MAT 3361, INTRODUCTION TO MATHEMATICAL LOGIC, Fall 2004

Lecture Notes: Analytic Tableaux

Peter Selinger

These notes are based on Raymond M. Smullyan, "First-order logic". Dover Publications, New York 1968.

1 Analytic Tableaux

Definition. A *signed formula* is an expression TX or FX, where X is an (unsigned) formula. Under a given valuation, a signed formula TX is called *true* if X is true, and *false* if X is false. Also, a signed formula FX is called *true* if X is false, and *false* if X is true.

We begin with the following observations about signed formulas:

Observation 1.1. *For all propositions X, Y:*

1a. $T(\neg X) \Rightarrow FX$.
1b. $F(\neg X) \Rightarrow TX$.
2a. $T(X \wedge Y) \Rightarrow TX$ *and* TY.
2b. $F(X \wedge Y) \Rightarrow FX$ *or* FY.
3a. $T(X \vee Y) \Rightarrow TX$ *or* TY.
3b. $F(X \vee Y) \Rightarrow FX$ *and* FY.
4a. $T(X \to Y) \Rightarrow FX$ *or* TY.
4b. $F(X \to Y) \Rightarrow TX$ *and* FY.

The method of analytic tableaux can be summarized as follows: To prove the validity of a proposition X, we assume FX and derive a contradiction, using the rules from Observation 1.1. In doing so, we follow a specific format which is illustrated in the following example.

Example 1.2. An analytic tableau proving the validity of $X = (p \vee (q \wedge r)) \to ((p \vee q) \wedge (p \vee r))$ is shown in Table 1. Note: the line numbers, such as (1), (2) etc, are not part of the formalism; they are only used for our discussion.

The initial premise on line (1) is of the form $F(Y \to Z)$, where $Y = (p \vee (q \wedge r))$ and $Z = ((p \vee q) \wedge (p \vee r))$. By rule 4b, we can conclude both TY and FZ.

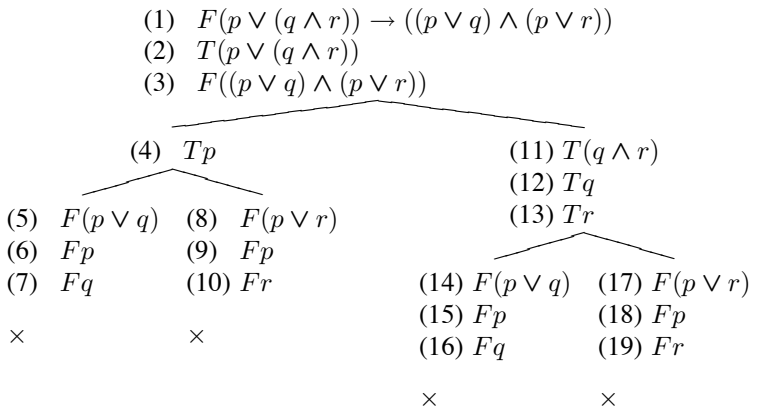


Table 1: An analytic tableau for $X = (p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r))$.

These are called the *direct consequences* of line (1), and we write them in lines (2) and (3), respectively. After we have done so, we say that line (1) has been *used*.

Now consider the formula in line (2), which is of the form $T(X \vee Y)$, where $X = p$ and $Y = q \wedge r$. From rule 3a, we may conclude that *either* TX *or* TY holds. Since the conclusion in this cases involves a choice between two possibilities, we say that the formula on line (2) *branches*. When using such a formula, the tableaux splits into two branches, one for each possibility. This has been done in lines (4) and (11).

Continuing in a similar fashion, we use up all the lines containing composite formulas. We say that a branch is *closed* if it contains both TX and FX, for some signed formula X. We mark closed branches with the symbol $\times$. A closed branch represents a contradiction. Since in this example, all branches are closed, we conclude that the original formula $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r))$ is valid.

As the example shows, note that there are essentially two types of signed formulas:

(A) signed formulas with direct consequences, which are $T(\neg X)$, $F(\neg X)$, $T(X \wedge Y)$, $F(X \vee Y)$, and $F(X \rightarrow Y)$, and

(B) signed formulas which branch, which are $F(X \wedge Y)$, $T(X \vee Y)$, and $T(X \rightarrow Y)$.

When using a formula of type (A), we simply add all of its direct consequences to each branch underneath the formula being used. When using a formula of type (B), we split each branch underneath the formula into two new branches. The rules for tableaux can be summarized schematically as follows:

$$\frac{T(\neg X)}{FX} \qquad \frac{F(\neg X)}{TX}$$

$$\frac{T(X \wedge Y)}{\begin{array}{c} TX \\ TY \end{array}} \qquad \frac{F(X \wedge Y)}{FX \mid FY}$$

$$\frac{T(X \vee Y)}{TX \mid TY} \qquad \frac{F(X \vee Y)}{\begin{array}{c} FX \\ FY \end{array}}$$

$$\frac{T(X \to Y)}{FX \mid TY} \qquad \frac{F(X \to Y)}{\begin{array}{c} TX \\ FY \end{array}}$$

Definition. A branch is said to be *complete* if every formula on it has been used. A tableau is said to be *completed* if every one of its branches is complete or closed. A tableau is said to be *closed* if all of its branches are closed. A tableau is said to be *open* if it is not closed, i.e., if it has at least one open branch.

We say that a formula X has been *proved by the tableaux method* if there exists a closed analytic tableau with origin FX.

Strategies. Our goal is to find a completed analytic tableau for a given formula. There are different strategies for deriving such a tableau.

Strategy 1 is to work systematically downwards: in this strategy, we never use a line until all lines above it have been used. When using this strategy, we are guaranteed to arrive at a completed tableau after a finite number of steps. However, strategy 1 is often more inefficient than the following strategy 2:

Strategy 2: give priority to lines of type (A). This means that we use up all lines of type (A) before using those of type (B). When following this strategy, we postpone the creation of new branches until absolutely necessary, thus keeping the size of the tableau smaller when compared to strategy 1.

Abbreviations. We often use the following shortcut notation when discussion signed formulas: We use the letter α to stand for any signed formula of type (A). In this case, we use α_1 and α_2 to denote the direct consequences (in the special case where there is only one direct consequence, we will set $\alpha_1 = \alpha_2$). All possibilities for α, α_1, and α_2 are summarized in the following table:

α	α_1	α_2
$T(X \wedge Y)$	TX	TY
$F(X \vee Y)$	FX	FY
$F(X \rightarrow Y)$	TX	FY
$T(\neg X)$	FX	FX
$F(\neg X)$	TX	TX

We also use the letter β to stand for any signed formula of type (B). In this case, we use β_1 and β_2 to denote the two alternative consequences. For reasons of symmetry, we further also allow β to also stand for a signed formula which is a negation, in which case we set $\beta_1 = \beta_2$. Thus, all possibilities for β, β_1, β_2 are summarized as follows:

β	β_1	β_2
$F(X \wedge Y)$	FX	FY
$T(X \vee Y)$	TX	TY
$T(X \rightarrow Y)$	FX	TY
$T(\neg X)$	FX	FX
$F(\neg X)$	TX	TX

With these conventions, the rules for tableaux can be written succinctly as follows:

$$\frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad\qquad \frac{\beta}{\beta_1 \mid \beta_2}$$

Definition. The *conjugate* of a signed formula FX is TX, and the conjugate of a signed formula TX is FX. We write $\bar{\varphi}$ for the conjugate of a signed formula φ.

We also observe the following: the conjugate of any α is some β, and in this case, $\overline{(\alpha_1)} = \beta_1$ and $\overline{(\alpha_2)} = \beta_2$. The conjugate of any β is some α, and in this case, $\overline{(\beta_1)} = \alpha_1$ and $\overline{(\beta_2)} = \alpha_2$. Moreover, for any signed formula φ, we have $\bar{\bar{\varphi}} = \varphi$.

2 Soundness and Completeness for Analytic Tableaux

Recall that we have called a branch of a tableau "complete" if every formula on it "has been used". With our convention on using the letters α and β for signed formulas, we may express this more precisely:

A branch θ of a tableau $\mathcal{T}$ is *complete* if for every $\alpha \in \theta$, both $\alpha_1, \alpha_2 \in \theta$, and for every $\beta \in \theta$, either $\beta_1 \in \theta$ or $\beta_2 \in \theta$.

As before, we say that a tableau $\mathcal{T}$ is *completed* if every branch θ of $\mathcal{T}$ is either closed or complete.

2.1 Tableaux and valuations

Let $[\![-]\!]$ be a valuation. We extend $[\![-]\!]$ to signed formulas in the obvious way by letting $[\![TX]\!] = [\![X]\!]$ and $[\![FX]\!] = 1 - [\![X]\!]$. Thus, FX is true under a given valuation iff X is false under that valuation.

Definition. Let $[\![-]\!]$ be a valuation. We say that a branch θ of a tableau $\mathcal{T}$ is *true* under $[\![-]\!]$ if for all $\varphi \in \theta$, $[\![\varphi]\!] = 1$. We say that $\mathcal{T}$ is *true* under $[\![-]\!]$ if there is at least one branch θ of $\mathcal{T}$ such that θ is true under $[\![-]\!]$.

2.2 Soundness

Soundness states that if a formula X is provable by the tableaux method, then X is a tautology.

Theorem 2.1 (Soundness). *Suppose X is a proposition, and $\mathcal{T}$ is a closed tableau with origin FX. Then X is a tautology.*

The proof depends on the following lemma:

Lemma 2.2. *Suppose $\mathcal{T}_1$ and $\mathcal{T}_2$ are tableaux such that $\mathcal{T}_2$ is an immediate extension of $\mathcal{T}_1$. Then $\mathcal{T}_2$ is true under every interpretation under which $\mathcal{T}_1$ is true.*

Proof. Suppose $\mathcal{T}_1$ is true under the given valuation $[\![-]\!]$. Then $\mathcal{T}_1$ has at least one true branch θ. Now $\mathcal{T}_2$ was obtained by adding one or two successors to the endpoint of some branch θ_1 of $\mathcal{T}_1$. If $\theta_1 \neq \theta$, then θ is still a branch of $\mathcal{T}_2$, hence $\mathcal{T}_2$ is true and we are done. Assume therefore that $\theta_1 = \theta$. Then θ was extended by one of the following operations:

(A) For some $\alpha \in \theta$, we have added α_1 or α_2, so $\theta \cup \{\alpha_1\}$ or $\theta \cup \{\alpha_2\}$ is a branch of $\mathcal{T}_2$. But $[\![\alpha]\!] = 1$, therefore $[\![\alpha_1]\!] = 1$ and $[\![\alpha_2]\!] = 1$, therefore $\mathcal{T}_2$ contains a true branch.

(B) For some $\beta \in \theta$, we have added both β_1 and β_2, so both $\theta \cup \{\beta_1\}$ and $\theta \cup \{\beta_2\}$ are branches of $\mathcal{T}_2$. But $[\![\beta]\!] = 1$, therefore $[\![\beta_1]\!] = 1$ or $[\![\beta_2]\!] = 1$, therefore $\mathcal{T}_2$ contains at least one true branch. □

Lemma 2.3. *Let $[\![-]\!]$ be a fixed valuation. For any tableau $\mathcal{T}$, if the origin of $\mathcal{T}$ is true under $[\![-]\!]$, then $\mathcal{T}$ is true under $[\![-]\!]$.*

Proof. This is an immediate consequence of the previous lemma, by induction: $\mathcal{T}$ is obtained from the origin by repeatedly extending the tableau in the sense of Lemma 2.2, at each step preserving truth. □

Proof of the Soundness Theorem: Let $\mathcal{T}$ be a closed tableau with origin FX, and let $[\![-]\!]$ be any valuation. Since $\mathcal{T}$ is closed, each branch contains some formula and its negation, and therefore $\mathcal{T}$ cannot be true under $[\![-]\!]$. From Lemma 2.3, it follows that the origin of $\mathcal{T}$ is false under $[\![-]\!]$, thus $[\![FX]\!] = 0$, thus $[\![X]\!] = 1$. Since $[\![-]\!]$ was arbitrary, it follows that X is a tautology. □

2.3 Completeness

Completeness is the converse of soundness: it states that if X is a tautology, then X is provable by the tableaux method. In fact we will prove something slightly stronger, namely, if X is a tautology, then *every* strategy for completing a tableaux for X will lead to a closed tableaux.

Theorem 2.4 (Completeness). *(a) Suppose X is a tautology. Then every completed tableau with origin FX must be closed.*

(b) Suppose X is a tautology. Then X is provable by the tableaux method.

The main ingredient in the proof is the notion of a Hintikka set.

Definition. Let S be a (finite or infinite) set of signed formulas. Then S is called a *Hintikka set* (or *downward saturated*) if it satisfies the following three conditions:

(H_0) There is no propositional variable p such that both $Tp \in S$ and $Fp \in S$.

(H_1) If $\alpha \in S$, then $\alpha_1 \in S$ and $\alpha_2 \in S$.

(H_2) If $\beta \in S$, then $\beta_1 \in S$ or $\beta_2 \in S$.

Note that, by definition, a complete non-closed branch θ is a Hintikka set.

If S is a set of signed formulas, we say that S is *satisfiable* if there exists a valuation $[\![-]\!]$ such that for all $\varphi \in S$, $[\![\varphi]\!] = 1$.

Lemma 2.5 (Hintikka Lemma). *Every Hintikka set is satisfiable.*

Proof. Let S be a Hintikka set, and define a valuation as follows: for any propositional variable p, let

$$\begin{array}{ll} [\![p]\!] = 1 & \text{if } Tp \in S, \\ [\![p]\!] = 0 & \text{if } Fp \in S, \\ [\![p]\!] = 1 & \text{if } Tp \not\in S \text{ and } Fp \not\in S. \end{array}$$

Note that, since S is a Hintikka set, we cannot have $Tp \in S$ and $Fp \in S$ at the same time. Thus, this is well-defined. We recursively extend $[\![-]\!]$ to composite formulas in the unique way.

We now claim that for all $\varphi \in S$, $[\![\varphi]\!] = 1$. This is proved by induction on φ. For atomic φ, this is true by definition. If φ is composite, then there are two cases:

(A) φ is some α. Then by (H_1), $\alpha_1 \in S$ and $\alpha_2 \in S$. By induction hypothesis, $[\![\alpha_1]\!] = 1$ and $[\![\alpha_2]\!] = 1$, therefore $[\![\alpha]\!] = 1$.

(B) φ is some β. Then by (H_2), $\beta_1 \in S$ or $\beta_2 \in S$. By induction hypothesis, $[\![\beta_1]\!] = 1$ or $[\![\beta_2]\!] = 1$, therefore $[\![\beta]\!] = 1$.

Thus, $[\![\varphi]\!] = 1$ for all $\varphi \in S$, and hence S is satisfiable as desired. □

Proof of the Completeness Theorem:

(a) Suppose X is a tautology, and $\mathcal{T}$ is some completed tableau with origin FX. Suppose θ is some branch of $\mathcal{T}$ which is not closed. Then θ is a Hintikka set by definition, hence satisfiable by the Hintikka Lemma. Thus, there exists some valuation $[\![-]\!]$ which makes θ true. Since $FX \in \theta$, we have $[\![FX]\!] = 1$, hence $[\![X]\!] = 0$, hence X is not a tautology, a contradiction. It follows that every branch of $\mathcal{T}$ is closed.

(b) It is easy to see that for any signed formula φ, there exists a completed tableau with origin φ. For example, such a tableau is obtained by following

Strategy 1 or Strategy 2 from Section 1. In particular, if X is a tautology, then there exists a completed tableau with origin FX, which is closed by (a), and hence X is provable by the tableaux method. □

2.4 Discussion of the proofs

We note the following features of the soundness and completeness proofs:

Soundness proof. The proof of soundness essentially proceeds by induction on tableaux, as is evident in the proof of Lemma 2.3. One fixes a valuation, then proves by induction that all derivations respect the given valuation.

This proof method is typical of soundness proofs in general. Compare this proof e.g. to the soundness proof for natural deduction in Lemma 1.5.1 of van Dalen's book. Most of the time, soundness proofs are relatively easy.

Completeness proof. The central part of any completeness proof is a satisfiability result: for a certain set of formulas, one must show that there exists a valuation making all the formulas true. To see why this is central, notice that the completeness property can be equivalently expressed as follows:

If X is *not* provable, then X is *not* a tautology.

Thus, it is natural to start by assuming that X is not provable (e.g., its analytic tableau does not close). Now one must prove that X is not a tautology, which amounts to finding a specific valuation which makes X false. In the case of analytic tableaux, this valuation is obtained using Hintikka's lemma.

Compare this to the completeness proof for natural deduction in Section 1.5 of van Dalen's book. It uses a completely different method, yet the central lemma is the one which allows one to construct a valuation, namely Lemma 1.5.11 (every consistent set is satisfiable). The method used for constructing a suitable valuation varies from proof system to proof system, and usually gets more difficult as features are added to the logic.

Propositional Proof Complexity: Past, Present, and Future

Paul Beame and Toniann Pitassi

Abstract. Proof complexity, the study of the lengths of proofs in propositional logic, is an area of study that is fundamentally connected both to major open questions of computational complexity theory and to practical properties of automated theorem provers. In the last decade, there have been a number of significant advances in proof complexity lower bounds. Moreover, new connections between proof complexity and circuit complexity have been uncovered, and the interplay between these two areas has become quite rich. In addition, attempts to extend existing lower bounds to ever stronger systems of proof have spurred the introduction of new and interesting proof systems, adding both to the practical aspects of proof complexity as well as to a rich theory. This note attempts to survey these developments and to lay out some of the open problems in the area.

1. Introduction

One of the most basic questions of logic is the following: Given a universally true statement (tautology) what is the length of the shortest proof of the statement in some standard axiomatic proof system? The propositional logic version of this question is particularly important in computer science for both theorem proving and complexity theory. Important related algorithmic questions are: Is there an efficient algorithm that will produce a proof of any tautology? Is there an efficient algorithm to produce the shortest proof of any tautology? Such questions of theorem proving and complexity inspired Cook's seminal paper on NP-completeness notably entitled "The complexity of theorem-proving procedures" [**35**] and were contemplated even earlier by Godel in his now well-known letter to von Neumann (see [**86**]).

The above questions have fundamental implications for complexity theory. As formalized by Cook and Reckhow [**37**], there exists a propositional proof system giving rise to short (polynomial-size) proofs of all tautologies if and only if NP equals co-NP. Cook and Reckhow were the first to propose a program of research aimed at attacking the NP versus co-NP problem by systematically studying and proving strong lower bounds for standard proof

systems of increasing complexity. This program has several important side effects.

First, standard proof systems are interesting in their own right. Almost all theorem-proving systems implement a deterministic or randomized procedure that is based on a standard propositional proof system, and thus upper and lower bounds on these systems shed light on the inherent complexity of any theorem-proving system upon which it is based. The most striking example is Resolution on which almost all propositional theorem provers (and even first-order theorem provers) are based.

Secondly, and of equal or greater importance, lower bounds on standard proof systems additionally prove that a certain class of algorithms for the satisfiability problem will fail to run in polynomial-time.

This program has led to many beautiful results as well as to new connections with circuit complexity within the last twenty years. In this article, we will try to highlight some of the main discoveries, with emphasis on the interplay between logic, (circuit) complexity theory, and combinatorics that has arisen. We omit all proofs; see [**92**] for a quite readable survey that includes detailed proofs of many of the earlier results.

In section 2, we define various proof systems that will be discussed throughout this article. In section 3, we review some of the main lower bounds that have been proven for standard proof systems, emphasizing the combinatorial techniques and connections to circuit complexity that have been shown. Finally in section 4, we list some of the main open questions and promising directions in the area.

2. Propositional proof systems

What exactly is a propositional proof? Cook and Reckhow were possibly the first to make this and related questions precise. They saw that it is useful to separate the idea of providing a proof from that of being efficient. Since there are only finitely many truth assignments to check, why not allow the statement itself as a proof? What extra value is there in a filled out truth table or a derivation using some axiom/inference scheme? The key observation is that a proof is easy to check, unlike the statement itself. Of course we also need to know the format in which the proof will be presented in order to make this check. That is, in order to identify some character string as a proof we must see it as an instance of some general format for presenting proofs. Therefore a *propositional proof system* S is defined to be a polynomial-time computable predicate S such that for all F,

$$F \in \text{TAUT} \quad \Leftrightarrow \quad \exists p.\ S(F,p). \tag{1}$$

That is, we identify a proof system with a polynomial time procedure that checks the correctness of proofs.[1] Property (1) ensures that the system S is

[1]Cook and Reckhow's definition is formally different although essentially equivalent to this one. They define a proof system as a polynomial-time computable *onto* function $f : \Sigma^* \rightarrow$TAUT which can be thought of as mapping each string viewed as potential proof

logically both sound and complete. The *complexity* $comp_S$ of a propositional proof system S is then defined to be the smallest bounding function $b : \mathbb{N} \to \mathbb{N}$ on the lengths of the proofs in S as a function of the tautologies being proved, i.e. for all F,

$$F \in \text{TAUT} \Leftrightarrow \exists p.|p| \leq b(|F|).\ S(F,p).$$

Efficient proof systems correspond to those of polynomial complexity; these are called *p-bounded*.

Given these definitions, many natural questions arise: How efficient are existing proof systems? How can one compare the relative efficiencies of proof systems? Can one classify proof systems using reduction as we do languages? Is there a proof system of optimal complexity (up to a polynomial)?

The key tool for comparing proof systems is p-simulation. A proof system T *p-simulates* a proof system S iff there is a polynomial-time computable function f mapping proofs in S into proofs in T, that is for all $F \in \text{TAUT}$, $S(F,p) \Leftrightarrow T(F,f(p))$. We use non-standard notation and write $S \leq_p T$ in this case. Clearly, it implies that $comp_T \leq comp_S^{O(1)}$. (One says that T *weakly p-simulates* S iff we have this latter condition but we do not know if such a reducing function f exists.) One says that S and T are *p-equivalent* iff each p-simulates the other. Obviously, two p-equivalent proof systems either are both p-bounded or neither is.

2.1. Frege and extended-Frege proofs. Cook and Reckhow did more than merely formalize the intuitive general notions of the efficiency of propositional proofs. They also identified two major classes of p-equivalent proof systems which they called Frege and extended-Frege systems in honor of Gottlob Frege who made some of the first attempts to formalize mathematics based on logic and set theory [**42, 43**] (and whose work is now best known as the unfortunate victim of Russell's famous paradox concerning the set of all sets that are not members of themselves).

A Frege system $\mathcal{F}$ is defined in terms of a finite, implicationally complete (enough to derive every true statement) set $\mathcal{A}_\mathcal{F}$ of axioms and inference rules. The general form of an inference rule is written as $\frac{A_1,\ldots,A_k}{B}$ where $A_1,\ldots,A_k$ and B are propositional formulas; the rule is an axiom if $k = 0$. A formula H follows from formulas $G_1,\ldots,G_k$ using this inference rule if there is a consistent set of substitutions σ of formulas for the variables appearing in the rule such that $G_i = A_i^\sigma$ for $i = 1,\ldots,k$ and $H = B^\sigma$.

For a Frege system $\mathcal{F}$, a typical set of axioms $\mathcal{A}_\mathcal{F}$ might include the axiom of the excluded middle $\frac{}{A \vee \neg A}$ or identity $\frac{}{A \to A}$ as well as the cut rule $\frac{A \vee C, \neg C \vee B}{A \vee B}$ or *modus ponens* $\frac{A, A \to B}{B}$. A proof of a tautology F in $\mathcal{F}$ consists of a finite sequence $F_1,\ldots,F_r$ of formulas, called *lines*, such that $F = F_r$

onto the tautology it proves. The analogous function f in our case would map (F,p) to F if $S(F,p)$ were true and would map it to a trivial tautology, $(x \vee \neg x)$, otherwise. In the converse direction one would define $S(F,p)$ to be true iff $f(p) = F$.

and each F_j either is an an instance of an axiom in $\mathcal{A}_\mathcal{F}$ or follows from some previous lines $F_{i_1} \ldots, F_{i_k}$ for $i_1, \ldots, i_k < j$ using some inference rule of $\mathcal{A}_\mathcal{F}$. An equivalent way of using a Frege system works backwards from $\neg F$ to derive a contradiction such as $p \wedge \neg p$. The size of a Frege proof is typically defined to be the total number of symbols occurring in the proof. The proof can also be tree-like or dag-like: in the tree-like case, each intermediate formula can be at used at most once in subsequent derivations; in the more general dag-like case, an intermediate formula can be used unboundedly many times. Krajíček [**60**] has shown that for Frege systems, there is not much loss in efficiency in going from a dag-like proof to a tree-like proof.

Various Frege systems (which have also been called Hilbert systems or Hilbert-style deduction systems) appear frequently in logic textbooks. However, it is difficult to find two logic textbooks that define precisely the same such system. Cook and Reckhow showed that these distinctions do not matter; namely, all Frege systems are p-equivalent. Furthermore, they showed that Frege systems were also p-equivalent to another class of proof systems appearing frequently in logic textbooks called sequent calculus or Gentzen systems. These systems manipulate pairs of sequences (or sets) of formulas, written as $\Gamma \rightarrow \Delta$, where Γ and Δ are sequences of formulas with the intended interpretation being that the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ. Therefore to prove a formula F in the sequent calculus, one proves the corresponding sequent $\rightarrow F$.

In any sequent calculus system, there is an underlying basis set of connectives, B. B can consist of unbounded fan-in or bounded fan-in connectives, and the only requirement is that B be a complete basis. (Typically, B is the standard basis consisting of $\wedge$, $\vee$ and $\neg$.) The only initial sequent is $A \rightarrow A$ for any formula A defined over B.

Additionally there are three types of rules for deriving new sequents from previous ones: (i) structural rules; (ii) logical rules; and (iii) the cut rule. Structural rules do not actually manipulate the underlying formulas of the sequent, but instead they allow one to operate on the sequences of formulas as sets rather than sequences. A typical structural rule is contraction which allows us to derive $\Gamma, A \rightarrow \Delta$ from $\Gamma, A, A \rightarrow \Delta$. The logical rules allow us to build larger formulas from previous ones, according to the truth-table definition of each of the connectives in B. More precisely, for each connective in B, there are two logical rules, one for introducing the connective on the left and one for introducing the connective on the right. For example, if $\wedge$ is in B, then the $\wedge$-left rule would allow us to derive $\Gamma, A \wedge B \rightarrow \Delta$ from $\Gamma, A, B \rightarrow \Delta$, and the $\wedge$-right rule would allow us to derive $\Gamma \rightarrow A \wedge B, \Delta$ from $\Gamma \rightarrow A, \Delta$ and $\Gamma \rightarrow B, \Delta$.

Of particular importance for sequent calculi is the cut rule: From $\Gamma, A \rightarrow \Delta$ and $\Gamma \rightarrow A, \Delta$, derive $\Gamma \rightarrow \Delta$. Gentzen showed that the cut rule is unnecessary but it may have a huge impact on proof length. If the cut rule is removed then one obtains a much weaker system than Frege systems. This system is known as analytic tableaux or cut-free LK. While analytic

tableaux are often more efficient than truth tables, somewhat surprisingly they cannot even p-simulate truth tables because their worst-case complexity is $\Omega(n!)$ rather than $O(n2^n)$ for truth tables (see [**92**]).

The other major class of proof systems identified by Cook and Reckhow includes systems that permit one to extend Frege proofs by introducing new propositional variables to stand for arbitrary formulas appearing in the proof. All such systems, which are called extended-Frege, are p-equivalent to each other. These systems appear to be much more succinct than Frege proofs and can conveniently express many mathemtical arguments quite naturally. It has been shown by Dowd (unpublished) and also by Krajíček and Pudlák [**61**] that extended-Frege proofs are also p-equivalent to substitution-Frege (sF) proofs. (In sF, one is allowed to use each line of a Frege proof immediately as if it were an axiom; that is, new lines follow from existing ones by substituting arbitrary formulas for their propositional variables.)

2.2. CNF refutations and Resolution. Using the construction of the standard reduction from SAT to 3-SAT, one can take an arbitrary propositional formula F and convert it to a CNF or 3-CNF formula in such a way that it has only polynomially larger size and is unsatisfiable iff the original formula was a tautology. To do this one adds new variables x_A to stand for each of its subformulas A and clauses to specify that the value at each connective is computed correctly as well as one clause of the form $\neg x_F$. In this way, one can consider any sound and complete system that produces refutations for CNF formulas as a general propositional proof system.

In the 1960's several such refutation systems were developed. The most powerful of these systems is Resolution [**84**], which in its propositional form is a very specialized form of a Frege proof system that can only manipulate clauses and has only one inference rule, the resolution rule

$$\frac{A \vee x,\ B \vee \neg x}{A \vee B}$$

called *resolution on variable* x and is a special form of cut. The contradictory formula to be derived is simply an empty clause (which can be seen as the result of resolving on clauses p and $\neg p$).

Resolution was pre-dated by two systems known as Davis-Putnam procedures which are still the most widely used in propositional theorem proving. The general idea of these procedures is to convert a problem on n variables to problems on $n-1$ variables by eliminating all references to some variable. The former [**40**] which we call DP does this by applying all possible uses of the resolution rule on a given variable to eliminate it. The latter [**39**], which we call DLL and is the form used today, branches based on the possible truth assignments to a given variable; although at first this does not look like Resolution, it is an easy argument to show that this second form is equivalent to the special class of tree-like Resolution proofs. As a proof system, Resolution is strictly stronger than DP [**50**] which is strictly stronger

than DLL [**92**]. The reasons for DLL's popularity are related to its proof search properties which we discuss below.

A more general but still restricted form of Resolution is called *regular* Resolution, which was introduced and analyzed by Tseitin [**89**]. A regular Resolution refutation is a Resolution refutation whose underlying directed acyclic graph has the property that along each path from the root (empty clause) to a leaf (initial clause), each variable is resolved upon at most once. It is not too hard to see that any minimal tree-like Resolution refutation is regular; also, the DP algorithm trivially produces a regular Resolution proof. For a period of 15 years after Tseitin's analysis, although there were improvements in the bounds derived for regular resolution [**49**], understanding general resolution seemed out of reach.

2.3. Circuit-complexity-based Proof Systems. One of the most powerful insights that has developed in the study of propositional proof complexity is that there is a parallel between circuit-based complexity classes and propositional proof systems. This insight was first made by Cook [**36**] where he established a close connection between polynomial-size extended Frege proofs and proofs using "polynomial-time" reasoning. (In more familiar terms, he showed that extended Frege proofs are the nonuniform analog of polynomial-time proofs systems such as PV or S^1_2, in the same way that polynomial-size circuits are the nonuniform analog of the complexity class P.) The same intuition (applied to other proof systems) was subsequently used to obtain other important results by Ajtai [**4, 1**] and Buss [**29**]. More generally, the parallel between circuit classes and proof systems has greatly broadened the range of proof systems that are typically considered and has led to new techniques for analyzing proof systems and circuit classes. We first briefly outline the general form of this correspondence and then we re-examine and refine some of the proof systems above in this light.

Typically, circuit-based complexity classes are defined by giving a structural characterization of a class of circuits and then placing some bound on the size of the circuits involved. For many circuit-based complexity classes C this size bound is polynomial. For any such class C we can consider a Frege-style proof system whose lines are circuits with the same structural characterization as the circuits defining C but which do not necessarily satisfy the size bound. The set of circuits of this type must be closed under substitution into any formula appearing in an axiom or inference rule of the system. Although the notation is not precise in general, we call such a proof system C-Frege. (Since we can always assume that our goal formula is in CNF, there is no problem representing it in C-Frege for virtually any non-trivial C.)

For example, the complexity class corresponding to the set of all polynomial size propositional formulas is NC^1 so NC^1-Frege would just be another name for Frege. It is also easy to observe that the extension rule of extended-Frege proofs builds circuits in terms of the original propositional variables

in which the new variables give the values computed by sub-circuits. Thus extended-Frege could also be called P/poly-Frege. Resolution is a Frege system that manipulates simple clauses (or, alternatively, terms if one views it dually as a proof system rather than a refutation system) but there isn't a convenient name for this complexity class of depth 1 formulas. Note that with this intuition it is clear that extended-Resolution, the natural generalization of Resolution that permits the introduction of new propositional variables, is p-equivalent to extended-Frege since it clearly can generate any circuit in P/poly with a polynomial number of extensions.

A very natural new proof system arising from this correspondence is a generalization of Resolution to arbitrary constant-depth, unbounded fan-in formulas/circuits (for constant depth there is no difference between polynomial-size formulas and circuits). This new system, AC^0-Frege, also known as constant- or bounded-depth Frege, first arose from the study of bounded first-order arithmetic (the word 'bounded' in that case derives from polynomial complexity bounds rather than from the depth). AC^0-Frege proofs derive from translations of proofs in certain systems of bounded first-order arithmetic [**70, 28**] which are restrictions/extensions of Peano arithmetic that model feasible inference. Although these motivations are important in the study of constructive logic, space considerations do not permit us to go into detail about them; we refer the interested reader to [**60**] where many of these connections are described in detail. Typically, lower bounds on the size of AC^0-Frege proofs can show that related first-order tautologies are unprovable in a given system of bounded arithmetic. These translations are analogous to those of Furst, Saxe, and Sipser [**47**], and Sipser [**87**] which convert oracle computations in the polynomial hierarchy to constant-depth unbounded fan-in circuits. In addition to AC^0-Frege, proof systems for $\mathsf{AC}^0[p]$-Frege and TC^0-Frege and their subclasses have also been studied extensively.

The correspondence between circuit classes and proof systems has not only been fruitful in developing ideas for new proof systems. It has also been the avenue for applying circuit lower bound techniques to propositional proofs. Some of the major progress of the last decade building on the original insight due to Ajtai [**4, 1**], has been in achieving lower bounds for AC^0-Frege proof systems and their extensions. In general, the intuition for this approach is that any tautology that needs to use in its proof some concept that is not representable in complexity class C will not be efficiently provable in C-Frege.

3. The State of the Art in Proof Complexity

We give a quick tour of the state of the art in propositional proof complexity. As is always inevitable in a short survey such as ours, space considerations do not permit us to do justice to the full range of results available. Although we will not always emphasize the connections very strongly, many

of these results have been inspired or derived from methods in circuit complexity, and conversely lower bounds for particular proof systems imply lower bounds for restricted families of algorithms for solving SAT. We expect this cross-fertilization to continue.

In keeping with the general program of proving that proof systems are not efficient, much work has been devoted to proving lower bounds on the sizes of proofs of specific tautologies. One can broadly characterize several classes of formulas for which these lower bounds have been shown. The first two classes consist of the propositional translations of combinatorial or counting principles. The first of these involves highly symmetric counting principles. A canonical example here is the translation of the pigeonhole principle, which prohibits 1-1 functions from m to n for $m > n$, as an unsatisfiable CNF formula $\neg PHP_n^m$ with mn atoms p_{ij} denoting whether or not i is mapped to j. Related principles include the counting principles $Count_p^n$ for $n \not\equiv 0 \pmod{p}$ which express the property that n cannot be perfectly partitioned into sets of size p and onto-PHP_n^m which only prohibits bijections rather than all 1-1 functions. The second class consists of much less symmetric counting principles, such as the 'odd-charged graph' principles for bounded-degree graphs, one for each graph, which express the property that the sum of the degrees in any of its sub-graphs is even; these are of particular interest when the underlying graph is a bounded-degree expander [**91**]. The third class of formulas are 'minterm-maxterm' formulas associated with any monotone function, which express the fact that any minterm and any maxterm of such a function must overlap; these are of particular interest when the function is known to require exponential-size monotone circuits. A related family of formulas is obtained by taking a language L that is in $\mathsf{NP} \cap \mathsf{coNP}$, and writing the formula expressing the fact that any instance x cannot have both a 'yes' witness and a 'no' witness. Finally, there are k-CNF formulas randomly chosen from an appropriate distribution.

We summarize known bounds for these formulas by considering the various proof systems one by one. Four basic methods can be identified for proving these lower bounds: (1) the bottleneck counting method; (2) the method of restrictions; (3) the interpolation method; (4) algebraic methods. The first three methods are quite specific whereas the last method is the youngest, probably the most powerful, and already has many facets.

Some of the state of our knowledge of proof complexity lower bounds can be summarized by the following chain

$$\text{Resolution} <_p \mathsf{AC}^0\text{-Frege} <_p \mathsf{AC}^0[p]\text{-Frege} \leq_p \mathsf{TC}^0\text{-Frege}.$$

The separations match all the major circuit complexity separations known with the notable exception of the last one. However, the prospect for proof complexity seems better than that for circuit complexity: The results of [**81**] indicate that to make further progress in circuit lower bounds will likely require very new, nonconstructive techniques. However such barriers do not currently exist in proof complexity: that is, proving superpolynomial lower

bounds for Frege systems might be no harder than what we currently know how to do. The techniques used to obtain some of the most recent results in proof complexity use insights from areas of mathematics apparently unrelated to those applied to show circuit complexity bounds.

3.1. Resolution. Resolution is the most well-studied model. Exponential lower bounds are now known for all of the major classes of formulas listed above. The first superpolynomial lower bound for Resolution was obtained by Tseitin in the 1960's for the odd-charged graph tautologies in the special case of regular Resolution [**89**]. Interestingly, obtaining an improvement of this bound to an exponential one by Galil [**49**] was a driving force behind some of the early work in the development of the theory of expander graphs [**48**].

There was a 15+ year gap before the first superpolynomial lower bound for proofs in general Resolution was obtained by Haken [**52**] who showed exponential lower bounds for the pigeonhole principle. Subsequently, exponential bounds have also been shown for the odd-charged graph formulas [**91**], random k-CNF formulas with various clause/variable ratios [**32, 46, 10, 9**], and minterm-maxterm formulas [**75**]. The proofs of all of the strongest forms of these bounds for Resolution, other than those for the minterm-maxterm formulas, involve a technique known as *bottleneck counting* due to Haken.

In this method, one views the proof as a directed acyclic graph of clauses and views the truth assignments as flowing from the root of the directed acyclic graph to a leaf, where an assignment flows through a clause C if and only if: (i) it flows through the parent clause of C and (ii) the assignment falsifies C. Each assignment can be seen to flow through a unique path in any Resolution refutation. The idea is to show that for the formula in question, there must exist a large set of truth assignments with the property that each must pass through a large clause. Since a large clause cannot falsify too many assignments, this implies that there must exist many large clauses and hence the proof must be large.

An essential lemma in any bottleneck counting argument is to show that any Resolution refutation of F must involve a large clause. Recently, it has been shown [**17**], using ideas from [**33**], that for a suitable choice of parameters this lemma is also sufficient, namely any Resolution refutation of small size can be converted into a refutation with only small clauses.

Another method used to obtain exponential Resolution lower bounds, used for example for the minterm-maxterm formulas, is the method of interpolation, which will be discussed in section 3.3.

In addition to lower bounds for general resolution, there is also practical interest in understanding the behavior of the special cases of DP and DLL algorithms. (See [**68, 34**].) Random k-CNF formulas have been of particular interest in this regard and there is a variety of results giving more precise bounds on their properties both as proof systems and as satisfiability algorithms at various clause/variable ratios [**32, 46, 10, 9**].

3.2. AC^0-Frege systems and their extensions. While Haken's bound for resolution was a major breakthrough, it is the paper by Ajtai [**4**] giving super-polynomial lower bounds for proofs of the pigeonhole principle in AC^0-Frege systems that has formed the basis of much of the research in proof complexity over the last decade. As mentioned above, this gave the first connection between the techniques of circuit complexity and those of proof complexity. Ajtai's result has been improved to exponential lower bounds and these apply to all the symmetric counting principles in the first class above, e.g. [**14, 74, 63**]. Despite this success, no lower bounds are known for AC^0-Frege proofs of formulas in the other classes above although there are certain other tautologies for which we know a superpolynomial separation as a function of the depth [**59**].

The *restriction method*, by which the lower bounds above were shown comes from Ajtai's paper [**4**]. The essence of this idea is to apply restrictions to try to simplify each of the formulas in the proof yet leave the input tautology still highly non-trivial. Thus the basic method is very similar to the random-restriction method, used to show that AC^0 cannot compute parity. However, it is necessarily more complex: Since the circuits appearing in a sound proof always compute the constant function 1, the usual simplification induced by restrictions applied to circuits must be replaced by one that includes a form of approximation as well.[2] Using this method, one shows that if the proof is too short, then there exists a restriction such that after applying the restriction to the short proof, what results is a very trivial proof of a formula of the same basic form, but on a reduced number of variables. Then a contradiction can be reached by showing that such a trivial proof cannot exist.

Once it is known that the pigeonhole principle is not provable in AC^0-Frege one can immediately obtain a stronger system by adding PHP_n^{n+1} as an axiom schema for arbitrary n; i.e., one is permitted to derive lines in the proof by substituting arbitrary formulas for the variables of some PHP_n^{n+1}. Ajtai [**5**] showed that even in this stronger system $Count_2^n$ does not have polynomial-size proofs. There is now some quite interesting structure known about the relative proof strength of these augmented AC^0-Frege systems in which some axiom scheme is added to the basic system.

In the system above in which PHP_n^{n+1} was added as an axiom schema, all of the $Count_p^n$ principles in fact are now known to require exponential size proofs [**15, 83**]; conversely, given any $Count_p^n$ axiom schema, any bounded-depth Frege proofs of PHP_n^{n+1} or onto-$PHP_n^{n+p^{\epsilon \log n}}$ requires exponential size but onto-PHP_n^{n+1} is trivial [**11**]. Thus PHP_n^{n+1} is exponentially stronger than onto-PHP_n^{n+1}. Quite precise conditions are now known

[2] [**4**] used the language of forcing to describe this approximation; the cleanest way of expressing this approximation is in terms of so-called k-evaluations which are described in [**13, 92, 11**] and are a modification of the definitions in [**63**].

under which exponential separations exist between the various $Count_p^n$ principles in this context [**23, 13, 11**].

The proofs for these results begin with the same restriction method strategy described above. However, with the additional axiom schema there is the further requirement that each line in the proof where the axiom schema is applied be simplified just as the rest of the proof is. The need to prove this latter requirement motivated the introduction of Nullstellensatz proofs [**13**] (see below) and the exponential separations are all derived from lower bounds on the degrees of such proofs.

The method involving Nullstellensatz degree lower bounds is not the only one that has been used for obtaining such separations. In proving the first super-polynomial separations between the various $Count_p^n$, Ajtai [**2**] used certain properties of concisely represented symmetric systems of linear equations [**3**] which he proved using structural results in the theory of representations of the symmetric group [**55**] over $GF(p)$ to show that the axiom schemas are appropriately simplified. This technique has recently been employed to give super-polynomial lower bounds for very powerful algebraic proof systems (see sections 3.4 and 3.5).

3.3. Cutting Planes and Interpolation. The method of using cutting planes for inference in the study of polytopes in integer programming was first described by Gomory [**51**], modified and shown to be complete by Chvátal [**30**], and first analyzed for its efficiency as a proof system in [**38**]. It is one of two classes of proof systems developed by representing unsatisfiably problems as integer or 0-1 programming problems, the other being a collection of systems due to Lovasz and Schrijver [**64**] which are described in detail in the open problems section.

Cutting Planes proofs manipulate integer linear inequalities: One can add inequalities or multiply them by positive constants but the truly powerful rule is the rounded division rule:

$$ca_1x_1 + ca_2x_2 + \ldots ca_kx_k \geq b \quad \Rightarrow \quad a_1x_1 + a_2x_2 + \ldots a_kx_k \geq \lceil b/c \rceil.$$

A refutation of a set of integer linear inequalities is a sequence of inequalities, where each inequality is either one of the original ones or follows from previous inequalities by applying one of the above rules, and where the final inequality is $1 \geq 0$. To refute a CNF formula, one first converts each clause into an equivalent integer linear inequality. Cutting planes proofs can simulate Resolution efficiently and easily prove all of the symmetric counting tautologies mentioned above.

Exponential lower bounds have been shown for cutting planes proofs of the minterm-maxterm formulas using a method called *interpolation*. In this method one begins with an unsatisfiable formula of the form $F = A(x, z) \wedge B(y, z)$ where we view x and y each as a vector of 'private' variables, and z as a vector of 'shared' variables. After any assignment τ to the common z variables is made, in the remaining formula $A(x, \tau) \wedge B(y, \tau)$, it must be the case that either A is unsatisfiable or B is unsatisfiable. The associated

interpolation problem for F takes as input an assignment τ to the common variables, and outputs A only when $A(x,\tau)$ is unsatisfiable, and outputs B only when $B(y,\tau)$ is unsatisfiable. Of course, sometimes both A and B may be acceptable answers. (This problem is called the interpolation problem since it is equivalent to Craig Interpolation. In the typical formulation, G is a tautological formula in the form $A'(x,z) \rightarrow B'(y,z)$. In our formulation F is $\neg G$, A is A', and B is $\neg B'$.)

For arbitrary F, it may be very difficult to solve the interpolation problem associated with F: if L is a decision problem in $\mathsf{NP} \cap \mathsf{co\text{-}NP}$, then if we define $A(x,z)$ to be a formula stating that x is a 'yes' witness for the instance z, and let $B(y,z)$ state that y is a 'no' witness for z, then the existence of a polynomial time algorithm for the interpolation problem would have surprising consequences for complexity theory! (See [**69**]). However it might still be that, whenever F (of the above form) has a short refutation in some proof system S, the interpolation problem associated with F has a polynomial-time solution. This possibility was first suggested by Krajíček. If this situation exists for proof system S, then we say that S has the *feasible interpolation property*. There is also a monotone version of feasible interpolation. Namely, $F = A(x,z) \wedge B(y,z)$ is monotone if z occurs only positively in A, and in this case the interpolation problem is monotone. S has the monotone feasible interpolation property if whenever F is monotone and has a short S-proof, then the associated interpolation problem has polynomial-size (uniform) monotone circuits.

Razborov [**80**] and independently Bonet, Pitassi and Raz [**20**] were the first to use the above idea to obtain exponential lower bounds for certain proof systems. In [**80**], a formula (formalizing that SAT does not have polynomial-size circuits) is constructed with the property that the associated interpolant problem has no polynomial-time circuits, under cryptographic assumptions. [**20**] constructs a monotone formula with the property that the associated interpolant problem has no monotone polynomial-time circuits (under no complexity assumptions). On the other hand they show that small-weight Cutting Planes has monotone feasible interpolation, thus implying exponential lower bounds. Pudlák [**75**] significantly extended the above ideas by showing that unrestricted Cutting Planes also has a form of monotone feasible interpolation. This combined with new exponential lower bounds for monotone real circuits [**75, 53**] gives unconditional lower bounds for Cutting Planes.

Feasible interpolation can thus give very good lower bounds for many proof systems (sometimes only under cryptographic assumptions). In addition to the unconditional lower bounds mentioned above, conditional lower bounds have been shown for all of the following systems: Resolution, Cutting Planes, Nullstellensatz [**33**], Polynomial Calculus [**76**], as well as any proof system where the underlying formulas in the proof have small probabilistic communication complexity [**20**]. Unfortunately there are strong negative results, showing that the interpolation method *cannot* be applied to give

lower bounds for the following proof systems, again under various cryptographic assumptions: Extended Frege [**62**], Frege, TC^0-Frege [**21**], and even AC^0-Frege [**65, 19**].

A disadvantage of the interpolation method for obtaining lower bounds is that it applies only to formulas of a very special form. Thus, for example, nothing is known about the length of the shortest Cutting Planes proofs for either the odd-charged graph formulas or for random formulas.

3.4. Algebraic proof systems. The Nullstellensatz [**13**] and Polynomial Calculus [**33**] proof systems are based on a special case of Hilbert's famous Nullstellensatz which relates the question of the non-existence of simultaneous zeros of a family of multivariable polynomials in certain fields to the question of the existence of coefficient polynomials witnessing that 1 is in the ideal generated by these polynomials. (In fact they use a generalization of this special case to rings as well as fields.)

To use this relation one first expresses an unsatisfiable Boolean formula as a system of constant-degree polynomial equations in some polynomial ring. For the propositional versions of the symmetric counting principles, these translations are quite natural, for example PHP_n^m when translated has polynomials $\sum_{j\leq n} x_{ij} - 1 = 0$ for each $i \leq m$, as well as $x_{ij}x_{i'j} = 0$ for each $i \neq i' \leq m$ and $j \leq n$. More generally, there are natural low-degree translations of arbitrary CNF formulas that are similar to those used in probabilistically checkable proofs (PCP) [**8, 41**]. To use these mechanisms to detect 0-1 solutions only, we add the equations $x^2 - x = 0$ for each variable x. Hilbert's Nullstellensatz implies that such a system $\{\vec{Q}(\vec{x}) = 0\}$ does not have a solution if and only if there exists a family of multi-variate polynomials $\vec{P}$ such that $\sum_i P_i(\vec{x})Q_i(\vec{x}) \equiv 1$. It is easy to see that the $x^2 - x = 0$ equations guarantee that degree at most n is sufficient.

The complexity in the Nullstellensatz proof system is simply the size of the dense representation of the coefficient polynomials and thus of the form $n^{O(d)}$ where d is the largest degree required. In the Polynomial Calculus proof system one does not need to explicitly write down these polynomials all at once but rather one can give a derivation that demonstrates their existence involving polynomials of low degree along the way. The size of each of these polynomials is also based on this dense representation.

A variety of Nullstellensatz lower bounds are known for the symmetric counting principles but no Nullstellensatz lower bounds are known for the other formulas above. (For example, the odd-charged graph tautologies have easy proofs over $\mathbb{Z}_2$.)

One drawback of the Nullstellensatz system (although not Polynomial Calculus) is that a simple chain of inference of length n using modus ponens requires non-constant degree $\Theta(\log n)$ [**26**]. It is even possible that certain principles may be proved using degree 2 using Polynomial Calculus but require degree $\Omega(\sqrt{n})$ Nullstellensatz proofs [**33**].

Although it is trivial to prove $Count_r^n$ in constant degree in $\mathbb{Z}_r$, degree lower bounds of $n^{\Omega(1)}$ have been shown in $\mathbb{Z}_r$ for PHP_n^m [**12**], $Count_s^n$ for most $s \neq r$ [**23**] and onto-$PHP_n^{n+r^{\omega(1)}}$ [**11**]. So far, all of the Nullstellensatz bounds mentioned were shown using the notion of a dual *design* [**12, 25**]. This is typically a combinatorial construction that guarantees that one can derive a solution to a set of dual equations that express the coefficient of the constant term in $\sum_i P_i \cdot Q_i$ as a linear combination of higher degree coefficients in the indeterminate coefficients of the P_i. Since $\sum_i P_i \cdot Q_i$ is supposed to represent the polynomial 1 this would be impossible. [**23**] introduced a nice technique (used in [**11**]) that makes such designs easier to construct.

Using a more general class of designs the degree bounds for PHP_n^m were were improved to degree $n/2+1$ by Razborov [**77**]. Remarkably, Razborov's result also applies to the Polynomial Calculus proof system and was not only the first non-trivial lower bound shown for that system but also is one of strongest known for the Nullstellensatz system. It involves an explicit computation of the Groebner basis of the ideal generated by the PHP_n^m equations. Razborov also extends this lower bound to obtain stronger, nearly linear, degree lower bounds in Polynomial Calculus for related tautologies as a function of the number of variables.

Recently, Krajicek [**58**] has shown non-constant degree lower bounds for $Count_s^n$ in the Polynomial Calculus proof system over $\mathbb{Z}_r$ by extending the results of Ajtai regarding symmetric linear equations and the structure of representations of the symmetric group [**3**] mentioned earlier.

3.5. $\mathsf{AC}^0[r]$-Frege systems. We have already seen how one can extend AC^0-Frege proofs by adding axioms for counting modulo r. A far more general, and in some sense, more natural way to add the power of modular counting to a proof system is to include it fundamentally in the structure of the objects about which one reasons; that is, to introduce modular counting connectives into the lines of the proofs themselves and new inference rules for manipulating these formulas. $\mathsf{AC}^0[r]$-Frege is precisely such a system. Even the Polynomial Calculus proof system modulo r may be viewed as a subsystem of an $\mathsf{AC}^0[r]$-Frege proof system in which all the modular connectives are at the top [**72**].

At present there are no lower bounds known for $\mathsf{AC}^0[r]$-Frege, even when r is a prime. One program for obtaining such bounds was laid out in [**23**]. where it is shown how to convert an $\mathsf{AC}^0[p]$-Frege proof of F to a Polynomial Calculus proof of a system that involves the polynomials for F plus certain low degree extension polynomials (which mimic the low degree approximations used by [**78, 88**] for $\mathsf{AC}^0[p]$ lower bounds).

3.6. Frege systems and TC^0-Frege systems. Just as $\mathsf{AC}^0[r]$-Frege proofs include counting modulo r as a first-class concept, so TC^0-Frege proofs

include counting up to a threshold as a first-class concept. Loosely speaking, TC^0-Frege proofs are proofs where the underlying class of formulas are small-weight, constant-depth threshold formulas. Thus, for example, Cutting Planes proofs can be viewed as a special case of restricted TC^0-Frege proofs.

We only have partial results on TC^0-Frege proofs. Maciel and Pitassi [**66**] have shown proof-theoretic analogues of circuit complexity constructions [**7**] to relate TC^0- and $\mathsf{AC}^0[p]$-Frege proofs. In particular they show how one can convert polynomial-size $\mathsf{AC}^0[2]$-Frege proofs into restricted quasipolynomial-size depth 3 TC^0-Frege proofs.

The potential list of candidate hard problems for TC^0-Frege is quite short, in part because there are efficient TC^0-Frege proofs for so many formulas for which lower bounds are known in other systems and because the basic techniques for dealing with these formulas fundamentally break down. There are polynomial-size TC^0-Frege proofs of all of the symmetric counting principles [**29**] as well as the odd-charged graph principles [**91**]. And, as mentioned above, the interpolation method cannot apply to TC^0-Frege assuming that factoring Blum integers is hard [**21**]. Since TC^0-Frege proofs are special cases of Frege proofs, the same problems apply to Frege proofs as well. Some candidates for hard tautologies for these systems have been suggested in [**18**].

3.7. Optimal proof systems. Research in proof complexity was originally motivated in part as a way of proving $\mathsf{NP} \neq \mathsf{co\text{-}NP}$, by proving superpolynomial lower bounds for increasingly powerful proof systems. An important question is whether such a chain of results will ever actually lead us to a proof of $\mathsf{NP} \neq \mathsf{co\text{-}NP}$. In other words, is there an optimal proof system? This question is quite important and is still open. However, some partial results have been obtained [**67, 61, 16**] relating this existence to the equivalence of certain complexity classes.

3.8. Proof Search. While lengths of proofs are important, it is also important to be able to *find* proofs quickly. Clearly, if we know that a proof system S is not polynomially-bounded, then no efficient deterministic procedure can exist that will produce short proofs of all tautologies. But is it possible to find short proofs of all tautologies that have short proofs? To this end, [**21**] defines a proof system S to be *automatizable* if there exists a deterministic algorithm that takes as input a tautology F, and outputs an S-proof of F in time polynomial in the size of the shortest S-proof of F.

Automatizability is very important for automated theorem proving, and is very similar to an older concept, k-provability. The k-provability problem for a proof system S is as follows. The input is a formula F, and a number k, and the input should be accepted if and only if F has an S-proof of size at most k. Clearly a proof system S is not automatizable if the k-provability problem cannot be approximated to within any polynomial factor.

Which proof systems are automatizable, and for which proof systems is k-provability hard? It has been shown [**6, 22, 54**] that the k-provability problem is NP-hard for essentially every standard propositional proof system, and furthermore using the PCP theorem, that the k-provability problem cannot be approximated to within any constant factor, unless $\mathsf{P} = \mathsf{NP}$.

The above hardness results show that finding good estimates of the proof length is hard, even for very simple proof systems such as Resolution. But it may still be that most proof systems are automatizable. However, under stronger assumptions, one can show that many proof systems are not automatizable. These results are shown by exploiting a connection between interpolation and automatizability. In particular, it can be shown that if a proof system S does not have feasible interpolation, then this implies that S is not automatizable. Thus, feasible interpolation gives us a formal tradeoff between the complexity/strength of S and the ability to *find* short proofs quickly. Using this connection, it has been shown that AC^0-Frege proofs as well as any proof system that can p-simulate AC^0-Frege, is not automatizable, under cryptographic assumptions. (See [**62, 21, 65, 19**].)

Are there any proof systems that are automatizable? Both the Nullstellensatz and Polynomial Calculus proof systems as well as DLL are actually search procedures as well as nondeterministic algorithms. But, unlike DLL, the Nullstellensatz and Polynomial Calculus algorithms are guaranteed to find short proofs if they exist [**13, 33**]; that is, they are automatizable. (Any proof of degree d in n variables may be found using linear algebra in time $n^{O(d)}$.) Nullstellensatz proofs may be exponentially smaller than bounded-depth Frege proofs, but they also may be exponentially larger than Resolution proofs [**33**]. Polynomial Calculus proofs are at worst quasi-polynomially larger than the best proofs under any DLL algorithm and from this one can derive a method of searching for short DLL proofs that is guaranteed to succeed [**33**]. In [**10**], this algorithm is converted to a direct search procedure for such proofs. It appears fruitful to investigate Polynomial Calculus proofs as a theorem-proving tool to see if they can be refined to compete with DLL algorithms. Preliminary results of this form appear in [**33**].

4. Open Problems

FIND HARD TAUTOLOGIES FOR TC^0-FREGE AND FREGE. No examples of tautologies are known for which Frege proofs even require a super-linear number of distinct subformulas. One difficult problem that is faced when trying to prove lower bounds for Frege or Extended Frege systems is that there is a surprising lack of hard candidate tautologies. Most of the lower bounds proven thus far have been for various counting principles, all of which have polynomial-size TC^0-Frege or Frege proofs. (Even the minterm-maxterm formulas can be viewed as an application of a counting principle.)

Some candidate hard examples have been suggested in [**18**] including random k-CNF formulas. Another family of examples that is of particular interest for complexity theory, are at least as hard to prove as many of

the classes of formulas discussed in this paper, but are only believed to be tautological are particular formulas given by Razborov [**80**] stating that NP is not contained in P/poly.

There is an even larger gap in tautologies that seem to separate extended Frege from Frege systems. In fact, we know of no convincing combinatorial tautologies that might have polynomial-size extended Frege proofs, but require exponential-size Frege proofs. In [**18**], several tautologies based on linear algebra are suggested to give a quasipolynomial separation between extended Frege and Frege systems. A very simple such example, suggested by Cook and Rackoff, is the propositional form of the Boolean matrix product implication $AB = I \Rightarrow BA = I$.

HOW HARD ARE RANDOM k-CNF FORMULAS? The only lower bounds known for unsatisfiability proofs of random formulas are for forms of Resolution. What about other proof systems such as: bounded-depth Frege proofs? cutting planes proofs? Nullstellensatz or polynomial calculus proofs? The absence of random unsatisfiable formulas in the list of lower bounds for systems other than Resolution is quite noteworthy, especially given the lack of good upper bounds for proofs of these formulas in any system, even extended-Frege.

Many NP-complete graph problems are easy on the average for the natural random graph probability distributions. Random k-CNF formulas under the analogous probability distributions seem surprisingly hard in the region of probabilities for which the formulas are likely unsatisfiable [**32, 9**]. Is k-UNSAT hard on the average in this sense?

For example, the best upper bound for any search algorithm for unsatisfiability proofs of random m-clause n-variable 3-CNF formulas is $2^{O(n^2/m)}$ with probability $1 - o(1)$ in n and this is tight for a class of DLL algorithms [**9**]. For Resolution the lower bound for this problem is nearly $2^{\Omega(n^5/m^4)}$ with probability $1 - O(1)$ in n [**9**]. Can these be improved?

SUPERPOLYNOMIAL LOWER BOUNDS FOR $\mathsf{AC}^0[r]$ FREGE?. Studying $\mathsf{AC}^0[r]$-Frege is a natural next step in proving lower bounds for proof systems, in particular when r is a prime. We already mentioned the program for obtaining such lower bounds in [**23**]. For this system we do have a natural candidate for a hard tautology, namely $Count_p^n$ for prime $p \neq r$. Such a lower bound would further the circuit/proof system correspondence by extending proof complexity lower bounds to the natural analogue of the Razborov-Smolensky circuit lower bounds [**78, 88**].

POLYNOMIAL CALCULUS IN A THEOREM PROVER? BETTER RESOLUTION PROOF SEARCH? Designing efficient theorem provers for the propositional calculus is an important practical question. To date, DLL algorithms are the champion theorem provers although they are theoretically quite weak as proof systems. A recent challenger seems promising: A variant of the Groebner basis algorithm has been used to find Polynomial Calculus proofs [**33**] and build a fairly efficient theorem prover. Can this be tuned to compete with DLL algorithms?

Clegg *et al.* [**33**] also give simulations of DLL and Resolution by the Polynomial Calculus. One method of improving the competitiveness of Polynomial Calculus as a theorem prover would be to improve these simulations. This is also related to the question of improving the more direct methods for proof search for Resolution and DLL [**10, 17**] that were inspired by these Polynomial Calculus simulations.

NEW PROOF SYSTEMS FROM NP-COMPLETE PROBLEMS. An appealing direction in proof complexity is the possibility of using natural domains that contain NP-hard problems to seek out new and interesting proof systems which reason about objects from radically different domains from Boolean formulas. Cutting planes come from integer programming, Nullstellensatz and Polynomial Calculus systems come from systems of polynomial equations. Pitassi and Urquhart [**73**] considered the Hajos calculus for non-3-colorability which they found, surprisingly, to be equivalent to extended-Frege proofs. It is likely that there is more to be mined in this search.

WEAK PIGEONHOLE PRINCIPLE AND THE LIMITS OF RESOLUTION. It is known that for $n < m < n^2/\log n$, PHP_n^m requires superpolynomial-size Resolution proofs [**52, 27**]. Originally it was conjectured that for any $m > n$, any Resolution refutation of PHP_n^m would require size exponential in n. However, this conjecture was shown to be false for large enough m [**24**]. Moreover, it appears that the standard method, the bottleneck counting technique, cannot be applied to obtain lower bounds for $m > n^2$. An interesting open problem is to prove lower bounds for PHP_n^m for $m > n^2$. This would likely give rise to a new lower bound method for Resolution. Additionally the complexity of the weak pigeonhole principle in various proof systems is interesting in its own right: it is known [**71**] that one can prove the existence of infinitely many primes in systems as weak as polynomial-size, bounded-depth Frege, assuming the weak-pigeonhole principle as an axiom schema. More generally, the weak pigeonhole principle can be used to carry out most combinatorial counting arguments, and is closely connected to approximate counting.

Partial results for the weak pigeonhole principle have recently been obtained [**82**]. There is a very tight connection between regular Resolution refutations and read-once branching programs which generalizes the equivalence between tree resolution and DLL mentioned earlier. Let F be an unsatisfiable formula in conjunctive normal form. The search problem associated with F takes as input a truth assignment τ to the underlying variables of F, and outputs a clause in F that is set to false by τ. Krajíček [**60**] has shown that for any F, the minimal size Resolution refutation of F is essentially equivalent to the minimal size read-once branching program to solve the related search problem. This idea was exploited in [**82**] to obtain some restricted lower bounds for Resolution proofs of weak versions of the pigeonhole principle.

Another, even older, principle for which no superpolynomial Resolution lower bounds are known is the domino-tiling principle. More details about this old problem can be found in [**90**].

LOVASZ-SCHRIJVER PROOF SYSTEMS. A variety of inference systems for 01-programming are described by Lovasz and Schrijver [**64**] in a paper that is primarily concerned with their implications for linear programming. Like cutting planes these proof systems represent statements using systems of linear inequalities, but unlike cutting planes, they replace rounding by an ability to take linear combinations of derived degree 2 terms in intermediate steps obtained by multiplying certain inequalities and including the equations $x^2 = x$ provided all degree 2 terms cancel. (There are several versions depending on whether one can (1) add the squares of arbitrary linear terms to the appropriate side of the inequalities or (2) multiply inequalities by x or $1 - x$ or, more generally, (3) multiply by any linear term previously shown to be positive.) Lovasz and Schrijver prove a number of properties of their systems that allow one to precisely determine the depth of the proofs involved but they do not consider the issue of proof size directly.

It turns out that, despite the absence of the rounded division of cutting planes, one can easily simulate Resolution and prove the pigeonhole principle in polynomial size in their weakest system. (To see an example of the system in action consider that if $1 - a - b \geq 0$, $1 - a - c \geq 0$, $1 - b - c \geq 0$ holds for $0 \leq a \leq 1$, $0 \leq b \leq 1$, $0 \leq c \leq 1$ then at most one of a, b, c is 1 and so $a+b+c \leq 1$. This is obtained by computing $0 \leq a(1-a-b) = a-a^2-ab = -ab$, $0 \leq a(1-a-c) = a-a^2-ac = -ac$, and $0 \leq (1-a)(1-b-c) = 1-a-b-c+ab+ac$. Adding these inequalities leads to the desired result.)

However other problems are less clear. In particular, Lovasz (private communication) suggested the problem of deriving the maximum independent set size of graphs that can be expressed as the line graphs of odd cliques. One can easily show that this problem is completely equivalent to the parity principle and requires linear depth in the number of vertices, but this is far from either an algorithm or a lower bound. This question is intimately tied to the question of whether these systems can simulate cutting planes efficiently. Assuming that they cannot, it may be even more interesting to consider what the combination of these new systems with cutting planes is capable of.

ODD-CHARGED GRAPHS HARD FOR AC^0-FREGE? Urquhart has suggested that a study of Tseitin's odd-charged graph tautologies [**89**] for appropriate graphs [**91**] in bounded-depth Frege systems might also lead to lower bounds for random formulas since it appears very difficult to apply Ajtai's program to them. No results are know for these even on depth 2 Frege systems.

MORE GENERAL LOWER BOUND TECHNIQUE FOR CUTTING PLANES? The interpolation method has been successfully applied to obtain unconditional lower bounds for Cutting Planes proofs. However, we are quite far from understanding more generally what types of tautologies are hard for Cutting

Planes. In particular, are random formulas hard? Other families of tautologies possibly hard for Cutting Planes are the odd-charged graph tautologies.

EXPONENTIAL BOUNDS FOR $Count_p^n$/ONTO-PHP IN POLYNOMIAL CALCULUS The lower bounds of [**58**] are barely super-polynomial and it seems unlikely that the methods used can produce much larger lower bounds. The lower bounds of Razborov [**77**] involve a direct computation of the low degree elements of the Groebner basis of the PHP_n^m polynomials. This computation relies heavily on the independence of the degree one PHP_n^m polynomials. All the other counting principles above do not have this property and it does not appear that the same method can be applied to them. Is there a more general methodology that does not require direct computation of the basis?

PROBABILISTICALLY CHECKABLE ALGEBRAIC PROOFS. The Nullstellensatz and Polynomial Calculus proof systems use the dense representation of multivariate polynomials. What happens if one modifies these proof systems to manipulate polynomials as straight-line programs that compute them rather than writing them out explicitly? One difficulty is that testing the equality of polynomials represented this way is not known to be efficiently computable deterministically. However, there are efficient probabilistic algorithms to do this check [**85**]. Such a proof system would lie outside our proof system definition since the verification predicate S would only be probabilistically checkable. See [**72**] for more details.

UNSATISFIABILITY THRESHOLD FOR RANDOM k-SAT. Although this is not a proof complexity question *per se* it is of interest in understanding the proof complexity of random k-CNF formulas. There have been a number of papers analyzing the satisfiability properties of these formulas as a function of their clause-variable ratios. Recently, it has been shown that there is a sharp threshold behavior for such formulas [**44**] but it is not known precisely where such a threshold lies or even if it approaches some fixed limit. In general it is known that it lies between $2^k/k$ and $2^k \ln 2$ [**32, 31**] and for $k = 3$ it is known to lie between 3.003 and 4.598 [**45, 57**] and is conjectured to be around 4.2 [**56**]. A related question is whether or not the satisfiability problem is easy right up to the threshold.

NATURAL PROOFS IN PROOF COMPLEXITY? In circuit complexity, Razborov and Rudich [**81**] suggest that, subject to some plausible cryptographic conjectures, current techniques will be inadequate for obtaining super-polynomial lower bounds for TC^0-circuits. To this point, proof complexity has made steady progress at matching the superpolynomial lower bounds currently known in the circuit world (albeit using different techniques), and the major remaining analogous result (a lower bound for $\mathsf{AC}^0[p]$-Frege proofs) also may be within reach. Unlike the circuit world, however, there is no analogue of Shannon's counting argument for size lower bounds for random functions and there does not seem any inherent reason for TC^0-Frege to be beyond current techniques. While it is true that one can show the failure of TC^0-Frege interpolation (also depending on cryptographic conjectures), this

applies to AC^0-Frege as well for which we do have lower bounds. Is there any analogue of natural proofs in proof complexity? (Razborov [**79**] has looked at the quite different question of examining particular proof systems and showing that any efficient proofs of lower bounds for circuits using such systems automatically naturalize.)

[1] M. Ajtai. The complexity of the pigeonhole principle. *Combinatorica*, 14(4):417–433, 1994.

[2] M. Ajtai. The independence of the modulo p counting principles. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pages 402–411, Montréal, Québec, Canada, May 1994.

[3] M. Ajtai. Symmetric systems of linear equations modulo p. Technical Report TR94-015, Electronic Colloquium in Computation Complexity, `http://www.eccc.uni-trier.de/eccc/`, 1994.

[4] Miklós Ajtai. The complexity of the pigeonhole principle. In *29th Annual Symposium on Foundations of Computer Science*, pages 346–355, White Plains, NY, October 1988. IEEE.

[5] Miklós Ajtai. Parity and the pigeonhole principle. In Samuel R. Buss and P. J. Scott, editors, *Feasible Mathematics*, pages 1–24, A Mathematical Sciences Institute Workshop, Ithaca, NY, 1990. Birkhäuser.

[6] M. Alekhnovich, S. Buss, S. Moran, and T. Pitassi. Minimum Propositional proof length is NP-hard to linearly approximate. Manuscript. 1998.

[7] E. Allender and U. Hertrampf. Depth reduction for circuits of unbounded fan-in. *Information and Computation*, 112(2):217–238, August 1994.

[8] S. Arora, C. Lund, Rajeev Motwani, M. Sudan, and Márió Szegedy. Proof verification and hardness of approximation problems. In *Proceedings 33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, PA, October 1992. IEEE.

[9] P. Beame, R. Karp, T. Pitassi, and M. Saks. On the complexity of unsatisfiability of random k-CNF formulas. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, Dallas, TX, May 1998.

[10] P. Beame and T. Pitassi. Simplified and improved resolution lower bounds. In *37th Annual Symposium on Foundations of Computer Science*, pages 274–282, Burlington, VT, October 1996. IEEE.

[11] P. Beame and S. Riis. More on the relative strength of counting principles. In P. Beame and S. Buss, editors, *Proof Complexity and Feasible Arithmetics*, DIMACS. American Mathematical Society, 1998. To appear.

[12] Paul W. Beame, Stephen A. Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of NP search problems. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, pages 303–314, Las Vegas, NV, May 1995.

[13] Paul W. Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, and Pavel Pudlák. Lower bounds on Hilbert's Nullstellensatz and propositional proofs. *Proc. London Math. Soc.*, 73(3):1–26, 1996.

[14] Paul W. Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, Pavel Pudlák, and Alan Woods. Exponential lower bounds for the pigeonhole principle. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, pages 200–220, Victoria, B.C., Canada, May 1992.

[15] Paul W. Beame and Toniann Pitassi. An exponential separation between the matching principle and the pigeonhole principle. In *8th Annual IEEE Symposium on Logic in Computer Science*, pages 308–319, Montreal, Quebec, June 1993.

[16] S. Ben-David and A. Gringauze. On the existence of propositional proof systems and oracle-relativized propositional logic. Technical Report TR98-021, Electronic Colloquium in Computation Complexity, `http://www.eccc.uni-trier.de/eccc/`, 1998.

[17] E. Ben Sasson and A. Wigderson. Private communication.
[18] M. Bonet, S. Buss, and T. Pitassi. Are there hard examples for Frege systems? In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 30–56. Birkhauser, 1995.
[19] M. Bonet, C. Domingo, and R. Gavalda. No feasible interpolation or automatization for AC^0-Frege proof systems. Manuscript. 1998.
[20] M. Bonet, T. Pitassi, and R. Raz. Lower bounds for cutting planes proofs with small coefficients. *Journal of Symbolic Logic*, 62(3):708–728, September 1997.
[21] M. Bonet, T. Pitassi, and R. Raz. No feasible interpolation for TC^0 frege proofs. In *38th Annual Symposium on Foundations of Computer Science*. IEEE, October 1997.
[22] S. Buss. On Godel's theorems on lengths of proofs II: Lower bounds for recognizing k symbol provability. In P. Clote and J. Remmel, editors, *Feasible mathematics II*, pages 57–90. Birkhauser-Boston, 1995.
[23] S. Buss, R. Impagliazzo, J. Krajíček, P. Pudlák, A. A. Razborov, and J. Sgall. Proof complexity in algebraic systems and bounded depth Frege systems with modular counting. *Computation Complexity*, 6(3):256–298, 1997.
[24] S. Buss and T. Pitassi. Resolution and the weak pigeonhole principle. In *Proceedings of Computer Science Logic*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
[25] S. R. Buss. Lower bounds on Nullstellensatz proofs via designs. In P. W. Beame and S. R. Buss, editors, *Proof Complexity and Feasible Arithmetics*, DIMACS, pages 59–71. American Math. Soc, 1997.
[26] S. R. Buss and T. Pitassi. Good degree bounds on Nullstellensatz refutations of the induction principle. In *Proceedings of the Eleventh Annual Conference on Computational Complexity (formerly: Structure in Complexity Theory)*, pages 233–242, Philadelphia, PA, May 1996. IEEE.
[27] S. R. Buss and G. Turan. Resolution proofs of generalized pigeonhole principles. *Theoretical Computer Science*, 62(3):311–317, December 1988.
[28] Samuel R. Buss. *Bounded Arithmetic*. Bibliopolis, Napoli, 1986. Volume 3 of Studies in Proof Theory.
[29] Samuel R. Buss. Polynomial size proofs of the pigeonhole principle. *Journal of Symbolic Logic*, 57:916–927, 1987.
[30] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4, 1973.
[31] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *33nd Annual Symposium on Foundations of Computer Science*, pages 620–627, Pittsburgh, PA, October 1992. IEEE.
[32] V. Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.
[33] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Gröbner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 174–183, Philadelphia, PA, May 1996.
[34] S. Cook and D. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *DIMACS Series in Theoretical Computer Science*, 1997.
[35] Stephen A. Cook. The complexity of theorem proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, May 1971.
[36] Stephen A. Cook. Feasibly constructive proofs and the propositional calculus. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, pages 83–97, Albuquerque, NM, May 1975.
[37] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[38] W. Cook, C. R. Coullard, and G. Turan. On the complexity of cutting plane proofs. *Discrete Applied Mathematics*, 18:25–38, 1987.

[39] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[40] M. Davis and H. Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7:201–215, 1960.

[41] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost *NP*-complete. In *Proceedings 32nd Annual Symposium on Foundations of Computer Science*, pages 2–12, San Juan, Puerto Rico, October 1991. IEEE.

[42] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinben Denkens*. Nebert, Halle, 1879.

[43] G. Frege. *Grundgesetze der Arithmetik*. Nebert, Halle, 1893,1901.

[44] E. Friedgut. Necessary and sufficient conditions for sharp thresholds of graph properties, and the k-sat problem. Preprint, May 1997.

[45] A. Frieze and S. Suen. Analysis of two simple heuristics on a random instance of k-SAT. *Journal of Algorithms*, 20(2):312–355, 1996.

[46] Xudong Fu. *On the complexity of proof systems*. PhD thesis, University of Toronto, 1995.

[47] M. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, April 1984.

[48] O. Gabber and Z. Galil. Explicit constructions of linear size superconcentrators. In *20th Annual Symposium on Foundations of Computer Science*, pages 364–370, New York, NY, 1979. IEEE.

[49] Z. Galil. On the complexity of regular resolution and the Davis-Putnam procedure. *Theoretical Computer Science*, 4:23–46, 1977.

[50] A. Goerdt. Regular resolution versus unrestricted resolution. *SIAM Journal on Computing*, 22(4):661–683, 1993.

[51] R.E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[52] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–305, 1985.

[53] A. Haken and S. A. Cook. An exponential lower bound for the size of monotone real circuits. Preprint, 1995.

[54] K. Iwama. Complexity of finding short resolution proofs. In I. Privara and P. Ruzicka, editors, *Lecture Notes in Computer Science 1295*, pages 309–318. Springer-Verlag, 1997.

[55] G. D. James. *The Representation Theory of the Symmetric Groups*. Number 682 in Lecture Notes in Mathematics. Springer-Verlag, 1978.

[56] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random formulas. *Science*, 264:1297–1301, May 1994.

[57] L. M. Kirousis, E. Kranakis, and D. Krizanc. Approximating the unsatisfiability threshold of random formulas. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 27–38, Barcelona, Spain, September 1996.

[58] J. Krajíček. On the degree of ideal membership proofs from uniform families of polynomials over finite fields. Manuscript, 1997.

[59] J. Krajicek. Lower bounds to the size of constant-depth propositional proofs. *Journal of Symbolic Logic*, 59(1):73–86, March 1994.

[60] J. Krajíček. *Bounded Arithmetic, Propositional Logic and Complexity Theory*. Cambridge University Press, 1996.

[61] J. Krajicek and P. Pudlák. Propositional proof systems, the consistency of first order theories and the complexity of computations. *J. Symbolic Logic*, 54(3):1063–1079, 1989.

[62] K. Krajicek and P. Pudlák. Some consequences of cryptographic conjectures for S_2^1 and EF. In D. Leivant, editor, *Logic and Computational Complexity*, pages 210–220. Springer-Verlag, 1995.

[63] J. Krajíček, P. Pudlák, and A Woods. Exponential lower bounds to the size of bounded depth Frege proofs of the pigeonhole principle. *Random Structures and Algorithms*, 7(1), 1995.

[64] L. Lovasz and A. Schrijver. Cones of matrices and set-functions and 0-1 optimization. *SIAM J. Optimization*, 1(2):166–190, 1991.

[65] A. Maciel and T. Pitassi. Non-automatizability of bounded-depth Frege proofs. Manuscript. 1998.

[66] A. Maciel and T. Pitassi. On $ACC^0[p^k]$ frege proofs. In *Proceedings of the Twenty Ninth Annual ACM Symposium on Theory of Computing*, pages 720–729, May 1997.

[67] J. Meßner and J. Toran. Optimal proof systems for propositional logic and complete sets. In *15th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, Paris, France, February 1998. Springer-Verlag.

[68] D. Mitchell. Hard problems for csp algorithms. In *Proceedings from AAAI '98*, 1998.

[69] D. Mundici. A lower bound for the complexity of Craig's interpolants in sentential logic. *Archiv fur Mathematische Logik und Grundlagenforschung*, 23(1-2):27–36, 1983.

[70] J. Paris and A. Wilkie. Counting problems in bounded arithmetic. In *Methods in Mathematical Logic: Proceedings of the 6th Latin American Symposium on Mathematical Logic 1983*, volume 1130 of *Lecture notes in Mathematics*, pages 317–340, Berlin, 1985. Springer-Verlag.

[71] J.B. Paris, A. J. Wilkie, and A. R. Woods. Provability of the pigeonhole principle and the existence of infinitely many primes. *Journal of Symbolic Logic*, 53:1235–1244, 1988.

[72] T. Pitassi. Algebraic propositional proof systems. In *DIMACS Series in Discrete Mathematics*, volume 31, pages 215–243. American Math. Soc, 1997.

[73] T. Pitassi and A. Urquhart. The complexity of the Hajós calculus. In *33nd Annual Symposium on Foundations of Computer Science*, pages 187–196, Pittsburgh, PA, October 1992. IEEE.

[74] Toniann Pitassi, Paul W. Beame, and Russell Impagliazzo. Exponential lower bounds for the pigeonhole principle. *Computational Complexity*, 3(2):97–140, 1993.

[75] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, September 1997.

[76] P. Pudlák and J. Sgall. Algebraic models of computation and interpolation for algebraic proof systems. In P. W. Beame and S. R. Buss, editors, *Proof Complexity and Feasible Arithmetics*, DIMACS, pages 279–295. American Math. Soc, 1997.

[77] A. A. Razborov. Lower bounds for the polynomial calculus. November 1996.

[78] A. A. Razborov. Lower bounds for the size of circuits with bounded depth with basis $\{\wedge, \oplus\}$. *Mat. Zametki*, 1987.

[79] A. A. Razborov. Bounded arithmetic and lower bounds in Boolean complexity. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 344–386. Birkhauser, 1995.

[80] A. A. Razborov. Unprovability of lower bounds on the circuit size in certain fragments of bounded arithmetic. *Izvestiiya of the RAN*, 59:201–224, 1995.

[81] A. A. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, August 1997.

[82] A. A. Razborov, A. Wigderson, and A. C. Yao. Read-once branching programs, rectangular proofs of the pigeonhole principle and the transversal calculus. In *Proceedings of the Twenty Ninth Annual ACM Symposium on Theory of Computing*, pages 739–748, El Paso, TX, May 1997.

[83] Søren Riis. *Independence in Bounded Arithmetic*. PhD thesis, Oxford University, 1993.

[84] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
[85] J. T. Schwartz. Probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, pages 701–717, 1980.
[86] M. Sipser. The history and status of the P versus NP question. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, pages 603–618, Victoria, B.C., Canada, May 1992.
[87] Michael Sipser. A complexity theoretic approach to randomness. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, Boston, MA, April 1983.
[88] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 77–82, New York, NY, May 1987.
[89] G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*. 1968.
[90] A. Urquhart. Manuscript, 1998.
[91] A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.
[92] A. Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1(4):425–467, December 1995.

Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350
E-mail address: beame@cs.washington.edu

Computer Science, University of Arizona Tucson, AZ 85721
E-mail address: toni@cs.arizona.edu

The Complexity of Analytic Tableaux

Noriko H. Arai*
National Institute for Informatics
Tokyo
arai@nii.ac.jp

Toniann Pitassi†
Department of Computer Science
University of Toronto
toni@cs.toronto.edu

Alasdair Urquhart‡
Department of Philosophy
University of Toronto
urquhart@cs.toronto.edu

ABSTRACT

The method of analytic tableaux is employed in many introductory texts and has also been used quite extensively as a basis for automated theorem proving. In this paper, we discuss the complexity of the system as a method for refuting contradictory sets of clauses, and resolve several open questions. We discuss the three forms of analytic tableaux: clausal tableaux, generalized clausal tableaux, and binary tableaux. We resolve the relative complexity of these three forms of tableaux proofs and also resolve the relative complexity of analytic tableaux versus resolution. We show that there is a quasi-polynomial simulation of tree resolution by analytic tableaux; this simulation cannot be improved, since we give a matching lower bound that is tight to within a polynomial.

1. INTRODUCTION

The method of analytic tableaux, or truth trees, is employed in many introductory logic texts; it is given a particularly elegant formulation in Smullyan's monograph [9]. It has also been used quite extensively as a basis for automated theorem proving. In this paper, we discuss the complexity of the system as a method of refuting contradictory sets of clauses. One of the results of the analysis is that the complexity depends crucially on the way in which the clauses are represented, and on the exact form of the decomposition rules employed.

An analytic tableaux tree for a set of clauses Σ has a root node with which we associate the clauses in Σ. For each internal node k of the tree, a particular clause associated with k is chosen and decomposed into l disjoint subclauses; the clauses associated with each of the l children of k are all of the clauses labelling k plus one of the l disjoint subclauses. A path in the proof tree is closed if the leaf node has associated with it a literal and its negation (and other clauses as well). An analytic tableaux tree for Σ is a refutation of Σ if all paths are closed.

There are various possible forms of the tableau decomposition rule, leading to different versions of analytic tableaux. The simplest form of the rule, leading to what we call *clausal tableaux* below, is one in which the decomposition of a clause into all of its component literals must be carried out in a single step. A general form of the rule, leading to what we call *generalized clausal tableaux*, allows the decomposition of a clause AB into two disjoint clauses A and B. As we shall see, the more general form of the clausal rule can result in drastically reduced complexity. One can also consider clauses as formulas built up from literals, using disjunction as a binary connective. The tableau rule then takes the form where a formula $A \vee B$ is decomposed into its immediate subformulas A and B. This form of analytic tableaux, the one used in Smullyan's monograph [9], we call *binary tableaux*.

The relative complexity of analytic tableaux versus tree resolution is interesting since both of these systems are used extensively in automated theorem proving. It has been known since the work of Cook and Reckhow [3] that there are sets of clauses that are easy to refute for tree resolution but hard for analytic tableaux. As a result, the question of the relative complexity of these two systems was generally held to be settled for many years. However, the question was reopened by Massacci [7, 6] who presented efficient binary analytic tableaux proofs for Cook and Reckhow's examples, thus pointing out that the lower bounds of [3] only hold for clausal analytic tableaux. Our paper gives a conclusive answer to this question of relative complexity and also shows the startling sensitivity of the analytic tableaux method to the form of the tableaux decomposition rule.

The paper is organized as follows. In §2, we present the lower bound of Cook and Reckhow for clausal tableaux, with improved parameters. In §3, we show that the speed-up observed by Massacci for binary analytic tableaux (for these examples) is highly dependent on the order in which the clauses are parenthesized. In particular, with one order of bracketing, we can construct quasi-polynomial size refutations, while with another order, exponential-size refutations are required.

This exponential separation does not hold if we adopt the generalized clausal form of the decomposition rule. In §4, we show that generalized clausal tableaux can quasi-polynomially simulate tree resolution. This generalizes the observation of Massacci about the speed-up of binary tab-

*Research supported by the Ministry of Education of Japan.
†Research supported by NSF grant CCR-9820831, US-Israel BSF grant 98-00349, and an NSERC grant.
‡Research supported by NSERC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
STOC'01, July 6-8, 2001, Hersonissos, Crete, Greece.
Copyright 2001 ACM 1-58113-349-9/01/0007 ...$5.00.

leaux over clausal tableaux. Finally in §5, we show that the simulation in §4 is essentially tight by giving a family of formulas with linear size tree-resolution proofs, but requiring quasi-polynomial size generalized clausal tableaux proofs.

Combining these results we obtain a nearly complete picture of the relative complexity of the various forms of analytic tableaux and resolution: Tree resolution polynomially simulates generalized clausal analytic tableaux, but the reverse simulation does not hold; generalized clausal analytic tableaux polynomially simulates both binary and clausal analytic tableaux, but the reverse simulations do not hold.

If Σ is a set of clauses and l is a literal, then we write $\Sigma\restriction l$ for the set of clauses that results from Σ by removing all clauses containing l and deleting $\overline{l}$ from any clause in which it occurs. We will use trees in describing both derivations in proof systems and in defining sets of clauses. If T is a binary tree with immediate subtrees U and V, then we write $T = U \oplus V$.

2. CLAUSAL TABLEAUX

In this section, we consider the simple form of the tableau where all formulas are clauses, and the tableau rule requires a clause to be decomposed into its component literals in one step. If Σ is a set of clauses, then a *clausal tableau for* Σ is a tree in which the interior nodes are associated with clauses from Σ; if a node is associated with a given clause, then the children of that node are labelled with the literals in the clause (the root of the tree remains unlabelled). (In the remainder of this section, the word "tableau" should be understood as referring to a a clausal tableau.) A tableau for Σ is a *refutation of* Σ if every branch in the tableau is closed (i.e. contains a literal and its negation). We shall also count a single unlabelled node as a refutation of the set of clauses $\{\Lambda\}$.

We define the *size* of a tableau refutation as the number of interior nodes in the tableau (this measure of complexity, omitting the leaves of the tree, is convenient for inductive proofs). If T is a tree in which every interior node has at most k children, then the number of leaves of T is bounded by $I(T)\cdot(k-1)+1$, where $I(T)$ is the number of interior nodes in T; if every interior node of T has exactly k children, then the number of leaves of T is exactly $I(T)\cdot(k-1)+1$. If Σ is a set of clauses, then $t(\Sigma)$ is defined to be the minimum size of a tableau refutation of Σ. Because of the simple structure of clausal tableau refutations, it is possible to prove exact lower bounds on their complexity. The basic tools are the following lemmas.

LEMMA 2.1. *In a tableau refutation of minimal size, no branch contains repeated literals.*

Proof. If a tableau refutation contains a branch with repeated literals, then it can be pruned as follows. Let T be a subtree of the tableau whose root is associated with a clause containing a literal l, and this literal l labels a node in the tableau on the path from the root of the tableau to T. Replace T with the immediate subtree of T whose root is labelled with l, but replacing the label on this subtree with the label on the root of T. The resulting tableau is still closed, and is smaller than the original. □

LEMMA 2.2. *If Σ is an unsatisfiable set of clauses, then $t(\Sigma)$ satisfies the recursive equation*

$$t(\Sigma) = \min\{t(\Sigma\restriction l_1)+\cdots+t(\Sigma\restriction l_n)+1 : l_1\vee\cdots\vee l_n\in\Sigma\}.$$

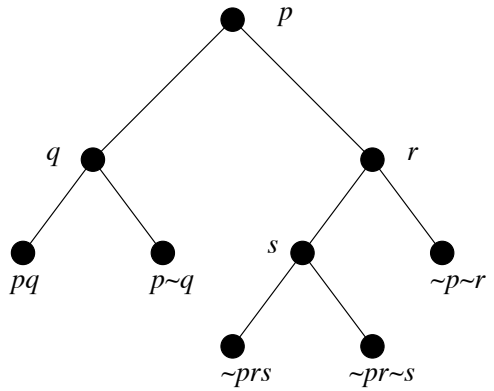


Figure 1: $\Sigma(T) = \{pq, p{\sim}q, {\sim}prs, {\sim}pr{\sim}s, {\sim}p{\sim}r\}$

Proof. For $C = l_1\vee\cdots\vee l_n\in\Sigma$, let T be a tableau refutation of Σ that is minimal among refutations that have C associated with their root. Let $T_1,\ldots,T_n$ be the immediate subtrees of T having $l_1,\ldots,l_n$ as labels on their roots. By Lemma 2.1, the literal l_i does not occur in T_i below the root of T_i; the complement of l_i may occur as the label of at least one leaf in T_i. Thus if we remove from T_i the leaves labelled with $\overline{l_i}$, the result is a refutation of $\Sigma\restriction l_i$. Hence the size of T_i is $t(\Sigma\restriction l_i)$, so that the size of T is

$$t(\Sigma\restriction l_1)+\cdots+t(\Sigma\restriction l_n)+1.$$

Choosing C to minimize this function, we obtain the equation of the lemma. □

Although clausal tableaux work well on simple examples, there are cases where any tableau refutation necessarily contains a great deal of repetition. This is shown by a set of examples due to Cook [1]. Cook's construction associates a set of clauses with a labelled binary tree as follows. Let T be a binary tree in which the interior nodes are labelled with distinct variables. We associate a set of clauses $\Sigma(T)$ with T, in such a way that each branch b in $\Sigma(T)$ has a clause $C_b\in\Sigma(T)$ associated with it. The variables in C_b are those labeling the nodes in b; if P is such a variable, then P is included in C_b if b branches to the left below the node labelled with P, otherwise C_b contains ${\sim}P$. Figure 2 shows a simple example.

Cook's clauses are the sets of clauses $\Sigma_n = \Sigma(T_n)$ associated with the complete binary tree T_n of depth n. To include the case where $n=0$, we take T_0 to consist of a single node, counted as an interior node; the set of clauses $\Sigma(T_0)$ is $\{\Lambda\}$.

If one of the variables in $\Sigma(T)$ is set to 0 or 1, then the resulting simplified set of clauses is also of the form $\Sigma(T')$ for some binary tree T'. Let l be a literal in $\Sigma(T)$, and P the variable in l. Define $T\restriction l$ to be the tree resulting from T by replacing the subtree whose root is labelled with P by either its immediate left or right subtree, depending on whether l is negated or not. Then it is easy to see that $\Sigma(T)\restriction l = \Sigma(T\restriction l)$.

The next lemma allows us to compute $t(T) = t(\Sigma(T))$ directly from the structure of T. It shows that for the set of clauses $\Sigma(T)$, there is an optimal refutation that is essentially unique.

LEMMA 2.3. *The function $t(T)$ satisfies the following recursion equations:*

1. *If T has only one node, then $t(T) = 1$;*

2. *If $T = U \oplus V$, then $t(T) = t(U)t(V) + \min\{t(U), t(V)\}$.*

Proof. If T has only one node, then $\Sigma(T) = \{\Lambda\}$, so $t(T) = 1$ (recall that by our convention, the unique node in a one-node tree counts as an interior node).

Assume the recursion equations hold for trees of size less than that of T, and let $T = U \oplus V$. Let $C = l_1 \vee \cdots \vee l_k$ be a clause in $\Sigma(T)$ that is associated with a branch ending in a leaf in U (the argument for branches in V is symmetrical). Define U_j for $2 \leq j \leq k$ to be the labelled tree $U \restriction l_j$. Let $t_C(T)$ be the size of a minimal tableau in which C is associated with the root. Then by Lemma 2.2,

$$\begin{aligned} t_C(T) &= t(T \restriction l_1) + \cdots + t(T \restriction l_k) + 1 \\ &= t((U \restriction l_1) \oplus V) + \cdots + t((U \restriction l_k) \oplus V) + 1 \\ &= t(V) + \sum_{j=2}^{k} [t(V)t(U_j) + \min\{t(V), t(U_j)\}] + 1 \\ &\quad \text{(by the induction hypothesis)} \\ &= t(V)[1 + \sum_{j=2}^{k} t(U_j)] + \sum_{j=2}^{k} \min\{t(V), t(U_j)\} + 1. \end{aligned} \tag{1}$$

By Lemma 2.2 again,

$$1 + \sum_{j=2}^{k} t(U_j) \geq t(U), \tag{2}$$

so by (1),

$$\begin{aligned} t_C(T) &\geq t(V)[1 + \sum_{j=2}^{k} t(U_j)] + \min\{t(V), [1 + \sum_{j=2}^{k} t(U_j)]\} \\ &\geq t(V)t(U) + \min\{t(V), t(U)\}. \end{aligned} \tag{3}$$

For the opposite inequality, assume that $t(U) \leq t(V)$ and that P is the variable labeling the root of T. Let l_1 be P or $\sim P$ according to whether U is the left or right subtree of T, let $l_2 \vee \cdots \vee l_k$ be the clause associated with the root of a minimal tableau refutation of $t(U)$, and C be the clause $l_1 \vee \cdots \vee l_k$. Then for every j, $t(U_j) \leq t(V)$, so that by (1),

$$\begin{aligned} t_C(T) &= t(V)[1 + \sum_{j=2}^{k} t(U_j)] + \sum_{j=2}^{k} t(U_j) + 1 \\ &= t(V)t(U) + t(U), \end{aligned} \tag{4}$$

completing the proof. □

THEOREM 2.4. *For $n > 0$,*

1. *The clauses Σ_n satisfy the recursion equations: $t(\Sigma_0) = 1$, $t(\Sigma_{n+1}) = t(\Sigma_n)[t(\Sigma_n) + 1]$;*

2. *$t(\Sigma_n) = \lfloor 2^{c2^n} \rfloor$, where $0.67618 < c < 0.67819$.*

Proof. The left and right subtrees of the complete binary tree T_n are isomorphic, so the first claim follows immediately from Lemma 2.3.

Let $z_n = t(\Sigma_n)$; we can estimate the size of z_n by using Aho and Sloane's analysis of doubly exponential sequences [4, §2.2.3]. Taking logarithms to the base 2, we have by the first part of the lemma,

$$\log z_{n+1} = 2 \log z_n + \log(1 + 1/z_n),$$

hence unrolling the recursion,

$$\log z_n = 2^n \sum_{k=0}^{n-1} \frac{\alpha_k}{2^{k+1}},$$

where $\alpha_k = \log(1 + 1/z_k)$. Defining

$$c_n = \sum_{k=0}^{n-1} \frac{\alpha_k}{2^{k+1}}, \quad c = \sum_{k=0}^{\infty} \frac{\alpha_k}{2^{k+1}},$$

we set $r_n = 2^n(c - c_n)$. We have

$$r_n = \sum_{k=0}^{\infty} \frac{\alpha_{n+k}}{2^{k+1}} < \alpha_n,$$

so $2^{r_n} < 2^{\alpha_n} = 1 + 1/z_n$. Hence, $2^{c2^n} - z_n = z_n(2^{r_n} - 1) < 1$, showing that $z_n = \lfloor 2^{c2^n} \rfloor$. The series for c converges rapidly; we find $c = 0.67618634966\ldots$ from $n = 5$. □

An exponential lower bound for the sets of clauses Σ_n was first stated (without proof) in [2]; the proof above is based on joint work of Cook and Urquhart, and is an improved version of the proof published in Urquhart [10]. The first published proof of the lower bound for clausal tableaux was given by Murray and Rosenthal [8], who gave a lower bound of $2^{2^{n-1}}$ for the size of clausal tableau refutations of Σ_n.

Theorem 2.4 has some significance for automated theorem proving based on clausal tableau methods. The set Σ_6 contains only 64 clauses of length 6, but the minimal tableau refutation for Σ_6 has 10,650,056,950,806 interior nodes. This shows that a practical implementation of the clausal tableau method should incorporate routines to eliminate repetition in tableau construction. This applies in particular to the model-elimination method [5], which can be considered as a version of clausal tableaux.

3. BINARY TABLEAUX

In recent work [7, 6] Massacci has shown that the sets of clauses Σ_n discussed in the previous section in fact have quasi-polynomial size analytic tableau refutations, provided we use binary tableaux. In this formulation, formulas are built up from binary disjunctions, and in an application of the tableau rule, formulas are decomposed into their immediate subformulas, rather being decomposed into their component literals, as is required in the case of clausal tableaux. Massacci's result revealed an unsuspected gap between these two formulations of the tableau system, and reopened the question of the relative complexity of the tableau system and resolution (in the form where resolution refutations are represented as trees).

A careful examination of Massacci's proofs reveals that they are strongly dependent on the way in which the formulas are parenthesized. In the case of the sets of clauses Σ_n, it is obvious that some variables are more significant than others. The variable labelling the root of the tree T_n occurs in all of the clauses in Σ_n, whereas a variable labelling a node immediately above a leaf occurs in only two clauses. Hence, assigning a value to the top variable results in a drastic reduction in the complexity of the problem since the set

of clauses is cut in half. However, setting a bottom variable results in only a tiny amount of progress. It is precisely for this reason that these examples are hard for clausal tableaux, since the decomposition rule forces us to assign values to variables low down in the tree T_n.

However, Massacci parenthesizes the formulas in such a way that all the clauses are given in the form $l_1 \vee (l_2 \vee \ldots l_n)\ldots)$, where the sequence of variables follows the sequence of nodes from the root of T_n to its leaves. This means that in a binary tableau refutation of Σ_n, we can give preference to the more significant variables. In this way, Massacci is able to construct binary tableau refutations of Σ_n of size bounded by $2^{O(n^2)}$, that is to say, a refutation of size quasi-polynomial in the size of the set of input clauses.

The observations above on the relative significance of variables suggest that we could prove a lower bound for binary tableaux similar to the lower bound of the preceding section if we parenthesized the set Σ_n in the opposite way, that is to say, if we represented the clauses in the form $(\ldots(l_1 \vee l_2) \vee \ldots l_{n-1}) \vee l_n$ This order of variables forces us to deal with the least significant variables first, and so should require a tableaux of a size comparable to that necessary in the case of clausal tableaux. In this section, we verify this conjecture.

By a *formula*, we mean a disjunction built up from literals using a binary disjunction connective. If Π is a set of formulas, then a *binary tableau for* Π is a tree in which the interior nodes are associated with formulas, in accordance with the following rules. If a formula A is associated with an internal node d, then it is either an initial formula belonging to Π, or occurs as a label on the branch determined by d; we say that A is the formula *decomposed at the node* d. If a node is associated with a given formula of the form $A \vee B$, then the children of that node are labeled with A and B respectively; we define the node labelled with A to be the *elder child*, and the node labelled with B to be the *younger child*. If an interior node d has a literal l associated with it, then the unique child of that node is labelled with l and defined to be the younger child of d. As in the case of clausal tableaux, the root of the tree is unlabelled. (In the remainder of this section, the phrase word "tableau" should be understood as referring to a binary tableau.) A tableau for Π is a *refutation of* Π if every branch in the tableau is closed (i.e. contains a literal and its negation).

Because of the more intricate nature of the binary decomposition rule, it is a little harder to analyse the complexity of binary tableaux than that of clausal tableaux. To make the recursion step in the lower bound proof easier, we define a complexity measure for binary tableaux that counts some, but not all, of the interior nodes. Let us define a node in a binary tableaux to be a *significant node* if it is either the root or a younger child of a node in the tableau. We define the *size* of a binary tableau refutation as the number of significant interior nodes in the tableau (as in the previous section, we define the unique node of a one-node tree to be an interior node). If Π is a set of clauses, then $u(\Pi)$ is defined to be the minimum size of a tableau refutation of Π.

Before proceeding to the lower bound, we require two lemmas on binary tableaux that are analogues of Lemmas 2.1 and 2.2. If A and B are formulas, then we say that A *subsumes* B if A is a subformula of B; for example, the formula $(p \vee \sim q)$ subsumes $(r \vee s) \vee (p \vee \sim q)$.

LEMMA 3.1. *If $\mathcal{T}$ is a binary tableau refutation of a set of formulas $\Pi \cup \{A, B\}$, where A subsumes B, then there is a tableau refutation $\mathcal{T}'$ of $\Pi \cup \{A\}$, where the size of $\mathcal{T}'$ is no bigger than the size of $\mathcal{T}$.*

Proof. Let Π be any contradictory set of formulas having a formula A as a member, and let C be a formula subsumed by A. We argue by induction on the complexity of C that if $\mathcal{T}$ is any tableau refutation of Π, and C occurs as a label on a node in $\mathcal{T}$, then we can shorten $\mathcal{T}$ by removing all nodes labelled with C.

Suppose that C is A itself, and that A labels a node e with a parent node d. Remove the subtree having the sibling of e as the root (if any), and then merge the nodes d and e. Remove the label A, retaining the original label on d (if there was one) as the label on the merged node. The result is still a closed tableau; the label A was never needed to justify a decomposition in $\mathcal{T}$, since it was a member of Π.

Assume that the Lemma holds for formulas subsumed by A, up to a given complexity, let B be subsumed by A, and suppose that $B \vee C$ labels a node e in the refutation with parent node d. By inductive assumption, there is no node in $\mathcal{T}$ labelled with B, from which it follows that in the subtree with e as root, $B \vee C$ is never decomposed. Remove the subtree having the sibling of e as the root (if any), and then merge the nodes d and e. Remove the label $B \vee C$, retaining the original label on d (if there was one) as the label on the merged node. The result is still a closed tableau, since the label $B \vee C$ is never needed to justify a decomposition in the subtree with e as root. □

We now define the sequence $\{\Pi_n\}$ of sets of formulas used in proving the lower bound for binary tableaux. It is essentially the same as the sequence $\{\Sigma_n\}$ used in the previous section, except that we parenthesize the clauses in a particular way. More generally, we associate a set of formulas $\Pi(T)$ with any binary tree T whose internal nodes are labelled with distinct variables. Let $l_1 l_2 \ldots l_k$ be a clause in $\Sigma(T)$, where the order of literals follows the order of the edges in the corresponding branch in T from the root to the leaf. Then the corresponding formula in $\Pi(T)$ is $(\ldots(l_1 \vee l_2)\ldots) \vee l_k$. We define Π_n to be the set of formulas $\Pi(T_n)$, where T_n is the complete binary tree of depth n. Since the size of a minimal tableau refutation of $\Pi(T)$ depends only on the structure of T, we shall write $u(T)$ as an abbreviation for $u(\Pi(T))$.

To prove a lower bound for Π_n, we first require a lemma that corresponds to Lemma 2.2. If Π is a set of formulas, and C is a disjunction that is a subformula of a formula in Π, then we define $\Pi[C]$ to be the set of formulas resulting from Π by removing all formulas subsumed by C, and replacing them by C. If Π is a contradictory set of formulas, and $C \in \Pi$, then we define $u(\Pi, C)$ to be the minimal size of a binary tableau refutation of Π, in which the formula decomposed at the root of the tableau is C. Obviously, we have $u(\Pi) = \min\{u(\Pi, C) : C \in \Pi\}$.

If T is a labelled binary tree, and e a leaf in T, then we write $u(T, e)$ for $u(\Pi(T), E)$, where E is the formula in $\Pi(T)$ associated with the branch in T ending in e. For a labelled tree T, we have $u(T) = \min\{u(T, e) :\ e \text{ a leaf in } T\}$.

LEMMA 3.2. *If Π is a contradictory set of formulas, and $A \vee l$ is a formula in Π, where l is a literal, then $u(\Pi, A \vee l) =$*

$u(\Pi[A]) + u(\Pi\restriction l)$, *if* A *is not a literal, while* $u(\Pi, A \vee l) = u(\Pi\restriction A) + u(\Pi\restriction l)$ *if* A *is a single literal.*

Proof. Suppose we are given a tableau refutation of Π of minimal size among tableaux in which $A \vee l$ is decomposed at the root. The size of the sub-tableau with l labelling its root is $u(\Pi\restriction l)$; this follows by the same argument as that used in Lemma 2.2.

If A is not a single literal, then the lefthand sub-tableau with A labelling its root is a refutation of $\Pi \cup \{A\}$, after the deletion of the label on its root (since the label on the root cannot be used to close a branch). By Lemma 3.1, a minimal tableau refutation of $\Pi \cup \{A\}$ is a refutation of $\Pi[A]$. The equation of the lemma follows because the node labelled with A does not contribute to the size of the refutation. On the other hand, if A is a single literal, then the size of the tableau is $u(\Pi\restriction A) + u(\Pi\restriction l)$, again by the argument of Lemma 2.2. □

As in the case of clausal tableaux, we can gain insight into the structure of tableau refutations of sets of formulas $\Pi(T)$ associated with labelled binary trees by describing logical operations on such sets of formulas in terms of operations on the underlying trees. Let T be a binary tree in which the interior nodes are labelled with distinct variables, and let e be a leaf in T. Let f be the sibling of e, and d their common parent. Then T_e^1 is the labelled tree obtained from T by removing the nodes e and f, and deleting the label on d, while T_e^2 is the tree obtained from T by removing e, removing the label on f, and then merging the nodes d and f (the label on the merged node is the same as the original label on d). Let $A \vee l$ be the formula associated with the branch in T that ends in the leaf e. Then it is easy to check that $\Pi(T)[A] = \Pi(T_e^1)$ and that $\Pi(T)\restriction l = \Pi(T_e^2)$.

LEMMA 3.3. *The function* $u(T)$ *satisfies the following recursion equations:*

1. *If* T *has only one node, then* $u(T) = 1$*;*
2. *If* $T = V \oplus W$*, and one of them, say* V*, is a one-element tree, then* $u(T) = u(W) + 1$*;*
3. *If* $T = V \oplus W$*, and neither* V *nor* W *is a one-element tree, then* $u(T) = u(V)u(W)$*.*

Proof. If T has only one node, then $\Pi(T) = \{\Lambda\}$, so $u(T) = 1$, since by our conventions, the unique node in a one-element tree counts as a significant node.

In the second case, let p be the variable labelling the root of T, and suppose that U is the lefthand subtree of T. In any refutation of $\Pi(T)$, there must be at least one node in the refutation labelled with p, from which it follows that $u(T) > u(W)$. Since there is clearly a refutation of size $u(W) + 1$, this must be a refutation of minimal size.

In the third case, let $\mathcal{T}$ be a refutation of $\Pi(T)$ in which the formula decomposed at the root of the tableau is associated with the branch ending in a leaf e. We assume that e belongs to the subtree V (the case for W is symmetrical). By Lemma 3.2, and the induction hypothesis, we have

$$\begin{aligned} u(T,e) &= u(T_e^1) + u(T_e^2) \\ &= u(V_e^1 \oplus W) + u(V_e^2 \oplus W) \\ &= u(V_e^1)u(W) + u(V_e^2)u(W) \\ &= (u(V_e^1) + u(V_e^2))u(W) \\ &= u(V,e)u(W). \end{aligned}$$

(We have assumed aboved that neither V_e^1 nor V_e^2 are one-element trees, but the reader can verify that the equation $u(T,e) = u(V,e)u(W)$ also holds when one or both are one-element trees.) We can minimize $u(V,e)u(W)$ by minimizing $u(V,e)$ over all leaves in T. Hence, $u(T) = u(V)u(W)$. □

THEOREM 3.4. *For* $n > 0$*,*

1. *The clauses* Π_n *satisfy the recursion equations* $u(\Pi_1) = 2$, $u(\Pi_{n+1}) = u(\Pi_n)^2$*;*
2. *Any binary tableau refutation of* Π_n *must contain at least* $2^{2^{n-1}}$ *nodes.*

Proof. The theorem is an immediate consequence of Lemma 3.3. □

COROLLARY 3.5. *Binary tableaux cannot p-simulate tree resolution.*

Proof. The sets of clauses Σ_n have linear-size tree resolution refutations; in fact, we can label the nodes of the labelled tree T with clauses so that the result is a tree refutation of the corresponding set of clauses $\Pi(T)$. □

4. GENERALIZED CLAUSAL TABLEAUX

The complexity gap between clausal and binary tableaux shows that it makes a very large difference in the complexity of proofs if we can give preference to certain variables; with the right choice of variables, we can produce an exponential speed-up over the simple clausal rule. This suggests a still more general form of tableau rule. In this formulation, we treat disjunctions of literals as clauses (that is, sets of literals), but instead of insisting on decomposing a clause in one step, we allow arbitrary decompositions of clauses into two subclauses. This form of the rule is clearly more powerful and flexible than either the simple clausal rule, or the binary rule. We shall use the phrase "generalized clausal tableaux" to describe tableaux constructed according to this rule.

If Σ is a set of clauses, then a *generalized tableau for* Σ is a tree in which an interior node d has associated with it a clause that is either a clause from Σ or occurs as a label on the branch ending in d. If a node is associated with a given clause C, then the children of the node are labelled with clauses C_1 and C_2, where C_1 and C_2 form a partition of C (the root of the tree remains unlabelled). If C consists of a single literal, then the unique child of the node associated with C is labelled with C itself. A tableau for Σ is a *refutation of* Σ if every branch in the tableau is closed (i.e. contains a literal and its negation).

We define a complexity measure for generalized clausal tableaux that is essentially the same as the measure used for clausal tableaux. That is to say, we define the complexity of a generalized clausal tableau to be the number of interior nodes in the tableau. If Σ is a set of clauses, then $g(\Sigma)$ is defined to be the minimum size of a generalized tableau refutation of Σ.

A *tree resolution refutation* of a contradictory set of clauses Σ is a binary tree in which the leaves are labelled with clauses from Σ, the root is labelled with the empty clause Λ, and an interior node in the tree is labelled with a resolvent of the two clauses labelling its children. We define the complexity of a tree resolution refutation to be the number of

leaves in the underlying tree. The *tree resolution complexity* of Σ, $r(\Sigma)$, is the minimal complexity of a tree resolution refutation of Σ.

The next theorem shows that the quasi-polynomial simulation of tree resolution by binary resolution shown by Massacci in the case of the examples of Cook and Reckhow can be generalized to a uniform quasi-polynomial simulation of tree resolution by general clausal tableaux. In the following theorem, all logarithms are to the base 2.

THEOREM 4.1. *If Σ is a contradictory set of clauses, then*

$$g(\Sigma) \leq r(\Sigma) \cdot |\Sigma|^{\log r(\Sigma)} = 2^{(\log r(\Sigma))(\log |\Sigma|+1)}.$$

Proof. The upper bound is proved by the recursive construction of a tableau refutation of Σ, starting from a tree resolution refutation of Σ. The proof is by induction on the number of variables in Σ. If Σ contains no variables, then $\Sigma = \{\Lambda\}$, $r(\Sigma) = g(\Sigma) = 1$, and the inequality is satisfied.

Assume that the upper bound holds for all sets of clauses with at most n variables, and let Σ contain $n+1$ variables. Let T be a minimal-size tree resolution refutation of Σ, with immediate subtrees U and V. Then there is a literal l so that U is a resolution proof of $\overline{l}$ of complexity $r(\Sigma\restriction l)$, while V is a resolution proof of l of complexity $r(\Sigma\restriction \overline{l})$; for a proof of this fact, see [10, §5]. We assume that the complexity of U is less than or equal to the complexity of V, so that $\log r(\Sigma\restriction l) \leq \log r(\Sigma) - 1$.

Two cases arise in the inductive step, the first where U is a one-element tree, and the second where it has more than one node. In the first case, $\Sigma\restriction l$ consists of the single literal $\overline{l}$. In this case, we construct the tableau refutation by starting from a minimal-size tableau refutation of $\Sigma\restriction \overline{l}$, adding $\overline{l}$ as a label on the root, then adding a new unlabelled root, together with appropriate leaves labelled with the literal l. The result is a refutation of Σ, and it has complexity

$$\begin{aligned} g(\Sigma\restriction \overline{l}) + 1 &\leq r(\Sigma\restriction \overline{l}) \cdot |\Sigma\restriction \overline{l}|^{\log r(\Sigma\restriction \overline{l})} + 1 \\ &\leq (r(\Sigma\restriction \overline{l}) + 1) \cdot |\Sigma|^{\log r(\Sigma)} \\ &= r(\Sigma) \cdot |\Sigma|^{\log r(\Sigma)} \end{aligned}$$

We now consider the second case, where the lefthand subtree U has at least two leaves. Let Σ_l be the set of all clauses in Σ containing the literal l; let $D_1 l, \ldots, D_k l$ be a list of all clauses in Σ_l. Since Σ is contradictory, $k < |\Sigma|$. Construct a generalized analytic tableau by decomposing $D_1 l$ at the root into D_1 and l; continue the construction by decomposing $D_2 l$ in the same way at the node labelled with D_1, $D_3 l$ at the node labelled D_2, and so on. Complete the construction by appending a tableau refutation of size $g(\Sigma\lfloor l)$ to all of the nodes labelled l, and a tableau of size $g(\Sigma\restriction \overline{l})$ to the node labelled D_k. (We can construct appropriate refutations of this size by adding some leaves to tableau refutations of $\Sigma\restriction l$ and $\Sigma\restriction \overline{l}$ respectively.) See Figure 2.

We estimate $g(\Sigma)$ by using the induction hypothesis, and

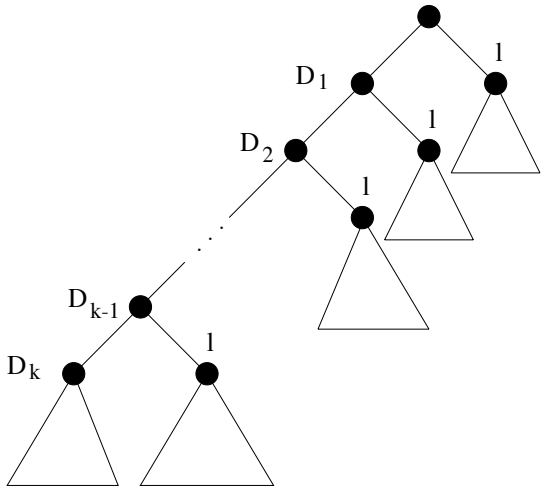


Figure 2: Inductive step for the second case

the inequalities $1 \leq \log r(\Sigma\restriction l) \leq \log r(\Sigma) - 1$:

$$\begin{aligned} g(\Sigma) &\leq (|\Sigma| - 1)(g(\Sigma\restriction l) + 1) + g(\Sigma\restriction \overline{l}) \\ &\leq (|\Sigma| - 1)(r(\Sigma\restriction l) \cdot |\Sigma\restriction l|^{\log r(\Sigma\restriction l)} + 1) \\ &\quad + r(\Sigma\restriction \overline{l}) \cdot |\Sigma\restriction \overline{l}|^{\log r(\Sigma\restriction \overline{l})} \\ &\leq |\Sigma| \cdot r(\Sigma\restriction l) \cdot |\Sigma|^{\log r(\Sigma)-1} + r(\Sigma\restriction \overline{l}) \cdot |\Sigma|^{\log r(\Sigma)} \\ &\leq r(\Sigma\restriction l) \cdot |\Sigma|^{\log r(\Sigma)} + r(\Sigma\restriction \overline{l}) \cdot |\Sigma|^{\log r(\Sigma)} \\ &= (r(\Sigma\restriction l) + r(\Sigma\restriction \overline{l}))|\Sigma|^{\log r(\Sigma)} \\ &= r(\Sigma) \cdot |\Sigma|^{\log r(\Sigma)}, \end{aligned}$$

completing the induction. □

COROLLARY 4.2. *Generalized clausal tableaux quasi polynomially simulates tree resolution.*

5. LOWER BOUND FOR UNRESTRICTED TABLEAUX

The upper bound of the preceding section leaves open the possibility that it might be improved to a polynomial simulation of tree resolution by generalized clausal tableaux. In this section, we show that no such improvement is possible by giving a lower bound for such tableaux that is polynomially related to the upper bound provided by Theorem 4.1.

The generalized tableaux of the preceding section were based on binary branching. The lower bound proved below holds for an even more general formulation of analytic tableaux, where we allow arbitrary decompositions of clauses, not just binary decomposition.

If Σ is a set of clauses, then an *unrestricted tableau for* Σ is a tree in which an interior node d has associated with it a clause that is either a clause from Σ or occurs as a label on the branch ending in d. If a node is associated with a given clause C, then the children of the node are labelled with clauses $C_1, \ldots, C_k$, where $C_1, \ldots, C_k$ forms a partition of C (the root of the tree remains unlabelled). If C consists of a single literal, then the unique child of the node associated with C is labelled with C itself. A tableau for Σ is a *refutation of* Σ if every branch in the tableau is closed (i.e. contains a literal and its negation).

The lower bound employs the sets of clauses Σ_n based on the complete binary tree T_n already used in §2. If T is a labelled binary tree, and $\Sigma(T)$ the corresponding set of clauses, constructed as in §2, then we write $h(T)$ as an abbreviation for $h(\Sigma(T))$, where $h(\Sigma)$ is the unrestricted tableau complexity of Σ.

If Σ is a set of clauses, and $D \subseteq C \in \Sigma$, then we write $\Sigma[D]$ for the set of clauses resulting from Σ by replacing all clauses in Σ subsumed by D by D.

LEMMA 5.1. *If $l \in D$, where $D \subseteq C \in \Sigma$, then $h(\Sigma \restriction l) \leq h(\Sigma[D])$.*

Proof. This follows from the fact that

$$h(\Sigma \restriction l) = h(\Sigma[D] \restriction l) \leq h(\Sigma[D]). \quad \square$$

A key feature of the sets of clauses Σ_n is that (as we already observed in §4), variables occurring higher in the tree T_n are more significant than those occurring lower down. We can make this observation precise as follows. Let l_1 and l_2 be literals occurring in the set of clauses $\Sigma(T)$, where T is a binary tree, and p_1 and p_2 the variables occurring in l_1 and l_2. We say that l_1 *dominates* l_2 if there is a branch in T so that both p_1 and p_2 label nodes in the branch, and the node labelled by p_1 is higher in the branch than the node labelled by p_2.

LEMMA 5.2. *If l_1 and l_2 are literals in the set of clauses $\Sigma(T)$, and l_1 dominates l_2, then $h(T \restriction l_1) \leq h(T \restriction l_2)$.*

Proof. If l_1 dominates l_2, then $\Sigma(T) \restriction l_1$ does not contain any clause having l_2 or $\overline{l_2}$ as a literal. Hence, $h(T \restriction l_1) = h((T \restriction l_2) \restriction l_1) \leq h(T \restriction l_2)$. $\square$

Define $h(m, N)$ to be the minimum of $h(T)$, where T is a binary tree of depth at most m, and containing at least $2^m - N$ internal nodes.

LEMMA 5.3. *If $N_1 \leq N_2$, then $h(m, N_1) \geq h(m, N_2)$.*

Proof. This follows from the fact that if T' results from T by deleting an interior node, then $h(T) \geq h(T')$. $\square$

LEMMA 5.4. *If $Q < 2^{p-1}$, then $h(p, Q-1) \geq h(p, Q) + h(p-1, Q-1)$.*

Proof. Let T be a binary tree of depth at most p, containing at least $2^p - Q - 1$ internal nodes, and such that $h(T) = h(p, Q-1)$. Furthermore, let us suppose that $T = U \oplus V$. Since $Q < 2^{p-1}$, it follows that neither U nor V is a one element tree.

Let C be the clause decomposed at the root of a minimal-size refutation of $\Sigma(T)$, and let $C_1, \ldots, C_k$ be the subclauses of C labelling the nodes in the refutation lying immediately below the root of the tableau. The clause C is a clause C_b associated with a branch b in T. Let l_1 be the highest literal occurring in C, l_2 the lowest literal in C (that is, to say, the variable in l_1 labels the root of T, while the variable in l_2 labels a node in b that has the leaf of b as a child).

One of the clauses C_i must contain the lowest literal l_2. Choose another clause C_j, where $i \neq j$, and let l_3 be the highest variable occurring in C_j (l_3 may or may not be l_1). Suppose that $T = U \oplus V$, and that the branch b belongs to the lefthand subtree U. Then $\Sigma(T) \restriction l_1 = \Sigma(T \restriction l_1) = \Sigma(V)$, and $\Sigma(T) \restriction l_2 = \Sigma(T')$, where T' has one fewer internal node than T. Lemma 5.1 and Lemma 5.2 imply that $h(T \restriction l_2) \leq h(\Sigma(T)[C_i])$, and $h(T \restriction l_1) \leq h(\Sigma(T)[C_j])$. Hence, $h(T) \geq h(T \restriction l_2) + h(T \restriction l_1)$. Since U contains at most $2^{p-1}-1$ internal nodes, it follows that V contains at least $2^{p-1} - Q$ internal nodes. It follows that

$$h(p, Q-1) \geq h(p, Q) + h(p-1, Q-1),$$

completing the proof of the lemma. $\square$

LEMMA 5.5. *If $rN + 1 < 2^{m-1}$, then for all j, where $0 \leq j \leq r$,*

$$h(m+j, (r-j)N + 1) \geq N^j.$$

Proof. The lemma is proved by induction on j, from $j = 0$ to $j = r$. The induction step is proved by introducing an auxiliary variable k, and proving by downward induction on k, from $k = N$ to $k = 0$ that

$$h(m+j, (r-j)N + k + 1) \geq (N-k)N^{j-1}.$$

Assume the above inequality holds up to $j-1$, and for parameter j, for $k = k+1$ to N. We now want to prove the inequality for j and k. By Lemmas 5.3 and 5.4, and the induction hypothesis,

$$\begin{aligned}
& h(m+j, (r-j)N + k + 1) \\
\geq\ & h(m+j, (r-j)N + k + 2) \\
& + h(m+j-1, (r-j)N + k + 1) \\
\geq\ & h(m+j, (r-j)N + (k+1) + 1 \\
& + h(m+j-1, (r-j+1)N + 1) \\
\geq\ & (N - (k+1)) \cdot N^{j-1} + N^{j-1} \\
=\ & (N-k) \cdot N^{j-1},
\end{aligned}$$

completing the induction step. $\square$

THEOREM 5.6. *The unrestricted tableau complexity of Σ_n is $2^{\Theta(n^2)}$.*

Proof. In Lemma 5.5, set $m = \lceil 2n/3 \rceil$, $r = \lfloor n/3 \rfloor$, $N = 2^{\lfloor n/3 \rfloor}$, and $j = r$. Then for n sufficiently large, $rN + 1 < 2^{m-1}$, showing that $h(\Sigma_n) = h(n, 1) \geq N^r = 2^{\Omega(n^2)}$. Theorem 4.1 produces an upper bound of $2^{O(n^2)}$, so the theorem follows. $\square$

COROLLARY 5.7. *Unrestricted clausal tableaux cannot p-simulate tree resolution.*

6. REFERENCES

[1] Stephen A. Cook. An exponential example for analytic tableaux. Manuscript, 1973.

[2] Stephen A. Cook and Robert A. Reckhow. On the lengths of proofs in the propositional calculus. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, 1974. See also corrections for above in *SIGACT News*, Vol. 6 (1974), pp. 15-22.

[3] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44:36–50, 1979.

[4] Daniel H. Greene and Donald E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhäuser, 1990. Third edition.

[5] D.W. Loveland. Mechanical theorem proving by model elimination. *Journal of the ACM*, 15:236–251, 1968.

[6] Fabio Massacci. The complexity of analytic and clausal tableaux. Forthcoming, *Theoretical Computer Science*.

[7] Fabio Massacci. Cook and Reckhow are wrong: Subexponential tableau proofs for their family of formulae. In Henri Pradé, editor, *13th European Conference on Artificial Intelligence*, pages 408–409. Morgan Kaufmann, 1998.

[8] Neil V. Murray and Erik Rosenthal. On the computational intractability of analytic tableau methods. *Bulletin of the IGPL*, Volume 2, Number 2:205–228, September 1994.

[9] Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag, New York, 1968. Reprinted by Dover, New York, 1995.

[10] Alasdair Urquhart. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 1:425–467, 1995.

The Completeness of Propositional Resolution A Simple and Constructive Proof

Jean Gallier

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
jean@saul.cis.upenn.edu

September 25, 2006

Abstract. It is well known that the resolution method (for propositional logic) is complete. However, completeness proofs found in the literature use an argument by contradiction showing that if a set of clauses is unsatisfiable, then it must have a resolution refutation. As a consequence, none of these proofs actually gives an algorithm for producing a resolution refutation from an unsatisfiable set of clauses. In this note, we give a simple and constructive proof of the completeness of propositional resolution which consists of an algorithm together with a proof of its correctness.

1 Introduction

The resolution method for (propositional) logic due to J.A. Robinson [4] (1965) is well-known to be a sound and complete procedure for checking the unsatisfiability of a set of clauses. However, it appears that the completeness proofs that can be found in the literature (for instance, Chang and Lee [1], Lewis and Papadimitriou [3], Robinson [5]) are existence proofs that proceed by contradiction to show that if a set of clauses is unsatisfiable, then it must have a resolution refutation because otherwise a satisfying assignment can be obtained. In particular, none of these proofs yields (directly) an algorithm producing a resolution refutation from an unsatisfiable set of clauses. In that sense, these proofs are nonconstructive. In Gallier [2] (1986), we gave a completeness proof based on an algorithm for converting a Gentzen-like proof (using sequents) into a resolution DAG (see Chapter 4). Such a method is more constructive than the others but, we found later on that it is possible to give a simple and constructive proof of the completeness of propositional resolution which consists of an algorithm together with a proof of its correctness. This algorithm and its correctness are the object of this note.

It should be noted that Judith Underwood gave other constructive proof procedures in her Ph.D. thesis, notably for the intuitionistic propositional calculus [6].

2 Review of Propositional Resolution

Recall that a *literal*, L, is either a propositional letter, P, or the negation, $\neg P$, of a propositional letter. A *clause* is a finite set of literals, $\{L_1, \ldots, L_k\}$, interpreted as the disjunction $L_1 \vee \cdots \vee L_k$ (when $k = 0$, this is the empty clause denoted $\square$). A set of clauses, $\Gamma = \{C_1, \ldots, C_n\}$, is interpreted as the conjunction $C_1 \wedge \cdots \wedge C_n$. For short, we write $\Gamma = C_1, \ldots, C_n$.

The *resolution method* (J.A. Robinson [4]) is a procedure for checking whether a set of clauses, Γ, is unsatisfiable. The resolution method consists in building a certain kind of labeled DAG whose leaves are labeled with clauses in Γ and whose interior nodes are labeled according to the *resolution rule*. Given two clauses $C = A \cup \{P\}$ and $C' = B \cup \{\neg P\}$ (where P is a propositional letter, $P \notin A$ and $\neg P \notin B$), the *resolvent of C and C'* is the clause

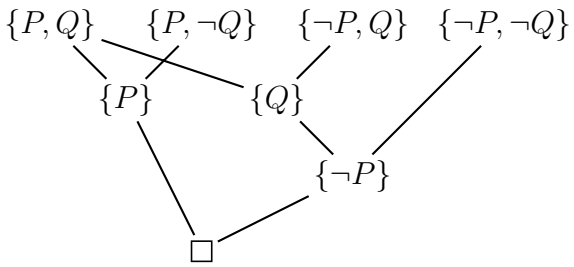
$$R = A \cup B$$

obtained by cancelling out P and $\neg P$. A *resolution DAG for* Γ is a DAG whose leaves are labeled with clauses from Γ and such that every interior node n has exactly two predecessors, n_1 and n_2 so that n is labeled with the resolvent of the clauses labeling n_1 and n_2. In a resolution step involving the nodes, n_1, n_2 and n, as above, we say that the two clauses C and C' labeling the nodes n_1 and n_2 are the *parent clauses* of the resolvent clause, R, labeling the node n. In a resolution DAG, D, a clause, C' is said to be a *descendant* of a clause, C, iff there is a (directed) path from some node labeled with C to a node labeled with C'. A *resolution refutation for* Γ is a resolution DAG with a single root whose label is the empty

clause. (For more details on the resolution method, resolution DAGs, etc., one may consult Gallier [2], Chapter 4, or any of the books cited in Section 1.)

Here is an example of a resolution refutation for the set of clauses

$$\Gamma = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\} :$$



3 Completeness of Propositional Resolution: An Algorithm and its Correctness

Let Γ be a set of clauses. Thus, Γ is either the empty clause, $\square$, or it is a conjunction of clauses, $\Gamma = C_1, \ldots, C_n$. We define the *complexity*, $c(C)$, of a clause, C, as the number of disjunction symbols in C; i.e., if C consists of a single literal (i.e., $C = \{L\}$, for some literal, L), then $c(C) = 0$, else if $C = \{L_1, \ldots, L_m\}$ (with $m \geq 2$) where the L_i's are literals, then $c(C) = m - 1$ (we also set $c(\square) = 0$). If Γ is a conjunction of clauses, $\Gamma = C_1, \ldots, C_n$, then we set

$$c(\Gamma) = c(C_1) + \cdots + c(C_n).$$

We now give a recursive algorithm, `buildresol`, for constructing a resolution DAG from any set of clauses and then prove its correctness, namely, that if the input set of clauses is unsatisfiable, then the output resolution DAG is a resolution refutation. This establishes the completeness of propositional resolution constructively.

Our algorithm makes use of two functions, `percolate`, and `graft`.

1. The function `percolate`(D, A, L)

The inputs are: a resolution DAG, D, some selected leaf of D labeled with a clause, A, and some literal, L. This function adds the literal L to the clause A to form the clause $A \cup \{L\}$ and then "percolates" L down to the root of D. More precisely, we construct the resolution DAG, D', whose underlying unlabeled DAG is identical to D, as follows: Since D and D' have the same unlabeled DAG we refer to two nodes of D of D' as *corresponding nodes* if they are identical in the underlying unlabeled DAG. Consider any resolution step of D. If both parent clauses are not descendants of the premise A, then the corresponding resolution step of D' is the same. If the parent clauses in D are C and C' where C' is a descendant of the premise A (resp. C is a descendant of the premise A) and if R is the

resolvent ot C and C' in D, then the corresponding parent nodes in D' are labeled with C and $C' \cup \{L\}$ and their resolvent node with $R \cup \{L\}$ (resp. the corresponding parent nodes in D' are labeled with $C \cup \{L\}$ and C' and their resolvent node wih $R \cup \{L\}$). If both parent clauses C and C' in D are descendant of the premise A, then the corresponding parent nodes in D' are labeled with $C \cup \{L\}$ and $C' \cup \{L\}$ and their resolvent node with $R \cup \{L\}$.

Observe that if $\Delta \cup \{A\}$ is the set of premises of D, then $\Gamma = \Delta \cup \{A \cup \{L\}\}$ is the set of premises of $\texttt{percolate}(D, A, L)$.

For example, if D is the resolution DAG shown below (in fact, a resolution refutation)

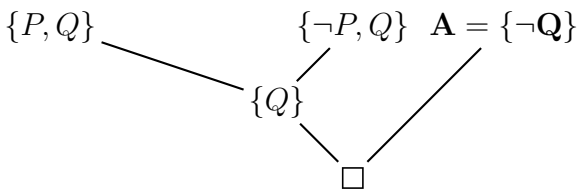


Figure 1: Resolution DAG D

then adding $L = \neg P$ to $A = \{\neg Q\}$ in D yields the resolution DAG D' produced by $\texttt{percolate}(D, A, L)$:

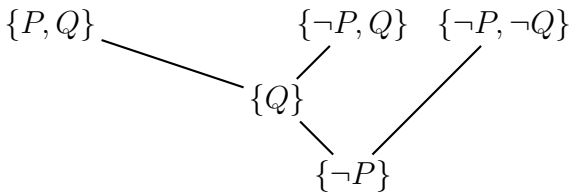


Figure 2: Resolution DAG $D' = \texttt{percolate}(D, A, L)$

2. The function $\texttt{graft}(D_1, D_2)$

Its inputs are two resolution DAGs, D_1 and D_2, where the clause, C, labeling the root of D_1 is identical to one of the premises of D_2. Then, this function combines D_1 and D_2 by connecting the links to the premise labeled C in D_2 to the root of D_1, also labeled C, obtaining the resolution DAG $\texttt{graft}(D_1, D_2)$.

For example, if D_1 and D_2 are the resolution refutation DAGs shown below

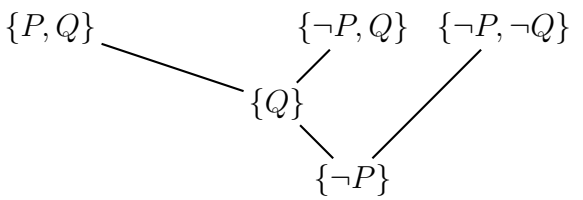


Figure 3: Resolution DAG D_1

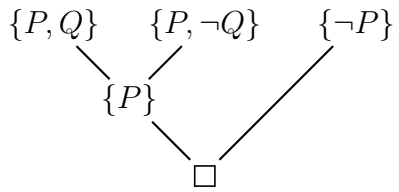


Figure 4: Resolution DAG D_2

we obtain the resolution DAG

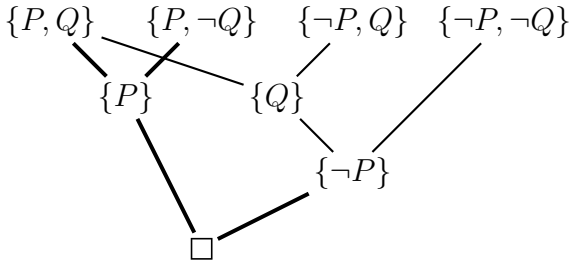


Figure 5: Resolution DAG $\texttt{graft}(D_1, D_2)$

where the edges coming from D_2 are indicated with thicker lines. The algorithm `buildresol` is shown below.

3. The algorithm `buildresol`(Γ)

The input to `buildresol` is a set of clauses, Γ.

function `buildresol`(Γ)
begin
if all clauses in Γ are literals **then**
  if Γ contains complementary literals L and $\neg L$,
  then return a resolution refutation with leaves L and $\neg L$
  else abort
  endif
else select any nonliteral clause, C, in Γ and select any literal, L, in C;
let $C = A \cup \{L\}$; let $\Gamma = \Delta \cup \{C\}$;
$D_1 = \texttt{buildresol}(\Delta \cup \{A\})$; $D_2 = \texttt{buildresol}(\Delta \cup \{L\})$; $D_1' = \texttt{percolate}(D_1, A, L)$;
  if D_1' is a resolution DAG
  then return D_1'
  else $D = \texttt{graft}(D_1', D_2)$; return D
  endif
endif
end

Finally, we prove the correctness of our recursive algorithm `buildresol`.

Theorem 3.1 *For every conjunction of clauses, Γ, if Γ is unsatisfiable, then the algorithm* `builresol` *outputs a resolution refutation for Γ. Therefore, propositional resolution is complete.*

Proof. We prove the correctness of the algorithm `buildresol` by induction on $c(\Gamma)$. Let $\Gamma = C_1, \ldots, C_n$. We may assume $\Gamma \neq \square$, since the case $\Gamma = \square$ is trivial. We proceed by induction on $c(\Gamma)$.

If $c(\Gamma) = 0$, then every clause, C_i, contains a single literal and if Γ is unsatisfiable, then there must be two complementary clauses, $C_i = \{P\}$ and $C_j = \{\neg P\}$, in Γ. Thus, we instantly get a resolution refutation by applying the resolution rule to $\{P\}$ and $\{\neg P\}$.

Otherwise, $c(\Gamma) > 0$, so there is some clause in Γ that contains at least two literals. Pick any such clause, C, and pick any literal, L, in C. Write $C = A \cup \{L\}$ with $A \neq \square$ and write $\Gamma = \Delta, C$ (Δ can't be empty since Γ is unsatisfiable). As $\Gamma = \Delta, A \cup \{L\}$ is unsatisfiable, *both* Δ, A and Δ, L must be unsatisfiable. However, observe that

$$c(\Delta, A) < c(\Gamma) \quad \text{and} \quad c(\Delta, L) < c(\Gamma).$$

Therefore, by the induction hypothesis, the algorithm `buildresol` produces two resolution refutations, D_1 and D_2, with sets of premises Δ, A and Δ, L, respectively. Now, consider the resolution DAG, $D_1' = \texttt{percolate}(D_1, A, L)$, obtained from D_1 by adding L to the clause A and letting L percolate down to the root.

Observe that in D_1', every clause that is a descendant of the premise $A \cup \{L\}$ is of the form $C \cup \{L\}$, where C is the corresponding clause in D_1. Therefore, the root of the new DAG D_1' obtained from D_1 is either labeled $\square$ (this may happen when the other clause in a resolution step involving a descendent of the clause A already contains L) or L. In the first case, D_1' is already a resolution refutation for Γ and we are done. In the second case, we can combine D_1' and D_2 using $\texttt{graft}(D_1', D_2)$ since the root of D_1' is also labeled L, one of the premises of D_2. Clearly, we obtain a resolution refutation for Γ. $\square$

As an illustration of our algorithm, consider the set of clauses

$$\Gamma = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$$

as above and pick $C = \{\neg P, \neg Q\}$, $L = \neg P$ and $A = \{\neg Q\}$. After the two calls $\texttt{buildresol}(\Delta \cup \{A\})$ and $\texttt{buildresol}(\Delta \cup \{L\})$, we get the resolution refutations D_1:

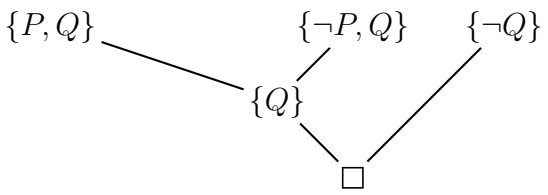


Figure 6: Resolution DAG $D_1 = \texttt{buildresol}(\Delta \cup \{A\})$

and D_2:

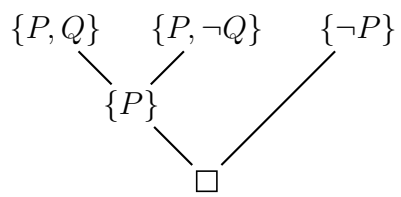


Figure 7: Resolution DAG $D_2 = \texttt{buildresol}(\Delta \cup \{L\})$

When we add $L = \neg P$ to $A = \{\neg Q\}$ in D_1, we get the resolution DAG $D_1' = \texttt{percolate}(D_1, A, L)$:

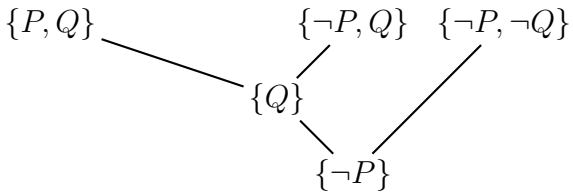


Figure 8: Resolution DAG $D_1' = \texttt{percolate}(D_1, A, L)$

Finally, we construct the resolution refutation $D = \texttt{graft}(D_1', D_2)$:

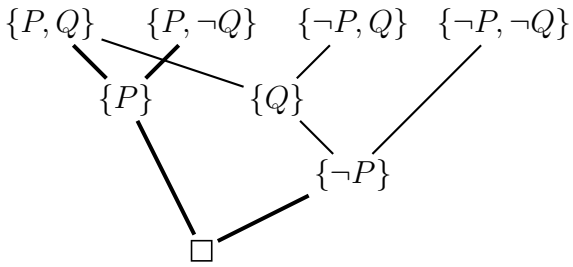


Figure 9: Resolution DAG $D = \texttt{graft}(D_1', D_2)$

where the edges coming from D_2 are indicated with thicker lines.

Observe that the proof of Theorem 3.1 proves that if Γ is unsatisfiable, then our algorithm succeeds no matter which clause containing at least two literals is chosen and no matter which literal is picked in such a clause.

Furthermore, as pointed out by one of the referees, although the proof of completeness is constructive in the sense that it shows an algorithm is correct, it does not explicitly use constructive logic. Nevertheless the logical proof can be recovered from the algorithm and it is constructive.

References

[1] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, first edition, 1973.

[2] Jean H. Gallier. *Logic For Computer Science*. Wiley, first edition, 1986.

[3] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, first edition, 1981.

[4] J.A. Robinson. A machine oriented logic based on the resolution principle. *J.ACM*, 12(1):23–41, 1965.

[5] J.A. Robinson. *Logic: Form and Function*. North-Holland, first edition, 1979.

[6] Judith Underwood. The tableau algorithm for intuitionistic propositional calculus as a constructive completeness proof. In Basin D., Fronhofer B., Hahnle R., Posegga J., and Schwind C., editors, *Second Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Marseille, France*, pages 245–248. Max–Planck–Institut fur Informatik, Saarbrucken, Germany, 1993.

Chapter 9

SLD-Resolution And Logic Programming (PROLOG)

9.1 Introduction

We have seen in Chapter 8 that the resolution method is a complete procedure for showing unsatisfiability. However, finding refutations by resolution can be a very expensive process in the general case. If subclasses of formulae are considered, more efficient procedures for producing resolution refutations can be found. This is the case for the class of Horn clauses. A Horn clause is a disjunction of literals containing at most one positive literal. For sets of Horn clauses, there is a variant of resolution called SLD-resolution, which enjoys many nice properties. SLD-resolution is a special case of a refinement of the resolution method due to Kowalski and Kuehner known as SL-resolution (Kowalski and Kuehner, 1970), a variant of Model Elimination (Loveland, 1978), and applies to special kinds of Horn clauses called definite clauses. We shall present SLD-resolution and show its completeness for Horn clauses.

SLD-resolution is also interesting because it is the main computation procedure used in PROLOG. PROLOG is a programming language based on logic, in which a computation is in fact a refutation. The idea to define a program as a logic formula and view a refutation as a computation is a very fruitful one, because it reduces the complexity of proving the correctness of programs. In fact, it is often claimed that logic programs are obviously correct, because these programs "express" the assertions that they should satisfy. However, this is not quite so, because the notion of correctness is relative, and

one still needs to define the semantics of logic programs in some independent fashion. This will be done in Subsection 9.5.4, using a model-theoretic semantics. Then, the correctness of SLD-resolution (as a computation procedure) with respect to the model-theoretic semantics will be proved.

In this chapter, as in Chapter 8, we begin by studying SLD-resolution in the propositional case, and then use the lifting technique to extend the results obtained in the propositional case to the first-order case. Fortunately, the lifting process goes very smoothly.

As in Chapter 4, in order to prove the completeness of SLD-resolution for propositional Horn clauses, we first show that Horn clauses have $GCNF'$-proofs of a certain kind, that we shall call $GCNF'$-proofs in SLD-form. Then, we show that every $GCNF'$-proof in SLD-form can be mapped into a linear SLD-refutation. Hence, the completeness proof for SLD-resolution is constructive.

The arguments used for showing that every unsatisfiable Horn clause has a $GCNF'$-proof in SLD-form are quite basic and combinatorial in nature. Once again, the central concept is that of *proof transformation*.

We conclude this chapter by discussing the notion of logic program and the idea of viewing SLD-resolution as a computation procedure. We provide a rigorous semantics for logic programs, and show the correctness and completeness of SLD-resolution with respect to this semantics.

The contents of Section 9.5 can be viewed as the theoretical foundations of logic programming, and PROLOG in particular.

9.2 $GCNF'$-Proofs in SLD-Form

First, we shall prove that every unsatisfiable propositional Horn clause has a $GCNF'$-proof of a certain type, called a proof in SLD-form. In order to explain the method for converting a $GCNF'$-proof into an SLD-resolution proof, it is convenient to consider the special case of sets of Horn clauses, containing exactly one clause containing no positive literals (clause of the form $\{\neg P_1, ..., \neg P_m\}$). Other Horn clauses will be called *definite clauses*.

9.2.1 The Case of Definite Clauses

These concepts are defined as follows.

Definition 9.2.1 A *Horn clause* is a disjunction of literals containing at most one positive literal. A Horn clause is a *definite clause* iff it contains a (single) positive literal. Hence, a definite clause is either of the form

$$\{Q\}, \quad \text{or} \quad \{\neg P_1, ..., \neg P_m, Q\}.$$

A Horn clause of the form

$$\{\neg P_1, ..., \neg P_m\}$$

is called a *negative clause* or *goal clause*.

For simplicity of notation, a clause $\{Q\}$ will also be denoted by Q. In the rest of this section, we restrict our attention to sets S of clauses consisting of definite clauses except for one goal clause. Our goal is to show that for a set S consisting of definite clauses and of a single goal B, if S is $GCNF'$-provable, then there is a proof having the property that whenever a $\vee : left$ rule is applied to a definite clause $\{\neg P_1, ..., \neg P_m, Q\}$, the rule splits it into $\{\neg P_1, ..., \neg P_m\}$ and $\{Q\}$, the sequent containing $\{Q\}$ is an axiom, and the sequent containing $\{\neg P_1, ..., \neg P_m\}$ does not contain $\neg Q$.

EXAMPLE 9.2.1

Consider the set S of Horn clauses with goal $\{\neg P_1, \neg P_2\}$ given by:

$$S = \{\{P_3\}, \{P_4\}, \{\neg P_1, \neg P_2\}, \{\neg P_3, \neg P_4, P_1\}, \{\neg P_3, P_2\}\}.$$

The following is a $GCNF'$-proof:

$$\dfrac{\dfrac{P_3, \neg P_3 \rightarrow \quad P_4, \neg P_4 \rightarrow}{P_3, P_4, \{\neg P_3, \neg P_4\} \rightarrow} \quad \dfrac{\neg P_1, P_1 \rightarrow \quad \dfrac{\neg P_2, P_2 \rightarrow \quad P_3, \neg P_3 \rightarrow}{P_3, \neg P_2, \{\neg P_3, P_2\} \rightarrow}}{P_3, \{\neg P_1, \neg P_2\}, P_1, \{\neg P_3, P_2\} \rightarrow}}{P_3, P_4, \{\neg P_1, \neg P_2\}, \{\neg P_3, \neg P_4, P_1\}, \{\neg P_3, P_2\} \rightarrow}$$

Another proof having the properties mentioned above is

$$\dfrac{\dfrac{\neg P_1, P_1 \rightarrow \quad \dfrac{P_3, \neg P_3 \rightarrow \quad P_4, \neg P_4 \rightarrow}{P_3, P_4, \{\neg P_3, \neg P_4\} \rightarrow}}{P_3, P_4, \neg P_1, \{\neg P_3, \neg P_4, P_1\} \rightarrow} \quad \dfrac{\neg P_2, P_2 \rightarrow \quad P_3, \neg P_3 \rightarrow}{P_3, \neg P_2, \{\neg P_3, P_2\} \rightarrow}}{P_3, P_4, \{\neg P_1, \neg P_2\}, \{\neg P_3, \neg P_4, P_1\}, \{\neg P_3, P_2\} \rightarrow}$$

Observe that in the above proof, the $\vee : left$ rule is first applied to the goal clause $\{\neg P_1, \neg P_2\}$, and then it is applied to split each definite clause $\{\neg Q_1, ..., \neg Q_m, Q\}$ into $\{\neg Q_1, ..., \neg Q_m\}$ and $\{Q\}$, in such a way that the sequent containing $\{Q\}$ is the axiom $Q, \neg Q \rightarrow$. Note also that each clause $\{\neg Q_1, ..., \neg Q_m\}$ resulting from splitting a definite clause as indicated above is the only goal clause in the sequent containing it.

9.2.2 $GCNF'$-Proofs in SLD-Form

The above example suggests that if a set of definite clauses with goal B is $GCNF'$-provable, it has a proof obtained by following rules described below, starting with a one-node tree containing the goal $B = \{\neg P_1, ..., \neg P_m\}$:

(1) If no leaf of the tree obtained so far contains a clause consisting of a single negative literal $\neg Q$ then,

As long as the tree is not a $GCNF'$-proof tree, apply the $\vee : left$ rule to each goal clause B of the form $\{\neg Q_1, ..., \neg Q_m\}$ $(m > 1)$ in order to form m immediate descendants of B, else

(2) For every goal clause consisting of a single negative literal $\neg Q$, find a definite clause $\{\neg P_1, ..., \neg P_k, Q\}$ (or Q when $k = 0$), and split $\{\neg P_1, ..., \neg P_k, Q\}$ using the $\vee : left$ rule in order to get the axiom $\neg Q, Q \rightarrow$ in one node, $\{\neg P_1, ..., \neg P_m\}$ in the other, and drop $\neg Q$ from that second node.

Go back to (1).

In is not clear that such a method works, and that in step (2), the existence of a definite clause $\{\neg P_1, ..., \neg P_k, Q\}$ such that Q cancels $\neg Q$ is guaranteed. However, we are going to prove that this is always the case. First, we define the type of proofs arising in the procedure described above.

Definition 9.2.2 Given a set S of clauses consisting of definite clauses and of a single goal B, a $GCNF'$-proof is in *SLD-form* iff the conditions below are satisfied:

For every node B in the tree that is not an axiom:

(1) If the set of clauses labeling that node does not contain any clause consisting of a single negative literal $\neg Q$, then it contains a single goal clause of the form $\{\neg Q_1, ..., \neg Q_m\}$ $(m > 1)$, and the $\vee : left$ rule is applied to this goal clause in order to form m immediate descendants of B.

(2) If the set of clauses labeling that node contains some single negative literal, for such a clause $\neg Q$, there is some definite clause

$$\{\neg P_1, ..., \neg P_k, Q\},$$

$(k > 0)$, such that the $\vee : left$ rule is applied to

$$\{\neg P_1, ..., \neg P_k, Q\}$$

in order to get the axiom $\neg Q, Q \rightarrow$ and a sequent containing the single goal clause $\{\neg P_1, ..., \neg P_k\}$.

We are now going to prove that if a set of clauses consisting of definite clauses and of a single goal clause if provable in $GCNF'$, then it has a proof in SLD-form. For this, we are going to perform proof transformations, and use simple combinatorial properties.

9.2.3 Completeness of Proofs in SLD-Form

First, we need to show that every $GCNF'$-provable set of clauses has a proof in which no weakenings takes place. This is defined as follows.

Definition 9.2.3 A $GCNF'$-proof is *without weakenings* iff every application of the $\vee : left$ rule is of the form:

$$\frac{\Gamma, A_1, ..., A_m \rightarrow \qquad \Gamma, B \rightarrow}{\Gamma, (A_1 \vee B), ..., (A_m \vee B) \rightarrow}$$

We have the following normal form lemma.

Lemma 9.2.1 If a set S of clauses is $GCNF'$-provable, then a $GCNF'$-proof without weakenings and in which all the axioms contain only literals can be constructed.

Proof: Since G' is complete, $S \rightarrow$ has a G'-proof T. By lemma 6.3.1 restricted to propositions, $S \rightarrow$ has a G'-proof T' in which all axioms are atomic. Using lemma 4.2.2, $S \rightarrow$ has a G'-proof T'' in which all axioms are atomic, and in which all applications of the $\vee : left$ rule precede all applications of the $\neg : left$ rule. The tree obtained from T'' by retaining the portion of the proof tree that does not contain $\neg : left$ inferences is the desired $GCNF'$-proof. □

The following permutation lemma is the key to the conversion to SLD-form.

Lemma 9.2.2 Let S be a set of clauses that has a $GCNF'$-proof T. Then, for any clause C in S having more than one literal, for any partition of the literals in C into two disjunctions A and B such $C = (A \vee B)$, there is a $GCNF'$-proof T' in which the $\vee : left$ rule is applied to $(A \vee B)$ at the root. Furthermore, if the proof T of S is without weakenings and all axioms contain only literals, the proof T' has the same depth as T.

Proof: Observe that representing disjunctions of literals as unordered sets of literals is really a convenience afforded by the associativity, commutativity and idempotence of $\vee$, but that this convenience does not affect the completeness of G'. Hence, no matter how C is split into a disjunction $(A \vee B)$, the sequent $\Gamma, (A \vee B) \rightarrow$ is G'-provable. By converting a G'-proof of $\Gamma, (A \vee B) \rightarrow$ given by lemma 9.2.1 into a $GCNF'$-proof, we obtain a $GCNF'$-proof without weakenings, and in which the $\vee : left$ rule is applied to A and B only after it is applied to $(A \vee B)$. If the $\vee : left$ rule applied at the root does not apply to $(A \vee B)$, it must apply to some other disjunction $(C \vee D)$. Such a proof T must be of the following form:

Tree T

$$\frac{\Pi_1 \qquad\qquad \Pi_2}{\Gamma, (A \vee B), (C \vee D) \rightarrow}$$

where Π_1 is the tree

$$\frac{\dfrac{\dfrac{T_1}{\Gamma_1, A \rightarrow} \qquad \dfrac{S_1}{\Gamma_1, B \rightarrow}}{\Gamma_1, (A \vee B) \rightarrow} \qquad \dfrac{\dfrac{T_m}{\Gamma_m, A \rightarrow} \qquad \dfrac{S_m}{\Gamma_m, B \rightarrow}}{\Gamma_m, (A \vee B) \rightarrow}}{R}$$

$$\Gamma, (A \vee B), C \rightarrow$$

and where Π_2 is the tree

$$\frac{\dfrac{\dfrac{T'_1}{\Delta_1, A \rightarrow} \qquad \dfrac{S'_1}{\Delta_1, B \rightarrow}}{\Delta_1, (A \vee B) \rightarrow} \qquad \dfrac{\dfrac{T'_n}{\Delta_n, A \rightarrow} \qquad \dfrac{S'_n}{\Delta_n, B \rightarrow}}{\Delta_n, (A \vee B) \rightarrow}}{S}$$

$$\Gamma, (A \vee B), D \rightarrow$$

In the above proof, we have indicated the nodes to which the $\vee : left$ rule is applied, nodes that must exist since all axioms consist of literals. The inferences above $\Gamma, (A \vee B), C$ and below applications of the $\vee : left$ rule to $(A \vee B)$ are denoted by R, and the similar inferences above $\Gamma, (A \vee B), D$ are denoted by S. We can transform T into T' by applying the $\vee : left$ rule at the root as shown below:

Tree T'

$$\frac{\Pi'_1 \qquad\qquad \Pi'_2}{\Gamma, (A \vee B), (C \vee D) \rightarrow}$$

where Π_1' is the tree

$$\frac{T_1}{\Gamma_1, A \rightarrow} \quad \frac{T_m}{\Gamma_m, A \rightarrow} \quad \frac{T_1'}{\Delta_1, A \rightarrow} \quad \frac{T_n'}{\Delta_n, A \rightarrow}$$

$$R \qquad\qquad S$$

$$\frac{\Gamma, A, C \rightarrow \qquad \Gamma, A, D \rightarrow}{\Gamma, A, (C \vee D) \rightarrow}$$

and where Π_2' is the tree

$$\frac{S_1}{\Gamma_1, B \rightarrow} \quad \frac{S_m}{\Gamma_m, B \rightarrow} \quad \frac{S_1'}{\Delta_1, B \rightarrow} \quad \frac{S_n'}{\Delta_n, B \rightarrow}$$

$$R \qquad\qquad S$$

$$\frac{\Gamma, B, C \rightarrow \qquad \Gamma, B, D \rightarrow}{\Gamma, B, (C \vee D) \rightarrow}$$

Clearly, $depth(T') = depth(T)$. □

Note that T' is obtained from T by permutation of inferences. We need another crucial combinatorial property shown in the following lemma.

Lemma 9.2.3 Let S be an arbitrary set of clauses such that the subset of clauses containing more than one literal is the nonempty set $\{C_1, ..., C_n\}$ and the subset consisting of the one-literal clauses is J. Assume that S is $GCNF'$-provable, and that we have a proof T without weakenings such that all axioms consist of literals. Then, every axiom is labeled with a set of literals of the form $\{L_1, ..., L_n\} \cup J$, where each literal L_i is in C_i, $i = 1, ..., n$.

Proof: We proceed by induction on proof trees. Since S contains at least one clause with at least two literals and the axioms only contain literals, $depth(T) \geq 1$. If T has depth 1, then there is exactly one application of the $\vee : rule$ and the proof is of the following form:

$$\frac{J, L_1 \rightarrow \qquad J, L_2 \rightarrow}{J, (L_1 \vee L_2) \rightarrow}$$

Clearly, the lemma holds.

If T is a tree of depth $k+1$, it is of the following form,

$$\dfrac{\dfrac{T_1}{\Gamma, A \rightarrow} \qquad \dfrac{T_2}{\Gamma, B \rightarrow}}{\Gamma, (A \vee B) \rightarrow}$$

where we can assume without loss of generality that $C_n = (A \vee B)$. By the induction hypothesis, each axiom of T_1 is labeled with a set of clauses of the form $\{L_1, ..., L_n\} \cup J$, where each literal L_i is in C_i for $i = 1, ..., n-1$, and either $L_n = A$ if A consists of a single literal, or L_n belongs to A. Similarly, each axiom of T_2 is labeled with a set of clauses of the form $\{L_1, ..., L_n\} \cup J$, where each literal L_i is in C_i for $i = 1, ..., n-1$, and either $L_n = B$ if B consists of a single literal, or L_n belongs to B. Since the union of A and B is C_n, every axiom of T is labeled with a set of clauses of the form $\{L_1, ..., L_n\} \cup J$, where each literal L_i is in C_i, $i = 1, ..., n$. Hence, the lemma holds. □

As a consequence, we obtain the following useful corollary.

Lemma 9.2.4 Let S be a set of Horn clauses. If S is $GCNF'$-provable, then S contains at least one clause consisting of a single positive literal, and at least one goal (negative) clause.

Proof: It S is an axiom, this is obvious. Otherwise, by lemma 9.2.3, if S is $GCNF'$-provable, then it has a proof T without weakenings such that every axiom is labeled with a set of literals of the form $\{L_1, ..., L_n\} \cup J$, where each literal L_i is in C_i, $i = 1, ..., n$, and J is the set of clauses in S consisting of a single literal. If J does not contain any positive literals, since every Horn clause C_i contains a negative literal say $\neg A_i$, the set $\{\neg A_1, ..., \neg A_n\} \cup J$ contains only negative literals, and so cannot be an axiom. If every clause in J is positive and every clause C_i contains some positive literal say A_i, then $\{A_1, ..., A_n\} \cup J$ contains only positive literals and cannot be an axiom. □

In order to prove the main theorem of this section, we will need to show that the provability of a set of Horn clauses with several goals (negative clauses) reduces to the case of a set of Horn clauses with a single goal.

Lemma 9.2.5 Let S be a set of Horn clauses consisting of a set J of single positive literals, goal clauses N_1,...,N_k, and definite clauses C_1,...,C_m containing at least two literals.

If S is $GCNF'$-provable, then there is some i, $1 \leq i \leq k$, such that

$$J \cup \{C_1, ..., C_m\} \cup \{N_i\}$$

is $GCNF'$-provable. Furthermore, if T is a $GCNF'$-proof of S without weakenings and such that the axioms contain only literals, $J \cup \{C_1, ..., C_m\} \cup \{N_i\}$ has a proof of depth less than or equal to the depth of T.

Proof: We proceed by induction on proof trees. Let T be a $GCNF'$-proof of S without weakenings and such that all axioms contain only literals.

Case 1: $J \cup \{C_1, ..., C_m\} \cup \{N_1, ..., N_k\}$ is an axiom. Then, one of the positive literals in J must be the conjugate of some negative clause N_i, and the lemma holds.

Case 2: The bottom $\vee : left$ rule is applied to one of the N_i. Without loss of generality, we can assume that it is $N_1 = \{\neg Q_1, ..., \neg Q_j, \neg P\}$.

Letting $\mathcal{C} = C_1, ..., C_m$, the proof is of the form

$$\dfrac{\dfrac{T_1}{J, \mathcal{C}, N_2, ..., N_k, \{\neg Q_1, ..., \neg Q_j\} \rightarrow} \qquad \dfrac{T_2}{J, \mathcal{C}, N_2, ..., N_k, \neg P \rightarrow}}{J, \mathcal{C}, N_1, ..., N_k \rightarrow}$$

Observe that the bottom sequents of T_1 and T_2 satisfy the conditions of the induction hypothesis. There are two subcases. If both

$$J, C_1, ..., C_m, \{\neg Q_1, ..., \neg Q_j\} \rightarrow \quad \text{and}$$
$$J, C_1, ..., C_m, \neg P \rightarrow$$

are provable, then

$$J, C_1, ..., C_m, \{\neg Q_1, ..., \neg Q_j, \neg P\} \rightarrow$$

is provable by application of the $\vee : rule$, and the lemma holds. If

$$J, C_1, ..., C_m, N_i \rightarrow$$

is provable for some i, $2 \leq i \leq k$, then the lemma also holds.

Case 3: The bottom $\vee : rule$ is applied to one of the C_i. Without loss of generality, we can assume that it is $C_1 = \{\neg Q_1, ..., \neg Q_j, P\}$. There are two subcases:

Case 3.1: Letting $\mathcal{N} = N_1, ..., N_k$, the proof is of the form

$$\dfrac{\dfrac{T_1}{J, C_2, ..., C_m, \mathcal{N}, \{\neg Q_1, ..., \neg Q_j\} \rightarrow} \qquad \dfrac{T_2}{J, P, C_2, ..., C_m, \mathcal{N} \rightarrow}}{J, C_1, ..., C_m, \mathcal{N} \rightarrow}$$

Again the induction hypothesis applies to both T_1 and T_2. If

$$J, C_2, ..., C_m, \{\neg Q_1, ..., \neg Q_j\} \rightarrow \quad \text{is provable and}$$
$$J, P, C_2, ..., C_m, N_i \rightarrow \quad \text{is provable}$$

for some i, $1 \leq i \leq k$, then by the $\vee : rule$,

$$J, C_1, ..., C_m, N_i \rightarrow$$

is also provable, and the lemma holds. If

$$J, C_2, ..., C_m, N_i \rightarrow$$

is provable for some i, $1 \leq i \leq k$, then

$$J, C_1, ..., C_m, N_i \rightarrow$$

is also provable (using weakening in the last $\vee : rule$).

Case 3.2: Letting $\mathcal{N} = N_1, ..., N_k$, the proof is of the form

$$\dfrac{\dfrac{T_1}{J, C_2, ..., C_m, \mathcal{N}, \{\neg Q_2, ..., \neg Q_j, P\} \rightarrow} \qquad \dfrac{T_2}{J, C_2, ..., C_m, \neg Q_1, \mathcal{N} \rightarrow}}{J, C_1, ..., C_m, \mathcal{N} \rightarrow}$$

Applying the induction hypothesis, either

$$J, C_2, ..., C_m, N_i, \{\neg Q_2, ..., \neg Q_j, P\}$$

is provable for some i, $1 \leq i \leq k$, and

$$J, C_2, ..., C_m, \neg Q_1 \rightarrow$$

is provable, and by the $\vee : rule$, $J, C_1, ..., C_m, N_i$ is provable and the lemma holds. Otherwise,

$$J, C_2, ..., C_m, N_i$$

is provable for some i, $1 \leq i \leq k$, and so $J, C_1, ..., C_m, N_i$ is also provable using weakening in the last $\vee : rule$. This concludes the proof. □

We are now ready to prove the main theorem of this section.

Theorem 9.2.1 (Completeness of proofs in SLD-form) If a set S consisting of definite clauses and of a single goal $B = \{\neg P_1, ..., \neg P_n\}$ is $GCNF'$-provable, then it has a $GCNF'$-proof in SLD-form.

Proof: Assume that S is not an axiom. By lemma 9.2.1, there is a $GCNF'$-proof T without weakenings, and such that all axioms consist of literals. We proceed by induction on the depth of proof trees. If $depth(T) = 1$, the proof is already in SLD-form (this is the base case of lemma 9.2.3). If $depth(T) > 1$, by n applications of lemma 9.2.2, we obtain a proof tree T' having the same depth as T, such that the i-th inference using the $\vee : left$ rule is applied to $\{\neg P_i, ..., \neg P_n\}$. Hence, letting $\mathcal{C} = C_1, ..., C_m$, the tree T' is of the form:

$$\dfrac{\dfrac{T_{n-1}}{J,\mathcal{C},\neg P_{n-1}\rightarrow} \qquad \dfrac{T_n}{J,\mathcal{C},\neg P_n\rightarrow}}{J,\mathcal{C},\{\neg P_{n-1},\neg P_n\}\rightarrow}$$

...

$$\dfrac{\dfrac{T_1}{J,\mathcal{C},\neg P_1\rightarrow} \qquad \dfrac{\dfrac{T_2}{J,\mathcal{C},\neg P_2\rightarrow} \qquad J,\mathcal{C},\{\neg P_3,...,\neg P_n\}\rightarrow}{J,\mathcal{C},\{\neg P_2,...,\neg P_n\}\rightarrow}}{J,\mathcal{C},\{\neg P_1,...,\neg P_n\}\rightarrow}$$

where J is the set of clauses consisting of a single positive literal, and each clause C_i has more than one literal. For every subproof rooted with $J, C_1, ..., C_m, \neg P_i \rightarrow$, by lemma 9.2.3, each axiom is labeled with a set of literals

$$\{L_1, ..., L_m\} \cup \{\neg P_i\} \cup J,$$

where each L_j is in C_j, $1 \leq j \leq m$. In particular, since each clause C_j contains a single positive literal A_j, for every i, $1 \leq i \leq n$, $\{A_1, ..., A_m\} \cup \{\neg P_i\} \cup J$ must be an axiom. Clearly, either some literal in J is of the form P_i, or there is some definite clause $C = \{\neg Q_1, ..., \neg Q_p, A_j\}$ among C_1,...,C_m, with positive literal $A_j = P_i$. In the first case, $J, C_1, ..., C_m, \neg P_i \rightarrow$ is an axiom and the tree T_i is not present. Otherwise, let $C' = \{C_1, ..., C_m\} - \{C\}$. Using lemma 9.2.2 again, we obtain a proof R_i of

$$J, C_1, ..., C_m, \neg P_i \rightarrow$$

(of depth equal to the previous one) such that the the $\vee : left$ rule is applied to C:

$$\dfrac{P_i, \neg P_i \rightarrow \qquad \dfrac{T_i'}{J, \{\neg Q_1, ..., \neg Q_p\}, C', \neg P_i \rightarrow}}{J, \{\neg Q_1, ..., \neg Q_p, P_i\}, C', \neg P_i \rightarrow}$$

Note that

$$J, \{\neg Q_1, ..., \neg Q_p\}, C', \neg P_i \rightarrow$$

has two goal clauses. By lemma 9.2.5, either

$$J, \{\neg Q_1, ..., \neg Q_p\}, C' \rightarrow$$

has a proof U_i, or

$$J, C', \neg P_i \rightarrow$$

has a proof V_i, and the depth of each proof is no greater than the depth of the proof R_i of $J, \{\neg Q_1, ..., \neg Q_p, P_i\}, C', \neg P_i \rightarrow$. In the second case, by performing a weakening in the last inference of V_i, we obtain a proof for $J, C_1 ..., C_m, \neg P_i \rightarrow$ of smaller depth than the original, and the induction hypothesis applies, yielding a proof in SLD-form for $J, C_1, ..., C_m, \neg P_i \rightarrow$. In the first case, $\neg P_i$ is dropped and, by the induction hypothesis, we also have a proof in SLD-form of the form:

$$\dfrac{P_i, \neg P_i \rightarrow \qquad \dfrac{T_i''}{J, \{\neg Q_1, ..., \neg Q_p\}, C' \rightarrow}}{J, \{\neg Q_1, ..., \neg Q_p, P_i\}, C', \neg P_i \rightarrow}$$

Hence, by combining these proofs in SLD-form, we obtain a proof in SLD-form for S. □

Combining theorem 9.2.1 and lemma 9.2.5, we also have the following theorem.

Theorem 9.2.2 Let S be a set of Horn clauses, consisting of a set J of single positive literals, goal clauses N_1,...,N_k, and definite clauses C_1,...,C_m containing at least two literals. If S is $GCNF'$-provable, then there is some i, $1 \leq i \leq k$, such that

$$J \cup \{C_1, ..., C_m\} \cup \{N_i\}$$

has a $GCNF'$-proof in SLD-form. □

Proof: Obvious by theorem 9.2.1 and lemma 9.2.5.

In the next section, we shall show how proofs in SLD-form can be converted into resolution refutations of a certain type.

PROBLEMS

9.2.1. Give a $GCNF'$-proof in SLD-form for each of the following sequents:

$$\{\neg P_3, \neg P_4, P_5\}, \{\neg P_1, P_2\}, \{\neg P_2, P_1\}, \{\neg P_3, P_4\}, \{P_3\}, \{\neg P_1, \neg P_2\}, \{\neg P_5, P_2\} \rightarrow$$

$$\{P_1\}, \{P_2\}, \{P_3\}, \{P_4\}, \{\neg P_1, \neg P_2, P_6\}, \{\neg P_3, \neg P_4, P_7\}, \{\neg P_6, \neg P_7, P_8\}, \{\neg P_8\} \rightarrow$$

$$\{\neg P_2, P_3\}, \{\neg P_3, P_4\}, \{\neg P_4, P_5\}, \{P_3\}, \{P_1\}, \{P_2\}, \{\neg P_1\},$$
$$\{\neg P_3, P_6\}, \{\neg P_3, P_7\}, \{\neg P_3, P_8\} \rightarrow$$

9.2.2. Complete the missing details in the proof of lemma 9.2.5.

9.2.3. Write a computer program for building proof trees in SLD-form for Horn clauses.

∗ **9.2.4.** Given a set S of Horn clauses, we define an H-tree for S as a tree labeled with propositional letters and satisfying the following properties:

(i) The root of T is labeled with **F** (false);

(ii) The immediate descendants of **F** are nodes labeled with propositional letters P_1,...,P_n such that $\{\neg P_1, ..., \neg P_n\}$ is some goal clause in S;

(iii) For every nonroot node in the tree labeled with some letter Q, either the immediate descendants of that node are nodes labeled with letters P_1,...,P_k such that $\{\neg P_1, ..., \neg P_k, Q\}$ is some clause in S, or this node is a leaf if $\{Q\}$ is a clause in S.

Prove that S is unsatisfiable iff it has an H-tree.

9.3 SLD-Resolution in Propositional Logic

SLD-refutations for sets of Horn clauses can be viewed as linearizations of $GCNF'$-proofs in SLD-form.

9.3.1 SLD-Derivations and SLD-Refutations

First, we show how to linearize SLD-proofs.

Definition 9.3.1 The *linearization procedure* is a recursive algorithm that converts a $GCNF'$-proof in SLD-form into a sequence of negative clauses according to the following rules:

(1) Every axiom $\neg P, P \rightarrow$ is converted to the sequence $< \{\neg P\}, \square >$.

(2) For a sequent $R \rightarrow$ containing a goal clause $N = \{\neg P_1, ..., \neg P_n\}$, with $n > 1$, if $\mathcal{C}_i$ is the sequence of clauses that is the linearization of the subtree with root the i-th descendant of the sequent $R \rightarrow$, construct the sequence obtained as follows:

Concatenate the sequences $\mathcal{C}'_1$,...,$\mathcal{C}'_{n-1}$, $\mathcal{C}_n$, where, for each i, $1 \leq i \leq n-1$, letting n_i be the number of clauses in the sequence $\mathcal{C}_i$, the sequence $\mathcal{C}'_i$ has $n_i - 1$ clauses such that, for every j, $1 \leq j \leq n_i - 1$, if the j-th clause of $\mathcal{C}_i$ is

$$\{B_1, ..., B_m\},$$

then the j-th clause of $\mathcal{C}'_i$ is

$$\{B_1, ..., B_m, \neg P_{i+1}, ..., \neg P_n\}.$$

(3) For every nonaxiom sequent $\Gamma, \neg P \rightarrow$ containing some negative literal $\neg P$, if the definite clause used in the inference is $\{\neg P_1, ..., \neg P_m, P\}$, letting $\Delta = \Gamma - \{\neg P_1, ..., \neg P_m, P\}$, then if the sequence of clauses for the sequent $\Delta, \{\neg P_1, ..., \neg P_m\} \rightarrow$ is $\mathcal{C}$, form the sequence obtained by concatenating $\neg P$ and the sequence $\mathcal{C}$.

Note that by (1), (2), and (3), in (2), the first clause of each $\mathcal{C}'_i$, $(1 \leq i \leq n-1)$, is

$$\{\neg P_i, \neg P_{i+1}, ..., \neg P_n\},$$

and the first clause of $\mathcal{C}_n$ is $\{\neg P_n\}$.

The following example shows how such a linearization is done.

EXAMPLE 9.3.1

Recall the proof tree in SLD-form given in example 9.2.1:

$$\dfrac{\dfrac{\dfrac{}{\neg P_1, P_1 \rightarrow} \quad \dfrac{\dfrac{}{P_3, \neg P_3 \rightarrow} \quad \dfrac{}{P_4, \neg P_4 \rightarrow}}{P_3, P_4, \{\neg P_3, \neg P_4\} \rightarrow}}{P_3, P_4, \neg P_1, \{\neg P_3, \neg P_4, P_1\} \rightarrow} \quad \dfrac{\dfrac{}{\neg P_2, P_2 \rightarrow} \quad \dfrac{}{P_3, \neg P_3 \rightarrow}}{P_3, \neg P_2, \{\neg P_3, P_2\} \rightarrow}}{P_3, P_4, \{\neg P_1, \neg P_2\}, \{\neg P_3, \neg P_4, P_1\}, \{\neg P_3, P_2\} \rightarrow}$$

The sequence corresponding to the left subtree is

$$< \{\neg P_1\}, \{\neg P_3, \neg P_4\}, \{\neg P_4\}, \square\} >$$

and the sequence corresponding to the right subtree is

$$< \{\neg P_2\}, \{\neg P_3\}, \square >$$

Hence, the sequence corresponding to the proof tree is

$$< \{\neg P_1, \neg P_2\}, \{\neg P_3, \neg P_4, \neg P_2\},$$
$$\{\neg P_4, \neg P_2\}, \{\neg P_2\}, \{\neg P_3\}, \square > .$$

This last sequence is an SLD-refutation, as defined below.

Definition 9.3.2 Let S be a set of Horn clauses consisting of a set D of definite clauses and a set $\{G_1, ..., G_q\}$ of goals. An *SLD-derivation* for S is a sequence $< N_0, N_1, ..., N_p >$ of negative clauses satisfying the following properties:

(1) $N_0 = G_j$, where G_j is one of the goals;

(2) For every N_i in the sequence, $0 \leq i < p$, if

$$N_i = \{\neg A_1, ..., \neg A_{k-1}, \neg A_k, \neg A_{k+1}, ..., \neg A_n\},$$

then there is some definite clause

$$C_i = \{\neg B_1, ..., \neg B_m, A_k\}$$

in D such that, if $m > 0$, then

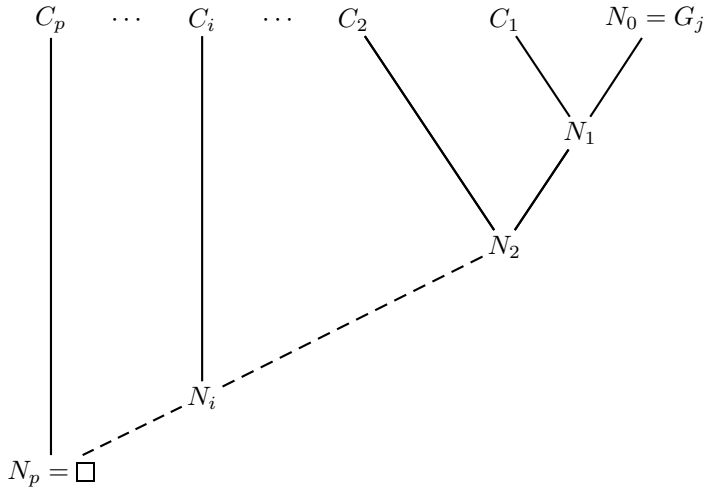
$$N_{i+1} = \{\neg A_1, ..., \neg A_{k-1}, \neg B_1, ..., \neg B_m, \neg A_{k+1}, ..., \neg A_n\}$$

else if $m = 0$ then

$$N_{i+1} = \{\neg A_1, ..., \neg A_{k-1}, \neg A_{k+1}, ..., \neg A_n\}.$$

An SLD-derivation is an *SLD-refutation* iff $N_p = \square$. The *SLD-resolution method* is the method in which a set of of Horn clauses is shown to be unsatisfiable by finding an SLD-refutation.

Note that an SLD-derivation is a linear representation of a resolution DAG of the following special form:



At each step, the clauses

$$\{\neg A_1, ..., \neg A_{k-1}, \neg A_k, \neg A_{k+1}, ..., \neg A_n\} \quad \text{and}$$
$$\{\neg B_1, ..., \neg B_m, A_k\}$$

are resolved, the literals A_k and $\neg A_k$ being canceled. The literal A_k is called the *selected atom* of N_i, and the clauses N_0, C_1, ..., C_p are the *input clauses*.

Such a resolution method is a form of linear input resolution, because it resolves the current clause N_k with some clause in the input set D.

By the soundness of the resolution method (lemma 4.3.2), the SLD-resolution method is sound.

EXAMPLE 9.3.2

The sequence

$$< \{\neg P_1, \neg P_2\}, \{\neg P_3, \neg P_4, \neg P_2\}, \\ \{\neg P_4, \neg P_2\}, \{\neg P_2\}, \{\neg P_3\}, \square >$$

of example 9.3.1 is an SLD-refutation.

9.3.2 Completeness of SLD-Resolution for Horn Clauses

In order to show that SLD-resolution is complete for Horn clauses, since by theorem 9.2.2 every set of Horn clauses has a $GCNF'$-proof in SLD-form, it is sufficient to prove that the linearization algorithm of definition 9.3.1 converts a proof in SLD-form to an SLD-refutation.

Lemma 9.3.1 (Correctness of the linearization process) Given any $GCNF'$-proof T in SLD-form, the linearization procedure outputs an SLD-refutation.

Proof: We proceed by induction on proofs. If T consists of an axiom, then the set S of Horn clauses contains a goal $\neg Q$ and a positive literal Q, and we have the SLD-refutation $< \{\neg Q\}, \square >$.

Otherwise, because it is in SLD-form, letting $\mathcal{C} = C_1, ..., C_m$, the tree T has the following structure:

$$\dfrac{\dfrac{T_{n-1}}{J, \mathcal{C}, \neg P_{n-1} \rightarrow} \qquad \dfrac{T_n}{J, \mathcal{C}, \neg P_n \rightarrow}}{J, \mathcal{C}, \{\neg P_{n-1}, \neg P_n\} \rightarrow}$$

...

$$\dfrac{\dfrac{T_1}{J, \mathcal{C}, \neg P_1 \rightarrow} \qquad \dfrac{\dfrac{T_2}{J, \mathcal{C}, \neg P_2 \rightarrow} \qquad J, \mathcal{C}, \{\neg P_3, ..., \neg P_n\} \rightarrow}{J, \mathcal{C}, \{\neg P_2, ..., \neg P_n\} \rightarrow}}{J, \mathcal{C}, \{\neg P_1, ..., \neg P_n\} \rightarrow}$$

Each tree T_i that is not an axiom is also in SLD-form and has the following shape:

$$\frac{P_i, \neg P_i \rightarrow \qquad \dfrac{T_i'}{J, \{\neg Q_1, ..., \neg Q_p\}, C' \rightarrow}}{J, \{\neg Q_1, ..., \neg Q_p, P_i\}, C', \neg P_i \rightarrow}$$

where $C' = \{C_1, ..., C_m\} - \{C\}$, for some definite clause $C = \{\neg Q_1, ..., \neg Q_p, P_i\}$.

By the induction hypothesis, each tree T_i' is converted to an SLD-refutation

$$Y_i =< \{\neg Q_1, ..., \neg Q_p\}, N_2, ..., N_q > .$$

By rule (3), the proof tree T_i is converted to the SLD-refutation X_i obtained by concatenating $\{\neg P_i\}$ and Y_i. But then,

$$X_i =< \{\neg P_i\}, \{\neg Q_1, ..., \neg Q_p\}, N_2, ..., N_q >$$

is an SLD-refutation obtained by resolving $\{\neg P_i\}$ with $\{\neg Q_1, ..., \neg Q_p, P_i\}$.

If T_i is an axiom then by rule (1) it is converted to $< \{\neg P_i\}, \square >$, which is an SLD-refutation.

Finally, rule (2) combines the SLD-refutations X_1,...,X_n in such a way that the resulting sequence is an SLD-refutation. Indeed, for every i, $1 \leq i \leq n-1$, X_i becomes the SLD-derivation X_i', where

$$X_i' =< \{\neg P_i, \neg P_{i+1}..., \neg P_n\}, \{\neg Q_1, ..., \neg Q_p, \neg P_{i+1}..., \neg P_n\},$$
$$N_2 \cup \{\neg P_{i+1}..., \neg P_n\}, ..., N_{q-1} \cup \{\neg P_{i+1}..., \neg P_n\} >,$$

and so the entire sequence X_1',...,X_{n-1}',X_n is an SLD-refutation starting from the goal $\{\neg P_1, ..., \neg P_n\}$. $\square$

As a corollary, we have the completeness of SLD-resolution for Horn clauses.

Theorem 9.3.1 (Completeness of SLD-resolution for Horn clauses) The SLD-resolution method is complete for Horn clauses. Furthermore, if the first negative clause is $\{\neg P_1, ..., \neg P_n\}$, for every literal $\neg P_i$ in this goal, there is an SLD-resolution whose first selected atom is P_i.

Proof: Completeness is a consequence of lemma 9.3.1 and theorem 9.2.2. It is easy to see that in the linearization procedure, the order in which the subsequences are concatenated does not matter. This implies the second part of the lemma. $\square$

Actually, since SLD-refutations are the result of linearizing proof trees in SLD-form, it is easy to show that any atom P_i such that $\neg P_i$ belongs to a negative clause N_k in an SLD-refutation can be chosen as the selected atom.

By theorem 9.2.2, if a set S of Horn clauses with several goals $N_1, ..., N_k$ is $GCNF'$-provable, then there is some goal N_i such that $S - \{N_1, ..., N_{i-1}, N_{i+1}, ..., N_k\}$ is $GCNF'$-provable. This does not mean that there is a unique such N_i, as shown by the following example.

EXAMPLE 9.3.2

Consider the set S of clauses:

$$\{P\}, \{Q\}, \{\neg S, R\}, \{\neg R, \neg P\}, \{\neg R, \neg Q\}, \{S\}.$$

We have two SLD-refutations:

$$< \{\neg R, \neg P\}, \{\neg R\}, \{\neg S\}, \square >$$

and

$$< \{\neg R, \neg Q\}, \{\neg R\}, \{\neg S\}, \square > .$$

In the next section, we generalize SLD-resolution to first-order languages without equality, using the lifting technique of Section 8.5.

PROBLEMS

9.3.1. Apply the linearization procedure to the proof trees in SLD-form obtained in problem 9.2.1.

9.3.2. Give different SLD-resolution refutations for the following sets of clauses:

$$\{P_1\}, \{P_2\}, \{P_3\}, \{P_4\}, \{\neg P_1, \neg P_2, P_6\}, \{\neg P_3, \neg P_4, P_7\}, \{\neg P_6, \neg P_7, P_8\}, \{\neg P_8\}.$$

$$\{\neg P_2, P_3\}, \{\neg P_3, P_4\}, \{\neg P_4, P_5\}, \{P_3\}, \{P_1\}, \{P_2\}, \{\neg P_1\}, \{\neg P_3, P_6\}, \{\neg P_3, P_7\}, \{\neg P_3, P_8\}.$$

9.3.3. Write a computer program implementing the linearization procedure.

9.4 SLD-Resolution in First-Order Logic

In this section we shall generalize SLD-resolution to first-order languages without equality. Fortunately, it is relatively painless to generalize results about

propositional SLD-resolution to the first-order case, using the lifting technique of Section 8.5.

9.4.1 Definition of SLD-Refutations

Since the main application of SLD-resolution is to PROLOG, we shall also revise our notation to conform to the PROLOG notation.

Definition 9.4.1 A *Horn clause* (in PROLOG notation) is one of the following expressions:

(i) $B : -A_1, ..., A_m$

(ii) B

(iii) $: -A_1, ..., A_m$

In the above, B, A_1,...,A_m are atomic formulae of the form $Pt_1...t_k$, where P is a predicate symbol of rank k, and t_1,...,t_k are terms.

A clause of the form (i) or (ii) is called a *definite clause*, and a clause of the form (iii) is called a *goal clause* (or *negative clause*).

The translation into the standard logic notation is the following:

The clause $B : -A_1, ..., A_m$ corresponds to the formula

$$(\neg A_1 \vee ... \vee \neg A_m \vee B);$$

The clause B corresponds to the atomic formula B;

The clause $: -A_1, ..., A_m$ corresponds to the formula

$$(\neg A_1 \vee ... \vee \neg A_m).$$

Actually, as in definition 8.2.1, it is assumed that a Horn clause is the universal closure of a formula as above (that is, of the form $\forall x_1...\forall x_n C$, where $FV(C) = \{x_1, ..., x_n\}$). The universal quantifiers are dropped for simplicity of notation, but it is important to remember that they are implicitly present.

The definition of SLD-derivations and SLD-refutations is extended by combining definition 9.3.2 and the definition of a resolvent given in definition 8.5.2.

Definition 9.4.2 Let S be a set of Horn clauses consisting of a set D of definite clauses and a set $\{G_1, ..., G_q\}$ of goals. An *SLD-derivation* for S is a sequence $< N_0, N_1, ..., N_p >$ of negative clauses satisfying the following properties:

(1) $N_0 = G_j$, where G_j is one of the goals;

(2) For every N_i in the sequence, $0 \leq i < p$, if

$$N_i =: -A_1, ..., A_{k-1}, A_k, A_{k+1}, ..., A_n,$$

then there is some definite clause $C_i = A : -B_1, ..., B_m$ in D such that A_k and A are unifiable, and for some most general unifier σ_i of A_k and $\rho_i(A)$, where (Id, ρ_i) is a separating substitution pair, if $m > 0$, then

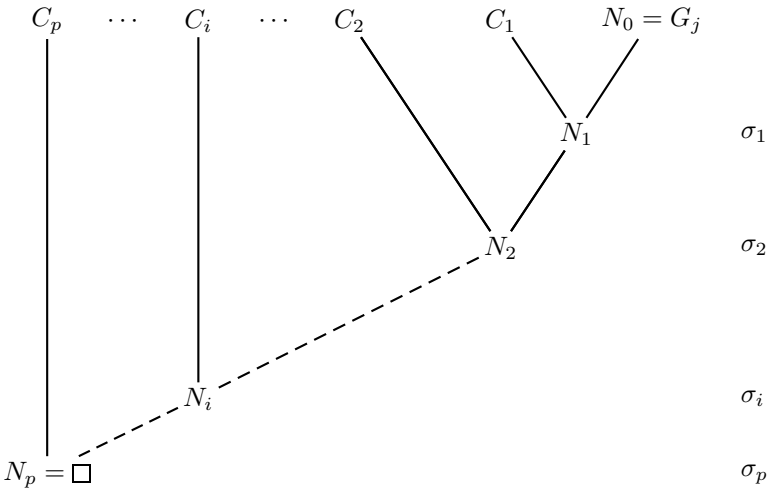
$$N_{i+1} =: -\sigma_i(A_1, ..., A_{k-1}, \rho_i(B_1), ..., \rho_i(B_m), A_{k+1}, ..., A_n)$$

else if $m = 0$ then

$$N_{i+1} =: -\sigma_i(A_1, ..., A_{k-1}, A_{k+1}, ..., A_n).$$

An SLD-derivation is an *SLD-refutation* iff $N_p = \square$.

Note that an SLD-derivation is a linear representation of a resolution DAG of the following special form:



At each step, the clauses

$$: -A_1, ..., A_{k-1}, A_k, A_{k+1}, ..., A_n$$

and

$$A : -B_1, ..., B_m$$

are resolved, the atoms A_k and $\rho_i(A)$ being canceled, since they are unified by the most general unifier σ_i. The literal A_k is called the *selected atom* of N_i, and the clauses N_0, C_1, ..., C_p are the *input clauses*.

When the derivation is a refutation, the substitution

$$\sigma = (\rho_1 \circ \sigma_1) \circ ... \circ (\rho_p \circ \sigma_p)$$

obtained by composing the substitutions occurring in the refutation is called the *result substitution* or *answer substitution*. It is used in PROLOG to extract the output of an SLD-computation.

Since an SLD-derivation is a special kind of resolution DAG, (a linear input resolution), its soundness is a consequence of lemma 8.5.2.

Lemma 9.4.1 (Soundness of SLD-resolution) If a set of Horn clauses has an SLD-refutation, then it is unsatisfiable.

Proof: Immediate from lemma 8.5.2. □

Let us give an example of an SLD-refutation in the first-order case.

EXAMPLE 9.4.1

Consider the following set of definite clauses, axiomatizing addition of natural numbers:

$$C_1 : add(X, 0, X).$$
$$C_2 : add(X, succ(Y), succ(Z)) : -add(X, Y, Z).$$

Consider the goal

$$B : -add(succ(0), V, succ(succ(0))).$$

We wish to show that the above set is unsatisfiable. We have the following SLD-refutation:

Goal clause	Input clause	Substitution
$: -add(succ(0), V, succ(succ(0)))$	C_2	
$: -add(succ(0), Y_2, succ(0))$	C_1	σ_1
□		σ_2

where

$$\sigma_1 = (succ(0)/X_1, succ(0)/Z_1, succ(Y_2)/V),$$
$$\sigma_2 = (succ(0)/X_2, 0/Y_2)$$

The variables X_1, Z_1, Y_2, X_2 were introduced by separating substitutions in computing resolvents. The result substitution is

$$(succ(0)/V, succ(0)/X_1, succ(0)/Z_1, succ(0)/X_2).$$

The interesting component is $succ(0)/V$. Indeed, there is a computational interpretation of the unsatisfiability of the set $\{C_1, C_2, B\}$. For this, it is necessary to write quantifiers explicitly and remember that goal clauses are negative. Observe that

$$\forall X C_1 \wedge \forall X \forall Y \forall Z C_2 \wedge \forall V B$$

is unsatisfiable, iff

$$\neg(\forall XC_1 \wedge \forall X\forall Y\forall ZC_2 \wedge \forall VB)$$

is valid, iff

$$(\forall XC_1 \wedge \forall X\forall Y\forall ZC_2) \supset \exists V\neg B$$

is valid. But $\exists V\neg B$ is actually

$$\exists V add(succ(0), V, succ(0)).$$

Since $(\forall XC_1 \wedge \forall X\forall Y\forall ZC_2)$ defines addition in the intuitive sense that any X, Y, Z satisfying the above sentence are such that $Z = X + Y$, we are trying to find some V such that $succ(0) + V = succ(succ(0))$, or in other words, compute the difference of $succ(succ(0))$ and $succ(0)$, which is indeed $succ(0)$!

This interpretation of a refutation showing that a set of Horn clauses is unsatisfiable as a computation of the answer to a query, such as

$$(\forall XC_1 \wedge \forall X\forall Y\forall ZC_2) \supset \exists V\neg B,$$

"find some V satisfying $\neg B$ and such that some conditional axioms $\forall XC_1$ and $\forall X\forall Y\forall ZC_2$ hold,"

is the essense of PROLOG. The set of clauses $\{C_1, C_2\}$ can be viewed as a *logic program*.

We will come back to the idea of refutations as computations in the next section.

9.4.2 Completeness of SLD-Resolution for Horn Clauses

The completeness of SLD-resolution for Horn clauses is shown in the following theorem.

Theorem 9.4.1 (Completeness of SLD-Resolution for Horn Clauses) Let **L** be any first-order language without equality. Given any finite set S of Horn clauses, if S is unsatisfiable, then there is an SLD-refutation with first clause some negative clause $: -B_1, ..., B_n$ in S.

Proof: We shall use the lifting technique provided by lemma 8.5.4. First, by the Skolem-Herbrand-Gödel theorem, if S is unsatisfiable, there is a set S_g of ground instances of clauses in S which is unsatisfiable. Since substitution instances of Horn clauses are Horn clauses, by theorem 9.3.1, there is an SLD-refutation for S_g, starting from some negative clause in S_g. Finally, we conclude by observing that if we apply the lifting technique of lemma 8.5.4, we obtain an SLD-refutation. This is because we always resolve a negative clause (N_i) against an input clause (C_i). Hence, the result is proved. □

From theorem 9.3.1, it is also true that if the first negative clause is $:-B_1, ..., B_n$, for every atom B_i in this goal, there is an SLD-resolution whose first selected atom is B_i. As a matter of fact, this property holds for any clause N_i in the refutation.

Even though SLD-resolution is complete for Horn clauses, there is still the problem of choosing among many possible SLD-derivations. The above shows that the choice of the selected atom is irrelevant. However, we still have the problem of choosing a definite clause $A :- B_1, ..., B_m$ such that A unifies with one of the atoms in the current goal clause $:- A_1, ..., A_{k-1}, A_k, A_{k+1}, ..., A_n$.

Such problems are important and are the object of current research in programming logic, but we do not have the space to address them here. The interested reader is referred to Kowalski, 1979, or Campbell, 1983, for an introduction to the methods and problems in programming logic.

In the next section, we discuss the use of SLD-resolution as a computation procedure for PROLOG.

PROBLEMS

9.4.1. Prove using SLD-resolution that the following set of clauses is unsatisfiable:

$$add(X, 0, X)$$
$$add(X, succ(Y), succ(Z)) :- add(X, Y, Z)$$
$$:- add(succ(succ(0)), succ(succ(0)), U).$$

9.4.2. Prove using SLD-resolution that the following set of clauses is unsatisfiable:

$$add(X, 0, X)$$
$$add(X, succ(Y), succ(Z)) :- add(X, Y, Z)$$
$$:- add(U, V, succ(succ(succ(0)))).$$

Find all possible SLD-refutations.

9.4.3. Using SLD-resolution, show that the following set of Horn clauses is unsatisfiable:

$$hanoi(N, Output) :- move(a, b, c, N, Output).$$
$$move(A, B, C, succ(M), Output) :- move(A, C, B, M, Out1),$$
$$\quad move(C, B, A, M, Out2),$$
$$\quad append(Out1, cons(to(A, B), Out2), Output).$$
$$move(A, B, C, 0, nil).$$
$$append(cons(A, L1), L2, cons(A, L3)) :- append(L1, L2, L3).$$
$$append(nil, L1, L1).$$
$$:- hanoi(succ(succ(0)), Z)$$

9.5 SLD-Resolution, Logic Programming (PROLOG)

We have seen in example 9.4.1 that an SLD-refutation for a set of Horn clauses can be viewed as a computation. This illustrates an extremely interesting use of logic as a *programming language*.

9.5.1 Refutations as Computations

In the past few years, Horn logic has been the basis of a new type of programming language due to Colmerauer named PROLOG. It is not the purpose of this book to give a complete treatment of PROLOG, and we refer the interested reader to Kowalski, 1979, or Clocksin and Mellish, 1981, for details. In this section, we shall lay the foundations of the programming logic PROLOG. It will be shown how SLD-resolution can be used as a computational procedure to solve certain problems, and the correctness and completeness of this approach will be proved.

In a logic programming language like PROLOG, one writes programs as sets of assertions in the form of Horn clauses, or more accurately, definite clauses, except for the goal. A set P of definite clauses is a *logic program*. As we said in Section 9.4, it is assumed that distinct Horn clauses are universally quantified.

Roughly speaking, a logic program consists of facts and assertions. Given such a logic program, one is usually interested in extracting facts that are consequences of the logic program P. Typically, one has a certain "query" (or goal) G containing some free variables z_1,...,z_q, and one wants to find term instances t_1, ..., t_q for the variables z_1, ..., z_q, such that the formula

$$P \supset G[t_1/z_1, ..., t_q/z_q]$$

is valid.

For simplicity, it will be assumed that the query is a positive atomic formula G. More complicated formulae can be handled (anti-Horn clauses), but we will consider this case later. In PROLOG, a goal statement G is denoted by $? - G$.

From a logical point of view, the problem is to determine whether the sentence

$$P \supset (\exists z_1 ... \exists z_q G)$$

is valid.

From a computational point of view, the problem is to find term values t_1,...,t_q for the variables z_1,...,z_q that make the formula

$$P \supset G[t_1/z_1, ..., t_q/z_q]$$

valid, and perhaps all such assignments.

Remarkably, SLD-resolution can be used not only as a proof procedure, but also a a computational procedure, because it returns a result substitution. The reason is as follows:

$$\begin{gathered}\text{The formula } P \supset (\exists z_1 ... \exists z_q G) \text{ is valid iff} \\ \neg(P \supset (\exists z_1 ... \exists z_q G)) \text{ is unsatisfiable iff} \\ P \wedge (\forall z_1 ... \forall z_q \neg G) \text{ is unsatisfiable.}\end{gathered}$$

But since G is an atomic formula, $\neg G$ is a goal clause : $-G$, and $P \wedge (\forall z_1 ... \forall z_q \neg G)$ is a conjuction of Horn clauses!

Hence, SLD-resolution can be used to test for unsatisfiability, and if it succeeds, it returns a result substitution σ. The crucial fact is that the components of the substitution σ corresponding to the variables z_1,...,z_q are answers to the query G. However, this fact is not obvious. A proof will be given in the next section. As a preliminary task, we give a rigorous definition of the semantics of a logic program.

9.5.2 Model-Theoretic Semantics of Logic Programs

We begin by defining what kind of formula can appear as a goal.

Definition 9.5.1 An *anti-Horn clause* is a formula of the form

$$\exists x_1 ... \exists x_m B,$$

where B is a conjunctions of literals $L_1 \wedge ... \wedge L_p$, with at most one negative literal and $FV(B) = \{x_1, ..., x_m\}$.

A *logic program* is a pair (P, G), where the *program* P is a set of (universal) Horn clauses, and the *query* G is a disjunction

$$(G_1 \vee ... \vee G_n)$$

of anti-Horn clauses $G_i = \exists y_1 ... \exists z_{m_i} B_i$.

It is also assumed that for all $i \neq j$, $1 \leq i, j \leq n$, the sets of variables $FV(B_i)$ and $FV(B_j)$ are disjoint. The union $\{z_1, ..., z_q\}$ of the sets of free variables occurring in each B_i is called the set of *output variables associated with* G.

Note that an anti-Horn clause is not a clause. However, the terminology is justified by the fact that the negation of an anti-Horn clause is a (universal) Horn clause, and that $\neg G$ is equivalent to a conjunction of universal Horn clauses.

Remark: This definition is more general than the usual definition used in PROLOG. In (standard) PROLOG, P is a set of definite clauses (that is,

P does not contain negative clauses), and G is a formula that is a conjunction of atomic formulae. It is shown in the sequel that more general queries can be handled, but that the semantics is a bit more subtle. Indeed, indefinite answers may arise.

EXAMPLE 9.5.1

The following is a logic program, where P consists of the following clauses:

$$rocksinger(jackson).$$
$$teacher(jean).$$
$$teacher(susan).$$
$$rich(X) : -rocksinger(X).$$
$$: -teacher(X), rich(X).$$

The query is the following disjunction:

$$? - \neg rocksinger(Y) \vee rich(Z)$$

EXAMPLE 9.5.2

The following is the program of a logic program:

$$hanoi(N, Output) : -move(a, b, c, N, Output).$$
$$move(A, B, C, succ(M), Output) : -move(A, C, B, M, Out1),$$
$$\quad move(C, B, A, M, Out2),$$
$$\quad append(Out1, cons(to(A, B), Out2), Output).$$
$$move(A, B, C, 0, nil).$$
$$append(cons(A, L1), L2, cons(A, L3)) : -append(L1, L2, L3).$$
$$append(nil, L1, L1).$$

The query is:

$$? - hanoi(succ(succ(succ(0))), Output).$$

The above program is a logical version of the well known problem known as the tower of Hanoi (see Clocksin and Mellish, 1981).

In order to give a rigorous definition of the semantics of a logic program, it is convenient to define the concept of a free structure. Recall that we are only dealing with first-order languages without equality, and that if the language has no constants, the special constant # is added to it.

Definition 9.5.2 Given a first-order language **L** without equality and with at least one constant, a *free structure* (or *Herbrand structure*) **H** is an **L**-structure with domain the set $H_{\mathbf{L}}$ of all closed **L**-terms, and whose interpretation function satisfies the following property:

(i) For every function symbol f of rank n, for all t_1,...,$t_n \in H_{\mathbf{L}}$,

$$f_{\mathbf{H}}(t_1, ..., t_n) = ft_1...t_n \quad \text{and}$$

(ii) For every constant symbol c,

$$c_{\mathbf{H}} = c.$$

The set of terms $H_{\mathbf{L}}$ is called the *Herbrand universe* of $\mathbf{L}$. For simplicity of notation, the set $H_{\mathbf{L}}$ is denoted as H when $\mathbf{L}$ is understood. The following lemma shows that free structures are universal. This lemma is actually not necessary for giving the semantics of Horn clauses, but it is of independent interest.

Lemma 9.5.1 A sentence X in NNF containing only universal quantifiers is satisfiable in some model iff it is satisfiable in some free structure.

Proof: Clearly, if X is satisfied in a free structure, it is satisfiable in some model. For the converse, assume that X has some model $\mathbf{A}$. We show how a free structure can be constructed from $\mathbf{A}$. We define the function $h : H \to A$ as follows:

For every constant c, $h(c) = c_{\mathbf{A}}$;

For every function symbol f of rank $n > 0$, for any n terms t_1,...,$t_n \in H$,

$$h(ft_1...t_n) = f_{\mathbf{A}}(h(t_1), ..., h(t_n)).$$

Define the interpretation of the free structure H such that, for any predicate symbol P of rank n, for any n terms t_1,...,$t_n \in H$,

$$\mathbf{H} \models P(t_1, ..., t_n) \quad \text{iff} \quad \mathbf{A} \models P(h(t_1), ..., h(t_n)). \qquad (*)$$

We now prove by induction on formulae that, for every assignment $s : \mathbf{V} \to H$, if $\mathbf{A} \models X[s \circ h]$, then $\mathbf{H} \models X[s]$.

(i) If X is a literal, this amounts to the definition $(*)$.

(ii) If X is of the form $(B \wedge C)$, then $\mathbf{A} \models X[s \circ h]$ implies that

$$\mathbf{A} \models B[s \circ h] \quad \text{and} \quad \mathbf{A} \models C[s \circ h].$$

By the induction hypothesis,

$$\mathbf{H} \models B[s] \quad \text{and} \quad \mathbf{H} \models C[s],$$

that is, $\mathbf{H} \models X[s]$.

(iii) If X is of the form $(B \vee C)$, then $\mathbf{A} \models X[s \circ h]$ implies that

$$\mathbf{A} \models B[s \circ h] \quad \text{or} \quad \mathbf{A} \models C[s \circ h].$$

By the induction hypothesis,

$$\mathbf{H} \models B[s] \quad \text{or} \quad \mathbf{H} \models C[s],$$

that is, $\mathbf{H} \models X[s]$.

(iv) X is of the form $\exists x B$. This case is not possible since X does not contain existential quantifiers.

(v) X is of the form $\forall x B$. If $\mathbf{A} \models X[s \circ h]$, then for every $a \in A$,

$$\mathbf{A} \models B[(s \circ h)[x := a]].$$

Now, since $h : H \to A$, for every $t \in H$, $h(t) = a$ for some $a \in A$, and so, for every t in H,

$$\mathbf{A} \models B[(s \circ h)[x := h(t)]], \text{ that is, } \mathbf{A} \models B[(s[x := t]) \circ h].$$

By the induction hypothesis, $\mathbf{H} \models B[s[x := t]]$ for all $t \in H$, that is, $\mathbf{H} \models X[s]$. □

It is obvious that lemma 9.5.1 also applies to sets of sentences. Also, since a formula is unsatisfiable iff it has no model, we have the following corollary:

Corollary Given a first-order language without equality and with some constant, a set of sentences in NNF and only containing universal quantifiers is unsatisfiable iff it is unsatisfiable in every free (Herbrand) structure. □

We now provide a rigorous semantics of logic programs.

Given a logic program (P, G), the question of interest is to determine whether the formula $P \supset G$ is valid. Actually, we really want more. If $\{z_1, ..., z_q\}$ is the the set of output variables occurring in G, we would like to find some (or all) tuple(s) $(t_1, ..., t_q)$ of ground terms such that

$$\models P \supset (B_1 \vee ... \vee B_n)[t_1/z_1, ..., t_q/z_q].$$

As we shall see, such tuples do not always exist. However, indefinite (or disjunctive) answers always exist, and if some conditions are imposed on P and G, definite answers (tuples of ground terms) exist.

Assume that $P \supset G$ is valid. This is equivalent to $\neg(P \supset G)$ being unsatisfiable. But $\neg(P \supset G)$ is equivalent to $P \wedge \neg G$, which is equivalent to a conjunction of universal Horn clauses. By the Skolem-Herbrand-Gödel

theorem (theorem 7.6.1), if $\{x_1, ..., x_m\}$ is the set of all universally quantified variables in $P \wedge \neg G$, there is *some set*

$$\{(t_1^1, ..., t_m^1), ..., (t_1^k, ..., t_m^k)\}$$

of m-tuples of ground terms such that the conjunction

$$(P \wedge \neg G)[t_1^1/x_1, ..., t_m^1/x_m] \wedge ... \wedge (P \wedge \neg G)[t_1^k/x_1, ..., t_m^k/x_m]$$

is unsatisfiable (for some $k \geq 1$). From this, it is not difficult to prove that

$$\models P \supset G[t_1^1/x_1, ..., t_m^1/x_m] \vee ... \vee G[t_1^k/x_1, ..., t_m^k/x_m].$$

However, we cannot claim that $k = 1$, as shown by the following example.

EXAMPLE 9.5.3

Let $P = \neg Q(a) \vee \neg Q(b)$, and $G = \exists x \neg Q(x)$. $P \supset G$ is valid, but there is no term t such that

$$\neg Q(a) \vee \neg Q(b) \supset \neg Q(t)$$

is valid.

As a consequence, the answer to a query may be indefinite, in the sense that it is a disjunction of substitution instances of the goal. However, definite answers can be ensured if certain restrictions are met.

Lemma 9.5.2 (Definite answer lemma) If P is a (finite) set of definite clauses and G is a query of the form

$$\exists z_1 ... \exists z_q (B_1 \wedge ... \wedge B_l),$$

where each B_i is an atomic formula, if

$$\models P \supset \exists z_1 ... \exists z_q (B_1 \wedge ... \wedge B_l),$$

then there is some tuple $(t_1, ..., t_q)$ of ground terms such that

$$\models P \supset (B_1 \wedge ... \wedge B_l)[t_1/z_1, ..., t_q/z_q].$$

Proof:

$$\models P \supset \exists z_1 ... z_q (B_1 \wedge ... \wedge B_l) \quad \text{iff}$$
$$P \wedge \forall z_1 ... \forall z_q (\neg B_1 \vee ... \vee \neg B_l) \quad \text{is unsatisfiable.}$$

By the Skolem-Herbrand-Gödel theorem, there is a set C of ground substitution instances of the clauses in $P \cup \{\neg B_1, ..., \neg B_l\}$ that is unsatisfiable. Since

the only negative clauses in C come from $\{\neg B_1, ..., \neg B_l\}$, by lemma 9.2.5, there is some substitution instance

$$(\neg B_1 \vee ... \vee \neg B_l)[t_1/z_1, ..., t_q/z_q]$$

such that

$$P' \cup \{(\neg B_1 \vee ... \vee \neg B_l)[t_1/z_1, ..., t_q/z_q]\}$$

is unsatisfiable, where P' is the subset of C consisting of substitution instances of clauses in P. But then, it is not difficult to show that

$$\models P \supset (B_1 \wedge ... \wedge B_l)[t_1/z_1, ..., t_q/z_q]. \qquad \square$$

The result of lemma 9.5.2 justifies the reason that in PROLOG only programs consisting of definite clauses and queries consisting of conjunctions of atomic formulae are considered. With such restrictions, definite answers are guaranteed. The above discussion leads to the following definition.

Definition 9.5.3 Given a logic program (P, G) with query $G = \exists z_1 ... \exists z_q B$ and with $B = (B_1 \vee ... \vee B_n)$, the *semantics* (or *meaning*) of (P, G) is the set

$$M(P,G) = \bigcup \{\{(t_1^1, ..., t_q^1), ..., (t_1^k, ..., t_q^k)\},\ k \geq 1,\ (t_1^k, ..., t_q^k) \in H^q \mid \models P \supset B[t_1^1/z_1, ..., t_q^1/z_q] \vee ... \vee B[t_1^k/z_1, ..., t_q^k/z_q]\}$$

of sets q-tuples of terms in the Herbrand universe H that make the formula

$$P \supset B[t_1^1/z_1, ..., t_q^1/z_q] \vee ... \vee B[t_1^k/z_1, ..., t_q^k/z_q]$$

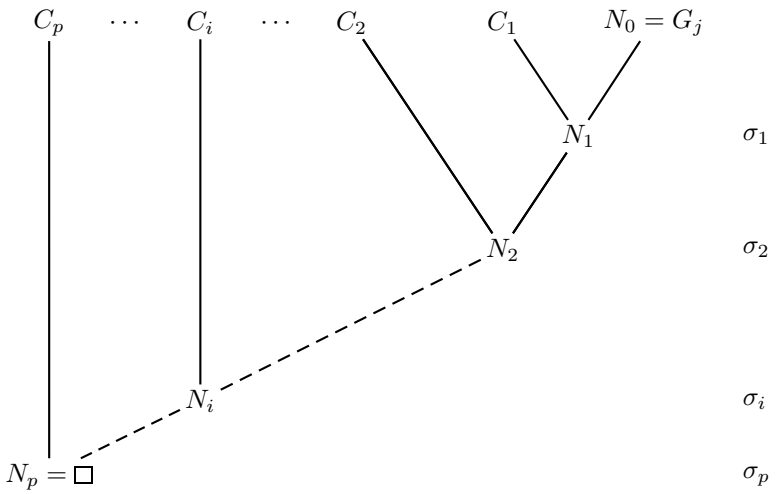
valid (in every free structure).

If P is a set of definite clauses and B is a conjunction of atomic formulae, $k = 1$.

9.5.3 Correctness of SLD-Resolution as a Computation Procedure

We now prove that for every SLD-refutation of the conjunction of clauses in $P \wedge \neg G$, the components of the result substitution σ restricted to the output variables belong to the semantics $M(P, G)$ of (P, G). We prove the following slightly more general lemma, which implies the fact mentioned above.

Lemma 9.5.3 Given a set P of Horn clauses, let $\mathcal{R}$ be an SLD-refutation



with result substitution σ (not necessarily ground). Let $\theta_p = \rho_p \circ \sigma_p$, and for every i, $1 \leq i \leq p-1$, let

$$\theta_i = (\rho_i \circ \sigma_i) \circ \theta_{i+1}.$$

(Note that $\sigma = \theta_1$, the result substitution.) The substitutions θ_i are also called *result substitutions*.) Then the set of quantifier-free clauses

$$\{\theta_1(N_0), \theta_1(C_1), ..., \theta_p(C_p)\}$$

is unsatisfiable (using the slight abuse of notation in which the matrix D of a clause $C = \forall x_1 ... \forall x_k D$ is also denoted by C).

Proof: We proceed by induction on the length of the derivation.

(i) If $p = 1$, N_0 must be a negative formula $: -B$ and C_1 a positive literal A such that A and B are unifiable, and it is clear that $\{\neg\theta_1(B), \theta_1(C_1)\}$ is unsatisfiable.

(ii) If $p > 1$, then by the induction hypothesis, taking N_1 as the goal of an SLD-refutation of length $p-1$ the set

$$\{\theta_2(N_1), \theta_2(C_2), ..., \theta_p(C_p)\}$$

is unsatisfiable. But N_0 is some goal clause

$$: -A_1, ..., A_{k-1}, A_k, A_{k+1}, ..., A_n,$$

and C_1 is some definite clause

$$A : -B_1, ..., B_m,$$

such that A and A_k are unifiable. Furthermore, the resolvent is given by

$$N_1 =: -\sigma_1(A_1, ..., A_{k-1}, \rho_1(B_1), ..., \rho_1(B_m), A_{k+1}, ..., A_n),$$

where σ_1 is a most general unifier, and we know that

$$\sigma_1(N_0) \wedge (\rho_1 \circ \sigma_1)(C_1) \supset N_1$$

is valid (by lemma 8.5.1). Since ρ_1 is a renaming substitution, it is the identity on N_0, and by the definition of θ_1, we have

$$\begin{aligned}\{\theta_2(\sigma_1(N_0)), \theta_2(\rho_1 \circ \sigma_1(C_1)), \theta_2(C_2), ..., \theta_p(C_p)\} \\ = \{\theta_1(N_0), \theta_1(C_1), \theta_2(C_2), ..., \theta_p(C_p)\}.\end{aligned}$$

If $\{\theta_1(N_0), \theta_1(C_1), ..., \theta_p(C_p)\}$ was satisfiable, since

$$\sigma_1(N_0) \wedge (\rho_1 \circ \sigma_1)(C_1) \supset N_1$$

is valid,

$$\{\theta_2(N_1), \theta_2(C_2), ..., \theta_p(C_p)\}$$

would also be satisfiable, a contradiction. Hence,

$$\{\theta_1(N_0), \theta_1(C_1), ..., \theta_p(C_p)\}$$

is unsatisfiable. □

Theorem 9.5.1 (Correctness of SLD-resolution as a computational procedure) Let (P, G) be a logic program with query $G = \exists z_1 ... \exists z_q B$, with $B = (B_1 \vee ... \vee B_n)$. For every SLD-refutation $\mathcal{R} =< N_0, N_1, ..., N_p >$ for the set of Horn clauses in $P \wedge \neg G$, if $\mathcal{R}$ uses (as in lemma 9.5.3) the list of definite clauses $< C_1, ..., C_p >$, the list of result substitutions (not necessarily ground) $< \theta_1, ..., \theta_p >$, and if $< \neg C_{i_1}, ..., \neg C_{i_k} >$ is the subsequence of $< N_0, C_1, ..., C_p >$ consisting of the clauses in $\{\neg B_1, ..., \neg B_n\}$ (with $\neg C_0 = N_0$), then

$$\models P \supset \theta_{i_1}(C_{i_1}) \vee ... \vee \theta_{i_k}(C_{i_k}).$$

Proof: Let P' be the set of formulae obtained by deleting the universal quantifiers from the clauses in P. By lemma 9.5.3, there is a sequence of clauses $< N_0, C_1, ..., C_p >$ from the set $P' \cup \{\neg B_1, ..., \neg B_n\}$ such that

$$\{\theta_1(N_0), \theta_1(C_1), ..., \theta_p(C_p)\}$$

is unsatisfiable. But then, it is easy to construct a proof of

$$P \supset \theta_{i_1}(C_{i_1}) \vee ... \vee \theta_{i_k}(C_{i_k})$$

(using $\forall : right$ rules as in lemma 8.5.4), and this yields the result. □

Note: The formulae C_{i_1},...,C_{i_k} are not necessarily distinct, but the substitutions θ_{i_1},...,θ_{i_k} might be.

Corollary Let (P, G) be a logic program such that P is a set of definite clauses and G is a formula of the form $\exists z_1 ... \exists z_q B$, where B is a conjunction of atomic formulae. For every SLD-refutation of the set of Horn clauses $P \wedge \neg G$, if σ is the result substitution and $(t_1/z_1, ..., t_q/z_q)$ is any ground substitution such that for every variable z_i in the support of σ, t_i is some ground instance of $\sigma(z_i)$ and otherwise t_i is any arbitrary term in H, then

$$\models P \supset B[t_1/z_1, ..., t_q/z_q].$$

Proof: First, observe that $\neg B$ must be the goal clause N_0. Also, if some output variable z_i does not occur in the support of the output substitution σ, this means that $\sigma(z_i) = z_i$. But then, it is immediate by lemma 9.5.3 that the result of substituting arbitrary terms in H for these variables in

$$\{\theta_1(N_0), \theta_1(C_1), ..., \theta_p(C_p)\}$$

is also unsatisfiable. □

Theorem 9.5.1 shows that SLD-resolution is a correct method for computing elements of $M(P, G)$, since every set $\{(t_1^1, ..., t_q^1), ..., (t_1^k, ..., t_q^k)\}$ of tuples of terms in H returned by an SLD-refutation (corresponding to the output variables) makes

$$P \supset B[t_1^1/z_1, ..., t_q^1/z_q] \vee ... \vee B[t_1^k/z_1, ..., t_q^k/z_q]$$

valid.

Remark: Normally, we are interested in tuples of terms in H, because we want the answers to be interpretable as *definite* elements of the Herbrand universe. However, by lemma 9.5.3, *indefinite answers* (sets of tuples of terms containing variables) have to be considered. This is illustrated in the next example.

EXAMPLE 9.5.4

Consider the logic program of example 9.5.1. The set of clauses corresponding to $P \wedge \neg G$ is the following:

$$\begin{aligned}
&rocksinger(jackson).\\
&teacher(jean).\\
&teacher(susan).\\
&rich(X) :- rocksinger(X).\\
&\ :- teacher(X), rich(X).\\
&rocksinger(Y).\\
&\ :- rich(Z)
\end{aligned}$$

Note the two negative clauses. There are four SLD-refutations, two with goal $: -teacher(X), rich(X)$, and two with goal $: -rich(Z)$.

(i) SLD-refutation with output $(jean/Y)$:

Goal clause	Input clause	Substitution
$: -teacher(X), rich(X)$	$teacher(jean)$	
$: -rich(jean)$	$rich(X) : -rocksinger(X)$	$(jean/X)$
$: -rocksinger(jean)$	$rocksinger(Y)$	$(jean/X_1)$
$\square$		$(jean/Y_1)$

The result substitution is $(jean/Y, jean/X)$. Also, Z is any element of the Herbrand universe.

(ii) SLD-refutation with output $(susan/Y)$: Similar to the above.

(iii) SLD-refutation with output $(jackson/Z)$:

Goal clause	Input clause	Substitution
$: -rich(Z)$	$rich(X) : -rocksinger(X)$	
$: -rocksinger(X_1)$	$rocksinger(jackson)$	(X_1/Z)
$\square$		$(jackson/X_1)$

Y is any element of the Herbrand universe.

(iv) SLD-refutation with output $(Y_1/Y, Y_1/Z)$:

Goal clause	Input clause	Substitution
$: -rich(Z)$	$rich(X) : -rocksinger(X)$	
$: -rocksinger(X_1)$	$rocksinger(Y)$	(X_1/Z)
$\square$		(Y_1/X_1)

In this last refutation, we have an indefinite answer that says that for any Y_1 in the Herbrand universe, $Y = Y_1$, $Z = Y_1$ is an answer. This is indeed correct, since the clause $rich(X) : -rocksinger(X)$ is equivalent to $\neg rocksinger(X) \vee rich(X)$, and so

$$\models P \supset (\neg rocksinger(Y_1) \vee rich(Y_1)).$$

We now turn to the completeness of SLD-resolution as a computation procedure.

9.5.4 Completeness of SLD-Resolution as a Computational Procedure

The correctness of SLD-resolution as a computational procedure brings up immediately the question of its completeness. For any set of tuples in $M(P,G)$, is there an SLD-refutation with that answer? This is indeed the case, as shown below. We state and prove the following theorem for the special case of definite clauses, leaving the general case as an exercise.

Theorem 9.5.2 Let (P,G) be a logic program such that P is a set of definite clauses and G is a goal of the form $\exists z_1 ... \exists z_q B$, where B is a conjunction $B_1 \wedge ... \wedge B_n$ of atomic formulae. For every tuple $(t_1,...,t_q) \in M(P,G)$, there is an SLD-refutation with result substitution σ and a (ground) substitution η such that the restriction of $\sigma \circ \eta$ to z_1,...,z_q is $(t_1/z_1,...,t_q/z_q)$.

Proof: By definition, $(t_1,...,t_q) \in M(P,G)$ iff

$$\begin{aligned} &\models P \supset (B_1 \wedge ... \wedge B_n)[t_1/z_1,...,t_q/z_q] \quad \text{iff} \\ &P \wedge (\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q] \quad \text{is unsatisfiable.} \end{aligned}$$

By theorem 9.5.1, there is an SLD-refutation with output substitution θ_1. Since

$$(\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q]$$

is the only negative clause, by lemma 9.5.3, for some sequence of clauses $< C_1,...,C_p >$ such that the universal closure of each clause C_i is in P,

$$\{\theta_1((\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q]), \theta_1(C_1),...,\theta_p(C_p)\}$$

is also unsatisfiable. If θ_1 is not a ground substitution, we can substitute ground terms for the variables and form other ground substitutions θ'_1,...,θ'_p such that,

$$\{\theta'_1((\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q]), \theta'_1(C_1),...,\theta'_p(C_p)\}$$

is still unsatisfiable. Since the terms t_1,...,t_q are ground terms,

$$\theta'_1((\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q]) = (\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q].$$

By theorem 9.3.1, there is a ground SLD-refutation $\mathcal{R}_g$ with sequence of input clauses

$$\begin{aligned} &< \{(\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q], C'_1,...,C'_r > \quad \text{for} \\ &\{(\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1,...,t_q/z_q], \theta'_1(C_1),...,\theta'_p(C_p)\}. \end{aligned}$$

By the lifting lemma (lemma 8.5.4), there is an SLD-refutation $\mathcal{R}$ with sequence of input clauses

$$\begin{aligned} &< \{\neg B_1,...,\neg B_n\}, C''_1,...,C''_r > \quad \text{for} \\ &\{\{\neg B_1,...,\neg B_n\}, C_1,...,C_p\}, \end{aligned}$$

such that for every pair of clauses N_i'' in $\mathcal{R}$ and N_i' in $\mathcal{R}_g$, $N_i' = \eta_i(N_i'')$, for some ground substitution η_i. Let $\eta = \eta_r$, and let σ be the result substitution of the SLD-refutation $\mathcal{R}$. It can be shown that

$$(\neg B_1 \vee ... \vee \neg B_n)[t_1/z_1, ..., t_q/z_q] = (\sigma \circ \eta)(\neg B_1 \vee ... \vee \neg B_n),$$

which shows that $(t_1/z_1, ..., t_q/z_q)$ is equal to the restriction of $\sigma \circ \eta$ to $z_1, ..., z_q$. □

9.5.5 Limitations of PROLOG

Theorem 9.5.1 and theorem 9.5.2 show that SLD-Resolution is a correct and complete procedure for computing the sets of tuples belonging to the meaning of a logic program. From a theoretical point of view, this is very satisfactory. However, from a practical point of view, there is still something missing. Indeed, we still need a procedure for producing SLD-refutations, and if possible, efficiently. It is possible to organize the set all SLD-refutations into a kind of tree (the search space), and the problem is then reduced to a tree traversal. If one wants to retain completeness, the kind of tree traversal chosen must be a breadth-first search, which can be very inefficient. Most implementations of PROLOG sacrifice completeness for efficiency, and adopt a depth-first traversal strategy.

Unfortunately, we do not have the space to consider these interesting issues, but we refer the interested reader to Kowalski, 1979, and to Apt and Van Emden, 1982, where the semantics of logic programming is investigated in terms of fixedpoints.

Another point worth noting is that not all first-order formulae (in Skolem form) can be expressed as Horn clauses. The main limitation is that negative premises are not allowed, in the sense that a formula of the form

$$B :- A_1, ..., A_{i-1}, \neg A, A_{i+1}, ..., A_n.$$

is not equivalent to any Horn clause (see problem 3.5.9).

This restriction can be somewhat overcome by the *negation by failure* strategy, but one has to be careful in defining the semantics of such programs (see Kowalski, 1979, or Apt and Van Emden, 1982).

PROBLEMS

9.5.1. (a) Give an SLD-resolution and the result substitution for the following set of clauses:

$French(Jean).$
$French(Jacques).$
$British(Peter).$
$likewine(X, Y) : -French(X), wine(Y).$
$likewine(X, Bordeaux) : -British(X).$
$wine(Burgundy).$
$wine(Bordeaux).$
$: -likewine(U, V).$

(b) Derive all possible answers to the query $likewine(U, V)$.

9.5.2. Give an SLD-resolution and the result substitution for the following set of clauses:

$append(cons(A, L1), L2, cons(A, L3)) : -append(L1, L2, L3).$
$append(nil, L1, L1).$
$: -append(cons(a, cons(b, nil)), cons(b, cons(c, nil)), Z)$

9.5.3. Give an SLD-resolution and the result substitution for the following set of clauses:

$hanoi(N, Output) : -move(a, b, c, N, Output).$
$move(A, B, C, succ(M), Output) : -move(A, C, B, M, Out1),$
$\quad move(C, B, A, M, Out2),$
$\quad append(Out1, cons(to(A, B), Out2), Output).$
$move(A, B, C, 0, nil).$
$append(cons(A, L1), L2, cons(A, L3)) : -append(L1, L2, L3).$
$append(nil, L1, L1).$
$: -hanoi(succ(succ(succ(0))), Z)$

9.5.4. Complete the proof of theorem 9.5.1 by filling in the missing details.

9.5.5. State and prove a generalization of theorem 9.5.2 for the case of arbitrary logic programs.

∗ **9.5.6** Given a set S of Horn clauses, an H-tree for S is a tree labeled with substitution instances of atomic formulae in S defined inductively as follows:

(i) A tree whose root is labeled with **F** (false), and having n immediate successors labeled with atomic formulae $B_1, ..., B_n$, where $: -B_1, ..., B_n$ is some goal clause in S, is an H-tree.

(ii) If T is an H-tree, for every leaf node u labeled with some atomic formulae X that is not a substitution instance of some atomic formula

B in S (a definite clause consisting of a single atomic formula), if X is unifiable with the lefthand side of any clause $A : -B_1, ..., B_k$ in S, if σ is a most general unifier of X and A, the tree T' obtained by applying the substitution σ to all nodes in T and adding the k ($k > 0$) immediate successors $\sigma(B_1)$,...,$\sigma(B_k)$ to the node u labeled with $\sigma(X) = \sigma(A)$ is an H-tree (if $k = 0$, the tree T becomes the tree T' obtained by applying the substitution σ to all nodes in T. In this case, $\sigma(X)$ is a substitution instance of an axiom.)

An H-tree for S is a proof tree iff all its leaves are labeled with substitution instances of axioms in S (definite clauses consisting of a single atomic formula).

Prove that S is unsatisfiable iff there is some H-tree for S which is a proof tree.

Hint: Use problem 9.2.4 and adapt the lifting lemma.

∗ **9.5.7** Complete the proof of theorem 9.5.2 by proving that the substitution $\varphi = (t_1/z_1, \ldots, t_q/z_q)$ is equal to the restriction of $\sigma \circ \eta$ to $z_1, \ldots, z_q$.

Hint: Let $\mathcal{R} =< N_0'', \ldots, N_r'' >$ be the nonground SLD-refutation obtained by lifting the ground SLD-refutation $\mathcal{R}_g =< N_0', \ldots, N_r' >$, and let $< \sigma_1'', \ldots, \sigma_r'' >$ be the sequence of unifiers associated with $\mathcal{R}$. Note that $\sigma = \sigma_1'' \circ \ldots \circ \sigma_r''$. Prove that there exists a sequence $< \eta_0, \ldots, \eta_r >$ of ground substitutions, such that, $\eta_0 = \varphi$, and for every i, $1 \leq i \leq r$, $\eta_{i-1} = \lambda_i \circ \eta_i$, where λ_i denotes the restriction of σ_i'' to the support of η_{i-1}. Conclude that $\varphi = \lambda_1 \circ \ldots \circ \lambda_r \circ \eta_r$.

Notes and Suggestions for Further Reading

The method of SLD-resolution is a special case of the SL-resolution of Kowalski and Kuehner (see Siekman and Wrightson, 1983), itself a derivative of Model Elimination (Loveland, 1978).

To the best of our knowledge, the method used in Sections 9.2 and 9.3 for proving the completeness of SLD-resolution for (propositional) Horn clauses by linearizing a Gentzen proof in SLD-form to an SLD-refutation is original.

For an introduction to logic as a problem-solving tool, the reader is referred to Kowalski, 1979, or Bundy, 1983. Issues about the implementation of PROLOG are discussed in Campbell, 1983. So far, there are only a few articles and texts on the semantic foundations of PROLOG, including Kowalski and Van Emden, 1976; Apt and Van Emden, 1982; and Lloyd, 1984. The results of Section 9.5 for disjunctive goals appear to be original.

Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)

ROBERT NIEUWENHUIS AND ALBERT OLIVERAS

Technical University of Catalonia, Barcelona, Spain

AND

CESARE TINELLI

The University of Iowa, Iowa City, Iowa

Abstract. We first introduce *Abstract DPLL*, a rule-based formulation of the Davis–Putnam–Logemann–Loveland (DPLL) procedure for propositional satisfiability. This abstract framework allows one to cleanly express practical DPLL algorithms and to formally reason about them in a simple way. Its properties, such as soundness, completeness or termination, immediately carry over to the modern DPLL implementations with features such as backjumping or clause learning.

We then extend the framework to Satisfiability Modulo background Theories (SMT) and use it to model several variants of the so-called *lazy approach* for SMT. In particular, we use it to introduce a few variants of a new, efficient and modular approach for SMT based on a general DPLL(X) engine, whose parameter X can be instantiated with a specialized solver $Solver_T$ for a given theory T, thus producing a DPLL(T) system. We describe the high-level design of DPLL(X) and its cooperation with $Solver_T$, discuss the role of *theory propagation*, and describe different DPLL(T) strategies for some theories arising in industrial applications.

Our extensive experimental evidence, summarized in this article, shows that DPLL(T) systems can significantly outperform the other state-of-the-art tools, frequently even in orders of magnitude, and have better scaling properties.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational logic*; *verification*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Deduction (e.g., natural, rule-based)*

General Terms: Theory, Verification

Additional Key Words and Phrases: SAT solvers, Satisfiability Modulo Theories

---

This work was partially supported by Spanish Ministry of Education and Science through the Logic Tools project TIN2004-03382 (all authors), the FPU grant AP2002-3533 (Oliveras) and National Science Foundation (NSF) grant 0237422 (Tinelli and Oliveras).

Authors' addresses: R. Nieuwenhuis and A. Oliveras, Technical University of Catalonia, Campus Nord—Edif. Omega, C. Jordi Girona, 1–3, 08034 Barcelona, Spain; C. Tinelli, University of Iowa, Department of Computer Science, 14 MacLean Hall, Iowa City, IA, 52242.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 0004-5411/06/1100-0937 $5.00

1. *Introduction*

The problem of deciding the satisfiability of propositional formulas (SAT) does not only lie at the heart of the most important open problem in complexity theory (P vs. NP), it is also at the basis of many practical applications in such areas as Electronic Design Automation, Verification, Artificial Intelligence, and Operations Research. Thanks to recent advances in SAT-solving technology, propositional solvers are becoming the tool of choice for attacking more and more practical problems.

Most state-of-the-art SAT solvers [Moskewicz et al. 2001; Goldberg and Novikov 2002; Eén and Sörensson 2003; Ryan 2004] today are based on different variations of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis and Putnam 1960; Davis et al. 1962]. Starting essentially with the work on the GRASP, SATO and Relsat systems [Marques-Silva and Sakallah 1999; Zhang 1997; Bayardo and Schrag 1997], the spectacular improvements in the performance of DPLL-based SAT solvers achieved in the last years are due to (i) better implementation techniques, such as the *two-watched literal* approach for unit propagation, and (ii) several conceptual enhancements on the original DPLL procedure, aimed at reducing the amount of explored search space, such as *backjumping* (a form of *non-chronological backtracking*), *conflict-driven lemma learning*, and *restarts*. These advances make it now possible to decide the satisfiability of industrial SAT problems with tens of thousands of variables and millions of clauses.

Because of their success, both the DPLL procedure and its enhancements have been adapted to handle satisfiability problems in more expressive logics than propositional logic. In particular, they have been used to build efficient algorithms for the *Satisfiability Modulo Theories (SMT)* problem: deciding the satisfiability of ground first-order formulas with respect to background theories such as the theory of equality, of the integer or real numbers, of arrays, and so on [Armando et al. 2000, 2004; Filliâtre et al. 2001; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Ganzinger et al. 2004; Bozzano et al. 2005]. SMT problems arise in many industrial applications, especially in formal verification (see Section 3 for examples). They may contain thousands of clauses like $p \vee \neg q \vee a = f(b - c) \vee g(g(b)) \neq c \vee a - c \leq 7$, with purely propositional atoms as well as atoms over (combined) theories, such as the theory of the integers, or of Equality with Uninterpreted Functions (EUF).

Altogether, many variants and extensions of the DPLL procedure exist today. They are typically described in the literature informally and with the aid of pseudocode fragments. Therefore, it has become difficult for the newcomer to understand the precise nature of all these procedures, and for the expert to formally reason about their properties.

The first main contribution of this article is to address these shortcomings by providing *Abstract DPLL*, a uniform, declarative framework for describing DPLL-based solvers, both for propositional satisfiability and for satisfiability modulo theories. The framework allows one to describe the essence of various prominent approaches and techniques in terms of simple transition rules and rule application strategies. By abstracting away heuristics and implementation issues, it facilitates the understanding of DPLL at a conceptual level as well as its correctness and termination. For DPLL-based SMT approaches, it moreover provides a clean formulation and a basis for comparison of the different approaches.

The second main contribution of this article is a new modular architecture for building SMT solvers in practice, called DPLL(T), and a careful study of *theory propagation*, a refinement of SMT methods that can have a crucial impact on their performance.

The architecture is based on a general DPLL(X) engine, whose parameter X can be instantiated with a specialized solver $Solver_T$ for a given theory T, thus producing a system DPLL(T). Such systems can be implemented extremely efficiently and have good scaling properties: our Barcelogic implementation of DPLL(T) won four divisions at the 2005 SMT Competition [Barrett et al. 2005] (for the other three existing divisions it had no $Solver_T$ yet). The insights provided by our Abstract DPLL framework were an important factor in the success of our DPLL(T) architecture and its Barcelogic implementation. For instance, the abstract framework helped us in understanding the interactions between the DPLL(X) engine and the solvers, especially concerning the different forms of theory propagation, as well as in defining a good interface between both.

Section 2 of this article presents the propositional version of Abstract DPLL. It models DPLL procedures by means of simple *transition systems*. While abstract and declarative in nature, these transition systems can explicitly model the salient conceptual features of state-of-the-art DPLL-based SAT solvers, thus bridging the gap between logic-based calculi for DPLL and actual implementations. Within the Abstract DPLL formalism, we discuss in a clean and uniform way properties such as soundness, completeness, and termination. These properties immediately carry over to modern DPLL implementations with features such as backjumping and learning.

For backjumping systems, for instance, we achieve this by modeling backjumping by a general rule that encompasses several backtracking strategies—including basic chronological backtracking—and explaining how different systems implement the rule. Similarly, we model learning by general rules that show how devices such as conflict graphs are just one possibility for computing new lemmas. We also provide a general and simple termination argument for DPLL procedures that does not depend on an exhaustive enumeration of truth assignments; instead, it relies on a notion of search progress neatly expressing that search advances with the deduction of new unit clauses—the higher up in the search tree the better—which is the very essence of backjumping.

In Section 3, we go beyond propositional satisfiability, and extend the framework to *Abstract DPLL Modulo Theories*. As in the purely propositional case, this again allows us to express—and formally reason about—a number of current DPLL-based techniques for SMT, such as the various variants of the so-called *lazy approach* [Armando et al. 2000, 2004; Filliâtre et al. 2001; Audemard et al. 2002; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Ball et al. 2004].

In Section 4, based on the Abstract DPLL Modulo Theories framework, we introduce our DPLL(T) approach for building SMT systems. We first describe two variants of DPLL(T), depending on whether theory propagation is done exhaustively or not. Once the DPLL(X) engine has been implemented, this approach becomes extremely flexible: a DPLL(T) system for a theory T is obtained by simply plugging in the corresponding theory solver $Solver_T$, which must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements. We discuss the design of DPLL(X) and describe how DPLL(X) and $Solver_T$ cooperate. We also show that practical T-solvers can be designed to include theory propagation in an efficient way. A nontrivial issue

is how to deal with conflict analysis and clause learning adequately in the context of theory propagation. Different options and possible problems for doing this are analyzed and discussed in detail in Section 5.

In Section 6, we discuss some experiments with our Barcelogic implementation of DPLL(T). The results show that it can significantly outperform the best state-of-the-art tools and, in addition, scales up very well.

This article consolidates and improves upon preliminary ideas and results presented at the JELIA [Tinelli 2002], LPAR [Nieuwenhuis and Oliveras 2003; Nieuwenhuis et al. 2005], and CAV [Ganzinger et al. 2004; Nieuwenhuis and Oliveras 2005a] conferences.

2. *Abstract DPLL in the Propositional Case*

We start this section with some formal preliminaries on propositional logic and on transition systems. Then we introduce several variants of Abstract DPLL and prove their correctness properties, showing at the same time how the different features of actual DPLL implementations are modeled by these variants.

2.1. Formulas, Assignments, and Satisfaction.

Let P be a fixed finite set of propositional symbols. If $p \in P$, then p is an *atom* and p and $\neg p$ are *literals* of P. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$. A *unit clause* is a clause consisting of a single literal. A (CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \cdots \wedge C_n$. When it leads to no ambiguities, we will sometimes also write such a formula in set notation $\{C_1, \ldots, C_n\}$, or simply replace the $\wedge$ connectives by commas.

A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. A literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. A literal is *defined* in M if it is either true or false in M. The assignment M is *total* over P if no literal of P is undefined in M. A clause C is true in M if at least one of its literals is in M. It is false in M if all its literals are false in M, and it is undefined in M otherwise. A formula F is true in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In that case, M is a *model* of F. If F has no models then it is *unsatisfiable*. If F and F' are formulas, we write $F \models F'$ if F' is true in all models of F. Then, we say that F' *is entailed by* F, or is a *logical consequence* of F. If $F \models F'$ and $F' \models F$, we say that F and F' are *logically equivalent*.

In what follows, (possibly subscripted or primed) lowercase l *always* denote literals. Similarly, C and D always denote clauses, F and G denote formulas, and M and N denote assignments. If C is a clause $l_1 \vee \cdots \vee l_n$, we sometimes write $\neg C$ to denote the formula $\neg l_1 \wedge \cdots \wedge \neg l_n$.

2.2. States and Transition Systems in Abstract DPLL.

DPLL can be fully described by simply considering that a *state* of the procedure is either the distinguished state *FailState* or a pair of the form $M \parallel F$, where F is a CNF formula, that is, a finite set of clauses, and M is, essentially, a (partial) assignment.

More precisely, M is a *sequence* of literals, never containing both a literal and its negation, where each literal has an *annotation*, a bit that marks it as a *decision* literal (see below) or not. Frequently, we will consider M just as a partial assignment, or as a set or conjunction of literals (and hence as a formula), ignoring both the annotations and the order between its elements.

The concatenation of two such sequences will be denoted by simple juxtaposition. When we want to emphasize that a literal l is annotated as a decision literal we will write it as l^{d}. We will denote the empty sequence of literals (or the empty assignment) by $\emptyset$. We say that a clause C is *conflicting* in a state $M \parallel F, C$ if $M \models \neg C$.

We will model each DPLL procedure by means of a set of states together with a binary relation $\Longrightarrow$ over these states, called the *transition relation*. As usual, we use infix notation, writing $S \Longrightarrow S'$ instead of $(S, S') \in \Longrightarrow$. If $S \Longrightarrow S'$ we say that there is a *transition* from S to S'. We denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$. We call any sequence of transitions of the form $S_0 \Longrightarrow S_1, \quad S_1 \Longrightarrow S_2, \cdots$ a *derivation*, and denote it by $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \cdots$. We call any subsequence of a derivation a *subderivation*.

In what follows, transition relations will be defined by means of conditional *transition rules*. For a given state S, a transition rule precisely defines whether there is a transition from S by this rule and, if so, to which state S'. Such a transition is called an *application step* of the rule.

A *transition system* is a set of transition rules defined over some given set of states. Given a transition system R, the transition relation defined by R will be denoted by $\Longrightarrow_R$. If there is no transition from S by $\Longrightarrow_R$, we will say that S is *final* with respect to R (examples of a transition system and a final state with respect to it can be found in Definition 2.1 and Example 2.2).

2.3. THE CLASSICAL DPLL PROCEDURE. A very simple DPLL system, faithful to the classical DPLL algorithm, consists of the following five transition rules. We give this system here mainly for explanatory and historical reasons. The informally stated results for it are easily obtained by adapting the more general ones given in Section 2.5.

Definition 2.1. The *Classical DPLL system* is the transition system $C\,l$ consisting of the following five transition rules. In this system, the literals added to M by all rules except Decide are annotated as non-decision literals.

UnitPropagate:

$$M \parallel F,\ C \vee l \quad \Longrightarrow \quad M\ l \parallel F,\ C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M. \end{cases}$$

PureLiteral:

$$M \parallel F \quad \Longrightarrow \quad M\ l \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M. \end{cases}$$

Decide:

$$M \parallel F \quad \Longrightarrow \quad M\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M. \end{cases}$$

Fail:

$$M \parallel F,\ C \quad \Longrightarrow \quad \mathit{FailState} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals.} \end{cases}$$

Backtrack:

$$M\ l^{\mathsf{d}}\ N \parallel F,\ C \quad \Longrightarrow \quad M\ \neg l \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C \\ N \text{ contains no decision literals.} \end{cases}$$

One can use the transition system Cl for deciding the satisfiability of an input formula F simply by generating an arbitrary derivation $\emptyset \parallel F \Longrightarrow_{\mathrm{Cl}} \cdots \Longrightarrow_{\mathrm{Cl}} S_n$, where S_n is a final state with respect to Cl. The applicability of each of the five rules is easy to check and, as we will see, their application always leads to finite derivations. Moreover, for every derivation like the above ending in a final state S_n, (i) F is unsatisfiable if, and only if, S_n is *FailState*, and (ii) if S_n is of the form $M \parallel F$ then M is a model of F. Note that in this Classical DPLL system the second component of a state remains unchanged, a property that does not hold for the other transition systems we introduce later.

We now briefly comment on what the different rules do. In the following, if M is a sequence of the form $M_0\, l_1\, M_1 \cdots l_k\, M_k$, where the l_i are all the decision literals in M, we say that the state $M \parallel F$ is *at decision level* k, and that all the literals of each $l_i\ M_i$ *belong to decision level* i.

—UnitPropagate: To satisfy a CNF formula, all its clauses have to be true. Hence, if a clause of F contains a literal l whose truth value is not defined by the current assignment M while all the remaining literals of the clause are false, then M must be extended to make l true.

—PureLiteral: If a literal l is *pure* in F, that is, it occurs in F while its negation does not, then F is satisfiable only if it has a model that makes l true. Thus, if M does not define l it can be extended to make l true.

—Decide: This rule represents a case split. An undefined literal l is chosen from F, and added to M. The literal is annotated as a *decision literal*, to denote that if $M\ l$ cannot be extended to a model of F then the alternative extension $M\ \neg l$ must still be considered. This is done by means of the Backtrack rule.

—Fail: This rule detects a *conflicting clause* C and produces the *FailState* state whenever M contains no decision literals.

—Backtrack: If a conflicting clause C is detected and Fail does not apply, then the rule backtracks one *decision level*, by replacing the most recent decision literal l^{d} by $\neg l$ and removing any subsequent literals in the current assignment. Note that $\neg l$ is annotated as a nondecision literal, since the other possibility l has already been explored.

Example 2.2. The following is a derivation in the Classical DPLL system, with each transition annotated by the rule that makes it possible. To improve readability we denote atoms by natural numbers, and negation by overlining.

$$
\begin{array}{rcl}
\emptyset & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}}\ \overline{2} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}}\ \overline{2}\ 3 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}}\ \overline{2}\ 3\ 4 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1}\ 4 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1}\ 4\ \overline{3}^{\mathsf{d}} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1}\ 4\ \overline{3}^{\mathsf{d}}\ 2 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4
\end{array}
\qquad
\begin{array}{l}
\Longrightarrow_{\mathrm{Cl}} \text{(Decide)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(Backtrack)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(Decide)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\phantom{\Longrightarrow_{\mathrm{Cl}}}
\end{array}
$$

The last state of this derivation is final. The (total) assignment in it is a model of the formula.

The Davis–Putnam procedure [Davis and Putnam 1960] was originally presented as a two-phase proof-procedure for first-order logic. The unsatisfiability of a formula was to be proved by first generating a suitable set of ground instances which then, in the second phase, were shown to be propositionally unsatisfiable.

Subsequent improvements, such as the Davis-Logemann-Loveland procedure of Davis et al. [1962], mostly focused on the propositional phase. What most authors now call the *DPLL Procedure* is a satisfiability procedure for propositional logic based on this propositional phase. Originally, this procedure amounted to the depth-first search algorithm with backtracking modeled by our Classical DPLL system.

2.4. MODERN DPLL PROCEDURES. The major modern DPLL-based SAT solvers do not implement the Classical DPLL system. For example, due to efficiency reasons the pure literal rule is normally only used as a preprocessing step—hence, we will not consider this rule in the following. Moreover, *backjumping*, a more general and more powerful backtracking mechanism, is now commonly used in place of chronological backtracking.

The usefulness of a more sophisticated backtracking mechanism for DPLL solvers is perhaps best illustrated with another example of derivation in the Classical DPLL system.

Example 2.3.

$$
\begin{aligned}
\emptyset \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Decide)}\\
1^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2 \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Decide)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4 \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Decide)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Backtrack)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2}
\end{aligned}
$$

Before the Backtrack step, the clause $6 \vee \overline{5} \vee \overline{2}$ is conflicting: it is false in the assignment $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6}$. This is a consequence of the unit propagation 2 of the decision 1^{d}, together with the decision 5^{d} and its unit propagation $\overline{6}$.

Therefore, one can infer that the decision 1^{d} is incompatible with the decision 5^{d}, that is, that the given clause set entails $\overline{1}\vee\overline{5}$. Similarly, it also entails $\overline{2}\vee\overline{5}$.

Such entailed clauses are called *backjump clauses* if their presence would have allowed a unit propagation at an earlier decision level. This is precisely what *backjumping* does: given a backjump clause, it goes back to that level and adds the unit propagated literal. For example, using $\overline{2}\vee\overline{5}$ as a backjump clause, the last Backtrack step could be replaced by a backjump to a state with first component $1^{\mathsf{d}}\ 2\ \overline{5}$.

We model all this in the next system with the Backjump rule, of which Backtrack is a particular case. In this rule, the clause $C' \vee l'$ is the backjump clause, where l'

is the literal that can be unit propagated ($\overline{5}$ in our example). Below we show that the rule is effective: a backjump clause can always be found.

Definition 2.4. The *Basic DPLL system* is the four-rule transition system B consisting of the rules UnitPropagate, Decide, Fail from Classical DPLL, and the following Backjump rule:

Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F,\ C \implies M\ l' \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N. \end{cases}$$

We call clause C in Backjump the *conflicting* clause and clause $C' \vee l'$ the *backjump* clause.

Chronological backtracking, modeled by Backtrack, always undoes the *last* decision l, going back to the previous level and adding $\neg l$ to it. *Conflict-driven* backjumping, as modeled by Backjump, is generally able to backtrack further than chronological backtracking by analyzing the reasons that produced the conflicting clause. Backjump can frequently undo *several* decisions at once, going back to a lower decision level than the previous level and adding some new literal to that lower level. It jumps over levels that are irrelevant to the conflict. In the previous example, it jumps over the decision 3^{d} and its consequence 4, which are totally unrelated with the reasons for the falsity of the conflicting clause $6 \vee \overline{5} \vee \overline{2}$. Moreover, intuitively, the search state $1^{\mathsf{d}}\ 2\ \overline{5}$ reached after Backjump is more *advanced* than the state $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5}$ reached after Backtrack. This notion of "being more advanced" is formalized in Theorem 2.10 below.

We show in the proof of Lemma 2.8 below that the literals of the backjump clause can always be chosen among the negations of the decision literals—although better choices usually exist. When the negations of *all* the decision literals are included in the backjump clause, the Backjump rule simulates the Backtrack rule of Classical DPLL. We remark that, in fact, Lemma 2.8 shows that, whenever a state $M \parallel F$ contains a conflicting clause, either Fail applies, if there are no decision literals in M, or otherwise Backjump applies.

Most modern DPLL implementations make additional use of backjump clauses: they add them to the clause set as *learned* clauses, also called *lemmas*, implementing what is usually called *conflict-driven learning*.

In Example 2.3, learning the clause $\overline{2} \vee \overline{5}$ will allow the application of UnitPropagate to any state whose assignment contains either 2 or 5. Hence, it will prevent any conflict caused by having both 2 and 5 in M. Reaching such *similar* conflicts frequently happens in industrial problems having some regular structure, and learning such lemmas has been shown to be very effective in improving performance.

Since a lemma is aimed at preventing future similar conflicts, when these conflicts are not very likely to be found again the lemma can be removed. In practice, a lemma is removed when its *relevance* (see, e.g., Bayardo and Schrag [1997]) or its *activity* level drops below a certain threshold; the activity can be, for example, the

number of times it becomes a unit or a conflicting clause [Goldberg and Novikov 2002].

To model lemma learning and removal we consider the following extension of the Basic DPLL system.

Definition 2.5. The *DPLL system with learning*, denoted by L, consists of the four transition rules of the Basic DPLL system and the two additional rules:

Learn:

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models C. \end{cases}$$

Forget:

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \qquad \textbf{if} \begin{cases} F \models C. \end{cases}$$

In any application step of Learn, the clause C is said to be *learned* if it did not already belong to F. Similarly, it is said to be *forgotten* by Forget.

Observe that the Learn rule allows one to add to the current formula F an arbitrary clause C entailed by F, as long as all the atoms of C occur in F or M. This models not only conflict-driven lemma learning but also any other techniques that produce consequences of F, such as limited forms of resolution (see the following example).

Similarly, the Forget rule can be used in principle to remove from F *any* clause that is entailed by the rest of F, not just those previously added to the clause set by Learn. The applicability of the two rules in their full scope, however, is limited in practice by the relative cost of determining such entailments in general.

The six rules of the DPLL system with learning model the high-level conceptual structure of DPLL implementations. These rules will allow us to formally reason about properties such as correctness or termination.

Example 2.6. We now show how the Backjump rule can be guided by means of a *conflict graph* for finding backjump clauses. In this example we assume a strategy that is followed in most SAT solvers: (i) Decide is applied only if no other Basic DPLL rule is applicable (Theorem 5.2 of Section 5 shows that this is not needed, but here we require it for simplicity) and (ii) after each application of Backjump, the backjump clause is learned.

Consider a state of the form $M \parallel F$ where, among other clauses, F contains:

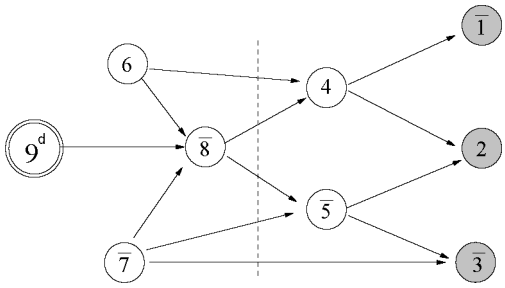
$$\overline{9} \vee \overline{6} \vee 7 \vee \overline{8} \quad 8 \vee 7 \vee \overline{5} \quad \overline{6} \vee 8 \vee 4 \quad \overline{4} \vee \overline{1} \quad \overline{4} \vee 5 \vee 2 \quad 5 \vee 7 \vee \overline{3} \quad 1 \vee \overline{2} \vee 3$$

and M is of the form: $\ldots 6 \ldots \overline{7} \ldots 9^{\mathsf{d}}\ \overline{8}\ \overline{5}\ 4\ \overline{1}\ 2\ \overline{3}$.

It is easy to see that this state can be reached after the last decision 9^{d} by six applications of UnitPropagate. For example, $\overline{8}$ is implied by 9, 6, and $\overline{7}$ because of the clause $\overline{9} \vee \overline{6} \vee 7 \vee \overline{8}$. A typical DPLL implementation will save the sequence of propagated literals and remember for each one of them the clause that caused its propagation. Now, in the state $M \parallel F$ above the clause $1 \vee \overline{2} \vee 3$ is conflicting, since M contains $\overline{1}$, 2 and $\overline{3}$. Using the saved information, the DPLL implementation can trace back the reasons for this conflicting clause. For example, the saved data will show that $\overline{3}$ was implied by $\overline{5}$ and $\overline{7}$, due to the clause $5 \vee 7 \vee \overline{3}$. The literal $\overline{5}$ was in turn implied by $\overline{8}$ and $\overline{7}$, and so on.

This way, *working backwards from the conflicting clause* and in the opposite order in which each literal was propagated, it is possible to build the following

conflict graph, where the nodes corresponding to the conflicting clause are shown in gray:



This figure shows the graph obtained when the decision literal of the current decision level (here, 9^{d}) is reached in this backwards process—which is why this node and the nodes belonging to earlier decision levels (in this example, literals 6 and $\overline{7}$) have no incoming arrows.

To find a backjump clause, it suffices to cut the graph into two parts. The first part must contain (at least) all the literals with no incoming arrows. The second part must contain (at least) all the literals with no outgoing arrows, that is, the negated literals of the conflicting clause (in our example, $\overline{1}$, 2 and $\overline{3}$). It is not hard to see that in such a cut no model of F can satisfy all the literals whose outgoing edges are cut.

For instance, consider the cut indicated by the dotted line in the graph, where the literals with cut outgoing edges are $\overline{8}$, $\overline{7}$, and 6. From these three literals, by unit propagation using five clauses of F, one can infer the negated literals of the conflicting clause. Hence, one can infer from F that $\overline{8}$, $\overline{7}$, and 6 cannot be simultaneously true, that is, one can infer the clause $8 \vee 7 \vee \overline{6}$. In this case, this is a possible backjump clause, that is, the clause $C' \vee l'$ in the definition of the Backjump rule, with the literal 8 playing the role of l'. The clause allows one to backjump to the decision level of $\overline{7}$ and add 8 to it. After that, the clause $8 \vee 7 \vee \overline{6}$ has to be learned to explain in future conflicts the presence of 8 as a propagation from 6 and $\overline{7}$.

The kind of cuts we have described produce backjump clauses provided that exactly one of the literals with cut outgoing edges belongs to the current decision level. The negation of this literal will act as the literal l' in the backjump rule. In the SAT literature, the literal is called a *Unique Implication Point (UIP)* of the conflict graph. Formally, UIPs are defined as follows. Let D be the set of all the literals of a conflicting clause C that have become false at the current decision level (this set is always nonempty, since Decide is applied only if Fail or Backjump do not apply). A UIP in the conflict graph of C is any literal that belongs to all paths in the graph from the current decision literal to the negation of a literal in D. Note that a conflict graph always contains at least one UIP, the decision literal itself, but in general it can contain more (in our example 9^{d} and $\overline{8}$ are both UIPs).

In practice, it is not actually necessary to build the conflict graph to produce a backjump clause; it suffices to work backwards from the conflicting clause, maintaining only a *frontier* list of literals yet to be expanded, until the *first* UIP (first in the reverse propagation ordering) has been reached [Marques-Silva and Sakallah 1999; Zhang et al. 2001].

The construction of the backjump clause can also be seen as a derivation in the resolution calculus, constructed according to the following *backwards conflict resolution* process. In our example, the clause $8 \vee 7 \vee \overline{6}$ is obtained by successive resolution steps on the conflicting clause, resolving away the literals $3, \overline{2}, 1, \overline{4}$ and 5, in the reverse order their negations were propagated, with the respective clauses that caused the propagations:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{5 \vee 7 \vee \overline{\mathbf{3}} \qquad 1 \vee \overline{2} \vee \mathbf{3}}{\overline{4} \vee 5 \vee \mathbf{2}} \qquad 5 \vee 7 \vee 1 \vee \overline{\mathbf{2}}}{\overline{4} \vee \overline{\mathbf{1}}} \qquad \overline{4} \vee 5 \vee 7 \vee \mathbf{1}}{\overline{6} \vee 8 \vee \mathbf{4}} \qquad 5 \vee 7 \vee \overline{\mathbf{4}}}{8 \vee 7 \vee \overline{\mathbf{5}}} \qquad \overline{6} \vee 8 \vee 7 \vee \mathbf{5}}{8 \vee 7 \vee \overline{6}}$$

The process stops once it generates a clause with only one literal of the current decision level, which is precisely the first UIP (in our example, the literal 8 in the clause $8 \vee 7 \vee \overline{6}$). Some SAT solvers, such as Siege, also learn some of the intermediate clauses in such resolution derivations [Ryan 2004].

2.5. CORRECTNESS OF DPLL WITH LEARNING. In this section, we show how the DPLL system with learning can be used as a decision procedure for the satisfiability of CNF formulas.

Deciding the satisfiability of an input formula F will be done by generating an arbitrary derivation of the form $\emptyset \parallel F \Longrightarrow_{\mathrm{L}} \cdots \Longrightarrow_{\mathrm{L}} S_n$ such that S_n is *final with respect to the Basic DPLL system*. Note that final states with respect to the DPLL system with Learning do not always exist, since the same clause could be learned and forgotten infinitely many times.

For all rules their applicability is easy to check and, as we will show in Theorem 2.11, if infinite subderivations with only Learn and Forget steps are avoided, one always reaches a state that is final with respect to the Basic DPLL system. This state S is moreover easily recognizable as final, because it is either *FailState* or of form $M \parallel F'$ where F' has no conflicting clauses and all of its literals are defined in M. Furthermore, similarly to the Classical DPLL system and as proved below in Theorem 2.12; in the first case, F is unsatisfiable, in the second case, it is satisfied by M.

We emphasize that these formal results apply to *any* procedure modeled by the DPLL system with learning, and can moreover be extended to DPLL Modulo Theories. This generalizes the less formal correctness proof for the concrete pseudo code of the Chaff algorithm given in Zhang and Malik [2003], which has the same underlying proof idea.

The starting point for our results is the next lemma, which lists a few properties that are invariant for all the states derived in the DPLL system with learning from initial states of the form $\emptyset \parallel F$.

LEMMA 2.7. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} M \parallel G$, *then the following hold.*

(1) *All the atoms in M and all the atoms in G are atoms of F.*
(2) *M contains no literal more than once and is indeed an assignment, that is, it contains no pair of literals of the form p and $\neg p$.*
(3) *G is logically equivalent to F.*

(4) *If M is of the form $M_0\ l_1\ M_1 \cdots l_n\ M_n$, where $l_1, \ldots, l_n$ are all the decision literals of M, then $F, l_1, \ldots, l_i \models M_i$ for all i in $0 \ldots n$.*

PROOF. Since all four properties trivially hold in the initial state $\emptyset \parallel F$, we only need to prove that all six rules preserve them. Consider a step $M' \parallel F' \Longrightarrow_{\mathrm{L}} M'' \parallel F''$ and assume all properties hold in $M' \parallel F'$. Property 1 holds in $M'' \parallel F''$ because the only atoms that may be added to M'' or F'' are the ones in F' or M', all of which belong to F. The side conditions of the rules clearly preserve Property 2. As for Property 3, only Learn and Forget may break the invariant. But learning (or forgetting) a clause C that is a logical consequence clearly preserves equivalence between F' and F''.

For the fourth property, consider that M' is of the form $M'_0\ l_1\ M'_1 \cdots l_n\ M'_n$, and $l_1, \ldots, l_n$ are all the decision literals of M'. If the step is an application of Decide, there is nothing to prove. For Learn or Forget, it easily follows since M' is M'' and F'' is logically equivalent to F'. The remaining rules are:

UnitPropagate: Since M'' will be of the form $M'l$ (we use l and C as in the definition of the rule), we only have to prove that $F, l_1, \ldots, l_n \models l$, which holds since (i) $F, l_1, \ldots, l_n \models M'$, (ii) $M' \models \neg C$, (iii) $C \vee l$ is a clause of F' and (iv) F and F' are equivalent.

Backjump: Assume that, in the Backjump rule, l^{d} is l_{j+1}, the $j+1$-th decision literal. Then (using l' and C' as in the definition of the rule), M'' is of the form $M'_0\ l_1\ M'_1 \cdots l_j\ M'_j\ l'$. We only need to show that $F, l_1, \ldots, l_j \models l'$. This holds as for the UnitPropagate case, since we have (i) $F, l_1, \ldots, l_j \models M'_0\ l_1\ M'_1 \cdots l_j\ M'_j$, (ii) $M'_0\ l_1\ M'_1 \cdots l_j\ M'_j \models \neg C'$, (iii) $F' \models C' \vee l'$ and (iv) F and F' are equivalent. □

The most interesting property of this lemma is probably Property 4. It shows that every nondecision literal added to an assignment M is a logical consequence of the previous decision literals of M and the initial formula F. In other words, we have that $F, l_1, \ldots, l_n \models M$. Hence, the only arbitrary additions to M are the ones made by Decide.

Another important property concerns the applicability of Backjump. Given a state with a conflicting clause, it may not be clear a priori whether Backjump is applicable or not, mainly due to the need to find an appropriate backjump clause. Below we show that, if there is a conflicting clause, it is always the case that either Backjump or Fail applies. Moreover, whenever the first precondition of Backjump holds ($M\ l^{\mathsf{d}}\ N \models \neg C$), a backjump clause $C' \vee l'$ always exists and can be easily computed.

LEMMA 2.8. *Assume that $\emptyset \parallel F \Longrightarrow^*_{\mathrm{L}} M \parallel F'$ and that $M \models \neg C$ for some clause C in F'. Then either Fail or Backjump applies to $M \parallel F'$.*

PROOF. If there is no decision literal in M, it is immediate that Fail applies. Otherwise, M is of the form $M_0\ l_1\ M_1 \cdots l_n\ M_n$ for some $n > 0$, where $l_1, \ldots, l_n$ are all the decision literals of M. Since $M \models \neg C$ we have, due to Lemma 2.7-4, that $F, l_1, \ldots, l_n \models \neg C$. If we now consider any i in $1 \cdots n$ such that $F, l_1, \ldots, l_i \models \neg C$, and any j in $0 \cdots i-1$ such that $F, l_1, \ldots, l_j, l_i \models \neg C$, we can show that then backjumping to decision level j is possible.

Let C' be the clause $\neg l_1 \vee \cdots \vee \neg l_j$, and note that M is also of the form $M'\ l_{j+1}\ N$. Then Backjump is applicable to $M \parallel F'$, yielding the state $M'\ \neg l_i \parallel F'$. That is because the clause $C' \vee \neg l_i$ satisfies all the side conditions of the Backjump rule:

(i) $F' \models C' \vee \neg l_i$ because $F, l_1, \ldots, l_j, l_i \models \neg C$, which implies, given that C is in F' and F' is equivalent to F (by Lemma 2.7-3), that $F, l_1, \ldots, l_j, l_i$ is unsatisfiable or, equivalently, that $F \models \neg l_1 \vee \cdots \vee \neg l_j \vee \neg l_i$; furthermore, $M' \models \neg C'$ by construction of C';

(ii) $\neg l_i$ is undefined in M' (by Lemma 2.7-2);

(iii) l_i occurs in M. □

It is interesting to observe that, the smaller one can choose the value j in the previous proof, the higher one can backjump. Note also that, if we construct the backjump clause as in the proof and take i to be n and j to be $n-1$ then the Backjump rule models standard backtracking.

We stress that backjump clauses need not be built as in the proof above, out of the decision literals of the current assignment. It follows from the termination and correctness results given in this section that in practice one is free to apply the backjump rule with *any* backjump clause. In fact, backjump clauses may be built to contain no decision literals at all, as is for instance possible in backjumping SAT solvers relying on the first UIP learning scheme illustrated in Example 2.6.

Given the previous lemma, it is easy to prove that final states with respect to Basic DPLL will be either *FailState* or $M \parallel F'$, where M is a model of the original formula F. More formally:

LEMMA 2.9. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} S$, *and S is final with respect to Basic DPLL, then S is either FailState, or it is of the form* $M \parallel F'$, *where*

(1) *all literals of* F' *are defined in* M,

(2) *there is no clause* C *in* F' *such that* $M \models \neg C$, *and*

(3) M *is a model of* F.

PROOF. Assume S is not *FailState*. If (1) does not hold, then S cannot be final, since Decide would be applicable. Similarly, for (2): by Lemma 2.8, either Fail or Backjump would apply. Together (1) and (2) imply that all clauses of F' are defined and true in M, and since by Lemma 2.7(3), F and F' are logically equivalent this implies that M is a model of F. □

We now prove termination of the Basic DPLL system.

THEOREM 2.10. *There are no infinite derivations of the form* $\emptyset \parallel F \Longrightarrow_{\mathrm{B}} S_1 \Longrightarrow_{\mathrm{B}} \cdots$.

PROOF. It suffices to define a well-founded strict partial ordering $\succ$ on states, and show that each step $M \parallel F \Longrightarrow_{\mathrm{B}} M' \parallel F$ is decreasing with respect to this ordering, that is, $M \parallel F \succ M' \parallel F$. Note that such an ordering must be entirely based on the first component of the states, because in this system without Learn and Forget the second component of states remains constant.

Let M be of the form $M_0\ l_1\ M_1 \cdots l_p\ M_p$, where $l_1, \ldots, l_p$ are all the decision literals of M. Similarly, let M' be $M'_0\ l'_1\ M'_1 \cdots l'_{p'}\ M'_{p'}$.

Let n be the number of distinct atoms (propositional variables) in F. By Lemma 2.7(1,2), we have that p, p' and the length of M and M' are always smaller than or equal to n.

For each assignment N, define $m(N)$ to be $n - \mathit{length}(N)$, that is, $m(N)$ is the number of literals "missing" in N for N to be total. Now define: $M \parallel F' \succ M' \parallel F''$ if

(i) there is some i with $0 \leq i \leq p$, p' such that

$$m(M_0) = m(M'_0), \ldots, m(M_{i-1}) = m(M'_{i-1}), \quad m(M_i) > m(M'_i) \text{ or}$$

(ii) $m(M_0) = m(M'_0), \ldots, m(M_p) = m(M'_p)$ and $m(M) > m(M')$.

Note that, in case (ii), we have $p' > p$, and all decision levels up to p coincide in number of literals. Comparing the number of missing literals in sequences is clearly a strict ordering (i.e., it is an irreflexive and transitive relation) and it is also well-founded, and hence this also holds for its lexicographic extension on tuples of sequences of bounded length. It is easy to see that all Basic DPLL rules are decreasing with respect to $\succ$ if *FailState* is added as an additional minimal element. The rules UnitPropagate and Backjump decrease by case (i) of the definition and Decide decreases by case (ii). □

It is nice to see in this proof that, in contrast to the classical, depth-first DPLL procedure, progress in backjumping DPLL procedures is not measured by the number of decision literals that have been *tried* with both truth values, but by the number of defined literals that are added to earlier decision levels. The Backjump rule makes progress in this sense by increasing by one the number of defined literals in the decision level it backjumps to. The lower this decision level is (i.e., the higher up in the depth-first search tree), the more progress is made with respect to $\succ$.

As an immediate consequence of this theorem, we obtain the termination of the DPLL system with learning if infinite subderivations with only Learn and Forget steps are avoided. The reason is that the other steps (the Basic DPLL ones) decrease the first components of the states with respect to the well-founded ordering, while the Learn and Forget steps do not modify that component.

THEOREM 2.11. *Every derivation* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}} S_1 \Longrightarrow_{\mathrm{L}} \cdots$ *by the DPLL system with Learning is finite if it contains no infinite subderivations consisting of only* Learn *and* Forget *steps.*

Note that this condition is very weak and easily enforced. Learn is typically only applied together with Backjump in order to learn the corresponding backjump clause. The theorem entails that such a strategy eventually reaches a state where only Learn and/or Forget apply, that is, a state that is final with respect to the Basic DPLL system. As already mentioned, by Lemma 2.9, this state is moreover easily recognizable because it is *FailState* or else it has the form $M \parallel G$ with all literals of G defined in M and no conflicting clause.

Actually, we could have alternatively defined a state $M \parallel G$ to be final if M is a *partial* assignment satisfying all clauses of G, hence allowing some literals of G to remain undefined. Then the correctness argument would have been exactly the same but without the use of Lemma 2.9—which now is needed mostly to show that the current definition of a final state $M \parallel G$ is a sufficient condition for M to be a model of G. However, in typical DPLL implementations, checking each time whether a partial assignment is a model of the current formula G is more expensive, because of the necessary additional bookkeeping, than just extending a partial model of G to a total one, which can be done with no search. But note that things may be different in the SMT case (see a brief discussion at the end of Section 3), or when the goal is to enumerate *all* models (perhaps in some compact representation) of the initial formula F.

We are now ready to prove that DPLL with learning provides a decision procedure for the satisfiability of CNF formulas.

THEOREM 2.12. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} S$ *where* S *is final with respect to Basic DPLL, then*

(1) S *is FailState if, and only if,* F *is unsatisfiable.*
(2) *If* S *is of the form* $M \parallel F'$ *then* M *is a model of* F.

PROOF. For Property 1, if S is *FailState* it is because there is some state $M \parallel F'$ such that $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} M \parallel F' \Longrightarrow_{\mathrm{L}} \mathit{FailState}$. By the definition of the Fail rule, there is no decision literal in M and there is a clause C in F' such that $M \models \neg C$. Since F and F' are equivalent by Lemma 2.7(3), we have that $F \models C$. However, if $M \models \neg C$, by Lemma 2.7(4), then also $F \models \neg C$, which implies that F is unsatisfiable. For the right-to-left implication, if S is not *FailState* it has to be of the form $M \parallel F'$. But then, by Lemma 2.9(3), M is a model of F and hence F is satisfiable.

For Property 2, if S is $M \parallel F'$, then, again by Lemma 2.9(3), M is a model of F. □

Note that the previous theorem does not guarantee confluence in the sense of rewrite systems, say. With unsatisfiable formulas, the only possible final (with respect to Basic DPLL) state for a sequence is *FailState*. If, on the other hand, the formula is satisfiable, different states that are final with respect to Basic DPLL may be reachable. However, all of them will be of the form $M \parallel F'$, with M a model of the original formula.

Although Theorem 2.12 was given for the relation $\Longrightarrow_{\mathrm{L}}$, it also holds for $\Longrightarrow_{\mathrm{B}}$, since the existence of Learn or Forget is not required in the proof.

THEOREM 2.13. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{B}}^{*} S$ *where* S *is final with respect to Basic DPLL, then*

(1) S *is FailState if, and only if,* F *is unsatisfiable.*
(2) *If* S *is of the form* $M \parallel F'$*, then* M *is a model of* F.

2.6. ABOUT PRACTICAL IMPLEMENTATIONS AND RESTARTS. State-of-the art SAT-solvers [Moskewicz et al. 2001; Goldberg and Novikov 2002; Eén and Sörensson 2003; Ryan 2004] essentially apply Abstract DPLL with Learning using efficient implementation techniques for UnitPropagate (such as the two-watched literal scheme for unit propagation [Moskewicz et al. 2001]), and good heuristics for selecting the decision literal when applying the Decide rule. As said, conflict analysis procedures for applying Backjump and the possibility of applying learning by other forms of resolution have also been well studied.

In addition, modern DPLL implementations *restart* the DPLL procedure whenever the search is not making enough progress according to some measure. The rationale behind this idea is that upon each restart, the additional knowledge of the search space compiled into the newly learned lemmas will lead the heuristics for Decide to behave differently, and possibly cause the procedure to explore the search space in a more compact way. The combination of learning and restarts has been shown to be powerful not only in practice, but also in theory. Essentially, any Basic DPLL derivation to *FailState* is equivalent to a *tree-like* refutation by resolution. But for some classes of problems tree-like proofs are always exponentially larger

than the smallest *general*, that is, DAG-like, resolution ones [Bonet et al. 2000]. The good news is that DPLL with learning and restarts becomes again equivalent to general resolution with respect to such notions of proof complexity [Beame et al. 2003].

In our formalism, restarts can be simply modeled by the following rule:

Definition 2.14. The Restart rule is:

$$M \parallel F \implies \emptyset \parallel F.$$

Adding the Restart rule to DPLL with Learning, it is obvious that all results of this section hold as long as one can ensure that a final state with respect to Basic DPLL is eventually reached. This is usually done in practice by periodically increasing the minimal number of Basic DPLL steps between each pair of restart steps. This is formalized below.

Definition 2.15. Consider a derivation by the DPLL system with learning extended with the Restart rule. We say that Restart *has increasing periodicity* in the derivation if, for each subderivation $S_i \implies \cdots \implies S_j \implies \cdots \implies S_k$ where the steps producing S_i, S_j, and S_k are the only Restart steps, the number of Basic DPLL steps in $S_i \implies \cdots \implies S_j$ is strictly smaller than in $S_j \implies \cdots \implies S_k$.

THEOREM 2.16. *Any derivation* $\emptyset \parallel F \implies S_1 \implies \cdots$ *by the transition system* L *extended with the* Restart *rule is finite if it contains no infinite subderivations consisting of only* Learn *and* Forget *steps, and* Restart *has increasing periodicity in it.*

PROOF. By contradiction, assume Der is an infinite derivation fulfilling the requirements. Let $\succ$ be the well-founded ordering on (the first components of) states defined in the proof of Theorem 2.10. In a subderivation of Der without Restart steps, at each step either this first component decreases with respect to $\succ$ (by the Basic DPLL steps) or it remains equal (by the Learn and Forget steps). Therefore, since there is no infinite subderivation consisting of only Learn and Forget steps, there must be infinitely many Restart steps in Der. Also, if between two states there is at least one Basic DPLL step and no Restart step, these states do not have the same first component. Therefore, if n denotes the (fixed, finite) number of different first components of states that exist for the given finite set of propositional symbols, there cannot be any subderivations with more than n Basic DPLL steps between two Restart steps. This contradicts the fact that there are infinitely many Restart steps if Restart has increasing periodicity in Der. □

In conclusion, in this section, we have formally described a large family of practical implementations of DPLL with learning and restarts, and proved that they provide a decision procedure for propositional satisfiability.

3. *Abstract DPLL Modulo Theories*

For many applications, encoding the problems into propositional logic is not the right choice. Frequently, a better alternative is to express the problems in a richer non-propositional logic, considering satisfiability with respect to a background theory T.

For example, some properties of timed automata are naturally expressed in *Difference Logic*, where formulas contain atoms of the form $a - b \leq k$, which are interpreted with respect to a background theory T of the integers, rationals or reals [Alur 1999]. Similarly, for the verification of pipelined microprocessors it is convenient to consider a logic of *Equality with Uninterpreted Functions (EUF)*, where the background theory T specifies a congruence [Burch and Dill 1994]. To mention just one further example, the conditions arising from program verification usually involve arrays, lists and other data structures, so it becomes very natural to consider satisfiability problems *modulo* the combined theory T of these data structures. In such applications, typical formulas consist of large sets of clauses such as:

$$p \quad \vee \quad \neg q \quad \vee \quad a = f(b - c) \quad \vee \quad read(s,\ f(b - c)) = d \quad \vee \quad a - g(c) \leq 7$$

containing purely propositional atoms as well as atoms over the combined theory. This is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory T: given a formula F, determine whether F is T-satisfiable, that is, whether there exists a model of T that is also a model of F.

In this section, we show that many of the existing techniques for handling SMT, of which SAT is a particular case if we consider T to be the empty theory, can be described and discussed within the Abstract DPLL framework.

3.1. Formal Preliminaries on Satisfiability Modulo Theories. Throughout this section, we consider the same definitions and notation given in Section 2 for the propositional case, except that here the set P over which formulas are built is a fixed finite set of *ground* (i.e., variable-free) first-order atoms, instead of propositional symbols.

In addition to these propositional notions, here we also consider some notions of first-order logic (see e.g., Hodges [1993]). A *theory* T is a set of closed first-order formulas. A formula F is *T-satisfiable* or *T-consistent* if $F \wedge T$ is satisfiable in the first-order sense. Otherwise, it is called *T-unsatisfiable* or *T-inconsistent*.

As in the previous section, a partial assignment M will sometimes also be seen as a conjunction of literals and hence as a formula. If M is a T-consistent partial assignment and F is a formula such that $M \models F$, that is, M is a (propositional) model of F, then we say that *M is a T-model of F*. If F and G are formulas, then *F entails G in T*, written $F \models_T G$, if $F \wedge \neg G$ is T-inconsistent. If $F \models_T G$ and $G \models_T F$, we say that F and G are *T-equivalent*. A *theory lemma* is a clause C such that $\emptyset \models_T C$.

The SMT problem for a theory T is the problem of determining, given a formula F, whether F is T-satisfiable, or, equivalently, whether F has a T-model.

As usual in SMT, given a background theory T, we will only consider the SMT problem for *ground* (and hence quantifier-free) CNF formulas F. Such formulas may contain *free* constants, that is, constant symbols not in the signature of T, which, as far as satisfiability is concerned, can be equivalently seen as existential variables. Other than free constants, all other predicate and function symbols in the formulas will instead come from the signature of T. From now on, when we say formula we will mean a formula satisfying these restrictions.

We will consider here only theories T such that the T-satisfiability of conjunctions of such ground literals is decidable. We will call any decision procedure for this problem a *T-solver*.

3.2. AN INFORMAL PRESENTATION OF SMT PROCEDURES. The current techniques for deciding the satisfiability of a ground formula F with respect to a background theory T can be broadly divided into two main categories: *eager* and *lazy*.

3.2.1. *Eager SMT Techniques.* In eager techniques, the input formula is translated using a satisfiability-preserving transformation into a propositional CNF formula which is then checked by a SAT solver for satisfiability (see, e.g., Bryant et al. [2001], Bryant and Velev [2002], and Strichman [2002]).

One of the strengths of this eager approach is that it can always use the best available SAT solver off the shelf. When the new generation of efficient SAT solvers such as Chaff [Moskewicz et al. 2001] became available, impressive results using the eager SMT approach were achieved by Bryant's group at CMU with the solver *UCLID* [Lahiri and Seshia 2004] for the verification of pipelined processors.

However, eager techniques are not very flexible: to make them efficient, sophisticated ad-hoc translations are required for each theory. For example, for EUF and for Difference Logic there exist the *per-constraint* encoding [Bryant and Velev 2002; Strichman et al. 2002], the *small domain* encoding (or *range-allocation* techniques), [Pnueli et al. 1999; Bryant et al. 2002; Talupur et al. 2004; Meir and Strichman 2005], and several hybrid approaches [Seshia et al. 2003]. The eager encoding approach can also handle integer linear arithmetic and the theory of arrays (see Seshia [2005]).

In spite of the effort spent in devising efficient translations, on many practical problems the translation process or the SAT solver run out of time or memory (see de Moura and Ruess [2004]). The current alternative techniques explained below are in many cases several orders of magnitude faster.

The correctness of the eager approach for SMT relies on the correctness of both the SAT solver and the translation, which is specific for each theory. It is out of the scope of this article to discuss the correctness of these ad-hoc translations. Assuming them to be correct, the correctness of the eager techniques follows from the results of Section 2.

3.2.2. *Lazy SMT Techniques.* As an alternative to the eager approach, one can use a specialized T-solver for deciding the satisfiability of conjunctions of theory literals. Then, a decision procedure for SMT is easily obtained by converting the given formula into disjunctive normal form (DNF) and using the T-solver to check whether any of the DNF conjuncts is satisfiable. However, the exponential blowup usually caused by the conversion into DNF makes this approach too inefficient.

A lot of research has then looked into ways to combine the strengths of specialized T-solvers with the strengths of state-of-the-art SAT solvers in dealing with the Boolean structure of formulas. The most widely used approach in the last few years is usually referred to as the *lazy* approach [Armando et al. 2000; Filliâtre et al. 2001; Audemard et al. 2002; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Armando et al. 2004; Ball et al. 2004]. In this approach, each atom occurring in a formula F to be checked for satisfiability is initially considered simply as a propositional symbol, *forgetting* about the theory T. Then the formula is given to a SAT solver. If the SAT solver determines it to be (propositionally) unsatisfiable, then F is T-unsatisfiable as well. If the SAT solver returns instead a propositional model M of F, then this assignment (seen as a conjunction of literals) is checked by a T-solver. If M is found T-consistent then it is a T-model of F. Otherwise, the T-solver builds a ground clause that is a logical consequence of T, that is, a theory

lemma, precluding that assignment. This lemma is added to F and the SAT solver is started again. This process is repeated until the SAT solver finds a T-model or returns unsatisfiable.

Example 3.1. Assume we are deciding with a lazy procedure the T-satisfiability of a large EUF formula, where T is the theory of equality, and assume that the model M found by the SAT solver contains, among many others, the four literals:

$$b=c, \quad f(b)=c, \quad a\neq g(b), \quad g(f(c))=a.$$

Then the T-solver detects that M is not a T-model, since

$$b=c \ \wedge \ f(b)=c \ \wedge \ g(f(c))=a \quad \models_T \quad a=g(b).$$

Therefore, the lazy procedure has to be restarted after the corresponding theory lemma has been added to the clause set. In principle, one can take as theory lemma simply the negation of M, that is, the disjunction of the negations of all the literals in M. However, this is usually not a good idea as the generated clause may end up containing thousands of literals. Lazy procedures are much more efficient if the T-solver is able instead to generate a small *explanation* of the T-inconsistency of M. In this example, the explanation could be simply the clause $b\neq c \ \vee \ f(b)\neq c \ \vee \ g(f(c))\neq a \ \vee \ a=g(b)$.

The main advantage of the lazy approach is its flexibility, since it can easily combine any SAT solver with any T-solver. More importantly, if the SAT solver used by the lazy SMT procedure is based on DPLL, then several refinements exist that make the SMT procedure much more efficient. Here we outline the most significant ones.

Incremental T-solver. The T-consistency of the assignment can be checked incrementally, while the assignment is being built by the DPLL procedure, without delaying the check until a propositional model has been found. This can save a large amount of useless work. It can be done fully eagerly, detecting T-inconsistencies as soon as they are generated, or, if that is too expensive, at regular intervals, for example, once every k literals added to the assignment. The idea was already mentioned in Audemard et al. [2002] under the name of *early pruning* and in Barrett [2003] under the name of *eager notification*. Currently, most SMT implementations work with incremental T-solvers. The incremental use of T-solvers poses different requirements on their implementation: to make the incremental approach effective in practice, the solver should (on average, say) be faster in processing one additional input literal l than in re-processing from scratch all previous inputs and l together. For many theories this can indeed be done; see, for example, Section 4.3, where we describe an incremental solver for Difference Logic.

On-line SAT Solver. When a T-inconsistency is detected by the incremental T-solver, one can ask the DPLL procedure simply to backtrack to some point where the assignment was still T-consistent, instead of restarting the search from scratch. For instance, if, in Abstract DPLL terms, the current state is of the form $M \parallel F$ and M has been detected to be T-inconsistent, then there is some subset $\{l_1 \cdots l_n\}$ of M such that $\neg l_1 \vee \cdots \vee \neg l_n$ is a theory lemma. This lemma can be added to the clause set, and, since it is conflicting, that is, it is false in M, Backjump or Fail can be applied. As we will formally prove below, after the backjump step this lemma is no longer needed for completeness and could be safely forgotten: the procedure

will search through all propositional models, finding a T-consistent one whenever it exists. Nevertheless, keeping theory lemmas can still be very useful for efficiency reasons, because it may cause an important amount of pruning later in the search. Theory lemmas are especially effective if they are small, as observed in, for example, de Moura and Rueß [2002] and Barrett [2003]. On-line SAT solvers (in combination with incremental T-solvers) are now common in SMT implementations, and state-of-the-art SAT solvers like zChaff or MiniSAT provide this functionality.

Theory Propagation. In the approach presented so far, the T-solver provides information only *after* a T-inconsistent partial assignment has been generated. In this sense, the T-solver is used only to *validate* the search *a posteriori*, not to *guide* it a priori. To overcome this limitation, the T-solver can also be used in a given DPLL state $M \parallel F$ to detect literals l occurring in F such that $M \models_T l$, allowing the DPLL procedure to move to the state $M\ l \parallel F$. We call this process *theory propagation*.

The idea of theory propagation was first mentioned in Armando et al. [2000] under the name of *Forward Checking Simplification*, and since then it has been applied, in limited form, in very few other systems (see Section 5). In contrast, theory propagation plays a major role in the DPLL(T) approach, introduced in Section 4 of this article. There we show that, somewhat against expectations, practical T-solvers can be designed to include this feature in an efficient way. A highly non-trivial issue is how to perform conflict analysis appropriately in the context of theory propagation. Different options and possible problems for doing this are analyzed and solved in detail in Section 5, something that, to our knowledge, had not been done before. In Section 6, we show that theory propagation, if handled well, has a crucial impact on the performance of SMT systems.

Exhaustive Theory Propagation. For some theories, it even pays off to perform *all* possible Theory Propagations before applying the Decide rule. This idea of exhaustive theory propagation is also introduced in the DPLL(T) approach presented here.

Lazy techniques that learn theory lemmas and do not perform any theory propagation in effect dump a large number of ground consequences of the theory into the clause set, duplicating theory information into the SAT solver. This duplication is instead completely unnecessary in a system with exhaustive theory propagation—and is greatly reduced with non-exhaustive theory propagation.

The reason is that any literal generated by unit propagation over a theory lemma can also be generated by theory propagation.[1]

For some logics, such as Difference Logic, for instance, exhaustive theory propagation usually yields speedups of several orders of magnitude, as we show in Section 6.

Using an incremental T-solver in combination with an on-line SAT solver is known to be crucial for efficiency. Possibly with the only exception of Verifun [Flanagan et al. 2003], an experimental system no longer under development, most, if not all, state-of-the-art SMT systems use incremental solvers. On the other hand, only a few SMT systems so far use theory propagation, as we will discuss in Section 5.

[1] But see the discussion about strategies with lazier theory propagation at the end of Section 5.1.

3.3. ABSTRACT DPLL MODULO THEORIES. In this section, we formalize the different enhancements of the lazy approach to Satisfiability Modulo Theories. We do this by adapting the Abstract DPLL framework for the propositional case presented in the previous section. One significant difference is that here we deal with ground first-order literals instead of propositional ones. Except for that, the rules Decide, Fail, UnitPropagate, and Restart remain unchanged: they will still regard all literals as syntactical items as in the propositional case. Only Learn, Forget and Backjump are slightly modified to work modulo theories: in these rules, entailment between formulas now becomes entailment in T. In addition, atoms of T-learned clauses can now also belong to M, and not only to F; this is required for Property 3.9 below, needed to recover from T-inconsistent states. Note that the theory version of Backjump below uses both the propositional notion of satisfiability ($\models$) and the first-order notion of entailment modulo theory ($\models_T$).

Definition 3.2. The rules T-Learn, T-Forget and T-Backjump are:

T-Learn:

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{cases}$$

T-Forget:

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \textbf{if} \begin{cases} F \models_T C \end{cases}$$

T-Backjump:

$$M\ l^{\mathsf{d}}\ N \parallel F, C \quad \Longrightarrow \quad M\ l' \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models_T C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N. \end{cases}$$

3.3.1. *Modeling the Naive Lazy Approach.* Using these rules, it is easy to model the basic lazy approach (without any of the refinements of incremental T-solvers, on-line SAT solvers or theory propagation). Each time a state $M \parallel F$ is reached that is final with respect to Decide, Fail, UnitPropagate, and T-Backjump, that is, final in a similar sense as in the previous section, M can be T-consistent or not. If it is, then M is indeed a T-model of F, as we will prove below. If M is not T-consistent, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. By one T-Learn step, the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$ can be learned and then Restart can be applied. As we will prove below, if these learned theory lemmas are never removed by the T-Forget rule, this strategy is terminating under similar requirements as those in the previous section, namely, the absence of infinite subderivations consisting of only Learn and Forget steps and the increasing periodicity of Restart steps. Then, the strategy is also sound and complete as stated in the previous section: the initial formula is T-unsatisfiable if, and only if, *FailState* is reached; moreover, if *FailState* is not reached then a T-model has been found.

3.3.2. *Modeling the Lazy Approach with an Incremental T-Solver.* Assume a state $M \parallel F$ has been reached where M is T-inconsistent. Note that in practice this

is detected by the incremental T-solver, and that this state need not be final now. Then, as in the naive lazy approach, there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. This theory lemma is then learned, producing the state $M \parallel F,\ \neg l_1 \vee \cdots \vee \neg l_n$. As in the previous case, Restart can then be applied and the same results hold.

3.3.3. *Modeling the Lazy Approach with an Incremental T-Solver and an on-Line SAT Solver.* As in the previous case, if a subset $\{l_1, \ldots, l_n\}$ of M is detected such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$, the theory lemma is learned, reaching the state $M \parallel F,\ \neg l_1 \vee \cdots \vee \neg l_n$. But now, since in addition we consider an online SAT solver, instead of completely restarting, the procedure *repairs* the T-inconsistency of the partial assignment by exploiting the fact that the recently learned theory lemma is a conflicting clause. As we show later, and similarly to what happened in the propositional case, if there is no decision literal in M then Fail applies, otherwise T-Backjump applies. Our results below prove that, even if the theory lemma is always forgotten immediately after backjumping, this approach is terminating, sound, and complete under similar conditions as the ones of the previous section.

3.3.4. *Modeling the Previous Refinements and Theory Propagation.* This requires the following additional rule:

Definition 3.3. The TheoryPropagate rule is:

$$M \parallel F \implies M\ l \parallel F \quad \textbf{if} \quad \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M. \end{cases}$$

The purpose of this rule is to prune the search by assigning a truth value to literals that are (propositionally) undefined by the current assignment M but T-entailed by it, rather than letting the Decide rule guess a value for them. As said, this sort of propagation can lead to dramatic improvements in performance. Below we prove that the correctness results mentioned for the previous three lazy approaches also hold in combination with arbitrary applications of this rule.

3.3.5. *Modeling the Previous Refinements and Exhaustive Theory Propagation.* Exhaustive theory propagation is modeled simply by assuming that TheoryPropagate is applied with a higher priority than Decide. The correctness of this approach follows immediately from the correctness of the previous one which had arbitrary applications of TheoryPropagate.

3.4. CORRECTNESS OF ABSTRACT DPLL MODULO THEORIES. Up to now, we have seen several different application strategies of (subsets) of the given rules, which lead to different SMT procedures. In this subsection we give a simple and uniform proof showing that all the approaches described in the previous subsection are indeed decision procedures for the SMT problem. The proofs are structured in the same way as the ones given in Section 2.5 for the propositional case, and hence here we focus on the variations and extensions that are needed.

Definition 3.4. The *Basic DPLL Modulo Theories system* consists of the rules Decide, Fail, UnitPropagate, TheoryPropagate, and T-Backjump.

Definition 3.5. The *Full DPLL Modulo Theories system*, denoted by FT, consists of the rules of Basic DPLL Modulo Theories and the rules T-Learn, T-Forget, and Restart.

As before, a decision procedure will be obtained by generating a derivation using the given rules with a particular strategy. However, here the aim of a derivation is to compute a state S to which the main theorem of this section, Theorem 3.10, can be applied, that is, a state S such that: (i) S is final with respect to the rules of Basic DPLL Modulo Theories and (ii) if S is of the form $M \parallel F$ then M is T-consistent.

Property 3.9 below provides a very general class of strategies in which such a state S is always reached, without violating the requirements of termination of Theorem 3.7 (also given below). Such a state S can be recognized in a similar way as in the propositional case: it is either *FailState* or it is of the form $M \parallel F$ where all the literals of F are defined in M, there are no conflicting clauses, and M is T-consistent.

The following lemma states invariants similar to the ones of Lemma 2.7 of the previous section.

LEMMA 3.6. *If* $\emptyset \parallel F \Longrightarrow_{\text{FT}}^{*} M \parallel G$, *then the following hold:*

(1) *All the atoms in* M *and all the atoms in* G *are atoms of* F.
(2) M *contains no literal more than once and is indeed an assignment, that is, it contains no pair of literals of the form* p *and* $\neg p$.
(3) G *is* T*-equivalent to* F.
(4) *If* M *is of the form* $M_0\ l_1\ M_1\ \cdots\ l_n\ M_n$, *where* $l_1, \ldots, l_n$ *are all the decision literals of* M, *then* $F, l_1, \ldots, l_i \models_T M_i$ *for all* i *in* $0 \cdots n$.

PROOF. As for Lemma 2.7, all rules preserve the properties. The new rule TheoryPropagate preserves them like UnitPropagate; the other rules as for their propositional versions. □

THEOREM 3.7 (TERMINATION). *Let* Der *be a derivation of the form:*
$\emptyset \parallel F = S_0 \Longrightarrow_{\text{FT}} S_1 \Longrightarrow_{\text{FT}} \cdots$
Then Der *is finite if the following two conditions hold:*

(1) Der *has no infinite subderivations consisting of only* T*-Learn and* T*-Forget steps.*
(2) *For every subderivation of* Der *of the form:*
$S_{i-1} \Longrightarrow_{\text{FT}} S_i \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_j \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_k$
where the only three Restart *steps are the ones producing* S_i, S_j, *and* S_k, *either:*
—*there are more Basic DPLL Modulo Theories steps in* $S_j \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_k$ *than in* $S_i \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_j$, *or*
—*a clause is learned*[2] *in* $S_j \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_k$ *that is not forgotten in* Der.

PROOF. The proof is a slight extension of the one of Theorem 2.16. The only new aspect is that some Restart steps are applied with non-increasing periodicity. But since for each one of them a new clause has been learned that is never forgotten in Der, there can only be finitely many of them. From this, a contradiction follows as in Theorem 2.16. □

[2] See Definition 2.5.

LEMMA 3.8. *If* $\emptyset \parallel F \Longrightarrow^{*}_{\mathrm{FT}} M \parallel F'$ *and there is some conflicting clause in* $M \parallel F'$*, that is,* $M \models \neg C$ *for some clause* C *in* F'*, then either* Fail *or* T*-*Backjump *applies to* $M \parallel F'$*.*

PROOF. As in Lemma 2.8. □

PROPERTY 3.9. *If* $\emptyset \parallel F \Longrightarrow^{*}_{\mathrm{FT}} M \parallel F'$ *and* M *is* T*-inconsistent, then either there is a conflicting clause in* $M \parallel F'$*, or else* T*-*Learn *applies to* $M \parallel F'$*, generating a conflicting clause.*

PROOF. If M is T-inconsistent, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. Hence, the conflicting clause $\neg l_1 \vee \cdots \vee \neg l_n$ is either in $M \parallel F'$, or else it can be learned by one T-Learn step. □

Lemma 3.8 and Property 3.9 show that a rule of Basic DPLL modulo theories is always applicable to a state of the form $M \parallel F$, or to its successor after a single T-Learn step, whenever a literal of F is undefined in M, or F contains a conflicting clause, or M is T-inconsistent. Together with Theorem 3.7 (Termination), this shows how to compute a state to which the following main theorem is applicable.

THEOREM 3.10. *Let* Der *be a derivation* $\emptyset \parallel F \Longrightarrow^{*}_{\mathrm{FT}} S$*, where (i)* S *is final with respect to Basic DPLL Modulo Theories, and (ii) if* S *is of the form* $M \parallel F'$ *then* M *is* T*-consistent. Then*

(1) *S is FailState if, and only if, F is T-unsatisfiable.*

(2) *If S is of the form $M \parallel F'$, then M is a T-model of F.*

PROOF. The first result follows from Lemmas 3.6(3), 3.6(4), as in Theorem 2.12. The second part is proved as in Lemma 2.9 of the previous section, but using Lemma 3.8 and Lemma 3.6(3), instead of Lemma 2.8 and Lemma 2.7(3). □

The previous theorem shows that a large family of practical approaches provide a decision procedure for satisfiability modulo theories. Note that the results of this section are independent from the theory T under consideration, the only (obviously necessary) requirement being the decidability of the T-consistency of conjunctions of ground literals.

We conclude this section by observing that, as in the propositional case, our definition of final state for Abstract DPLL Modulo Theories forces the assignment M in a state of the form $M \parallel G$ to be total. We remarked in the previous section that the alternative definition of final state where M can be partial as long as it satisfies G is inefficient in practice in the SAT case. With theories, however, this is not always true. Depending on the theory T and the available T-solver, it may be considerably more expensive to insist on extending a satisfying partial assignment to a total one than to check periodically whether the current assignment has become a model of the current formula. The reason is that by Theorem 3.10 one can stop the search with a final state $M \parallel G$ only if M is also T-consistent, and T-consistency checks can have a high cost, especially when the T-satisfiability of conjunction of literals is NP-hard. We have maintained the same definition of final state for both Abstract DPLL and Abstract DPLL Modulo Theories mainly for simplicity, to make the lifting of the former to the latter clearer. We stress though that as in

the previous section essentially the same correctness proof applies if one uses the alternative definition of final state in this section.

4. *The DPLL(T) Approach*

We have now seen an abstract framework that allows one to model a large number of complete and terminating strategies for SMT. In this section, we describe the DPLL(T) approach for Satisfiability Modulo Theories, a general modular architecture on top of which actual implementations of such SMT strategies can be built. This architecture is based on a general DPLL engine, called DPLL(X), that is not dependent on any particular theory T. Instead, it is parameterized by a solver for a theory T of interest. A DPLL(T) system for T is produced by instantiating the parameter X with a module $Solver_T$ that can handle conjunctions of literals in T, that is, a T-solver.

The basic idea is similar to the $CLP(X)$ scheme for constraint logic programming [Jaffar and Maher 1994]: provide a clean and modular, but at the same time efficient, integration of specialized theory solvers within a general-purpose engine, in our case one based on DPLL.

The DPLL(T) architecture presented here combines the advantages of the eager and lazy approaches to SMT. On the one hand, the architecture allows for very efficient implementations, as witnessed by our system, Barcelogic, which implements DPLL(T) for a number of theories and compares very favorably with other SMT systems—see Section 6. On the other hand, DPLL(T) has the flexibility of the lazy approaches: more general logics can be dealt with by simply plugging in other solvers into the general DPLL(X) engine, provided that these solvers conform to a minimal interface.

4.1. Overall Architecture of DPLL(T).

At each state $M \parallel F$ of a derivation, the DPLL(X) engine knows M and F, but it treats all literals and clauses as purely propositional ones. As a consequence, all the needed theory-based inferences are exclusively localized in the theory solver $Solver_T$, which knows M but not the current F.

For the purposes of this article, it is not necessary to precisely define the interface between DPLL(X) and $Solver_T$. It suffices to know that $Solver_T$ provides operations that can be called by DPLL(X) to:

—Notify $Solver_T$ that a certain literal has been set to true.

—Ask $Solver_T$ to check whether the current partial assignment M, as a conjunction of literals, is T-inconsistent. This request can be made by DPLL(X) with different degrees of *strength*: for theories where deciding T-inconsistency is in general expensive, it might be more convenient to use cheaper, albeit incomplete, T-inconsistency checks for most of the derivation, and resort to a more expensive but complete check only when necessary.[3]

It is required that when $Solver_T$ detects a T-inconsistency it is also able to identify a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. This theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$, which we will call the *(theory) explanation* of the T-inconsistency, is then communicated by $Solver_T$ to the engine.

[3] Note that, according to the correctness results of Abstract DPLL modulo theories, a *decision* of T-inconsistency is only needed when a final state with respect to the Basic DPLL rules is reached.

—Ask $Solver_T$ to identify currently undefined input literals that are T-consequences of M. Again, this request can be made by DPLL(X) with different degrees of strength. $Solver_T$ answers with a (possibly empty) list of literals of the input formula that are newly detected T-consequences of M. Note that for this operation $Solver_T$ needs to know the set of input literals.

—Ask $Solver_T$ to provide a justification for the T-entailment of some theory propagated literal l. This is needed for the following reasons. In a concrete implementation of the DPLL(X) engine, backjumping is typically guided by a conflict graph, as explained in Example 2.6. But there is a difference with respect to the purely propositional case: a literal l at a node in the graph can now also be due to an application of theory propagation. Hence, building the graph requires that $Solver_T$ be able to recover and return as a justification of l a (preferably small, non-redundant) subset $\{l_1, \ldots, l_n\}$ of literals of the assignment M that T-entailed l when l was T-propagated. Computing that subset amounts to generating the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n \vee l$. We will call this lemma the *(theory) explanation* of l. (See Example 5.1, and also Section 4.3 and Section 5 for more details and refinements.)

—Ask $Solver_T$ to undo the last n notifications that a literal has been set to true.

In the rest of this section, we describe two concrete SMT strategies for the Abstract DPLL modulo theories framework, and show how they can be implemented using the DPLL(T) architecture.

The first one, described in Section 4.2, performs exhaustive theory propagation in a very eager way: in a state $M \parallel F$, TheoryPropagate is immediately applied whenever some input literal l is T-entailed by M. Therefore, $Solver_T$ is required to detect *all* such entailments immediately after a literal is set to true. In contrast, the second DPLL(T) system, described in Section 4.4, allows $Solver_T$ to sometimes fail to detect some entailed literals.

Each system is accompanied by a concrete motivating example of a theory of practical relevance, namely Difference Logic and EUF Logic, respectively. For Difference Logic, an efficient design for $Solver_T$ is described in Section 4.3.

Further refinements of theory propagation and conflict-driven clause learning are discussed in more detail in Section 5.

4.2. DPLL(T) WITH EXHAUSTIVE THEORY PROPAGATION AND DIFFERENCE LOGIC. Here, we deal with a particular application strategy of the rules of Abstract DPLL Modulo Theories modeling exhaustive theory propagation. We show how it can be implemented using the DPLL(T) architecture, and explain the roles of the DPLL(X) engine and the theory solver $Solver_T$ in it.

$Solver_T$ processes the input formula, stores the list of all literals occurring in it, and hands it over to DPLL(X), which treats it as a purely propositional CNF. After that, the various Abstract DPLL rules are applied by DPLL(X) as described below:

TheoryPropagate. Immediately after $Solver_T$ is notified that a literal l has been added to M, (e.g., as a consequence of UnitPropagate or Decide), $Solver_T$ is also requested to provide *all* input literals that are T-consequences of $M\, l$ but not of M alone. Then, for each one of them, TheoryPropagate is immediately applied by DPLL(X). Note that, this way, M never becomes T-inconsistent, a property that can be exploited by $Solver_T$ to increase its efficiency (see the next subsection for

the case of Difference Logic), and by the DPLL(X) engine since it will never need to ask for the T-consistency of M.

UnitPropagate. If TheoryPropagate is not applicable, DPLL(X) tries to apply UnitPropagate next, possibly triggering more rounds of theory propagation, and stops if it discovers a conflicting clause. (In a concrete implementation all this can be implemented with the commonly used two-watched-literals scheme.)

Backjump and Fail. If DPLL(X) detects a conflicting clause, it immediately applies T-Backjump or Fail, depending respectively on whether the current assignment contains a decision literal or not. (In a concrete implementation, an appropriate backjump clause can be computed as explained in the next section.) At each backjump, DPLL(X) tells $Solver_T$ how many literals have been removed from the assignment.

***T*-Learn.** Immediately after each T-Backjump application, the T-Learn rule is applied to learn the backjump clause. This is possible because this clause is always a T-consequence of the formula F in the current state $M \parallel F$. Note that, as explained in Section 3.2 for the case of exhaustive theory propagation, theory lemmas (clauses C such that $\emptyset \models_T C$) are never learned, since they are useless in this context.

Restart. For correctness with respect to the Abstract DPLL modulo theories framework, one must guarantee that Restart has increasing periodicity. Typically, this is achieved by only applying Restart when certain system parameters reach some prescribed limits, such as the number of conflicts or the number of new units derived, and increasing this restart limit periodically.

***T*-Forget.** For correctness with respect to Abstract DPLL modulo theories, it suffices to apply this rule only to previously T-learned clauses. This is what is usually done, removing part of these clauses according to their activity (e.g., the number of times involved in recent conflicts).

Decide. In this strategy, DPLL(X) applies Decide only if none of the other Basic DPLL rules apply. The choice of the decision literal is well known to have a strong impact on the search behavior. Numerous heuristics for this purpose exist.

4.3. DESIGN OF $Solver_T$ FOR DIFFERENCE LOGIC. To provide an example in this article of $Solver_T$ for a given T, here we briefly outline the design of a theory solver for Difference Logic. Despite its simplicity, Difference Logic has been used to express important practical problems, such as verification of timed systems, scheduling problems or the existence of paths in digital circuits with bounded delays.

In Difference Logic, the background theory T can be the theory of the integers, the rationals or the reals, depending on the application. Input formulas are restricted to Boolean combinations of atoms of the form $a \leq b + k$, where a and b are free constants and k is a (possibly negative) integer, rational or real constant. Over the integers, atoms of the form $a < b + k$ can be equivalently written as $a \leq b + (k - 1)$; for instance, $a < b + 7$ becomes $a \leq b + 6$. A similar transformation exists for the rationals and reals, by decreasing k by a small enough amount ε. For a given input formula, the ε to be applied to its literals can be computed in linear time [Schrijver 1987; Armando et al. 2004]. Similarly, negations and equalities can also be removed, and one can assume that all literals are of the form $a \leq b + k$. Their conjunction can be seen as a weighted graph G with an edge $a \xrightarrow{k} b$ for each literal $a \leq b + k$. Independently of whether T is the theory

of the integers, the rationals or the reals, such a conjunction is T-satisfiable if, and only if, there is no cycle in G with negative accumulated weight. Therefore, once all literals are of the form $a \leq b + k$, the specific theory does not matter any more.

4.3.1. *Initial Setup.* As said, for the initial setup of DPLL(T), $Solver_T$ reads the input CNF, stores the list of all literals occurring in it, and hands it over to DPLL(X) as a purely propositional CNF. For efficiency reasons, it is important that, in this CNF, the relation between literals and their negations is made explicit. For example, over the integers, if $a \leq b + 2$ and $b \leq a - 3$ occur in the input, then, since one is equivalent to the negation of the other, they should be abstracted by a propositional variable and its negation. This can be detected by using a canonical form during this setup process. For instance, one can impose an ordering on the free constants and require that the smallest one, say a in the example above, be always on the left-hand side of the $\leq$ symbol. So here we would have that $b \leq a - 3$ is handled as the negation of $a \leq b + 2$.

For reasons we will see below, $Solver_T$ also builds a data structure containing, for each constant symbol, the number of input literals it occurs in, and the list of all these literals.

4.3.2. *DPLL(X) Sets the Truth Value of a Literal.* Then, $Solver_T$ adds the corresponding edge to the graph. Here we will write $a_0 \xrightarrow{k\,*} a_n$ if there is a path in the graph of the form $a_0 \xrightarrow{k_1} a_1 \xrightarrow{k_2} \cdots \xrightarrow{k_{n-1}} a_{n-1} \xrightarrow{k_n} a_n$ with $n \geq 0$ and where $k = k_1 + \cdots + k_n$ is called the length of this path.

Note that one can assume that DPLL(X) does not communicate to $Solver_T$ any redundant edges (i.e., edges already entailed by G), since such consequences would already have been communicated by $Solver_T$ to DPLL(X). Similarly, DPLL(X) will not communicate to $Solver_T$ any edges that are inconsistent with the graph. Therefore, there will be no cycles of negative length. Here, $Solver_T$ must return to DPLL(X) all input literals that are new consequences of the graph once the new edge has been added. Essentially, for detecting the new consequences of a new edge $a \xrightarrow{k} b$, $Solver_T$ needs to check all paths $a_i \xrightarrow{k_i\,*} a \xrightarrow{k} b \xrightarrow{k'_j\,*} b_j$ and see whether there is any input literal that follows from $a_i \leq b_j + (k_i + k + k'_j)$, that is, an input literal of the form $a_i \leq b_j + k'$, with $k' \geq k_i + k + k'_j$. For checking all such paths from a_i to b_j that pass through the new edge from a to b, the graph is kept in double adjacency list representation. Then a standard single-source-shortest-path algorithm starting from a can be used for computing all a_i with their corresponding minimal k_i (and similarly for the b_j). Its cost, for M literals containing N different constant symbols, is $O(N \cdot M)$.

While doing this, the visited nodes are marked and inserted into two lists, one for the a_i's and one for the b_j's. At the same time, two counters are kept measuring the total number of input literals containing the a_i's and, respectively, the b_j's.

Then, if, without loss of generality, the a_i's are the ones that occur in less input literals, we check, for each input literal l containing some a_i, whether the other constant in l is some of the found b_j, and whether l is entailed or not (this can be checked in constant time since previously all b_j have been marked). The asymptotic worst-case cost of this part is $O(L)$, where L is the number of different input literals. In our experience, this is much faster than the $O(N^2)$ check of the Cartesian product of a_i's and b_j's.

4.3.3. *Implementation of Explain and Backtrack.* Whenever the mth edge is added to the graph, the edge is annotated with its *insertion number* m. When a literal l of the form $a \leq b + k$ is returned as a T-consequence of the mth edge, this m is recorded together with l. If later on the explanation for l is required, a path in the graph from a to b of length at most k is searched, using a depth-first search as before, but without traversing any edges with insertion number greater than m. This not only improves efficiency, but it is also needed for not returning "too new" explanations, which may create cycles in the implication graph (see Section 5). Each time DPLL(X) backjumps, it communicates to $Solver_T$ how many edges it has to remove, for example, up to some insertion number m. According to our experiments, the best way (with negligible cost) for dealing with this in $Solver_T$ is the naive one, that is, using a trail stack of edges with their insertion numbers and all their associated T-consequences.

4.4. DPLL(T) WITH NONEXHAUSTIVE THEORY PROPAGATION AND EUF LOGIC. For some logics, such as the logic of Equality with Uninterpreted Functions (EUF), exhaustive theory propagation is not the best strategy. In EUF, atoms consist of ground equations between terms, and the theory T consists of the axioms of reflexivity, symmetry, and transitivity of '=', as well as the *monotonicity* axioms, saying, for all f, that $f(x_1 \cdots x_n) = f(y_1 \cdots y_n)$ whenever $x_i = y_i$ for all i in $1 \cdots n$ (see also Example 3.1).

Our experiments with EUF revealed that a nonexhaustive strategy behaves better in practice than one with exhaustive theory propagation. More precisely, we found that detecting exhaustively all *negative* equality consequences is very expensive, whereas all positive equalities can propagated efficiently by means of a congruence closure algorithm [Downey et al. 1980]. It is beyond the scope of this article to describe the design of a theory solver for EUF. We refer the reader to Nieuwenhuis and Oliveras [2003] for a description and discussion of a modern incremental, backtrackable congruence closure algorithm for this purpose. We point out that efficiently retrieving explanations (for constructing the conflict graph and generating theory lemmas) inside an incremental congruence closure algorithm is nontrivial. Increasingly better techniques have been developed in de Moura et al. [2004] Stump and Tan [2005], and Nieuwenhuis and Oliveras [2005b].

We describe below an application strategy of Abstract DPLL Modulo Theories for DPLL(T) with nonexhaustive theory propagation. The emphasis will be on those aspects that differ from the exhaustive case, and on how and when T-inconsistent partial assignments M are detected and repaired.

TheoryPropagate. In this strategy, when $Solver_T$ is asked for new T-consequences, it may return only an incomplete list. Therefore, DPLL(X) can no longer maintain the invariant that the partial assignment is always T-consistent as in the exhaustive case of Section 4.2. For this reason, it is no longer necessary to ask for T-consequences as eagerly as in the exhaustive case. Instead, for efficiency reasons it is better to ask $Solver_T$ for new T-consequences only if no Basic DPLL rule other than Decide applies and the current assignment is T-consistent. For each returned T-consequence, TheoryPropagate is immediately applied by DPLL(X).

UnitPropagate. DPLL(X) applies this rule while possible unless it detects a conflicting clause.

Backjump and Fail. DPLL(X) may apply T-Backjump or Fail due to two possible situations. The first one is when it detects a conflicting clause, as usual. The second one is due to a T-inconsistency of the current partial assignment M.

$Solver_T$ is asked to check the T-consistency of M each time no Basic DPLL rule other than Decide applies—and before being asked for theory consequences. When M is T-inconsistent $Solver_T$ identifies a subset $\{l_1, \ldots, l_n\}$ of it such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$, and returns the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$ as an explanation of the inconsistency. *DPLL*(X) then handles the lemma as a conflicting clause, applying T-Backjump or Fail to it.

***T*-Learn.** Immediately after each T-Backjump application, the T-Learn rule is applied for learning the backjump clause.

Now, in backjumps due to T-inconsistencies, the backjump clause may sometimes be the theory lemma denoting the T-inconsistency itself (if it has only one literal of the current decision level). Therefore, in this case, sometimes theory lemmas will be learned. Another possibility is to always learn the theory lemma coming from a T-inconsistency, even if it is not the backjump clause. This may be useful, because it prevents the same T-inconsistency from occurring again.

Restart. This rule is applied as in the exhaustive strategy of Section 4.2.

***T*-Forget.** It is also applied as in the exhaustive case, but in this case among the (less active) lemmas that are removed there are also theory lemmas. This is again less simple than in the exhaustive case, because different forgetting policies could be applied to the two kinds of lemmas. Note that, in any case, none of the lemmas needs to be kept for completeness.

Decide. This rule is applied as in the exhaustive strategy of Section 4.2.

We conclude this section by summarizing the key differences between the two strategies for exhaustive and nonexhaustive theory propagation, described in Sections 4.2 and 4.4: in the former, which we applied to Difference Logic, the partial model never becomes T-inconsistent, since *all* input literals that are T-consequences are immediately set to true. In contrast, the DPLL(T) system described in Section 4.4, and applied to EUF logic, allows, for efficiency reasons, $Solver_T$ to fail sometimes to detect some entailed literals, and hence it must be able to recover from T-inconsistent partial assignments.

5. *Theory Propagation Strategies and Conflict Analysis*

The idea of theory propagation was first mentioned in Armando et al. [2000] under the name of *Forward Checking Simplification*, in the context of temporal reasoning. The authors suggest that a literal l can be propagated if $\neg l$ is inconsistent with the current state, but they also imply that this is expensive "since it requires a number of T-consistency checks roughly equal to the number of literals in [the whole formula] φ". A similar notion called *Enhanced Early Pruning* is mentioned in Audemard et al. [2002] in the context of the MathSAT system, but nothing is said about when and how it is applied, and how it relates to conflict analysis. Also, the new system Yices (see Section 6) appears to apply some form of theory propagation. Except for these systems and ours, and a forthcoming version of CVC Lite based on the work described here, we are not aware of any other systems that apply theory propagation,

nor of any other descriptions of theory propagation in the literature outside our own previous work on the subject.

We remark that the techniques proposed in, for example, Armando et al. [2004] and Flanagan et al. [2003], where the input formula is statically augmented with theory lemmas encoding transitivity constraints, may have effects similar to theory propagation. However, eagerly learning all such constraints is usually too expensive and typically only a subset of them is ever used at run-time.

In Ganzinger et al. [2004], we showed that, somewhat against expectations, practical T-solvers can be designed to do theory propagation efficiently. To the best of our knowledge, before that, the methods for detecting a theory consequence l were essentially based on sending $\neg l$ to the theory solver, and checking whether a T-inconsistency was derived.

Some essential and nontrivial issues about theory propagation have remained largely unstudied until now:

—when to compute the explanations for the theory propagated literals;

—how to handle conflict analysis adequately in the context of theory propagation;

—how eagerly to perform theory propagation.

In this section, we analyze these issues in detail. We point out that thinking in terms of Abstract DPLL Modulo Theories was crucial in giving us a sufficient understanding for doing this analysis in the first place, especially by helping us clearly separate correctness concerns from efficiency ones. We start with a running example illustrating some of the questions above.

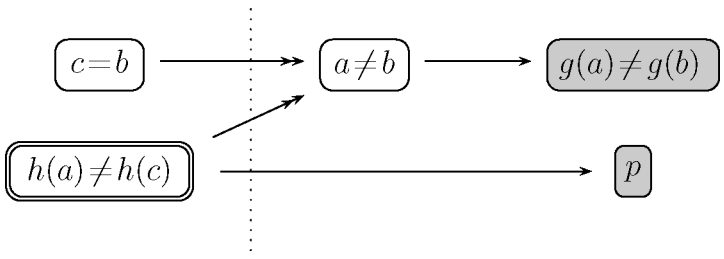
Example 5.1. Consider EUF logic and a clause set F containing, among others:

$$(1)\ a=b \ \vee\ g(a)\neq g(b)$$
$$(2)\ h(a)=h(c) \ \vee\ p$$
$$(3)\ g(a)=g(b) \ \vee\ \neg p.$$

Now consider a state of the form $M,\ c=b,\ f(a)\neq f(b) \parallel F,$ and the following sequence of derivation steps:

Step:	**New literal:**	**Reason:**
Decide	$h(a)\neq h(c)$	
TheoryPropagate	$a\neq b$	since $h(a)\neq h(c)\wedge c=b \models_T a\neq b$
UnitPropagate	$g(a)\neq g(b)$	because of $a\neq b$ and Clause 1
UnitPropagate	p	because of $h(a)\neq h(c)$ and Clause 2.

In the resulting state, Clause 3 is conflicting. When seen as a conflict graph, as done in Example 2.6 for the propositional case, the situation looks as follows:



In this graph, the double arrows $\twoheadrightarrow$ indicate theory propagations, whereas the single arrows denote unit propagations. The backjump clause $h(a) = h(c) \ \vee \ c \neq b$ can be produced by considering the indicated cut in the graph, as in Example 2.6. This clause can also be obtained by the backwards resolution process on the conflicting clause illustrated in Example 2.6, specifically, by resolving in reverse chronological order with the clauses that caused propagations, until a clause with exactly one literal from the current decision level is derived.

The only difference here with respect to the propositional case is that now we can have theory propagated literals as well. For each one of these literals, resolution is done with the theory lemma explaining its propagation (here, the leftmost premise of the last resolution step):

$$\dfrac{\dfrac{h(a)=h(c)\vee c\neq b\vee \mathbf{a}\neq\mathbf{b} \qquad \dfrac{a=b\vee \mathbf{g(a)}\neq\mathbf{g(b)} \qquad \dfrac{h(a)=h(c)\vee \mathbf{p} \qquad g(a)=g(b)\vee\neg\mathbf{p}}{\mathbf{g(a)}=\mathbf{g(b)}\vee h(a)=h(c)}}{h(a)=h(c)\vee \mathbf{a}=\mathbf{b}}}{h(a)=h(c) \ \vee \ c\neq b}$$

The resulting clause can be used as a backjump clause, in the same way as in Example 2.6 for the propositional case.

In what follows, we argue that in general it is not a good idea to compute these theory lemmas (or *explanations*) immediately, during theory propagation. Instead, it is usually better to compute each of them only as needed in resolution steps during conflict analysis. We also explain what problems may occur in delaying the computation of explanations until they are really needed, and give detailed results showing when and how a backjump clause can be found. □

5.1. When to Compute Explanations for the Theory Propagated Literals. Each time a theory propagation step of the form $M \parallel F \implies M\, l \parallel F$ takes place, this is because $l_1 \wedge \cdots \wedge l_n \models_T l$ for some subset $\{l_1, \ldots, l_n\}$ of M. Now, a very simple way of managing theory propagated literals for the purposes of conflict analysis is to use T-Learn immediately after each such a theory propagation step, adding the corresponding theory lemma $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ to the current formula. After that, the theory propagated literal l can be simply seen as a unit propagated literal by the newly learned clause. Hence, when a conflicting clause is detected, the backjump clause can be computed exactly as in the propositional case, as explained in Example 2.6.

Unfortunately, this approach, used for instance in the latest version of the MathSAT system [Bozzano et al. 2005], has some important drawbacks. We have done extensive experiments, running our DPLL(T) implementations on all the formulas available in the SMT-LIB benchmark library [Ranise and Tinelli 2003; Tinelli and Ranise 2005] for the logics EUF, RDL, IDL and UFIDL (see the next section). In these experiments, we have counted (i) the number of theory propagation steps and (ii) the number of times theory propagated literals are involved in a conflict, in other words, the number of resolution steps with explanations.

It turns out that, on average, theory propagations are around 250 times more frequent than resolution steps with explanations. For almost all examples, the ratio lies between 20 and 1500. Hence, immediately computing an explanation each time a theory propagation takes place, as done in MathSAT, is bound to be highly inefficient: on average just one of these lemmas out of every 250 is ever going

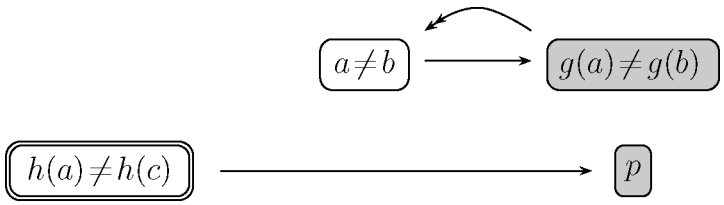
to be used (possibly, even less than that, as each theory propagated literal may occur in more than one conflict). The cost of generating explanations is twofold: it is the cost incurred by $Solver_T$ in computing the clause and that incurred by DPLL(X) in inserting the clause in the clause database and maintaining it under propagation.

There is however a potential advantage in the MathSAT approach, for strategies where TheoryPropagate is applied only if no other rule except Decide is applicable (this is what we did for EUF in Section 4.4). Assume as before that TheoryPropagate applies to a state $M \parallel F$ because $l_1 \wedge \cdots \wedge l_n \models_T l$ for for some subset $\{l_1, \ldots, l_n\}$ of M. Also assume that, due to a previous TheoryPropagate step, the explanation $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ is still present in F, although, due to backtracking, l has become again undefined in M. Then the effect of theory propagating l can now be achieved more efficiently by a unit propagation step with the clause $\neg l_1 \vee \cdots \vee \neg l_n \vee l$. If this leads to a conflict at the current decision level before TheoryPropagate is tried, then a gain in efficiency may be obtained. If, on the other hand, no conflict occurs before applying TheoryPropagate, then it is likely that repeated work is done by $Solver_T$, rediscovering the fact that l is a T-consequence of M.

For some theory solvers, it may be possible that, when computing a T-consequence, there is only a low additional cost in computing its explanation as well at the same time. But even then one usually would not want to pay the time and memory cost of adding the lemma as a new clause—since in many cases this is going to be wasted work and space. One could simply store the lemma as a passive clause, that is, not active in the DPLL procedure, or store some information on how to compute it later.

5.2. HANDLING CONFLICT ANALYSIS IN THE CONTEXT OF THEORY PROPAGATION. In the previous subsection we have argued that it is preferable to generate explanations only at the moment they are needed for conflict analysis. Here we analyze the possible problems that arise in doing so, and discuss when and how it is still possible to compute a backjump clause. In a state of the form $M_1\, l\, M_2\, l'\, M_3 \parallel F$, we say that l is *older* than l', and that l' is *newer* than l.

Too new explanations:
Let us first revisit Example 5.1. After the four steps, where Clause 3 is conflicting, if $Solver_T$ is asked to compute the explanation of $a \neq b$, it can also return $g(a) \neq g(b)$, instead of the "real" explanation $h(a) = h(c) \vee c \neq b \vee a \neq b$. Indeed $a \neq b$ is a T-consequence of $g(a) \neq g(b)$ as well. But $g(a) \neq g(b)$ is a *too new explanation*: it did not even belong to the partial assignment at the time $a \neq b$ was propagated, and was in fact deduced by UnitPropagate from $a \neq b$ itself and Clause 1. Too new explanations are problematic because they can cause cycles in the conflict graph:

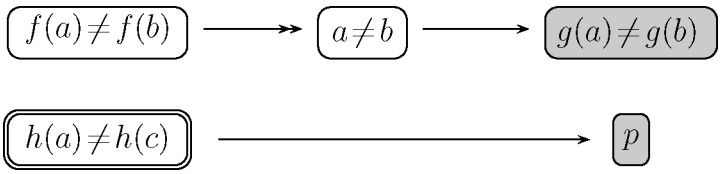


For the conflict graph above, the backwards resolution process computing the backjump clause does in fact loop:

$$\dfrac{g(a)=g(b)\vee \mathbf{a}\neq\mathbf{b} \qquad \dfrac{a=b\vee \mathbf{g(a)}\neq\mathbf{g(b)} \qquad \dfrac{h(a)=h(c)\vee \mathbf{p} \quad g(a)=g(b)\vee\neg\mathbf{p}}{\mathbf{g(a)}=\mathbf{g(b)}\vee h(a)=h(c)}}{h(a)=h(c)\vee \mathbf{a}=\mathbf{b}}}{g(a)=g(b)\ \vee\ h(a)=h(c)}$$

Therefore, to make sure that a backjump clause can be found, $Solver_T$ should never return too new explanations. A sufficient condition is to require that all literals in the explanation of a literal l be older than l. In other words, if the current state is of the form $M\ l\ N \parallel F$, then all literals in the explanation of l should occur in M.

Too old explanations:
In our example, when $Solver_T$ was asked to compute the explanation of $a \neq b$, it could also have returned $f(a) \neq f(b)$. This literal was already available before $a \neq b$ was obtained, but, as mentioned in Section 4.4, $Solver_T$ might have failed to detect $a \neq b$ as a negative consequence of it. It is interesting to observe that, with $f(a) = f(b) \vee a \neq b$ as the explanation of $a \neq b$, the resulting conflict graph has no unique implication point (UIP). In fact, there is not even a path from the current decision literal $h(a) \neq h(c)$ to the conflicting literal (of the current decision level) $g(a) \neq g(b)$:



However, looking at what happens with the backwards resolution procedure, one can see that it still produces a backjump clause, that is, a clause with exactly one literal from the current decision level:

$$\dfrac{f(a)=f(b)\vee \mathbf{a}\neq\mathbf{b} \qquad \dfrac{a=b\vee \mathbf{g(a)}\neq\mathbf{g(b)} \qquad \dfrac{h(a)=h(c)\vee \mathbf{p} \quad g(a)=g(b)\vee\neg\mathbf{p}}{\mathbf{g(a)}=\mathbf{g(b)}\vee h(a)=h(c)}}{h(a)=h(c)\vee \mathbf{a}=\mathbf{b}}}{h(a)=h(c)\ \vee\ f(a)=f(b)}$$

The following theorem states that in the backwards resolution process too old explanations are never a problem. It follows that just disallowing too new explanations suffices to guarantee that a backjump clause is always found.

THEOREM 5.2. *Assume that for any state of the form $M \parallel F$ and for any l in M due to* TheoryPropagate*, the explanation of l produced by $Solver_T$ contains no literals newer than l in M. Then, if some clause C is conflicting in a state S, either* Fail *applies to S, or else the backwards resolution process applied to C reaches a backjump clause.*

PROOF. Let d be the largest of the decision levels of the literals in C, and let D be the (nonempty) set of all literals of C that have become false at decision level d. If D is a singleton, C itself is a backjump clause. Otherwise, we can apply the

backwards conflict resolution process, resolving away literals of decision level d, until we reach a backjump clause having exactly one literal of level d. This process always terminates because each resolution step replaces a literal of decision level d by a finite number (zero in the case of a too old explanation) of strictly older literals of level d. The process is also guaranteed to produce a clause with just one literal of decision level d because, except for the decision literal itself, every literal of decision level d is resolvable. □

Note that the previous theorem is rather general by making no assumptions on the strategy followed in applying the DPLL rules. Also note that the theorem holds in the purely propositional case a well, where the theory T is empty and the theorem's assumption is vacuously true as TheoryPropagate never applies. Its generality entails that, for instance, one can apply Decide even in the presence of a conflicting clause, or if UnitPropagate also applies. In contrast, in Zhang and Malik [2003], the correctness proof of the Chaff algorithm assumes the fixed standard strategy in which unit propagation is done exhaustively before making any new decisions, which is considered an "important invariant". Theorem 5.2 instead shows that it is unproblematic for conflict analysis if a literal l is unit propagated at a decision level d when in fact it could have been propagated already at an earlier level. The reason is simply that in the backwards resolution step resolving on l replaces it by zero literals of level d, in perfect analogy to what happens to theory propagated literals with a too old explanation.

5.3. THE DEGREE OF EAGERNESS BY WHICH THEORY PROPAGATION SHOULD BE PERFORMED. So far, we have seen two possible strategies for theory propagation.

The first one, which we defined for Difference Logic, requires that $Solver_T$ returns *all* theory consequences (Section 4.2). In that strategy, TheoryPropagate is invoked each time a new literal is added to the current partial assignment. This is done to ensure that the partial assignment never becomes T-inconsistent.

The second strategy, defined for EUF logic, assumes that $Solver_T$ may return only some subset of the theory consequences, and applies TheoryPropagate only if no rule other than TheoryPropagate or Decide is applicable (Section 4.4).

However, there may also be expensive theories where one does not want to do full theory propagation (or check T-consistency) before every Decide step, but instead invoke it in some cheaper, incomplete way. The complete check is only required at the leaves of the search tree, that is, each time a propositional model has been found, in order to decide its T-consistency (this coincides with what is done in the naive lazy approach). The MathSAT approach [Bozzano et al. 2005] is based on a similar hierarchical view, where cheaper checks are performed more eagerly than expensive ones.

6. *Experiments with an Implementation of DPLL(T)*

We have experimented the DPLL(T) architecture with various implementations collaboratively developed in Barcelona and Iowa. We describe here our most advanced implementation, Barcelogic, developed mostly in Barcelona. The system follows the strategies presented in Section 4, and its solvers are as described in Section 4.3 and in Nieuwenhuis and Oliveras [2003, 2005b]. Its DPLL(X) engine implements state-of-the-art techniques such as the two-watched literal scheme for unit propagation, the first-UIP learning scheme, and VSIDS-like decision heuristics

[Moskewicz et al. 2001; Zhang et al. 2001]. The T-Forget rule is currently applied by DPLL(X) after each restart, removing a portion of the learned clauses according to their activity level [Goldberg and Novikov 2002], defined as the number of times they were involved in a conflict since the last restart.

6.1. THE 2005 SMT COMPETITION. The effectiveness of Barcelogic was shown at the 2005 SMT Competition [Barrett et al. 2005]. The competition used problems from the SMT-LIB library [Tinelli and Ranise 2005], a fairly large collection of benchmarks (around 1300) coming from such diverse areas as software and hardware verification, bounded model checking, finite model finding, and scheduling. These benchmarks were in the standard format of SMT-LIB [Ranise and Tinelli 2003], and were classified into 7 competition divisions according to their background theory and some additional syntactic restrictions. For each division, around 50 benchmarks were randomly chosen and given to each entrant system with a time limit of 10 minutes per benchmark.

Barcelogic entered and won all four divisions for which it had a theory solver: EUF, IDL and RDL (respectively, integer and real Difference Logic), and UFIDL (combining EUF and IDL). At least 10 systems participated in each of these divisions. Among the competitors were well-known SMT solvers such as SVC [Barrett et al. 1996], CVC [Barrett et al. 2002], CVC-Lite [Barrett and Berezin 2004], MathSAT [Bozzano et al. 2005], and two very recent successors of ICS [Filliâtre et al. 2001]: Yices (by Leonardo de Moura) and Simplics (by Dutertre and de Moura). Apart from EUF and Difference Logic, these systems also support other theories such as arrays (except MathSAT), and linear arithmetic (SVC only over the reals).

It is well-known that in practical problems over richer (combined) theories usually a large percentage of the work still goes into EUF and Difference Logic. For example, in [Bozzano et al. 2005] it is mentioned that in many calls a general solver is not needed: "very often, the unsatisfiability can be established in less expressive, but much easier, sub-theories". Similarly, Seshia and Bryant [2004], which deals with quantifier-free Presburger arithmetic, states that it has been found by them and others that literals are "mainly" difference logic.

The competition was run on 2.6 GHz, 512 MB, Pentium 4 Linux machines, with a 512 KB cache. For each division, the results of the best three systems are shown in the following table, where **Total time** is the total time in seconds spent by each system, with a timeout of 600 seconds, and **Time solved** is the time spent on the solved problems only:

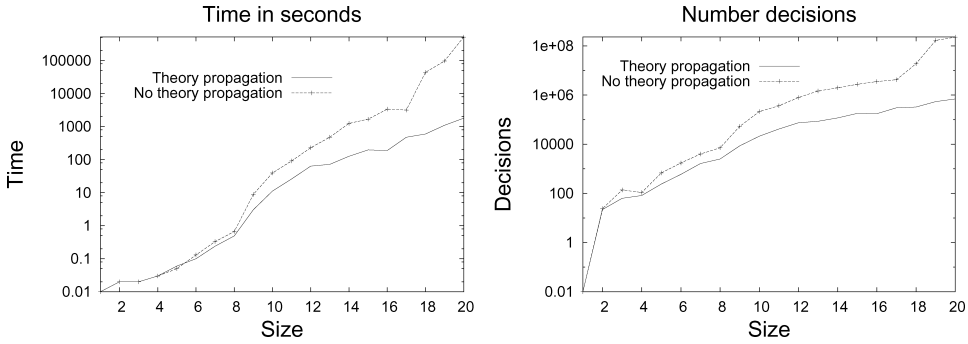
	Top-3 systems	# problems solved	Total time	Time solved
EUF (50 problems):	Barcelogic	39	8358	1758
	Yices	37	9601	1801
	MathSAT	33	12386	2186
RDL (50 pbms.):	Barcelogic	41	6341	940
	Yices	37	9668	1868
	MathSAT	37	10408	2608
IDL (51 pbms.):	Barcelogic	47	3531	1131
	Yices	47	4283	1883
	MathSAT	46	4295	1295
UFIDL (49 pbms.):	Barcelogic	45	2705	305
	Yices	36	9789	1989
	MathSAT	22	17255	1055

Not only did Barcelogic solve more problems than each of the other systems, it also did so in considerably less time, even—and in spite of the fact that it solved more problems—if only the time spent on the solved problems is counted.

6.2. EXPERIMENTS ON THE IMPACT OF THEORY PROPAGATION. In our experience, the overhead produced by theory propagation is almost always compensated by a significant reduction of the search space. In Ganzinger et al. [2004] we presented extensive experimental results showing its effectiveness in our DPLL(T) approach for EUF logic. In Nieuwenhuis and Oliveras [2005a] we discussed a large number of experiments for Difference Logic, with additional emphasis on the good scaling properties of the approach. The new SMT solver Yices now also heavily relies on theory propagation.

Of course, theory propagation may not pay off in certain specific problems where the theory plays an insignificant role, that is, where reasoning is done almost entirely at the Boolean level. Such situations can be detected on the fly by computing the percentage of conflicts caused by theory propagations. If this number is very low, theory propagation can be switched off automatically, or applied more lazily, to speed up the computation. (This is done in a forthcoming release of our system.)

In the following two figures, Barcelogic with and without theory propagation is compared, on the same type of machine as in the previous subsection, in terms of run time (in seconds) and number of decisions (applications of Decide) on a typical real-world Difference Logic suite (fisher6-mutex, see Tinelli and Ranise [2005]), consisting of 20 problems of increasing size.



The figures show the typical behavior on the larger problems where the theory plays a significant role: both the run time and the number of decisions are orders of magnitude smaller in the version with theory propagation (note that times and decisions are plotted on a logarithmic scale). In both cases, the DPLL(X) engine used was exactly the same, although in the exhaustive theory case some parts of the code were never executed (e.g., theory lemma learning).

6.3. EXPERIMENTS COMPARING BARCELOGIC WITH THE EAGER APPROACH. For completeness, we finally compare Barcelogic with UCLID, the best-known tool implementing the eager translation approach to SMT [Lahiri and Seshia 2004]. We show below run time results (in seconds) for three typical series of benchmarks for UFIDL coming from different methods for pipelined processor verification given in [Manolios and Srinivasan 2005a, 2005b] (more precisely, for the BIDW case (i) flushing, (ii) commitment good MA and (iii) commitment GFP). The benchmarks

were run on the same type of machine as in the previous two subsections, but this time with a one hour timeout. We used Siege [Ryan 2004] as the final SAT solver for UCLID, since it performed better than any other available SAT solver on these problems.

	UCLID	BLogic	UCLID	BLogic	UCLID	BLogic
6-stage:	258	1	3596	5	19	1
7-stage:	835	3	>3600	8	58	1
8-stage:	3160	15	>3600	18	226	1
9-stage:	>3600	23	>3600	18	664	1
10-stage:	>3600	54	>3600	29	>3600	2

We emphasize that these results are typical for the pipelined processor verification problems coming from this source, a finding that has also independently been reproduced by P. Manolios (private communication). We refer the reader to the results given in Ganzinger et al. [2004], showing that our approach also dominates UCLID in the pure EUF case, as well as for EUF with *integer offsets* (interpreted successor and predecessor symbols).

7. *Conclusions*

We have shown that the Abstract DPLL formalism introduced here can be very useful for understanding and formally reasoning about a large variety of DPLL-based procedures for SAT and SMT.

In particular, we have used it here to describe several variants of a new, efficient, and modular approach for SMT, called DPLL(T). Given a DPLL(X) engine, a DPLL(T) system for a theory T is obtained by simply plugging in the corresponding theory solver $Solver_T$, which must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements.

We are currently working on several—in our opinion very promising—ways to improve and extend both the abstract framework and the DPLL(T) architecture.

The abstract framework can be extended to deal more effectively with theories where the satisfiability of conjunctions of literals is already NP-hard by lifting, from the theory solver to the DPLL(X) engine, some or all of the case analysis done by the theory solver. Along those lines, the framework can also be nicely extended to a Nelson-Oppen style combination framework for handling formulas over several theories. The resulting $DPLL(T_1, \ldots, T_n)$ architecture can deal modularly and efficiently with the combined theories.

Preliminary experiments reveal that other applications of the DPLL(T) framework can produce competitive decision procedures as well for completely different (at least on the surface) kinds of problems. For example, optimization aspects of problems such as pseudo-Boolean constraints can be nicely expressed and efficiently solved in this framework by recasting them as particular SMT problems.

ACKNOWLEDGMENTS. We would like to thank Roberto Sebastiani for a number of insightful discussions and comments on the lazy SMT approach and DPLL(T). We are also grateful to the anonymous referees for their helpful suggestions on improving this article.

REFERENCES

ALUR, R. 1999. Timed automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)* (Trento, Italy), N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York, 8–22.

ARMANDO, A., CASTELLINI, C., AND GIUNCHIGLIA, E. 2000. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning* (Durham, UK), S. Biundo and M. Fox, Eds. Lecture Notes in Computer Science, vol. 1809. Springer-Verlag, New York, 97–108.

ARMANDO, A., CASTELLINI, C., GIUNCHIGLIA, E., AND MARATEA, M. 2004. A SAT-based decision procedure for the Boolean combination of difference constraints. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*. Lecture Notes in Computer Science. Springer-Verlag, New York.

AUDEMARD, G., BERTOLI, P., CIMATTI, A., KORNILOWICZ, A., AND SEBASTIANI, R. 2002. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *Proceedings of the CADE-18*. Lecture Notes in Computer Science, vol. 2392. Springer-Verlag, New York, 195–210.

BALL, T., COOK, B., LAHIRI, S. K., AND ZHANG, L. 2004. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York. 457–461.

BARRETT, C., DE MOURA, L., AND STUMP, A. 2005. SMT-COMP: Satisfiability modulo theories competition. In *Proceedings of the 17th International Conference on Computer Aided Verification*, K. Etessami and S. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 20–23. (See www.csl.sri.com/users/demoura/smt-comp.)

BARRETT, C., DILL, D., AND STUMP, A. 2002. Checking satisfiability of first-order formulas by incremental translation into SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, New York.

BARRETT, C., DILL, D. L., AND LEVITT, J. 1996. Validity checking for combinations of theories with equality. In *Proceedings of the 1st International Conference on Formal Methods in Computer Aided Design*. Lecture Notes in Computer Science, vol. 1166. Springer-Verlag, New York, 187–201.

BARRETT, C. W. 2003. Checking validity of quantifier-free formulas in combinations of first-order theories. Ph.D. dissertation. Stanford University, Stanford, CA.

BARRETT, C. W., AND BEREZIN, S. 2004. CVC lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 515–518.

BAYARDO, R. J. J., AND SCHRAG, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)* (Providence, RI), 203–208.

BEAME, P., KAUTZ, H., AND SABHARWAL, A. 2003. On the power of clause learning. In *Proceedings of IJCAI-03, 18th International Joint Conference on Artificial Intelligence* (Acapulco, MX).

BONET, M. L., ESTEBAN, J. L., GALESI, N., AND JOHANNSEN, J. 2000. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput. 30*, 5, 1462–1484.

BOZZANO, M., BRUTTOMESSO, R., CIMATTI, A., JUNTTILA, T. V. ROSSUM, P., SCHULZ, S., AND SEBASTIANI, R. 2005. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference (TACAS)*. Lecture Notes in Computer Science, vol. 3440. Springer-Verlag, New York, 317–333.

BRYANT, R., GERMAN, S., AND VELEV, M. 2001. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Computational Logic 2*, 1, 93–134.

BRYANT, R., LAHIRI, S., AND SESHIA, S. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, New York.

BRYANT, R. E., AND VELEV, M. N. 2002. Boolean satisfiability with transitivity constraints. *ACM Trans. Computational Logic 3*, 4, 604–627.

BURCH, J. R., AND DILL, D. L. 1994. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 818. Springer-Verlag, New York. 68–80.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Commun. ACM 5*, 7, 394–397.

DAVIS, M., AND PUTNAM, H. 1960. A computing procedure for quantification theory. *J ACM 7*, 201–215.

DE MOURA, L., AND RUEß, H. 2002. Lemmas on demand for satisfiability solvers. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*. 244–251.

DE MOURA, L., AND RUESS, H. 2004. An experimental evaluation of ground decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 162–174.

DE MOURA, L., RUEß, H., AND SHANKAR, N. 2004. Justifying equality. In *Proceedings of the 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning* (Cork, Ireland).

DOWNEY, P. J., SETHI, R., AND TARJAN, R. E. 1980. Variations on the common subexpressions problem. *J. ACM 27*, 4, 758–771.

EÉN, N., AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 502–518.

FILLIÂTRE, J.-C., OWRE, S., RUEß, H., AND SHANKAR, N. 2001. ICS: Integrated canonization and solving (tool presentation). In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'2001)*. G. Berry, H. Comon, and A. Finkel, Eds. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, New York, 246–249.

FLANAGAN, C., JOSHI, R., OU, X., AND SAXE, J. B. 2003. Theorem proving using lazy proof explanation. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2725. Springer-Verlag, New York.

GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2004. DPLL(T): Fast Decision Procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 175–188.

GOLDBERG, E., AND NOVIKOV, Y. 2002. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Symposium on Design, Automation, and Test in Europe (DATE '02)*. 142–149.

HODGES, W. 1993. *Model Theory*. Enclyclopedia of mathematics and its applications, vol. 42. Cambridge University Press, Cambridge, MA.

JAFFAR, J., AND MAHER, M. 1994. Constraint Logic Programming: A Survey. *J. Logic Prog. 19/20*, 503–581.

LAHIRI, S. K., AND SESHIA, S. A. 2004. The UCLID Decision Procedure. In *Computer Aided Verification, 16th International Conference (CAV)*. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 475–478.

MANOLIOS, P., AND SRINIVASAN, S. K. 2005a. A computationally efficient method based on commitment refinement maps for verifying pipelined machines. In *Proceedings of the ACM IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. ACM, New York.

MANOLIOS, P., AND SRINIVASAN, S. K. 2005b. Refinement maps for efficient verification of processor models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE)*. IEEE Computer Society, 1304–1309.

MARQUES-SILVA, J., AND SAKALLAH, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput. 48*, 5 (May), 506–521.

MEIR, O., AND STRICHMAN, O. 2005. Yet another decision procedure for equality logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)* (Edimburgh, Scotland). K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 307–320.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

NIEUWENHUIS, R., AND OLIVERAS, A. 2003. Congruence Closure with Integer Offsets. In *Proceedings of the 10h International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, M. Vardi and A. Voronkov, Eds. Lecture Notes in Artificial Intellegence, vol. 2850. Springer-Verlag, New York, 2850. 78–90.

NIEUWENHUIS, R., AND OLIVERAS, A. 2005a. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)* (Edimburgh, Scotland). K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 321–334.

NIEUWENHUIS, R., AND OLIVERAS, A. 2005b. Proof-producing congruence closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA'05)* (Nara, Japan). J. Giesl, Ed. Lecture Notes in Computer Science, vol. 3467. Springer-Verlag, New York, 453–468.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2005. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of the 11th International Conference Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. F. Baader and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 3452. Springer-Verlag, New York, 36–50.

PNUELI, A., RODEH, Y., SHTRICHMAN, O., AND SIEGEL, M. 1999. Deciding equality formulas by small domains instantiations. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York. 455–469.

RANISE, S., AND TINELLI, C. 2003. The SMT-LIB format: An initial proposal. In *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*. Miami.

RYAN, L. 2004. Efficient algorithms for clause-learning SAT solvers. M.S. dissertation, School of Computing Science, Simon Fraser University.

SCHRIJVER, A. 1987. *Theory of Linear and Integer Programming*. Wiley, New York.

SESHIA, S., LAHIRI, S., AND BRYANT, R. 2003. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proceedings of the 40th Design Automation Conference (DAC)*. 425–430.

SESHIA, S. A. 2005. Adaptive eager Boolean encoding for arithmetic reasoning in verification. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA.

SESHIA, S. A., AND BRYANT, R. E. 2004. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Computer Society Press, Los Alamitos, CA, 100–109.

STRICHMAN, O. 2002. On solving presburger and linear arithmetic with SAT. In *Proceedings of the Formal Methods in Computer-Aided Design, 4th International Conference (FMCAD 2002)* (Portland, OR). M. Aagaard and J. W. O'Leary, Eds. Lecture Notes in Computer Science, vol. 2517. Springer-Verlag, New York, 160–170.

STRICHMAN, O., SESHIA, S. A., AND BRYANT, R. E. 2002. Deciding separation formulas with SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol 2404. Springer-Verlag, New York. 209–222.

STUMP, A., AND TAN, L.-Y. 2005. The algebra of equality proofs. In *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05* (Nara, Japan). J. Giesl, Ed. Lecture Notes in Computer Science, vol. 3467. Springer-Verlag, New York, 469–483.

TALUPUR, M., SINHA, N., STRICHMAN, O., AND PNUELI, A. 2004. Range allocation for separation logic. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*. (Boston, MA July 13–17). Lecture Notes in Computer Science, Springer-Verlag, New York, 148–161.

TINELLI, C. 2002. A DPLL-based calculus for ground satisfiability modulo theories. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence*. Lecture Notes in Artificial Intelligence, vol. 2424. Springer-Verlag, New York, 308–319.

TINELLI, C., AND RANISE, S. 2005. *SMT-LIB: The Satisfiability Modulo Theories Library*. http://goedel.cs.uiowa.edu/smtlib/.

ZHANG, H. 1997. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*. Springer-Verlag, New York, 272–275.

ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. W., AND MALIK, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*. 279–285.

ZHANG, L., AND MALIK, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the 2003 Design, Automation and Test in Europe Conference (DATE 2003)*. IEEE Computer Society Press, Los Alamitos, CA, 10880–10885.

RECEIVED DECEMBER 2005; REVISED FEBRUARY 2006 AND JULY 2006; ACCEPTED NOVEMBER 2006

A Machine Program for Theorem-Proving†

Martin Davis, George Logemann, and Donald Loveland

Institute of Mathematical Sciences, New York University

The programming of a proof procedure is discussed in connection with trial runs and possible improvements.

In [1] is set forth an algorithm for proving theorems of quantification theory which is an improvement in certain respects over previously available algorithms such as that of [2]. The present paper deals with the programming of the algorithm of [1] for the New York University, Institute of Mathematical Sciences' IBM 704 computer, with some modifications in the algorithm suggested by this work, with the results obtained using the completed algorithm. Familiarity with [1] is assumed throughout.

Changes in the Algorithm and Programming Techniques Used

The algorithm of [1] consists of two interlocking parts. The first part, called the *QFl-Generator*, generates (from the formula whose proof is being attempted) a growing propositional calculus formula in conjunctive normal form, the "quantifier-free lines." The second part, the *Processor*, tests, at regular stages in its "growth," the consistency of this propositional calculus formula. An inconsistent set of quantifier-free lines constitutes a proof of the original formula.

The algorithm of [1] used in testing for consistency proceeded by successive elimination of atomic formulas, first eliminating *one-literal* clauses (one-literal clause rule), and then atomic formulas all of whose occurrences were positive or all of whose occurrences were negative (affirmative-negative rule). Finally, the remaining atomic formulas were to have been eliminated by the rule:

III. *Rule for Eliminating Atomic Formulas.* Let the given formula F be put into the form

$$(A \vee p) \ \& \ (B \vee \bar{p}) \ \& \ R$$

where A, B, and R are free of p. (This can be done simply by grouping together the clauses containing p and "factoring out" occurrences of p to obtain A, grouping the clauses containing $\bar{p}$ and "factoring out" $\bar{p}$ to obtain B, and grouping the remaining clauses to obtain R.) Then F is inconsistent if and only if $(A \vee B) \ \& R$ is inconsistent.

After programming the algorithm using this form of Rule III, it was decided to replace it by the following rule:

III*. *Splitting Rule.* Let the given formula F be put in the form

$$(A \vee p) \ \& \ (B \vee \bar{p}) \ \& R$$

where A, B, and R are free of p. Then F is inconsistent if and only if $A \ \& R$ and $B \ \& R$ are both inconsistent.

JUSTIFICATION OF RULE III*. For[1] $p = 0$, $F = A \& R$; for $p = 1$, $F = B \ \& R$.

The forms of Rule III are interchangeable; although theoretically they are equivalent, in actual applications each has certain desirable features. We used Rule III* because of the fact that Rule III can easily increase the number and the lengths of the clauses in the expression rather quickly after several applications. This is prohibitive in a computer if ones fast access storage is limited. Also, it was observed that after performing Rule III, many duplicated and thus redundant clauses were present. Some success was obtained by causing the machine to systematically eliminate the redundancy; but the problem of total length increasing rapidly still remained when more complicated problems were attempted. Also use of Rule III can seldom yield new one-literal clauses, whereas use of Rule III* often will.

In programming Rule III*, we used auxiliary tape storage. The rest of the testing for consistency is carried out using only fast access storage. When the "Splitting Rule" is used one of the two formulas resulting is placed on tape. Tape memory records are organized in the cafeterial stack-of-plates scheme: the last record written is the first to be read.

In the program written for the IBM 704, the matrix and conjunction of quantifier-free lines are coded into cross-referenced associated (or linked) memory tables by the QFL-Generator and then analyzed by the Processor. In particular, the QFL-Generator is programmed to read in the matrix M in suitably coded Polish (i.e., "parenthesis-free") form. The conversion to a quantifier-free matrix in conjunctive normal form requires, of course, a certain amount of pencil work on the formula, which could have been done by the computer. In doing this, we departed from [1], by not using prenex normal form. The steps are:

(1) Write all truth-functional connectives in terms of $\sim$, &, $\vee$.

(2) Move all $\sim$'s inward successively (using de Morgan laws) until they either are cancelled (with another $\sim$) or acting on an atomic formula.

(3) Now, replace existential quantifiers by function symbols (cf. [1], p. 205), drop universal quantifiers, and place in conjunctive normal form. A simple one-to-one assembler was written to perform the final translation of the matrix M into octal numbers.

It will be recalled that the generation of quantifier-free lines is accomplished by successive substitutions of "constants" for the variables in the matrix M. In the program

† The research reported in this document has been sponsored by the Mathematical Sciences Directorate, Air Force Office of Scientific Research, under Contract No. AF 49(638)-777.

[1] As in [1], 1 stands for "truth", and 0 for "falsehood".

the constants are represented by the successive positive integers.

For a matrix containing n individual variables, the n-tuples of positive integers are generated in a sequence of increasing norm such that all n-tuples with a given norm are in decreasing n-ary numerical order. Here we define the norm of $(j_1, \cdots, j_n) = j_1 + \cdots + j_n = \|j_i\|$. Other norms could have been used. For example, Gilmore [2] takes for $\|j_i\|$ the maximum of $j_i, \cdots, j_n$. In [1] a more complicated norm is indicated.

Substitutions of successive n-tuples into the matrix cause new constants to appear in the matrix. The program numbers constants in their order of appearance. Thus, the constants are ordered by the program in a manner depending upon the input data. By rearranging the clauses of a formula a different order would in general be created. In some cases, whether or not the program could actually prove the validity of a given formula (without running out of fast access storage) depended on how one shuffled the punched-data deck before reading it into the assembler! Thus, the variation in ordering of constants did affect by a factor of 10 (from 500 to 5000) the number of lines needed to prove the validity of:

$$(e)(Ed)(x)(y)[S(x, y, d) \rightarrow T(x, y, e)] \rightarrow (e)(x)(Ed)(y)[S(x, y, d) \rightarrow T(x, y, e)]$$

(This valid formula may be thought of as asserting that uniform continuity implies continuity if we set:

$$S(x, y, d) \leftrightarrow |x - y| < d$$

$$T(x, y, e) \leftrightarrow |f(x) - f(y)| < e.)$$

In storing the quantifier-free lines, two tables are used. The first, called the *conjunction table*, is a literal image of the quantifier-free lines in which one machine word corresponds to one literal, i.e., to p or $\sim p$ where p is an atomic formula. The lines in the second, or *formula table* are themselves heads of two chain lists giving the occurrences of p and $\sim p$ respectively in the conjunction table. In addition, included for formula p in the formula table are counts of the number of clauses in which p and $\sim p$ occur and total number of all literals in these clauses; the formula table is itself a two-way linked list. A third short list of those literals is kept in which are entered all formulas to which the one-lateral clause and affirmative-negative rules apply; this is called the *ready list*. If the program tries to enter p and $\sim p$ into the ready list, an inconsistency has been found; the machine stops.

The totality of the processing rules requires only two basic operations: a subroutine to *delete* the occurrences of a literal p or $\sim p$ from the quantifier-free lines, and a routine to *eliminate* from them all the clauses in which p or $\sim p$ occur.

We may observe that only the deletion program can create new one-literal clauses, and likewise applications of the affirmative-negative rule can come only from the elimination program.

The machine thus performs the one literal-clause and affirmative-negative rules as directed by the ready list until the ready list is empty. It is possible that the choice of p to be eliminated first is quite critical in determining the length of computation required to reach a conclusion: a program to choose p is used, but no tests were made to vary this segment of the program beyond a random selection, namely the first entry in the formula table. To perform Rule III*, one saves the appropriate tables with some added reference information in a tape record, then performs an elimination on $\sim p$ and a deletion on p. At a consequent discovery of consistency, one must generate more quantifier free lines; the QFL-generator is recalled. Otherwise, at finding an inconsistency, the machine must check to see if there are any records on the Rule III* tape: if none, the quantifier-free lines were inconsistent; otherwise, it reads in the last record.

If one uses Rule III (which we did in an early version of our program), an entirely different code is needed. The problem is precisely that of mechanizing the application of the distributive law.

Results Obtained in Running the Program

At the time the programming of the algorithm was undertaken, we hoped that some mathematically meaningful and, perhaps nontrivial, theorems could be proved. The actual achievements in this direction were somewhat disappointing. However, the program performed as well as expected on the simple predicate calculus formulas offered as fare for a previous proof procedure program. (See Gilmore [1].) In particular, the well-formed formula

$$(Ex)(Ey)(z)\{F(x, y) \rightarrow (F(y, z) \ \& F(z, z))) \ \& ((F(x, y) \ \& G(x, y)) \rightarrow (G(x, z) \ \& G(z, z)))\}$$

which was beyond the scope of Gilmore's program was proved in under two minutes with the present program. Gilmore's program was halted at the end of 7 "substitutions", (quantifier-free lines) after an elapsed period of about 21 minutes. It was necessary for the present program to generate approximately 60 quantifier-free lines before the inconsistency appeared.[2] Indeed, the "uniform continuity implies continuity" example mentioned above required over 500 quantifier-free lines to be generated and was shown to be valid in just over two minutes. This was accomplished by nearly filling the machine to capacity with generated quantifier-free lines (2000 lines in this case) before applying any of the rules of reduction.

Rather than describe further successes of the program, it will be instructive to consider in detail a theorem that the program was incapable of proving and to examine the cause for this. This particular example is one the authors originally had hoped the program could prove, an elementary group theory problem. In essence, it is to show that in a group a left inverse is also a right inverse.

[2] In [1], a hand calculation of this example using the present scheme showed inconsistency at 25 quantifier-free lines. The discrepancy is due to a different rule for generation of constants.

It is, in fact, quite easy to follow the behavior of the proof procedure on this particular example as it parallels the usual approach to the problem. The problem may be formulated as follows:

Axioms: 1. $e \cdot x = x$

2. $I(x) \cdot x = e$

3. $(x \cdot y) \cdot z = w \Rightarrow x \cdot (y \cdot z) = w$

4. $x \cdot (y \cdot z) = w \Rightarrow (x \cdot y) \cdot z = w$

Conclusion: $x \cdot I(x) = e$

The letter e is interpreted as the identity element and the function I as the inverse function. The associative law has been split into two clauses for convenience.

A proof is as follows:

1. $I(I(x)) \cdot I(x) = e$ by Axiom 2
2. $e \cdot x = x$ by Axiom 1
3. $I(x) \cdot x = e$ by Axiom 2
4. $I(I(x)) \cdot e = x$ by Axiom 3, taking $(I(I(x)), I(x), x)$ for (x, y, z)
5. $e \cdot I(x) = I(x)$ by Axiom 1
6. $I(I(x)) \cdot I(x) = e$ by Axiom 2
7. $I(I(x)) \cdot e = x$ step 4
8. $x \cdot I(x) = e$ by Axiom 4, taking $(I(I(x)), e, I(x))$ for (x, y, z)

Step 8 is the desired result.

To formalize this proof would require adjoining axioms of equality. To avoid this, one can introduce the predicate of three arguments $P(x, y, z)$, interpreted as $x \cdot y = z$. The theorem reformulated becomes:

Axioms: 1. $P(e, x, x)$

2. $P(I(x), x, e)$

3. $\sim P(x, y, u) \vee \sim P(u, z, w) \vee \sim P(y, z, v) \vee P(x, v, w)$

4. $\sim P(y, z, v) \vee \sim P(x, v, w) \vee \sim P(x, y, u) \vee P(u, z, w)$

Conclusion: $P(x, I(x), e)$.

The theorem to be proved valid is the implication of the conjunction of the four axioms with the conclusion, the universal quantifiers appearing outside the matrix.

To complete the preparation of the well-formed formula for encoding for the assembler, it is necessary to negate the conclusion. (cf. [1], p. 204.)

The single existential quantifier has no dependence on the universal quantifiers, hence leads to the constant function s when this existential quantifier is replaced by a function symbol. (cf. [1], p. 205.]

The conclusion then becomes

$$\sim P(s, I(s), e).$$

The conjunction of this with the four axioms gives the desired form.

As seen from the proof previously noted the quantifier-free clauses needed to produce the inconsistency are

1. $P(I(I(s)), I(s), e)$
2. $P(e, s, s)$
3. $P(I(s), s, e)$
4. $\sim P(I(I(s)), I(s), e) \vee \sim P(e, s, s) \vee \sim P(I(s), s, e) \vee P(I(I(s)), e, s)$
5. $P(e, I(s), I(s))$
6. $\sim P(e, I(s), I(s)) \vee \sim P(I(I(s)), I(s), e) \vee \sim P(I(I(s), e, s) \vee P(s, I(s), e)$
7. $\sim P(s, I(s), e)$

(It is quite clear in this case that successive applications of the one-literal clause rule reducing this set to

$$P(s, I(s), e) \,\&\, \sim P(s, (I(s), e).)$$

The question to be considered is: how many quantifier-free lines must be generated by the present program to realize these required lines? The constants are generated in the following:

1. e
2. s
3. $I(s)$
4. $I(e)$
5. $I(I(s))$

etc.

(The constants are identified directly with their index e.g. the 6-tuple (1, 1, 1, 1, 1, 1) represents (e, e, e, e, e, e). As this is the first substitution, the program assigns in order, reading the well-formed formula backwards and from the inside out for nesting functions: e, s, $I(s)$, $I(e)$, $I(I(s))$. The $I(I(s))$ appears when x is assigned $I(s)$, no new entries occurring until this time. Note that there are 6 free variables (u, v, w, x, y, z) in the matrix).

The program generates the needed n-tuples by producing all possible n-tuples of integers whose sum N of entries is fixed, $N = n, n + 1, \cdots$. Thus it is only necessary to consider the n-tuple which has the maximum sum of entries. In this case, the substitution $u = s$, $v = I(s)$, $w = e$, $x = I(I(s))$, $y = e$, $z = I(s)$ (required for axiom 4 to produce the clause 6 in the "proof" above in a quantifier-free line) gives the n-tuple with maximum sum. The n-tuple is seen to be (2, 3, 1, 5, 1, 3), the sum equals 15. The combinatorial expression $\binom{N}{n}$ gives the total number of n-tuples of positive integers whose sum is less than or equal[3] to N.

[3] To see this, consider a sequence of $N+1$ ones. Flag n of these. The flag is to be interpreted "sum all 1's, including the flagged 1, to the next flag and consider this sum as an entry in the n-tuple". Placing an unflagged 1 on the extreme left, leaving it fixed, consider the possible permutations of all other symbols. The different sequences total $\binom{N}{n}$ and, regarding the set of 1's starting with the last flagged 1 as overflow, this is seen to represent precisely the desired n-tuples.

It is seen that to prove this theorem at least $\binom{14}{6} = 3003$ lines must be generated and that the inconsistency will be found on or before $\binom{15}{6} = 5005$ lines have been generated.[4]

The present program generated approximately 1300 quantifier-free lines. This number of quantifier-free lines was accomplished holding all major tables simultaneously in core memory, limited to 32,768 "words". (This was done to insure a reasonable time factor for any problem, within possible scope of the program. For this reason also, the entire program was coded in SAP with many time-saving devices employed.)

The authors believe that a reprogramming to make use of tape storage of tables might realize a factor of 4 for the total number of quantifier-free lines attainable before running time became prohibitive. This would be just sufficient for this problem. That realizing this extra capacity is really uninteresting is seen by noting that if the conclusion was placed before the axioms, altering the validity of the matrix not at all, the element $I(c)$ would be generated before $I(s)$ and the needed n-tuple would sum to 16. Then $\binom{16}{6} = 8008$ becomes the upper bound, beyond the capacity of the projected program. Other formulations of the same problem result in quite unapproachable figures for the number of quantifier-free lines needed. (For another example illustrating the same situation, see Prawitz [3].)

The existing program allows one to think of working with a capacity of 1000 or 2000 quantifier-free lines instead of a capacity of 10 or 20, the previous limit. The time required to generate additional quantifier-free lines is independent of the number of quantifier-free lines already existing. Against this linear growth of number of quantifier-free lines generated, there is, in a meaningful sense, an extreme nonlinear growth in the number of quantifier-free lines to be considered with increasingly more "difficult" problems. This is true of simple enumeration schemes of the nature considered here. It seems that the most fruitful future results will come from reducing the number of quantifier-free lines that need be considered, by excluding, in some sense, "irrelevant" quantifier-free lines. Some investigation in this area has already been done (see Prawitz [3]).

[4] If the rule for generating n-tuples had been, for each m, to generate all n-tuples possible such that each entry assumes a positive integral value less than or equal to m, it is clear that at least 4^6 (= 4096) quantifier-free lines would be needed and 5^6 (= 15625) lines would suffice to guarantee a solution. If no more information were available, one sees an intuitive advantage, in this case, for using the previous method. In general, the authors see no preference for either method, in contrast to some previous suggestions that the latter method seemed intuitively better.

REFERENCES

1. Davis, Martin, and Putman, Hilary. A computing procedure for quantification theory. *J. ACM 7* (1960), 201–215.
2. Gilmore, P. C. A proof method for quantification theory. *IBM J. Res. Dev. 4* (1960), 28–35.
3. Prawitz, Dag. An improved proof procedure. *Theoria 26*, 2 (1960), 102–139.

Nonlinear Regression and the Solution of Simultaneous Equations

Robert M. Baer

University of California, Berkeley

If one has a set of observables $(z_1, \cdots, z_m)$ which are bound in a relation with certain parameters $(a_1, \cdots, a_n)$ by an equation $\zeta(z_1, \cdots ; a_1, \cdots) = 0$, one frequently has the problem of determining a set of values of the a_i which minimizes the sum of squares of differences between observed and calculated values of a distinguished observable, say z_m. If the solution of the above equation for z_m,

$$z_m = \eta(z_1, \cdots ; a_1, \cdots)$$

gives rise to a function η which is nonlinear in the a_i, then one may rely on a version of Gaussian regression [1, 2] for an iteration scheme that converges to a minimizing set of values. It is shown here that this same minimization technique may be used for the solution of simultaneous (not necessarily linear) equations.

Modifications of the technique, while necessary for convergence in some problems, are extraneous to the argument and shall be ignored. The Gaussian procedure may then be defined as follows.

If $a_i(h)$ denotes the values of the parameters at the hth iteration, then $a_i(h + 1) = a_i(h) + \epsilon_i(h)$, where the corrections $\epsilon_i(h)$ are the solution to the set of equations

$$\sum_j A_{ij}\epsilon_j + B_i = 0 \qquad (i = 1, \cdots, n) \tag{1}$$

where

$$A_{ij} = \sum_k \frac{\partial \eta_k}{\partial a_i}\frac{\partial \eta_k}{\partial a_j} \tag{2}$$

and

$$B_i = \sum_k [\eta_k - z_m(k)]\frac{\partial \eta_k}{\partial a_i}. \tag{3}$$

An Extensible SAT-solver
[extended version 1.2]

Niklas Eén, Niklas Sörensson

Chalmers University of Technology, Sweden
{een,nik}@cs.chalmers.se

Abstract. In this article, we present a small, complete, and efficient SAT-solver in the style of conflict-driven learning, as exemplified by CHAFF. We aim to give sufficient details about implementation to enable the reader to construct his or her own solver in a very short time. This will allow *users* of SAT-solvers to make domain specific extensions or adaptions of current state-of-the-art SAT-techniques, to meet the needs of a particular application area. The presented solver is designed with this in mind, and includes among other things a mechanism for adding arbitrary boolean constraints. It also supports solving a series of related SAT-problems efficiently by an incremental SAT-interface.

1 Introduction

The use of SAT-solvers in various applications is on the march. As insight on how to efficiently encode problems into SAT is increasing, a growing number of problem domains are successfully being tackled by SAT-solvers. This is particularly true for the *electronic design automation* (EDA) industry [BCCFZ99,Lar92]. The success is further magnified by current state-of-the-art solvers being extended and adapted to meet the specific characteristics of these problem domains [ARMS02,ES03].

However, modifying an existing solver, even with a thorough understanding of both the problem domain and of modern SAT-techniques, can become a time consuming and bewildering journey into the mysterious inner workings of a ten-thousand-line software package. Likewise, writing a solver from scratch can also be a daunting task, as there are numerous pitfalls hidden in the intricate details of a correct and efficient solver. The problem is that although the *techniques* used in a modern SAT-solver are well documented, the details necessary for an *implementation* have not been adequately presented before.

In the fall of 2002, the authors implemented the solvers SATZOO and SATNIK. In order to sufficiently understand the implementation tricks needed for a modern SAT-solver, it was necessary to consult the source-code of previous implementations.[1] We find that the material contained therein can be made more accessible, which is desirable for the SAT-community. Thus, the principal goal of this article is to bridge the gap between existing descriptions of SAT-techniques and their actual implementation.

We will do this by presenting the code of a minimal SAT-solver MINISAT, based on the ideas for conflict-driven backtracking [MS96], together with watched literals and dynamic variable ordering [MZ01]. The original C++ source code

[1] LIMMAT at http://www.inf.ethz.ch/personal/biere/projects/limmat/
ZCHAFF at http://www.ee.princeton.edu/~chaff/zchaff

(downloadable from `http://www.cs.chalmers.se/~een`) for MiniSat is under 600 lines (not counting comments), and is the result of rethinking and simplifying the designs of Satzoo and Satnik without sacrificing efficiency. We will present all the relevant parts of the code in a manner that should be accessible to anyone acquainted with either C++ or Java.

The presented code includes an incremental SAT-interface, which allows for a series of related problems to be solved with potentially huge efficiency gains [ES03]. We also generalize the expressiveness of the SAT-problem formulation by providing a mechanism for arbitrary *constraints* over boolean variables to be defined. Paragraphs discussing implementation alternatives are marked "[Discussion]" and can be skipped on a first reading.

From the documentation in this paper we hope it is possible for *you* to implement a fresh SAT-solver in your favorite language, or to grab the C++ version of MiniSat from the net and start modifying it to include new and interesting ideas.

2 Application Programming Interface

We start by presenting MiniSat's external interface, with which a user application can specify and solve SAT-problems. A basic knowledge about SAT is assumed (see for instance [MS96]). The types ***var***, ***lit***, and ***Vec*** for variables, literals, and vectors respectively are explained in detail in section 4.

class *Solver* – *Public interface*		
var	*newVar*	()
bool	*addClause*	(***Vec***⟨***lit***⟩ literals)
bool	*add...*	(...)
bool	*simplifyDB*	()
bool	*solve*	(***Vec***⟨***lit***⟩ assumptions)
Vec⟨***bool***⟩ model		– *If found, this vector has the model.*

The "*add...*" method should be understood as a place-holder for additional constraints implemented in an extension of MiniSat.

For a standard SAT-problem, the interface is used in the following way: Variables are introduced by calling *newVar()*. From these variables, clauses are built and added by *addClause()*. Trivial conflicts, such as two unit clauses $\{x\}$ and $\{\overline{x}\}$ being added, can be detected by *addClause()*, in which case it returns False. From this point on, the solver state is undefined and must not be used further. If no such trivial conflict is detected during the clause insertion phase, *solve()* is called with an empty list of assumptions. It returns False if the problem is *unsatisfiable*, and True if it is *satisfiable*, in which case the model can be read from the public vector "model".

The *simplifyDB()* method can be used before calling *solve()* to simplify the set of problem constraints (often called the *constraint database*). In our implementation, *simplifyDB()* will first propagate all unit information, then remove all satisfied constraints. As for *addClause()*, the simplifier can sometimes detect a

conflict, in which case FALSE is returned and the solver state is, again, undefined and must not be used further.

If the solver returns *satisfiable*, new constraints can be added repeatedly to the existing database and *solve()* run again. However, more interesting sequences of SAT-problems can be solved by the use of *unit assumptions*. When passing a non-empty list of assumptions to *solve()*, the solver temporarily assumes the literals to be true. After finding a model or a contradiction, these assumptions are undone, and the solver is returned to a usable state, even when *solve()* return FALSE, which now should be interpreted as *unsatisfiable under assumptions*.

For this to work, calling *simplifyDB()* before *solve()* is no longer optional. It is the mechanism for detecting conflicts independent of the assumptions – referred to as a *top-level* conflict from now on – which puts the solver in an undefined state. We wish to remark that the ability to pass unit assumptions to *solve()* is more powerful than it might appear at first. For an example of its use, see [ES03].

An alternative interface would be for *solve()* to return one of three values: *satisfiable, unsatisfiable,* or *unsatisfiable under assumptions*. This is indeed a less error-prone interface as there is no longer a pre-condition on the use of *solve()*. The current interface, however, represents the smallest modification of a non-incremental SAT-solver. The early non-incremental version of SATZOO was made compliant to the above interface by adding just 5 lines of code. [Discussion]

3 Overview of the SAT-solver

This article will treat the popular style of SAT-solvers based on the DPLL algorithm [DLL62], backtracking by conflict analysis and clause recording (also referred to as *learning*) [MS96], and boolean constraint propagation (BCP) using *watched literals* [MZ01].[2] We will refer to this style of solver as a *conflict-driven SAT-solver*.

The components of such a solver, and indeed a more general constraint solver, can be conceptually divided into three categories:

- **Representation.** Somehow the SAT-instance must be represented by internal data structures, as must any derived information.
- **Inference.** Brute force search is seldom good enough on its own. A solver also needs some mechanism for computing and propagating the direct implications of the current state of information.
- **Search.** Inference is almost always combined with search to make the solver complete. The search can be viewed as another way of deriving information.

A standard conflict-driven SAT-solver can represent *clauses* (with two literals or more) and *assignments*. Although the assignments can be viewed as unit-clauses, they are treated specially in many ways, and are best viewed as a separate type of information.

The only inference mechanism used by a standard solver is *unit propagation*. As soon as a clause becomes *unit* under the current assignment (all literals except

[2] This includes SAT-solvers such as: ZCHAFF, LIMMAT, BERKMIN.

one are false), the remaining unbound literal is set to true, possibly making more clauses unit. The process is continued until no more information can be propagated.

The search procedure of a modern solver is the most complex part. Heuristically, variables are picked and assigned values (*assumptions* are made), until the propagation detects a *conflict* (all literals of a clause have become false). At that point, a so called *conflict clause* is constructed and added to the SAT problem. Assumptions are then canceled by backtracking until the conflict clause becomes unit, from which point this unit clause is propagated and the search process continues.

MiniSat is extensible with arbitrary boolean constraints. This will affect the *representation*, which must be able to store these constraints; the *inference*, which must be able to derive unit information from these constraints; and the *search*, which must be able to analyze and generate conflict clauses from the constraints. The mechanism we suggest for managing general constraints is very lightweight, and by making the dependencies between the SAT-algorithm and the constraints implementation explicit, we feel it rather adds to the clarity of the solver than obscures it.

Propagation. The propagation procedure of MiniSat is largely inspired by that of Chaff [MZ01]. For each literal, a list of constraints is kept. These are the constraints that *may* propagate unit information (variable assignments) if the literal becomes True. For clauses, no unit information can be propagated until all literals except one have become False. Two unbound literals p and q of the clause are therefore selected, and references to the clause are added to the lists of $\overline{p}$ and $\overline{q}$ respectively. The literals are said to be *watched* and the lists of constraints are referred to as *watcher lists*. As soon as a watched literal becomes True, the constraint is invoked to see if information may be propagated, or to select new unbound literals to be watched.

A feature of the watcher system for clauses is that on backtracking, no adjustment to the watcher lists need to be done. Backtracking is therefore very cheap. However, for other constraint types, this is not necessarily a good approach. MiniSat therefore supports the optional use of *undo lists* for those constraints; storing what constraints need to be updated when a variable becomes unbound by backtracking.

Learning. The learning procedure of MiniSat follows the ideas of Marques-Silva and Sakallah in [MS96]. The process starts when a constraint becomes conflicting (impossible to satisfy) under the current assignment. The conflicting constraint is then asked for a set of variable assignments that make it contradictory. For a clause, this would be all the literals of the clause (which are False under a conflict). Each of the variable assignments returned must be either an *assumption* of the search procedure, or the result of some *propagation* of a constraint. The propagating constraints are in turn asked for the set of variable assignments that forced the propagation to occur, continuing the analysis backwards. The procedure is repeated until some termination condition is fulfilled, resulting in a set of variable assignments that implies the conflict. A clause prohibiting that particular assignment is added to the clause database. This *learnt*

clause must always, by construction, be implied by the original problem constraints.

Learnt clauses serve two purposes: they drive the backtracking (as we shall see) and they speed up future conflicts by "caching" the reason for the conflict. Each clause will prevent only a constant number of inferences, but as the recorded clauses start to build on each other and participate in the unit propagation, the accumulated effect of learning can be massive. However, as the set of learnt clauses increase, propagation is slowed down. Therefore, the number of learnt clauses is periodically reduced, keeping only the clauses that seem useful by some heuristic.

Search. The search procedure of a conflict-driven SAT-solver is somewhat implicit. Although a recursive definition of the procedure might be more elegant, it is typically described (and implemented) iteratively. The procedure will start by selecting an unassigned variable x (called the *decision variable*) and assume a value for it, say TRUE. The consequences of $x=$TRUE will then be propagated, possibly resulting in more variable assignments. All variables assigned as a consequence of x is said to be from the same *decision level*, counting from 1 for the first assumption made and so forth. Assignments made before the first assumption (decision level 0) are called *top-level*.

All assignments will be stored on a stack in the order they were made; from now on referred to as the *trail*. The trail is divided into decision levels and is used to undo information during backtracking.

The decision phase will continue until either all variables have been assigned, in which case we have a model, or a conflict has occurred. On conflicts, the learning procedure will be invoked and a conflict clause produced. The trail will be used to undo decisions, one level at a time, until precisely one of the literals of the learnt clause becomes unbound (they are all FALSE at the point of conflict). By construction, the conflict clause cannot go directly from conflicting to a clause with two or more unbound literals. If the clause remains unit for several decision levels, it is advantageous to chose the lowest level (referred to as *backjumping* or *non-chronological backtracking* [MS96]).

```
loop
    propagate()    – propagate unit clauses
    if not conflict then
        if all variables assigned then
            return SATISFIABLE
        else
            decide()    – pick a new variable and assign it
    else
        analyze()    – analyze conflict and add a conflict clause
        if top-level conflict found then
            return UNSATISFIABLE
        else
            backtrack()    – undo assignments until conflict clause is unit
```

An important part of the procedure is the heuristic for *decide()*. Like CHAFF, MINISAT uses a dynamic variable order that gives priority to variables involved in recent conflicts.

[Discussion] Although this is a good default order, domain specific heuristics have successfully been used in various areas to improve the performance [Stri00]. Variable ordering is a traditional target for improving SAT-solvers.

Activity heuristics. One important technique introduced by CHAFF [MZ01] is a dynamic variable ordering based on activity (referred to as the VSIDS heuristic). The original heuristic imposes an order on *literals*, but borrowing from SATZOO, we make no distinction between p and $\overline{p}$ in MINISAT.

Each variable has an *activity* attached to it. Every time a variable occurs in a recorded conflict clause, its activity is increased. We will refer to this as *bumping*. After recording the conflict, the activity of all the variables in the system are multiplied by a constant less than 1, thus *decaying* the activity of variables over time. Recent increments count more than old. The current sum determines the activity of a variable.

In MINISAT we use a similar idea for clauses. When a learnt clause is used in the analysis process of a conflict, its activity is bumped. Inactive clauses are periodically removed.

Constraint removal. The constraint database is divided into two parts: the *problem constraints* and the *learnt clauses*. As we have noted, the set of learnt clauses can be periodically reduced to increase the performance of propagation. Learnt clauses are used to crop future branches in the search tree, so we risk getting a bigger search space instead. The balance between the two forces is delicate, and there are SAT-instances for which a big learnt clause set is advantageous, and others where a small set is better. MINISAT's default heuristic starts with a small set and gradually increases the size.

Problem constraints can also be removed if they are satisfied at the top-level. The API method *simplifyDB()* is responsible for this. The procedure is particularly important for incremental SAT-problems, where techniques for clause removal build on this feature.

Top-level solver. Although the pseudo-code for the search procedure presented above suffices for a simple conflict-driven SAT-solver, a solver *strategy* can improve the performance. A typical strategy applied by modern conflict-driven SAT-solvers is the use of *restarts* to escape from futile parts of the search tree. In MINISAT we also vary the number of learnt clauses kept at a given time. Furthermore, the *solve()* method of the API supports incremental assumptions, not handled by the above pseudo-code.

4 Implementation

The following conventions are used in the code. Atomic types start with a lowercase letter and are passed by value. Composite types start with a capital letter and are passed by reference. Blocks are marked only by indentation level. The

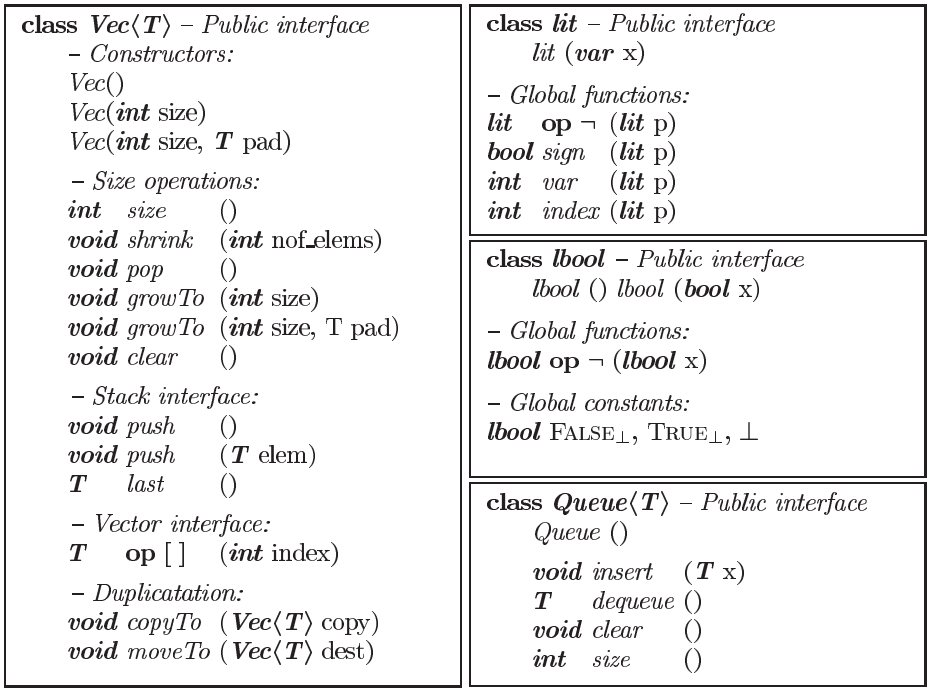


Fig. 1. *Basic abstract data types used throughout the code.* The vector data type can push a default constructed element by the *push*() method with no argument. The *moveTo*() method will move the contents of a vector to another vector in constant time, clearing the source vector. The literal data type has an *index*() method which converts the literal to a "small" integer suitable for array indexing. The *var*() method returns the underlying variable of the literal, and the *sign*() method if the literal is signed (FALSE for x and TRUE for $\overline{x}$).

bottom symbol $\bot$ will always mean *undefined*; the symbol FALSE will be used to denote the boolean false.

We will use, but not specify an implementation of, the following abstract data types: ***Vec***$\langle \boldsymbol{T} \rangle$ an extensible vector of type $\boldsymbol{T}$; ***lit*** the type of literals containing a special literal $\bot_{lit}$; ***lbool*** for the lifted boolean domain containing elements $\text{TRUE}_\bot$, $\text{FALSE}_\bot$, and $\bot$; ***Queue***$\langle \boldsymbol{T} \rangle$ a queue of type $\boldsymbol{T}$. We also use ***var*** as a type synonym for ***int*** (for implicit documentation) with the special constant $\bot_{var}$. The interfaces of the abstract data types are presented in *Figure 1*.

4.1 The solver state

A number of things need to be stored in the solver state. *Figure 2* shows the complete set of member variables of the solver type of MINISAT. Together with the state variables we define some short helper methods in *Figure 3*, as well as the interface of *VarOrder* (*Figure 4*), explained in section 4.6.

[Discussion]

The state does *not* contain a boolean "conflict" to remember if a top-level conflict has been reached. Instead we impose as an invariant that the solver must never be in a conflicting state. As a consequence, any method that puts the solver

in a conflicting state must communicate this. Using the solver object after this point is illegal. The invariant makes the interface slightly more cumbersome to use, but simplifies the implementation, which is important when extending and experimenting with new techniques.

4.2 Constraints

MiniSat can handle arbitrary constraints over boolean variables through the abstraction presented in *Figure 5*. Each constraint type needs to implement methods for constructing, removing, propagating and calculating reasons. In addition, methods for simplifying the constraint and updating the constraint on backtrack can be specified. We explain the meaning and responsibilities of these methods in detail:

Constructor. The constructor may only be called at the top-level. It must create and add the constraint to appropriate watcher lists after enqueuing any unit information derivable under the current top-level assignment. Should a conflict arise, this must be communicated to the caller.

Remove. The remove method supplants the destructor by receiving the solver state as a parameter. It should dispose the constraint and remove it from the watcher lists.

Propagate. The propagate method is called if the constraint is found in a watcher list during propagation of unit information p. The constraint is removed from the list and is required to insert itself into a new or the same watcher list. Any unit information derivable as a consequence of p should be enqueued. If successful, True is returned; if a conflict is detected, False is returned. The constraint may add itself to the undo list of $var(p)$ if it needs to be updated when p becomes unbound.

Simplify. At the top-level, a constraint may be given the opportunity to simplify its representation (returns False) or state that the constraint is satisfied under the current assignment and can be removed (returns True). A constraint must *not* be simplifiable to produce unit information or to be conflicting; in that case the propagation has not been correctly defined.

Undo. During backtracking, this method is called if the constraint added itself to the undo list of $var(p)$ in *propagate()*. The current variable assignments are guaranteed to be identical to that of the moment before *propagate()* was called.

Calculate Reason. This method is given a literal p and an empty vector. The constraint is the *reason* for p being true, that is, during propagation, the current constraint enqueued p. The received vector is extended to include a set of assignments (represented as literals) implying p. The current variable assignments are guaranteed to be identical to that of the moment before the constraint propagated p. The literal p is also allowed to be the special constant $\bot_{lit}$ in which case the reason for the clause being *conflicting* should be returned through the vector.

```
class Solver
  – Constraint database
  Vec⟨Constr⟩ constrs     – List of problem constraints.
  Vec⟨Clause⟩ learnts     – List of learnt clauses.
  double      cla_inc     – Clause activity increment – amount to bump with.
  double      cla_decay   – Decay factor for clause activity.

  – Variable order
  Vec⟨double⟩ activity    – Heuristic measurement of the activity of a variable.
  double      var_inc     – Variable activity increment – amount to bump with.
  double      var_decay   – Decay factor for variable activity.
  VarOrder    order       – Keeps track of the dynamic variable order.

  – Propagation
  Vec⟨Vec⟨Constr⟩⟩ watches – For each literal 'p', a list of constraints watching 'p'.
                            A constraint will be inspected when 'p' becomes true.
  Vec⟨Vec⟨Constr⟩⟩ undos   – For each variable 'x', a list of constraints that need to
                            update when 'x' becomes unbound by backtracking.
  Queue⟨lit⟩  propQ       – Propagation queue.

  – Assignments
  Vec⟨lbool⟩  assigns     – The current assignments indexed on variables.
  Vec⟨lit⟩    trail       – List of assignments in chronological order.
  Vec⟨int⟩    trail_lim   – Separator indices for different decision levels in 'trail'.
  Vec⟨Constr⟩ reason      – For each variable, the constraint that implied its value.
  Vec⟨int⟩    level       – For each variable, the decision level it was assigned.
  int         root_level  – Separates incremental and search assumptions.
```

Fig. 2. Internal state of the solver.

```
int   Solver.nVars()          return assigns.size()
int   Solver.nAssigns()       return trail.size()
int   Solver.nConstraints()   return constrs.size()
int   Solver.nLearnts()       return learnts.size()
lbool Solver.value(var x)     return assigns[x]
lbool Solver.value(lit p)     return sign(p) ? ¬assigns[var(p)] : assigns[var(p)]
int   Solver.decisionLevel()  return trail_lim.size()
```

Fig. 3. Small helper methods. For instance, *nLearnts()* returns the number of learnt clauses.

```
class VarOrder – Public interface
  VarOrder (Vec⟨lbool⟩ ref_to_assigns, Vec⟨double⟩ ref_to_activity)

  void newVar()       – Called when a new variable is created.
  void update(var x)  – Called when variable has increased in activity.
  void updateAll()    – Called when all variables have been assigned new activities.
  void undo(var x)    – Called when variable is unbound (may be selected again).
  var  select()       – Called to select a new, unassigned variable.
```

Fig. 4. Assisting ADT for the dynamic variable ordering of the solver. The constructor takes references to the assignment vector and the activity vector of the solver. The method *select()* will return the unassigned variable with the highest activity.

```
class Constr
  virtual void remove    (Solver S)                                        – must be defined
  virtual bool propagate (Solver S, lit p)                                 – must be defined
  virtual bool simplify  (Solver S)                                        – defaults to return false
  virtual void undo      (Solver S, lit p)                                 – defaults to do nothing
  virtual void calcReason (Solver S, lit p, Vec<lit> out_reason)           – must be defined
```

Fig. 5. Abstract base class for constraints.

The code for the *Clause* constraint is presented in *Figure 7*. It is also used for learnt clauses, which are unique in that they can be added to the clause database while the solver is not at top-level. This makes the constructor code a bit more complicated than it would be for a normal constraint.

Implementing the *addClause()* method of the solver API is just a matter of calling *Clause_new()* and pushing the new constraint on the "constrs" vector, storing the list of problem constraints. For completeness, we also display the code for creating variables in the solver (*Figure 6*).

```
var Solver.newVar()
    int index
    index = nVars()
    watches .push()
    watches .push()
    undos   .push()
    reason  .push(NULL)
    assigns .push(⊥)
    level   .push(-1)
    activity .push(0)
    order   .newVar()
    return index
```

Fig. 6. Creates a new SAT variable in the solver.

[Discussion] There are a number of tricks for smart-coding that can be used in a C++ implementation of *Clause*. In particularly the "lits" vector can be implemented as an zero-sized array placed last in the class, and then extra memory allocated for the clause to contain the data. We observed a 20% speedup for this trick. Furthermore, memory can be saved by not storing activity for problem clauses.

[Discussion] Of the methods defining a constraint, *propagate()* should be the primary target for efficient implementation. The SAT-solver spends about 80% of the time propagating, so the method will be called frequently. In SATZOO a performance gain was achieved by remembering the position of the last watched literal and start looking for a new literal to watch from that position. Further speedups may be achieved by specializing the code for small clause sizes.

4.3 Propagation

Given the mechanism for adding constraints, we now move on to describe the propagation of unit information on these constraints.

The propagation routine keeps a set of literals (unit information) that is to be propagated. We call this the *propagation queue*. When a literal is inserted into the queue, the corresponding variable is immediately assigned. For each literal in the queue, the watcher list of that literal determines the constraints that may be affected by the assignment. Through the interface described in the previous section, each constraint is asked by a call to its *propagate()* method if more unit information can be inferred, which will then be enqueued. The process continues until either the queue is empty or a conflict is found.

```
class Clause : public Constr
  bool      learnt
  float     activity
  Vec⟨lit⟩ lits

  – Constructor – creates a new clause and adds it to watcher lists:
  static bool Clause_new(Solver S, Vec⟨lit⟩ ps, bool learnt, Clause out_clause)
      "Implementation in Figure 8"

  – Learnt clauses only:
  bool locked(Solver S)
      return S.reason[var(lits[0])] == this

  – Constraint interface:
  void remove(Solver S)
      removeElem(this, S.watches[index(¬lits[0])])
      removeElem(this, S.watches[index(¬lits[1])])
      delete this

  bool simplify(Solver S)          – only called at top-level with empty prop. queue
      int j = 0
      for (int i = 0; i < lits.size(); i++)
          if (S.value(lits[i]) == True⊥)
              return True
          else if (S.value(lits[i]) == ⊥)
              lits[j++] = lits[i] – false literals are not copied (only occur for i ≥ 2)
      lits.shrink(lits.size() − j)
      return False

  bool propagate(Solver S, lit p)
      – Make sure the false literal is lits[1]:
      if (lits[0] == ¬p)
          lits[0] = lits[1], lits[1] = ¬p

      – If 0th watch is true, then clause is already satisfied.
      if (S.value(lits[0]) == True⊥)
          S.watches[index(p)].push(this)          – re-insert clause into watcher list
          return True

      – Look for a new literal to watch:
      for (int i = 2; i < size(); i++)
          if (S.value(lits[i]) != False⊥)
              lits[1] = lits[i], lits[i] = ¬p
              S.watches[index(¬lits[1])].push(this) – insert clause into watcher list
              return True

      – Clause is unit under assignment:
      S.watches[index(p)].push(this)
      return S.enqueue(lits[0], this)                    – enqueue for propagation

  void calcReason(Solver S, lit p, vec⟨lit⟩ out_reason)
      – invariant: (p == ⊥) or (p == lits[0])
      for (int i = ((p == ⊥) ? 0 : 1); i < size(); i++)
          out_reason.push(¬lits[i])          – invariant: S.value(lits[i]) == False⊥
      if (learnt) S.claBumpActivity(this)
```

Fig. 7. Implementation of the *Clause* constraint.

```
bool Clause_new(Solver S, Vec⟨lit⟩ ps, bool learnt, Clause out_clause)
  out_clause = NULL

  – Normalize clause:
  if (!learnt)
      if ("any literal in ps is true")       return TRUE
      if ("both p and ¬p occurs in ps") return TRUE
      "remove all false literals from ps"
      "remove all duplicates from ps"

  if (ps.size() == 0)
      return FALSE
  else if (ps.size() == 1)
      return S.enqueue(ps[0])                          – unit facts are enqueued
  else
      – Allocate clause:
      Clause c = new Clause
      ps.moveTo(c.lits)
      c.learnt    = learnt
      c.activity = 0                                   – only relevant for learnt clauses

      if (learnt)
          – Pick a second literal to watch:
          "Let max_i be the index of the literal with highest decision level"
          c.lits[1] = ps[max_i], c.lits[max_i] = ps[1]

          – Bumping:
          S.claBumpActivity(c)  – newly learnt clauses should be considered active
          for (int i = 0; i < ps.size(); i++)
              S.varBumpActivity(ps[i])  – variables in conflict clauses are bumped

      – Add clause to watcher lists:
      S.watches[index(¬c.lits[0])].push(c)
      S.watches[index(¬c.lits[1])].push(c)
      out_clause = c

      return TRUE
```

Fig. 8. Constructor function for clauses. Returns FALSE if top-level conflict is detected. 'out_clause' may be set to NULL if the new clause is already satisfied under the current top-level assignment. **Post-condition:** 'ps' is cleared. For learnt clauses, all literals will be false except 'lits[0]' (this by design of the *analyze*() method). For the propagation to work, the second watch must be put on the literal which will first be unbound by backtracking. (Note that none of the learnt-clause specific things needs to be done for a user defined constraint type.)

An implementation of this procedure is displayed in *Figure 9*. It starts by dequeuing a literal and clearing the watcher list for that literal by moving it to "tmp". The propagate method is then called for each constraint of "tmp". This will re-insert watches into new lists. Should a conflict be detected during the traversal of "tmp", the remaining watches will be copied back to the original watcher list, and the propagation queue cleared.

The method for enqueuing unit information is relatively straightforward. Note that the same fact can be enqueued several times, as it may be propagated from different constraints, but it will only be put on the propagation queue once.

It may be that later enqueuings have a "better" reason (determined heuristically) and a small performance gain was achieved in SATZOO by changing reason if the new reason was smaller than the previously stored. The changing affects the conflict clause generation described in the next section. [Discussion]

4.4 Learning

This section describes the conflict-driven clause learning. It was first described in [MS96] and is one of the major advances of SAT-technology in the last decade.

We describe the basic conflict-analysis algorithm by an example. Assume the database contains the clause $\{x, y, z\}$ which just became unsatisfied during propagation. This is our conflict. We call $\overline{x} \wedge \overline{y} \wedge \overline{z}$ the reason set of the conflict. Now x is false because $\overline{x}$ was propagated from some constraint. We ask that constraint to give us the reason for propagating $\overline{x}$ (the *calcReason()* method). It will respond with another conjunction of literals, say $u \wedge v$. These were the variable assignment that implied $\overline{x}$. The constraint may in fact have been the clause $\{\overline{u}, \overline{v}, \overline{x}\}$. From this little analysis we know that $u \wedge v \wedge \overline{y} \wedge \overline{z}$ must also lead to a conflict. We may prohibit this conflict by adding the clause $\{\overline{u}, \overline{v}, y, z\}$ to the clause database. This would be an example of a *learnt* conflict clause.

In the example, we picked only one literal and analyzed it one step. The process of expanding literals with their reason sets can be continued, in the extreme case until all the literals of the conflict set are decision variables (which were not propagated by any constraints). Different learning schemes based on this process have been proposed. Experimentally the "First Unique Implication Point" (First UIP) heuristic has been shown effective [ZM01]. We will not give the definition of UIPs here, but just state the algorithm: In a breadth-first manner, continue to expand literals of the current decision level, until there is just one left.

In the code for *analyze()*, displayed in *Figure 10*, we make use of the fact that a breadth-first traversal can be achieved by inspecting the trail backwards. Especially, the variables of the reason set of p is always before p in the trail. Furthermore, in the algorithm we initialize p to $\bot_{lit}$, which will make *calcReason()* return the reason for the conflict.

Assuming x to be the unit information that causes the conflict, an alternative implementation would be to calculate the reason for $\overline{x}$ and just add x to that set. The code would be slightly more cumbersome but the contract for *calcReason()* would be simpler, as we no longer need the special case for $\bot_{lit}$. [Discussion]

```
Constr Solver.propagate()
  while (propQ.size() > 0)
      lit p = propQ.dequeue()          – 'p' is now the enqueued fact to propagate
      Vec⟨Constr⟩ tmp                  – 'tmp' will contain the watcher list for 'p'
      watches[index(p)].moveTo(tmp)

      for (int i = 0; i < tmp.size(); i++)
          if (!tmp[i].propagate(this, p))
              – Constraint is conflicting; copy remaining watches to 'watches[p]'
              – and return constraint:
              for (int j = i+1; j < tmp.size(); j++)
                watches[index(p)].push(tmp[j])
              propQ.clear()
              return tmp[i]
  return Null
```

```
bool Solver.enqueue(lit p, Constr from = Null)
  if (value(p) != ⊥)
      if (value(p) == False⊥)
          – Conflicting enqueued assignment
          return False
      else
          – Existing consistent assignment – don't enqueue
          return True
  else
      – New fact, store it
      assigns [var(p)] = lbool(!sign(p))
      level   [var(p)] = decisionLevel()
      reason  [var(p)] = from
      trail.push(p)
      propQ.insert(p)
      return True
```

Fig. 9. ***propagate():*** Propagates all enqueued facts. If a conflict arises, the *conflicting* clause is returned, otherwise Null. ***enqueue():*** Puts a new fact on the propagation queue, as well as immediately updating the variable's value in the assignment vector. If a conflict arises, False is returned and the propagation queue is cleared. The parameter 'from' contains a reference to the constraint from which 'p' was propagated (defaults to Null if omitted).

Finally, the analysis not only returns a conflict clause, but also the backtracking level. This is the lowest decision level for which the conflict clause is unit. It is advantageous to backtrack as far as possible [MS96], and is referred to as *back-jumping* or *non-chronological backtracking* in the literature.

4.5 Search

The search method in *Figure 13* works basically as described in section 3 but with the following additions:

Restarts. The first argument of the search method is "nof_conflicts". The search for a model or a contradiction will only be conducted for this many

```
void Solver.analyze(Constr confl, Vec⟨lit⟩ out_learnt, Int out_btlevel)
  Vec⟨bool⟩ seen(nVars(), FALSE)
  int       counter  = 0
  lit       p        = ⊥lit
  Vec⟨lit⟩  p_reason
  out_learnt.push()                          – leave room for the asserting literal
  out_btlevel = 0
  do
      p_reason.clear()
      confl.calcReason(this, p, p_reason)        – invariant here: confl != NULL

      – TRACE REASON FOR P:
      for (int j = 0; j < p_reason.size(); j++)
          lit q = p_reason[j]
          if (!seen[var(q)])
              seen[var(q)] = TRUE
              if (level[var(q)] == decisionLevel())
                  counter++
              else if (level[var(q)] > 0)  – exclude variables from decision level 0
                  out_learnt.push(¬q)
                  out_btlevel = max(out_btlevel, level[var(q)])

      – SELECT NEXT LITERAL TO LOOK AT:
      do
          p     = trail.last()
          confl = reason[var(p)]
          undoOne()
      while (!seen[var(p)])
      counter−−
  while (counter > 0)
  out_learnt[0] = ¬p
```

Fig. 10. Analyze a conflict and produce a reason clause. **Pre-conditions:** (1) 'out_learnt' is assumed to be cleared. (2) Current decision level must be greater than root level. **Post-conditions:** (1) 'out_learnt[0]' is the asserting literal at level 'out_btlevel'. **Effect:** Will undo part of the trail, but not beyond last decision level.

```
void Solver.record(Vec⟨lit⟩ clause)
  Clause c              – will be set to created clause, or NULL if 'clause[]' is unit
  Clause_new(this, clause, TRUE, c)                    – cannot fail at this point
  enqueue(clause[0], c)                                – cannot fail at this point
  if (c != NULL) learnts.push(c)
```

Fig. 11. Record a clause and drive backtracking. Pre-condition: 'clause[0]' must contain the asserting literal. In particular, 'clause[]' must not be empty.

conflicts. If failing to solve the SAT-problem within the bound, all assumptions will be canceled and $\perp$ returned. The surrounding solver strategy will then restart the search, possibly with a new set of parameters.

Reduce. The second argument, "nof_learnts", sets an upper limit on the number of learnt clauses that are kept. Once this number is reached, *reduceDB()* is called. Clauses that are currently the reason for a variable assignment are said to be *locked* and cannot be removed by *reduceDB()*. For this reason, the limit is extended by the number of assigned variables, which approximates the number of locked clauses.

Parameters. The third argument to the search method groups some tuning constants. In the current version of MiniSat, it only contains the decay factors for variables and clauses.

Root-level. To support incremental SAT, the concept of a *root-level* is introduced. The root-level acts a bit as a new top-level. Above the root-level are the incremental assumptions passed to *solve()* (if any). The search procedure is not allowed to backtrack above the root-level, as this would change the incremental assumptions. If we reach a conflict at root-level, the search will return False.

A problem with the approach presented here is conflict clauses that are unit. For these, *analyze()* will always return a backtrack level of 0 (top-level). As unit clauses are treated specially, they are never added to the clause database. Instead they are enqueued as facts to be propagated (see the code of *Clause_new()*). There would be no problem if this was done at top-level. However, the search procedure will only undo until root-level, which means that the unit fact will be enqueued there. Once *search()* has solved the current SAT-problem, the surrounding solver strategy will undo any incremental assumption and put the solver back at the top-level. By this the unit clause will be forgotten, and the next incremental SAT problem will have to infer it again.

A solution to this is to store the learnt unit clauses in a vector and re-insert them at top-level before the next call to *solve()*. The reason for omitting this in MiniSat is that we have not seen any performance gain by this extra handling in our applications [ES03,CS03]. Simplicity thus dictates that we leave it out of the presentation.

Simplify. Provided the root-level is 0 (no assumptions were passed to *solve()*) the search will return to the top-level every time a unit clause is learnt. At that point it is legal to call *simplifyDB()* to simplify the problem constraints according to the top-level assignment. If a stronger simplifier than presented here is implemented, a contradiction may be found, in which case the search should be aborted. As our simplifier is not stronger than normal propagation, it can never reach a contradiction, so we ignore the return value of *simplify()*.

```
void Solver.undoOne()
    lit      p   = trail.last()
    var      x   = var(p)
    assigns [x] = ⊥
    reason  [x] = NULL
    level   [x] = -1
    order.undo(x)
    trail.pop()
    while (undos[x].size() > 0)
        undos[x].last().undo(this, p)
        undos[x].pop()
```

```
bool Solver.assume(lit p)
    trail_lim.push(trail.size())
    return enqueue(p)
```

```
void Solver.cancel()
    int c = trail.size() − trail_lim.last()
    for (; c != 0; c−−)
        undoOne()
    trail_lim.pop()
```

```
void Solver.cancelUntil(int level)
    while (decisionLevel() > level)
        cancel()
```

Fig. 12. ***assume():*** returns FALSE if immediate conflict. **Pre-condition:** propagation queue is empty. ***undoOne():*** unbinds the last variable on the trail. ***cancel():*** reverts to the state before last *push*(). **Pre-condition:** propagation queue is empty. ***cancelUntil():*** cancels several levels of assumptions.

4.6 Activity heuristics

The implementation of activity is shown in *Figure 14*. Instead of actually multiplying all variables by a decay factor after each conflict, we bump variables with larger and larger numbers. Only relative values matter. Eventually we will reach the limit of what is representable by a floating point number. At that point, all activities are scaled down.

In the *VarOrder* data type of MINISAT, the list of variables is kept sorted on activity at all time. The backtracking will always accurately choose the most active variable. The original suggestion for the VSIDS dynamic variable ordering was to sort periodically.

The polarity of a literal is ignored in MINISAT. However, storing the latest polarity of a variable might improve the search when restarts are used, but it remains to be empirically supported. Furthermore, the interface of *VarOrder* can be used for other variable heuristics. In SATZOO, an initial static variable order computed from the clause structure was particularly successful on many problems. [Discussion]

4.7 Constraint removal

The methods for reducing the set of learnt clauses as well as the top-level simplification procedure can be found in *Figure 15*.

When removing learnt clauses, it is important not to remove so called *locked* clauses. Locked clauses are those participating in the current backtracking branch by being the reason (through propagation) for a variable assignment. The reduce procedure keeps half of the learnt clauses, except for those which have decayed below a threshold limit. Such clauses can occur if the set of active constraints is very small.

Top-level simplification can be seen as a special case of propagation. Since [Discussion]

```
lbool Solver.search(int nof_conflicts, int nof_learnts, SearchParams params)
  int conflictC = 0
  var_decay = 1 / params.var_decay
  cla_decay = 1 / params.cla_decay
  model.clear()

  loop
      Constr confl = propagate()
      if (confl != NULL)
          – CONFLICT
          conflictC++
          Vec⟨lit⟩ learnt_clause
          int backtrack_level
          if (decisionLevel() == root_level)
              return FALSE⊥
          analyze(confl, learnt_clause, backtrack_level)
          cancelUntil(max(backtrack_level, root_level))
          record(learnt_clause)
          decayActivities()
      else
          – NO CONFLICT
          if (decisionLevel() == 0)
              – Simplify the set of problem clauses:
              simplifyDB()                    – our simplifier cannot return false here

          if (learnts.size()−nAssigns() ≥ nof_learnts)
              – Reduce the set of learnt clauses:
              reduceDB()

          if (nAssigns() == nVars())
              – Model found:
              model.growTo(nVars())
              for (int i = 0; i < nVars(); i++)
                 model[i] = (value(i) == TRUE⊥)
              cancelUntil(root_level)
              return TRUE⊥
          else if (conflictC ≥ nof_conflicts)
              – Reached bound on number of conflicts:
              cancelUntil(root_level)                              – force a restart
              return ⊥
          else
              – New variable decision:
              lit p = lit(order.select())       – may have heuristic for polarity here
              assume(p)                                          – cannot return false
```

Fig. 13. Search method. Assumes and propagates until a conflict is found, from which a conflict clause is learnt and backtracking performed until search can continue. **Precondition:** root_level == *decisionLevel*().

```
void Solver.varBumpActivity(var x)
  if ((activity[x] += var_inc) > 1e100)
      varRescaleActivity()
  order.update(x)

void Solver.varDecayActivity()
  var_inc *= var_decay

void Solver.varRescaleActivity()
  for (int i = 0; i < nVars(); i++)
      activity[i] *= 1e-100
  var_inc *= 1e-100
```

```
void Solver.claBumpActivity(Clause c)
void Solver.claDecayActivity()
void Solver.claRescaleActivity()
    – Similarly implemented.

void Solver.decayActivities()
    varDecayActivity()
    claDecayActivity()
```

Fig. 14. Bumping of variable and clause activities.

it is performed under no assumption, anything learnt can be kept forever. The freedom of not having to store derived information separately, with the ability to undo it later, makes it easier to implement stronger propagation.

4.8 Top-level solver

The method implementing MiniSat's top-level strategy can be found in *Figure 16*. It is responsible for making the incremental assumptions and setting the root level. Furthermore, it completes the simple backtracking search with restarts, which are performed less and less frequently. After each restart, the number of allowed learnt clauses is increased.

The code contains a number of hand-tuned constants that have shown to perform reasonable on our applications [ES03,CS03]. The top-level strategy, however, is a productive target for improvements (possibly application dependent). In Satzoo, the top-level strategy contains an initial phase where a static variable ordering is used.

5 Conclusions and Related Work

By this paper, we have provided a minimal reference implementation of a modern conflict-driven SAT-solver. Despite the abstraction layer for boolean constraints, and the lack of more sophisticated heuristics, the performance of MiniSat is comparable to state-of-the-art SAT-solvers. We have tested MiniSat against zChaff and Berkmin 5.61 on 177 SAT-instances. These instances were used to tune Satzoo for the *SAT 2003 Competition*. As Satzoo solved more instances and series of problems, ranging over all three categories (*industrial, handmade,* and *random*), than any other solver in the competition, we feel that this is a good test-set for the overall performance. No extra tuning was done in MiniSat; it was just run once with the constants presented in the code. At a time-out of 10 minutes, MiniSat solved 158 instances, while zChaff solved 147 instances and Berkmin 157 instances.

Another approach to incremental SAT and non-clausal constraints was presented by Aloul, Ramani, Markov, and Sakallah in their work on Satire and PBS [WKS01,ARMS02]. Our implementation differs in that it has a simpler

```
void Solver.reduceDB()
  int     i, j
  double lim = cla_inc / learnts.size()

  sortOnActivity(learnts)
  for (i=j=0; i < learnts.size()/2; i++)
      if (!learnts[i].locked(this))
          learnts[i].remove(this)
      else
          learnts[j++] = learnts[i]
  for (; i < learnts.size(); i++)
      if (!learnts[i].locked(this)
      && learnts[i].activity() < lim)
          learnts[i].remove(this)
      else
          learnts[j++] = learnts[i]
  learnts.shrink(i − j)
```

```
bool Solver.simplifyDB()
  if (propagate() != NULL)
      return FALSE

  for (int type = 0; type < 2; type++)
      Vec⟨Constr⟩ cs = type ?
        (Vec⟨Constr⟩)learnts : constrs
      int j = 0
      for (int i = 0; i < cs.size(); i++)
          if (cs[i].simplify(this))
              cs[i].remove(this)
          else
              cs[j++] = cs[i]
      cs.shrink(cs.size()−j)
  return TRUE
```

Fig. 15. ***reduceDB():*** Remove half of the learnt clauses minus some locked clauses. A locked clause is a clauses that is reason to a current assignment. Clauses below a certain lower bound activity are also be removed. ***simplifyDB():*** Top-level simplify of constraint database. Will remove any satisfied constraint and simplify remaining constraints under current (partial) assignment. If a top-level conflict is found, FALSE is returned. **Pre-condition:** Decision level must be zero. **Post-condition:** Propagation queue is empty.

```
bool Solver.solve(Vec⟨lit⟩ assumps)
    SearchParams params(0.95, 0.999)
    double nof_conflicts = 100
    double nof_learnts   = nConstraints()/3
    lbool  status        = ⊥

    – PUSH INCREMENTAL ASSUMPTIONS:
    for (int i = 0; i < assumps.size(); i++)
        if (!assume(assumps[i]) || propagate() != NULL)
            cancelUntil(0)
            return FALSE
    root_level = decisionLevel()

    – SOLVE:
    while (status == ⊥)
        status = search((int)nof_conflicts, (int)nof_learnts, params)
        nof_conflicts *= 1.5
        nof_learnts   *= 1.1

    cancelUntil(0)
    return status == TRUE⊥
```

Fig. 16. Main solve method. **Pre-condition:** If assumptions are used, *simplifyDB*() must be called right before using this method. If not, a top-level conflict (resulting in a non-usable internal state) cannot be distinguished from a conflict under assumptions.

notion of incrementality, and that it contains a well documented interface for non-clausal constraints.

Finally, a set of reference implementations of modern SAT-techniques is present in the OpenSAT project.[3] However, the project aim for completeness rather than minimal exposition, as we have chosen in this paper.

6 Exercises

1. Write the code for an *AtMost* constraint. The constraint is satisfied if at most n out of m specified literals are true.
2. Implement a generator for (generalized) pigeon-hole formulas using the new constraints. The generator should take three arguments: number of pigeons, number of holes, and hole capacity. Each pigeon must reside in some pigeon-hole. No hole may contain more pigeons than its capacity.
3. Make an incremental version that adds one pigeon to the problem at a time until the problem becomes unsatisfiable.

References

[ARMS02] F. Aloul, A. Ramani, I. Markov, K. Sakallah. **"Generic ILP vs. Specialized 0-1 ILP: an Update"** in *International Conference on Computer Aided Design (ICCAD)*, 2002.

[BCCFZ99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu. **"Symbolic Model Checking using SAT procedures instead of BDDs"** in *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[CS03] K. Claessen, N. Sörensson. **"New Techniques that Improve MACE-style Finite Model Finding"** in *CADE-19, Workshop W4. Model Computation – Principles, Algorithms, Applications*, 2003.

[DLL62] M. Davis, G. Logemann, D. Loveland. **"A machine program for theorem proving"** in *Communications of the ACM*, vol 5, 1962.

[ES03] N. Eén, N. Sörensson. **"Temporal Induction by Incremental SAT Solving"** in *Proc. of First International Workshop on Bounded Model Checking*, 2003. ENTCS issue 4 volume 89.

[Lar92] T. Larrabee. **"Test Pattern Generation Using Boolean Satisfiability"** in *IEEE Transactions on Computer-Aided Design*, vol. 11-1, 1992.

[MS96] J.P. Marques-Silva, K.A. Sakallah. **"GRASP – A New Search Algorithm for Satisfiability"** in *ICCAD. IEEE Computer Society Press*, 1996

[MZ01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. **"Chaff: Engineering an Efficient SAT Solver"** in *Proc. of the 38th Design Automation Conference*, 2001.

[Stri00] O. Strichman **"Tuning SAT checkers for Bounded Model Checking"** in *Proc. of 12th Intl. Conf. on Computer Aided Verification*, LNCS:1855, Springer-Verlag 2000

[WKS01] J. Whittemore, J. Kim, K. Sakallah. **"SATIRE: A New Incremental Satisfiability Engine"** in *Proc. 38th Conf. on Design Automation*, ACM Press 2001.

[ZM01] L. Zhang, C.F. Madigan, M.W. Moskewicz, S. Malik. **"Efficient Conflict Driven Learning in Boolean Satisfiability Solver"** in *Proc. of the International Conference on Computer Aided Design (ICCAD)*, 2001.

[3] http://www.opensat.org

Appendix – What is missing from Satzoo?

In order to reduce the size of MiniSat to a minimum, all non-essential parts of Satzoo/Satnik were left out. Since Satzoo won two categories of the *SAT 2003 Competition*, we chose to present the missing parts here for completeness.

Initial strategies:

- *Burst of random variable orders.* Before anything else, Satzoo runs several passes of about 10-100 conflicts each with the variable order initiated to random. For satisfiable problems, Satzoo can sometimes stumble upon the solution by this strategy. For hard (typically unsatisfiable) problems, important clauses can be learnt in this phase that is outside the "local optimum" that the activity driven variable heuristic will later get stuck in.
- *Static variable ordering.* The second phase of Satzoo is to compute a static variable ordering taking into account how the variables of different clauses relates to each other (see *Figure 17*). Variables often occurring together in clauses will be put close in the variable order. Satzoo uses this static ordering for at least 5000 conflicts and does not stop until progress is halted severely. The static ordering often counters the effect of "shuffling" the problem (changing the order of clauses). The authors believe this phase to be the most important feature left out of MiniSat, and an important part of the success of Satzoo in the competition.[4]

Extra variable decision heuristics:

- *Variable of recent importance.* Inspired by the SAT-solver Berkmin, occasionally variables from recent (unsatisfied) recorded clauses are picked.
- *Random.* About 1% of the time, a random variable is selected for branching. This simple strategy seems to crack some extra problems without incurring any substantial overhead for other problems. Give it a try!

Other:

- *Equivalent variable substitution.* The binary clauses are checked for cyclic implications. If a cycle is found, a representative is selected and all other variables in the cycle is replaced by this representative in the clause database. This yields a smaller database and fewer variables. The simplification is done periodically, but is most important in the initial phase (some problems can be very redundant).
- *Garbage collection.* Satzoo implements its own memory management which allows clauses to be stored more compactly.
- *0-1-programming.* Pseudo-boolean constraints are supported by Satzoo. This can of course easily be added to MiniSat through the constraint interface.

[4] The provided code currently has no further motivation beyond the authors' intuition. Indeed it was added as a quick hack two days before the competition.

```
void Solver.staticVarOrder()

  – Clear activity:
  for (int i = 0; i < nVars(); i++) activity[i] = 0

  – Do simple variable activity heuristic:
  for (int i = 0; i < clauses.size(); i++)
      Clause c   = clauses[i]
      double add = pow2(−size(c))
      for (int j = 0; j < size(c); j++) activity[var(c[j])] += add

  – Calculate the initial "heat" of all clauses:
  Vec⟨Vec⟨int⟩⟩          occurs(2*nVars()) – Map literal to list of clause indices
  Vec⟨Pair⟨double,int⟩⟩ heat(clauses.size())    – Pairs of heat and clause index
  for (int i = 0; i < clauses.size(); i++)
      Clause c   = clauses[i]
      double sum = 0
      for (int j = 0; j < size(c); j++)
          occurs[index(c[j])].push(i)
          sum += activity[var(c[j])]
      heat[i] = Pair_new(sum, i)

  – Bump heat for clauses whose variables occur in other hot clauses:
  double iter_size = 0
  for (int i = 0; i < clauses.size(); i++)
      Clause c = clauses[i]
      for (int j = 0; j < size(c); j++) iter_size += occurs[index(c[j])].size()
  int    iterations = min((int)(((double)literals / iter_size) * 100), 10)
  double disapation = 1.0 / iterations
  for (int c = 0; c < iterations; c++)
      for (int i = 0; i < clauses.size(); i++)
          Clause c = clauses[i]
          for (int j = 0; j < size(c); j++)
              Vec⟨int⟩ os = occurs[index(c[j])]
              for (int k = 0; k < os.size(); k++)
                heat[i].fst += heat[os[k]].fst * disapation

  – Set activity according to hot clauses:
  sort(heat)
  for (int i = 0; i < nVars(); i++) activity[i] = 0

  double extra = 1e200
  for (int i = 0; i < heat.size(); i++)
      Clause& c = clauses[heat[i].snd]
      for (int j = 0; j < size(c); j++)
          if (activity[var(c[j])] == 0)
              activity[var(c[j])] = extra
              extra *= 0.995

  order.updateAll()
  var_inc = 1
```

Fig. 17. The static variable ordering of SATZOO. The code is defined only for clauses, not for arbitrary constraints. It must be adapted before it can be used in MINISAT.

Chapter 2

Natural Deduction

> Ich wollte zunächst einmal einen Formalismus aufstellen, der dem wirklichen Schließen möglichst nahe kommt. So ergab sich ein „Kalkül des natürlichen Schließens".[1]
>
> — Gerhard Gentzen
> *Untersuchungen über das logische Schließen* [Gen35]

In this chapter we explore ways to define logics, or, which comes to the same thing, ways to give meaning to logical connectives. Our fundamental notion is that of a *judgment* based on *evidence*. For example, we might make the judgment "*It is raining*" based on visual evidence. Or we might make the judgment "*'A implies A' is true for any proposition A*" based on a derivation. The use of the notion of a judgment as conceptual prior to the notion of proposition has been advocated by Martin-Löf [ML85a, ML85b]. Certain forms of judgments frequently recur and have therefore been investigated in their own right, prior to logical considerations. Two that we will use are *hypothetical judgments* and *parametric jugments* (the latter are sometimes called *general judgments* or *schematic judgments*).

A hypothetical judgment has the form "J_2 *under hypothesis* J_1". We consider this judgment evident if we are prepared to make the judgment J_2 once provided with evidence for J_1. Formal evidence for a hypothetical judgment is a *hypothetical derivation* where we can freely use the hypothesis J_1 in the derivation of J_2. Note that hypotheses need not be used, and could be used more than once.

A parametric judgment has the form "J *for any* a" where a is a *parameter* which may occur in J. We make this judgment if we are prepared to make the judgment $[O/a]J$ for arbitrary objects O of the right category. Here $[O/a]J$ is our notation for substituting the object O for parameter a in the judgment J. Formal evidence for a parametric judgment J is a *parametric derivation* with free occurrences of the parameter a.

[1] First I wanted to construct a formalism which comes as close as possible to actual reasoning. Thus arose a "calculus of natural deduction".

Formal evidence for a judgment in form of a derivation is usually written in two-dimensional notation:

$$\begin{array}{c}\mathcal{D}\\ J\end{array}$$

if $\mathcal{D}$ is a derivation of J. For the sake of brevity we sometimes use the alternative notation $\mathcal{D} :: J$. A hypothetical judgment is written as

$$\begin{array}{c}\overline{}\,u\\ J_1\\ \vdots\\ J_2\end{array}$$

where u is a label which identifies the hypothesis J_1. We use the labels to guarantee that hypotheses which are introduced during the reasoning process are not used outside their scope.

The separation of the notion of judgment and proposition and the corresponding separation of the notion of evidence and proof sheds new light on various styles that have been used to define logical systems.

An axiomatization in the style of Hilbert [Hil22], for example, arises when one defines a judgment "*A is true*" without the use of hypothetical judgments. Such a definition is highly economical in its use of judgments, which has to be compensated by a liberal use of implication in the axioms. When we make proof structure explicit in such an axiomatization, we arrive at combinatory logic [Cur30].

A categorical logic [LS86] arises (at least in the propositional case) when the basic judgment is not truth, but entailment "*A entails B*". Once again, presentations are highly economical and do not need to seek recourse in complex judgment forms (at least for the propositional fragment). But derivations often require many hypotheses, which means that we need to lean rather heavily on conjunction here. Proofs are realized by morphisms which are an integral part of the machinery of category theory.

While these are interesting and in many ways useful approaches to logic specification, neither of them comes particularly close to capturing the practice of mathematical reasoning. This was Gentzen's point of departure for the design of a system of *natural deduction* [Gen35]. From our point of view, this system is based on the simple judgment "*A is true*", but relies critically on hypothetical and parametric judgments. In addition to being extremely elegant, it has the great advantage that one can define all logical connectives without reference to any other connective. This principle of modularity extends to the meta-theoretic study of natural deduction and simplifies considering fragments and extension of logics. Since we will consider many fragments and extension, this *orthogonality* of the logical connectives is a critical consideration. There is another advantage to natural deduction, namely that its proofs are isomorphic to the terms in a λ-calculus via the so-called Curry-Howard isomorphism [How69], which establishes many connections to functional programming.

Finally, we arrive at the *sequent calculus* (also introduced by Gentzen in his seminal paper [Gen35]) when we split the single judgment of truth into two: "*A is an assumption*" and "*A is true*". While we still employ the machinery of parametric and hypothetical judgments, we now need an explicit rule to state that "*A is an assumption*" is sufficient evidence for "*A is a true*". The reverse, namely that if "*A is true*" then "*A may be used as an assumption*" is the Cut rule which he proved to be redundant in his *Hauptsatz*. For Gentzen the sequent calculus was primarily a technical device to prove consistency of his system of natural deduction, but it exposes many details of the fine structure of proofs in such a clear manner that many logic presentations employ sequent calculi. The laws governing the structure of proofs, however, are more complicated than the Curry-Howard isomorphism for natural deduction might suggest and are still the subject of study [Her95, Pfe95].

We choose natural deduction as our definitional formalism as the purest and most widely applicable. Later we justify the sequent calculus as a calculus of proof search for natural deduction and explicitly relate the two forms of presentation.

We begin by introducing natural deduction for intuitionistic logic, exhibiting its basic principles.

2.1 Intuitionistic Natural Deduction

The system of natural deduction we describe below is basically Gentzen's system NJ [Gen35] or the system which may be found in Prawitz [Pra65]. The calculus of natural deduction was devised by Gentzen in the 1930's out of a dissatisfaction with axiomatic systems in the Hilbert tradition, which did not seem to capture mathematical reasoning practices very directly. Instead of a number of axioms and a small set of inference rules, valid deductions are described through inference rules only, which at the same time explain the meaning of the logical quantifiers and connectives in terms of their proof rules.

A language of (first-order) *terms* is built up from *variables* x, y, *etc.*, *function symbols* f, g, *etc.*, each with a unique arity, and *parameters* a, b, *etc.* in the usual way.

$$\textit{Terms} \quad t \quad ::= \quad x \mid a \mid f(t_1, \ldots, t_n)$$

A constant c is simply a function symbol with arity 0 and we write c instead of $c()$. Exactly which function symbols are available is left unspecified in the general development of predicate logic and only made concrete for specific theories, such as the theory of natural numbers. However, variables and parameters are always available. We will use t and s to range over terms.

The language of *propositions* is built up from *predicate symbols* P, Q, *etc.* and terms in the usual way.

$$\begin{array}{lll} \textit{Propositions} & A & ::= \quad P(t_1, \ldots, t_n) \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid A_1 \vee A_2 \mid \neg A \\ & & \quad\quad \mid \bot \mid \top \mid \forall x.\, A \mid \exists x.\, A \end{array}$$

A propositional constant P is simply a predicate symbol with no arguments and we write P instead of $P()$. We will use A, B, and C to range over propositions. Exactly which predicate symbols are available is left unspecified in the general development of predicate logic and only made concrete for specific theories.

The notions of *free* and *bound* variables in terms and propositions are defined in the usual way: the variable x is bound in propositions of the form $\forall x.\ A$ and $\exists x.\ A$. We use parentheses to disambiguate and assume that $\wedge$ and $\vee$ bind more tightly than $\supset$. It is convenient to assume that propositions have no free individual variables; we use parameters instead where necessary. Our notation for substitution is $[t/x]A$ for the result of substituting the term t for the variable x in A. Because of the restriction on occurrences of free variables, we can assume that t is free of individual variables, and thus capturing cannot occur.

The main judgment of natural deduction is "C *is true*" written as C *true*, from hypotheses A_1 *true*, $\ldots$, A_n *true*. We will model this as a hypothetical judgment. This means that certain structural properties of derivations are tacitly assumed, independently of any logical inferences. In essence, these assumptions explain what hypothetical judgments are.

Hypothesis. If we have a hypothesis A *true* than we can conclude A *true*.

Weakening. Hypotheses need not be used.

Duplication. Hypotheses can be used more than once.

Exchange. The order in which hypotheses are introduced is irrelevant.

In natural deduction each logical connective and quantifier is characterized by its *introduction rule(s)* which specifies how to infer that a conjunction, disjunction, *etc.* is true. The *elimination rule* for the logical constant tells what other truths we can deduce from the truth of a conjunction, disjunction, *etc.* Introduction and elimination rules must match in a certain way in order to guarantee that the rules are meaningful and the overall system can be seen as capturing mathematical reasoning.

The first is a *local soundness* property: if we introduce a connective and then immediately eliminate it, we should be able to erase this detour and find a more direct derivation of the conclusion without using the connective. If this property fails, the elimination rules are too strong: they allow us to conclude more than we should be able to know.

The second is a *local completeness* property: we can eliminate a connective in a way which retains sufficient information to reconstitute it by an introduction rule. If this property fails, the elimination rules are too weak: they do not allow us to conclude everything we should be able to know.

We provide evidence for local soundness and completeness of the rules by means of *local reduction* and *expansion* judgments, which relate proofs of the same proposition.

One of the important principles of natural deduction is that each connective should be defined only in terms of inference rules without reference to other

logical connectives or quantifiers. We refer to this as *orthogonality* of the connectives. It means that we can understand a logical system as a whole by understanding each connective separately. It also allows us to consider fragments and extensions directly and it means that the investigation of properties of a logical system can be conducted in a modular way.

We now show the introduction and elimination rules, local reductions and expansion for each of the logical connectives in turn. The rules are summarized on page 2.1.

Conjunction. $A \wedge B$ should be true if both A and B are true. Thus we have the following introduction rule.

$$\frac{A\ true \qquad B\ true}{A \wedge B\ true}\,\wedge\mathrm{I}$$

If we consider this as a complete definition, we should be able to recover both A and B if we know $A \wedge B$. We are thus led to two elimination rules.

$$\frac{A \wedge B\ true}{A\ true}\,\wedge\mathrm{E_L} \qquad \frac{A \wedge B\ true}{B\ true}\,\wedge\mathrm{E_R}$$

To check our intuition we consider a deduction which ends in an introduction followed by an elimination:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A\ true\end{array} \qquad \begin{array}{c}\mathcal{E}\\ B\ true\end{array}}{A \wedge B\ true}\,\wedge\mathrm{I}}{A\ true}\,\wedge\mathrm{E_L}$$

Clearly, it is unnecessary to first introduce the conjunction and then eliminate it: a more direct proof of the same conclusion from the same (or fewer) assumptions would be simply

$$\begin{array}{c}\mathcal{D}\\ A\ true\end{array}$$

Formulated as a transformation or *reduction* between derivations we have

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A\ true\end{array} \qquad \begin{array}{c}\mathcal{E}\\ B\ true\end{array}}{A \wedge B\ true}\,\wedge\mathrm{I}}{A\ true}\,\wedge\mathrm{E_L} \;\Longrightarrow_R\; \begin{array}{c}\mathcal{D}\\ A\ true\end{array}$$

and symmetrically

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A\ true\end{array} \qquad \begin{array}{c}\mathcal{E}\\ B\ true\end{array}}{A \wedge B\ true}\,\wedge\mathrm{I}}{B\ true}\,\wedge\mathrm{E_R} \;\Longrightarrow_R\; \begin{array}{c}\mathcal{E}\\ B\ true\end{array}$$

The new judgment

$$\begin{array}{c}\mathcal{D}\\ A\ true\end{array} \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{E}\\ A\ true\end{array}$$

relates derivations with the same conclusion. We say $\mathcal{D}$ *locally reduces to* $\mathcal{E}$. Since local reductions are possible for both elimination rules for conjunction, our rules are locally sound. To show that the rules are locally complete we show how to reintroduce a conjunction from its components in the form of a local expansion.

$$\begin{array}{c}\mathcal{D}\\ A \wedge B\ true\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A \wedge B\ true\end{array}}{A\ true}\wedge\mathrm{E_L} \qquad \dfrac{\begin{array}{c}\mathcal{D}\\ A \wedge B\ true\end{array}}{B\ true}\wedge\mathrm{E_R}}{A \wedge B\ true}\wedge\mathrm{I}$$

Implication. To derive $A \supset B\ true$ we assume $A\ true$ and then derive $B\ true$. Written as a hypothetical judgment:

$$\dfrac{\begin{array}{c}\dfrac{}{A\ true}\,u\\ \vdots\\ B\ true\end{array}}{A \supset B\ true}\supset\mathrm{I}^u$$

We must be careful that the hypothesis $A\ true$ is available only in the derivation above the premiss. We therefore label the inference with the name of the hypothesis u, which must not be used already as the name for a hypothesis in the derivation of the premiss. We say that the hypothesis $A\ true$ labelled u is *discharged* at the inference labelled $\supset\mathrm{I}^u$. A derivation of $A \supset B\ true$ describes a construction by which we can transform a derivation of $A\ true$ into a derivation of $B\ true$: we substitute the derivation of $A\ true$ wherever we used the assumption $A\ true$ in the hypothetical derivation of $B\ true$. The elimination rule expresses this: if we have a derivation of $A \supset B\ true$ and also a derivation of $A\ true$, then we can obtain a derivation of $B\ true$.

$$\dfrac{A \supset B\ true \qquad A\ true}{B\ true}\supset\mathrm{E}$$

The local reduction rule carries out the substitution of derivations explained above.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{A\ true}\,u\\ \mathcal{D}\\ B\ true\end{array}}{A \supset B\ true}\supset\mathrm{I}^u \qquad \begin{array}{c}\mathcal{E}\\ A\ true\end{array}}{B\ true}\supset\mathrm{E} \quad \Longrightarrow_R \quad \dfrac{\begin{array}{c}\dfrac{\mathcal{E}}{A\ true}\,u\\ \mathcal{D}\\ B\ true\end{array}}{}$$

The final derivation depends on all the hypotheses of $\mathcal{E}$ and $\mathcal{D}$ except u, for which we have substituted $\mathcal{E}$. An alternative notation for this substitution of derivations for hypotheses is $[\mathcal{E}/u]\mathcal{D} :: B\ true$. The local reduction described above may significantly increase the overall size of the derivation, since the deduction $\mathcal{E}$ is substituted for each occurrence of the assumption labeled u in $\mathcal{D}$ and may thus be replicated many times. The local expansion simply rebuilds the implication.

$$\begin{array}{c}\mathcal{D}\\ A \supset B\ true\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\mathcal{D}}{A \supset B\ true} \qquad \dfrac{}{A\ true}\,u}{\dfrac{B\ true}{A \supset B\ true}\,\supset\mathrm{I}^u}\,\supset\mathrm{E}$$

Disjunction. $A \vee B$ should be true if either A is true or B is true. Therefore we have two introduction rules.

$$\frac{A\ true}{A \vee B\ true}\,\vee\mathrm{I_L} \qquad \frac{B\ true}{A \vee B\ true}\,\vee\mathrm{I_R}$$

If we have a hypothesis $A \vee B\ true$, we do not know how it might be inferred. That is, a proposed elimination rule

$$\frac{A \vee B\ true}{A\ true}\,?$$

would be incorrect, since a deduction of the form

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ B\ true\end{array}}{A \vee B\ true}\,\vee\mathrm{I_R}}{A\ true}\,?$$

cannot be reduced. As a consequence, the system would be *inconsistent*: if we have at least one theorem (B, in the example) we can prove every formula (A, in the example). How do we use the assumption $A \vee B$ in informal reasoning? We often proceed with a proof by cases: we prove a conclusion C under the assumption A and also show C under the assumption B. We then conclude C, since either A or B by assumption. Thus the elimination rule employs two hypothetical judgments.

$$\frac{A \vee B\ true \qquad \begin{array}{c}\dfrac{}{A\ true}\,u\\ \vdots\\ C\ true\end{array} \qquad \begin{array}{c}\dfrac{}{B\ true}\,w\\ \vdots\\ C\ true\end{array}}{C\ true}\,\vee\mathrm{E}^{u,w}$$

Now one can see that the introduction and elimination rules match up in two reductions. First, the case that the disjunction was inferred by $\vee\mathrm{I}_\mathrm{L}$.

$$\dfrac{\dfrac{\mathcal{D}}{A\ true}}{A \vee B\ true}\vee\mathrm{I}_\mathrm{L} \qquad \dfrac{\overline{}\,u}{A\ true} \quad \mathcal{E}_1 \quad C\ true \qquad \dfrac{\overline{}\,w}{B\ true} \quad \mathcal{E}_2 \quad C\ true}{C\ true}\vee\mathrm{E}^{u,w} \qquad \Longrightarrow_R \qquad \dfrac{\dfrac{\mathcal{D}}{A\ true}\,u}{\mathcal{E}_1\ C\ true}$$

The other reduction is symmetric.

$$\dfrac{\dfrac{\mathcal{D}}{B\ true}}{A \vee B\ true}\vee\mathrm{I}_\mathrm{R} \qquad \dfrac{\overline{}\,u}{A\ true} \quad \mathcal{E}_1 \quad C\ true \qquad \dfrac{\overline{}\,w}{B\ true} \quad \mathcal{E}_2 \quad C\ true}{C\ true}\vee\mathrm{E}^{u,w} \qquad \Longrightarrow_R \qquad \dfrac{\dfrac{\mathcal{D}}{B\ true}\,w}{\mathcal{E}_2\ C\ true}$$

As in the reduction for implication, the resulting derivation may be longer than the original one. The local expansion is more complicated than for the previous connectives, since we first have to distinguish cases and then reintroduce the disjunction in each branch.

$$\dfrac{\mathcal{D}}{A \vee B\ true} \qquad \Longrightarrow_E \qquad \dfrac{\mathcal{D}}{A \vee B\ true} \qquad \dfrac{\dfrac{\overline{}\,u}{A\ true}}{A \vee B\ true}\vee\mathrm{I}_\mathrm{L} \qquad \dfrac{\dfrac{\overline{}\,w}{B\ true}}{A \vee B\ true}\vee\mathrm{I}_\mathrm{R}}{A \vee B\ true}\vee\mathrm{E}^{u,w}$$

Negation. In order to derive $\neg A$ we assume A and try to derive a contradiction. Thus it seems that negation requires falsehood, and, indeed, in most literature on constructive logic, $\neg A$ is seen as an abbreviation of $A \supset \bot$. In order to give a self-contained explanation of negation by an introduction rule, we employ a judgment that is parametric in a propositional parameter p: If we can derive *any* p from the hypothesis A we conclude $\neg A$.

$$\dfrac{\begin{array}{c}\overline{}\,u\\ A\ true\\ \vdots\\ p\ true\end{array}}{\neg A\ true}\neg\mathrm{I}^{p,u} \qquad\qquad \dfrac{\neg A\ true \qquad A\ true}{C\ true}\neg\mathrm{E}$$

The elimination rule follows from this view: if we know $\neg A\ true$ and $A\ true$ then we can conclude any formula C is true. In the form of a local reduction:

$$\dfrac{\dfrac{\dfrac{}{A\ true}\,u}{\dfrac{\mathcal{D}}{p\ true}}\,\neg\mathrm{I}^{p,u} \qquad \dfrac{\mathcal{E}}{A\ true}}{C\ true}\,\neg\mathrm{E} \quad \Longrightarrow_R \quad \dfrac{\mathcal{E}}{A\ true}\,u \\ [C/p]\mathcal{D} \\ C\ true$$

The substitution $[C/p]\mathcal{D}$ is valid, since $\mathcal{D}$ is parametric in p. The local expansion is similar to the case for implication.

$$\dfrac{\mathcal{D}}{\neg A\ true} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\mathcal{D}}{\neg A\ true} \qquad \dfrac{}{A\ true}\,u}{\dfrac{p\ true}{\neg\ trueA}\,\neg\mathrm{I}^{p,u}}\,\neg\mathrm{E}$$

Truth. There is only an introduction rule for $\top$:

$$\dfrac{}{\top\ true}\,\top\mathrm{I}$$

Since we put no information into the proof of $\top$, we know nothing new if we have an assumption $\top$ and therefore we have no elimination rule and no local reduction. It may also be helpful to think of $\top$ as a 0-ary conjunction: the introduction rule has 0 premisses instead of 2 and we correspondingly have 0 elimination rules instead of 2. The local expansion allows the replacement of any derivation of $\top$ by $\top\mathrm{I}$.

$$\dfrac{\mathcal{D}}{\top\ true} \quad \Longrightarrow_E \quad \dfrac{}{\top\ true}\,\top\mathrm{I}$$

Falsehood. Since we should not be able to derive falsehood, there is no introduction rule for $\bot$. Therefore, if we can derive falsehood, we can derive everything.

$$\dfrac{\bot\ true}{C\ true}\,\bot\mathrm{E}$$

Note that there is no local reduction rule for $\bot\mathrm{E}$. It may be helpful to think of $\bot$ as a 0-ary disjunction: we have 0 instead of 2 introduction rules and we correspondingly have to consider 0 cases instead of 2 in the elimination rule. Even though we postulated that falsehood should not be derivable, falsehood could clearly be a consequence of contradictory assumption. For example, $A \wedge$

$\neg A \supset \bot$ *true* is derivable. While there is no local reduction rule, there still is a local expansion in analogy to the case for disjunction.

$$\begin{array}{c}\mathcal{D}\\ \bot\ true\end{array} \quad \Longrightarrow_E \quad \dfrac{\begin{array}{c}\mathcal{D}\\ \bot\ true\end{array}}{\bot\ true}\,\bot\mathrm{E}$$

Universal Quantification. Under which circumstances should $\forall x.\ A$ be true? This clearly depends on the domain of quantification. For example, if we know that x ranges over the natural numbers, then we can conclude $\forall x.\ A$ if we can prove $[0/x]A$, $[1/x]A$, *etc.* Such a rule is not effective, since it has infinitely many premisses. Thus one usually retreats to rules such as induction. However, in a general treatment of predicate logic we would like to prove statements which are true for *all* domains of quantification. Thus we can only say that $\forall x.\ A$ should be provable if $[a/x]A$ is provable for a new parameter a about which we can make no assumption. Conversely, if we know $\forall x.\ A$, we know that $[t/x]A$ for any term t.

$$\dfrac{[a/x]A\ true}{\forall x.\ A\ true}\,\forall\mathrm{I}^a \qquad\qquad \dfrac{\forall x.\ A\ true}{[t/x]A\ true}\,\forall\mathrm{E}$$

The label a on the introduction rule is a reminder the parameter a must be "new", that is, it may not occur in any undischarged assumption in the proof of $[a/x]A$ or in $\forall x.\ A$ itself. In other words, the derivation of the premiss must be parametric in a. The local reduction carries out the substitution for the parameter.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ [a/x]A\ true\end{array}}{\forall x.\ A\ true}\,\forall\mathrm{I}}{[t/x]A\ true}\,\forall\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}[t/a]\mathcal{D}\\ [t/x]A\ true\end{array}$$

Here, $[t/a]\mathcal{D}$ is our notation for the result of substituting t for the parameter a throughout the deduction $\mathcal{D}$. For this substitution to preserve the conclusion, we must know that a does not already occur in A. Similarly, we would change the hypotheses if a occurred free in any of the undischarged hypotheses of $\mathcal{D}$. This might render a larger proof incorrect. As an example, consider the formula $\forall x.\ \forall y.\ P(x) \supset P(y)$ which should clearly not be true for all predicates P. The

following is *not* a deduction of this formula.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(a)\ true}\,u}{\forall x.\ P(x)\ true}\,\forall \mathrm{I}^{a}?}{P(b)\ true}\,\forall \mathrm{E}}{P(a) \supset P(b)\ true}\,\supset \mathrm{I}^{u}}{\forall y.\ P(a) \supset P(y)\ true}\,\forall \mathrm{I}^{b}}{\forall x.\ \forall y.\ P(x) \supset P(y)\ true}\,\forall \mathrm{I}^{a}$$

The flaw is at the inference marked with "?," where a is free in the hypothesis labelled u. Applying a local proof reduction to the (incorrect) $\forall$I inference followed by $\forall$E leads to the the assumption $[b/a]P(a)$ which is equal to $P(b)$. The resulting derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P(b)\ true}\,u}{P(a) \supset P(b)\ true}\,\supset \mathrm{I}^{u}}{\forall y.\ P(a) \supset P(y)\ true}\,\forall \mathrm{I}^{b}}{\forall x.\ \forall y.\ P(x) \supset P(y)\ true}\,\forall \mathrm{I}^{a}$$

is once again incorrect since the hypothesis labelled u should read $P(a)$, not $P(b)$.

The local expansion for universal quantification is much simpler.

$$\begin{array}{c}\mathcal{D}\\ \forall x.\ A\ true\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \forall x.\ A\ true\end{array}}{[a/x]A\ true}\,\forall \mathrm{E}}{\forall x.\ A\ true}\,\forall \mathrm{I}^{a}$$

Existential Quantification. We conclude that $\exists x.\ A$ is true when there is a term t such that $[t/x]A$ is true.

$$\dfrac{[t/x]A\ true}{\exists x.\ A\ true}\,\exists \mathrm{I}$$

When we have an assumption $\exists x.\ A$ we do not know for which t it is the case that $[t/x]A$ holds. We can only assume that $[a/x]A$ holds for some parameter a about which we know nothing else. Thus the elimination rule resembles the

one for disjunction.

$$\dfrac{\exists x.\ A\ true \qquad \begin{array}{c}\overline{}\,u\\ [a/x]A\ true\\ \vdots\\ C\ true\end{array}}{C\ true}\ \exists \mathrm{E}^{a,u}$$

The restriction is similar to the one for $\forall$I: the parameter a must be new, that is, it must not occur in $\exists x.\ A$, C, or any assumption employed in the derivation of the second premiss. In the reduction rule we have to perform two substitutions: we have to substitute t for the parameter a and we also have to substitute for the hypothesis labelled u.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ [t/x]A\ true\end{array}}{\exists x.\ A}\ \exists \mathrm{I} \qquad \begin{array}{c}\overline{}\,u\\ [a/x]A\ true\\ \mathcal{E}\\ C\ true\end{array}}{C\ true}\ \exists \mathrm{E}^{a,u} \qquad \Longrightarrow_R \qquad \begin{array}{c}\dfrac{\begin{array}{c}\mathcal{D}\\ [t/x]A\ true\end{array}}{[t/a]A\ true}\,u\\ [t/a]\mathcal{E}\\ C\ true\end{array}$$

The proviso on occurrences of a guarantees that the conclusion and hypotheses of $[t/a]\mathcal{E}$ have the correct form. The local expansion for existential quantification is also similar to the case for disjunction.

$$\begin{array}{c}\mathcal{D}\\ \exists x.\ A\ true\end{array} \qquad \Longrightarrow_E \qquad \dfrac{\begin{array}{c}\mathcal{D}\\ \exists x.\ A\ true\end{array} \qquad \dfrac{\begin{array}{c}\overline{}\,u\\ [a/x]A\ true\end{array}}{\exists x.\ A\ true}\ \exists \mathrm{I}}{\exists x.\ A\ true}\ \exists \mathrm{E}^{a,u}$$

Here is a simple example of a natural deduction. We attempt to show the process by which such a deduction may have been generated, as well as the final deduction. The three vertical dots indicate a gap in the derivation we are trying to construct, with hypotheses and their consequences shown above and the desired conclusion below the gap.

$$\begin{array}{c}\vdots\\ A \wedge (A \supset B) \supset B\ true\end{array} \qquad \rightsquigarrow \qquad \dfrac{\begin{array}{c}\overline{}\,u\\ A \wedge (A \supset B)\ true\\ \vdots\\ B\ true\end{array}}{A \wedge (A \supset B) \supset B\ true}\ \supset \mathrm{I}^u$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{A \wedge (A \supset B)\ true}\,u}{A\ true}\wedge\mathrm{E_L} \qquad \vdots \\ B\ true}{A \wedge (A \supset B) \supset B\ true}\supset\mathrm{I}^u \qquad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{A \wedge (A \supset B)\ true}\,u}{A\ true}\wedge\mathrm{E_L} \qquad \dfrac{\dfrac{}{A \wedge (A \supset B)\ true}\,u}{A \supset B\ true}\wedge\mathrm{E_R} \\ \vdots \\ B\ true}{A \wedge (A \supset B) \supset B\ true}\supset\mathrm{I}^u$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{A \wedge (A \supset B)\ true}\,u}{A \supset B\ true}\wedge\mathrm{E_R} \qquad \dfrac{\dfrac{}{A \wedge (A \supset B)\ true}\,u}{A\ true}\wedge\mathrm{E_L}}{B\ true}\supset\mathrm{E} \\ \vdots \\ \dfrac{B\ true}{A \wedge (A \supset B) \supset B\ true}\supset\mathrm{I}^u$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{A \wedge (A \supset B)\ true}\,u}{A \supset B\ true}\wedge\mathrm{E_R} \qquad \dfrac{\dfrac{}{A \wedge (A \supset B)\ true}\,u}{A\ true}\wedge\mathrm{E_L}}{B\ true}\supset\mathrm{E} \\ \dfrac{B\ true}{A \wedge (A \supset B) \supset B\ true}\supset\mathrm{I}^u$$

The symbols A and B in this derivation stand for arbitrary propositions; we can thus established a judgment parametric in A and B. In other words, every instance of this derivation (substituting arbitrary propositions for A and B) is a valid derivation.

Below is a summary of the rules of intuitionistic natural deduction.

Introduction Rules

$$\frac{A\ true \qquad B\ true}{A \wedge B\ true}\ \wedge\mathrm{I}$$

$$\frac{A\ true}{A \vee B\ true}\ \vee\mathrm{I}_{\mathrm{L}} \qquad \frac{B\ true}{A \vee B\ true}\ \vee\mathrm{I}_{\mathrm{R}}$$

$$\frac{\begin{array}{c}\overline{}\,u\\ \vdots\\ B\ true\end{array}}{A \supset B\ true}\ \supset\mathrm{I}^{u}$$

$$\frac{\begin{array}{c}\overline{}\,u\\ \vdots\\ p\ true\end{array}}{\neg A\ true}\ \neg\mathrm{I}^{p,u}$$

$$\frac{}{\top\ true}\ \top\mathrm{I}$$

no $\bot$ *introduction*

$$\frac{[a/x]A\ true}{\forall x.\ A\ true}\ \forall\mathrm{I}^{a}$$

$$\frac{[t/x]A\ true}{\exists x.\ A\ true}\ \exists\mathrm{I}$$

Elimination Rules

$$\frac{A \wedge B\ true}{A\ true}\ \wedge\mathrm{E}_{\mathrm{L}} \qquad \frac{A \wedge B\ true}{B\ true}\ \wedge\mathrm{E}_{\mathrm{R}}$$

$$\frac{A \vee B\ true \qquad \begin{array}{c}\overline{}\,u\\ \vdots\\ C\ true\end{array} \qquad \begin{array}{c}\overline{}\,w\\ \vdots\\ C\ true\end{array}}{C\ true}\ \vee\mathrm{E}^{u,w}$$

$$\frac{A \supset B\ true \qquad A\ true}{B\ true}\ \supset\mathrm{E}$$

$$\frac{\neg A\ true \qquad A\ true}{C\ true}\ \neg\mathrm{E}$$

no $\top$ *elimination*

$$\frac{\bot\ true}{C\ true}\ \bot\mathrm{E}$$

$$\frac{\forall x.\ A\ true}{[t/x]A\ true}\ \forall\mathrm{E}$$

$$\frac{\exists x.\ A\ true \qquad \begin{array}{c}\overline{[a/x]A\ true}\,u\\ \vdots\\ C\ true\end{array}}{C\ true}\ \exists\mathrm{E}^{a,u}$$

2.2 Classical Logic

The inference rules so far only model *intuitionistic logic*, and some classically true propositions such as $A \vee \neg A$ (for an arbitrary A) are not derivable, as we will see in Section 3.5. There are three commonly used ways one can construct a system of *classical natural deduction* by adding one additional rule of inference. $\bot_C$ is called *Proof by Contradiction* or *Rule of Indirect Proof*, $\neg\neg_C$ is the *Double Negation Rule*, and XM is referred to as *Excluded Middle*.

$$\dfrac{\dfrac{\dfrac{}{\neg A}\,u}{\vdots}}{\dfrac{\bot}{A}\,\bot^u_C} \qquad \dfrac{\neg\neg A}{A}\,\neg\neg_C \qquad \dfrac{}{A \vee \neg A}\,\mathrm{XM}$$

The rule for classical logic (whichever one chooses to adopt) breaks the pattern of introduction and elimination rules. One can still formulate some reductions for classical inferences, but natural deduction is at heart an intuitionistic calculus. The symmetries of classical logic are much better exhibited in sequent formulations of the logic. In Exercise 2.3 we explore the three ways of extending the intuitionistic proof system and show that they are equivalent.

Another way to obtain a natural deduction system for classical logic is to allow multiple conclusions (see, for example, Parigot [Par92]).

2.3 Localizing Hypotheses

In the formulation of natural deduction from Section 2.1 correct use of hypotheses and parameters is a global property of a derivation. We can localize it by annotating each judgment in a derivation by the available parameters and hypotheses. We give here a formulation of natural deduction for intuitionistic logic with localized hypotheses, but not parameters. For this we need a notation for hypotheses which we call a *context*.

$$\textit{Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, u{:}A$$

Here, "$\cdot$" represents the empty context, and $\Gamma, u{:}A$ adds hypothesis $A\ \textit{true}$ labelled u to Γ. We assume that each label u occurs at most once in a context in order to avoid ambiguities. The main judgment can then be written as $\Gamma \vdash A$, where

$$\cdot, u_1{:}A_1, \ldots, u_n{:}A_n \vdash A$$

stands for

$$\dfrac{\dfrac{}{A_1\ \textit{true}}\,u_1 \quad \ldots \dfrac{}{A_n\ \textit{true}}\,u_n}{\vdots}$$
$$A\ \textit{true}$$

in the notation of Section 2.1.

We use a few important abbreviations in order to make this notation less cumbersome. First of all, we may omit the leading "$\cdot$" and write, for example, $u_1{:}A_1, u_2{:}A_2$ instead of $\cdot, u_1{:}A_1, u_2{:}A_2$. Secondly, we denote concatenation of contexts by overloading the comma operator as follows.

$$\begin{array}{rcl} \Gamma, \cdot & = & \Gamma \\ \Gamma, (\Gamma', u{:}A) & = & (\Gamma, \Gamma'), u{:}A \end{array}$$

With these additional definitions, the localized version of our rules are as follows.

Introduction Rules	Elimination Rules
$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge\mathrm{I}$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge\mathrm{E_L} \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge\mathrm{E_R}$
$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,\vee\mathrm{I_L} \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,\vee\mathrm{I_R}$	$\dfrac{\Gamma \vdash A \vee B \qquad \Gamma, u{:}A \vdash C \qquad \Gamma, w{:}B \vdash C}{\Gamma \vdash C}\,\vee\mathrm{E}^{u,w}$
$\dfrac{\Gamma, u{:}A \vdash B}{\Gamma \vdash A \supset B}\,\supset\mathrm{I}^{u}$	$\dfrac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\supset\mathrm{E}$
$\dfrac{\Gamma, u{:}A \vdash p}{\Gamma \vdash \neg A}\,\neg\mathrm{I}^{p,u}$	$\dfrac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash C}\,\neg\mathrm{E}$
$\dfrac{}{\Gamma \vdash \top}\,\top\mathrm{I}$	*no* $\top$ *elimination*
no $\bot$ *introduction*	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash C}\,\bot\mathrm{E}$
$\dfrac{\Gamma \vdash [a/x]A}{\Gamma \vdash \forall x.\, A}\,\forall\mathrm{I}^{a}$	$\dfrac{\Gamma \vdash \forall x.\, A}{\Gamma \vdash [t/x]A}\,\forall\mathrm{E}$
$\dfrac{\Gamma \vdash [t/x]A}{\Gamma \vdash \exists x.\, A}\,\exists\mathrm{I}$	$\dfrac{\Gamma \vdash \exists x.\, A \qquad \Gamma, u{:}[a/x]A \vdash C}{\Gamma \vdash C}\,\exists\mathrm{E}^{a,u}$

We also have a new rule for hypotheses which was an implicit property of the hypothetical judgments before.

$$\frac{}{\Gamma_1, u{:}A, \Gamma_2 \vdash A}\,u$$

Other general assumptions about hypotheses, namely that they may be used arbitrarily often in a derivation and that their order does not matter, are indirectly

reflected in these rules. Note that if we erase the context Γ from the judgments throughout a derivation, we obtain a derivation in the original notation.

When we discussed local reductions in order to establish local soundness, we used the notation

$$\begin{array}{c} \mathcal{D} \\ \hline A\ \mathit{true} \end{array} u \qquad \begin{array}{c} \mathcal{E} \\ C\ \mathit{true} \end{array}$$

for the result of substituting the derivation $\mathcal{D}$ of A *true* for all uses of the hypothesis A *true* labelled u in $\mathcal{E}$. We would now like to reformulate the property with localized hypotheses. In order to prove that the (now explicit) hypotheses behave as expected, we use the principle of *structural induction* over derivations. Simply put, we prove a property for all derivations by showing that, whenever it holds for the premisses of an inference, it holds for the conclusion. Note that we have to show the property outright when the rule under consideration has no premisses. Such rules are the base cases for the induction.

Theorem 2.1 (Structural Properties of Hypotheses) *The following properties hold for intuitionistic natural deduction.*

1. *(Exchange) If* $\Gamma_1, u_1{:}A, \Gamma_2, u_2{:}B, \Gamma_3 \vdash C$ *then* $\Gamma_1, u_2{:}B, \Gamma_2, u_1{:}A, \Gamma_3 \vdash C$.
2. *(Weakening) If* $\Gamma_1, \Gamma_2 \vdash C$ *then* $\Gamma_1, u{:}A, \Gamma_2 \vdash C$.
3. *(Contraction) If* $\Gamma_1, u_1{:}A, \Gamma_2, u_2{:}A, \Gamma_3 \vdash C$ *then* $\Gamma_1, u{:}A, \Gamma_2, \Gamma_3 \vdash C$.
4. *(Substitution) If* $\Gamma_1, u{:}A, \Gamma_2 \vdash C$ *and* $\Gamma_1 \vdash A$ *then* $\Gamma_1, \Gamma_2 \vdash C$.

Proof: The proof is in each case by straightforward induction over the structure of the first given derivation.

In the case of exchange, we appeal to the inductive assumption on the derivations of the premisses and construct a new derivation with the same inference rule. Algorithmically, this means that we exchange the hypotheses labelled u_1 and u_2 in every judgment in the derivation.

In the case of weakening and contraction, we proceed similarly, either adding the new hypothesis $u{:}A$ to every judgment in the derivation (for weakening), or replacing uses of u_1 and u_2 by u (for contraction).

For substitution, we apply the inductive assumption to the premisses of the given derivation $\mathcal{D}$ until we reach hypotheses. If the hypothesis is different from u we can simply erase $u{:}A$ (which is unused) to obtain the desired derivation. If the hypothesis is $u{:}A$ the derivation looks like

$$\mathcal{D} = \quad \frac{}{\Gamma_1, u{:}A, \Gamma_2 \vdash A}\, u$$

so $C = A$ in this case. We are also given a derivation $\mathcal{E}$ of $\Gamma_1 \vdash A$ and have to construct a derivation $\mathcal{F}$ of $\Gamma_1, \Gamma_2 \vdash A$. But we can just repeatedly apply weakening to $\mathcal{E}$ to obtain $\mathcal{F}$. Algorithmically, this means that, as expected, we

substitute the derivation $\mathcal{E}$ (possibly weakened) for uses of the hypotheses $u{:}A$ in $\mathcal{D}$. Note that in our original notation, this weakening has no impact, since unused hypotheses are not apparent in a derivation. □

It is also possible to localize the derivations themselves, using *proof terms*. As we will see in Section 2.4, these proof terms form a λ-calculus closely related to functional programming. When parameters, hypotheses, and proof terms are all localized our main judgment becomes decidable. In the terminology of Martin-Löf [ML94], the main judgment is then *analytic* rather than *synthetic*. We no longer need to go outside the judgment itself in order to collect evidence for it: An analytic judgment encapsulates its own evidence.

2.4 Proof Terms

The basic judgment of the system of natural deduction is the derivability of a formula A, written as $\vdash A$. It has been noted by Howard [How69] that there is a strong correspondence between (intuitionistic) derivations and λ-terms. The formulas A then act as types classifying λ-terms. In the propositional case, this correspondence is an isomorphism: formulas are isomorphic to types and derivations are isomorphic to simply-typed λ-terms. These isomorphisms are often called the *propositions-as-types* and *proofs-as-programs* paradigms.

If we stopped at this observation, we would have obtained only a fresh interpretation of familiar deductive systems, but we would not be any closer to the goal of providing a language for reasoning about properties of programs. However, the correspondences can be extended to first-order and higher-order logics. Interpreting first-order (or higher-order) formulas as types yields a significant increase in expressive power of the type system. However, maintaining an isomorphism during the generalization to first-order logic is somewhat unnatural and cumbersome. One might expect that a proof contains more information than the corresponding program. Thus the literature often talks about *extracting programs from proofs* or *contracting proofs to programs*. We do not discuss program extraction further in these notes.

We now introduce a notation for derivations to be carried along in deductions. For example, if M represents a proof of A and N represents a proof of B, then the pair $\langle M, N\rangle$ can be seen as a representation of the proof of $A \wedge B$ by $\wedge$-introduction. We write $\Gamma \vdash M : A$ to express the judgment *M is a proof term for A under hypotheses Γ*. We also repeat the local reductions and expansions from the previous section in the new notation. For local expansion we state the proposition whose truth must established by the proof term on the left-hand side. This expresses restrictions on the application of the expansion rules.

Conjunction. The proof term for a conjunction is simply the pair of proofs of the premisses.

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \wedge B}\,\wedge\mathrm{I}$$

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathrm{fst}\, M : A}\,\wedge\mathrm{E_L} \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathrm{snd}\, M : B}\,\wedge\mathrm{E_R}$$

The local reductions now lead to two obvious local reductions of the proof terms. The local expansion is similiarly translated.

$$\begin{array}{rcl} \mathrm{fst}\,\langle M, N\rangle & \longrightarrow_R & M \\ \mathrm{snd}\,\langle M, N\rangle & \longrightarrow_R & N \\ M : A \wedge B & \longrightarrow_E & \langle \mathrm{fst}\, M, \mathrm{snd}\, M\rangle \end{array}$$

Implication. The proof of an implication $A \supset B$ will be represented by a function which maps proofs of A to proofs of B. The introduction rule explicitly forms such a function by λ-abstraction and the elimination rule applies the function to an argument.

$$\frac{\Gamma, u{:}A \vdash M : B}{\Gamma \vdash (\lambda u{:}A.\ M) : A \supset B}\,\supset\mathrm{I}^u \qquad \frac{\Gamma \vdash M : A \supset B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}\,\supset\mathrm{E}$$

The binding of the variable u in the conclusion of $\supset$I correctly models the intuition that the hypothesis is discharged and not available outside deduction of the premiss. The abstraction is labelled with the proposition A so that we can later show that the proof term uniquely determines a natural deduction. If A were not given then, for example, $\lambda u.\ u$ would be ambigous and serve as a proof term for $A \supset A$ for any formula A. The local reduction rule is β-reduction; the local expansion is η-expansion.

$$\begin{array}{rcl} (\lambda u{:}A.\ M)\, N & \longrightarrow_R & [N/u]M \\ M : A \supset B & \longrightarrow_E & \lambda u{:}A.\ M\, u \end{array}$$

In the reduction rule, bound variables in M that are free in N must be renamed in order to avoid variable capture. In the expansion rule u must be new—it may not already occur in M.

Disjunction. The proof term for disjunction introduction is the proof of the premiss together with an indication whether it was inferred by introduction on the left or on the right. We also annotate the proof term with the formula which did not occur in the premiss so that a proof term always proves exactly one proposition.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{inl}^B\, M : A \vee B}\,\vee\mathrm{I_L} \qquad \frac{\Gamma \vdash N : B}{\Gamma \vdash \mathrm{inr}^A\, N : A \vee B}\,\vee\mathrm{I_R}$$

The elimination rule corresponds to a case construction.

$$\frac{\Gamma \vdash M : A \vee B \qquad \Gamma, u{:}A \vdash N_1 : C \qquad \Gamma, w{:}B \vdash N_2 : C}{\Gamma \vdash (\,\mathbf{case}\ M\ \mathbf{of}\ \mathrm{inl}\, u \Rightarrow N_1 \mid \mathrm{inr}\, w \Rightarrow N_2) : C}\ \vee\mathrm{E}^{u,w}$$

Since the variables u and w label assumptions, the corresponding proof term variables are *bound* in N_1 and N_2, respectively. The two reduction rules now also look like rules of computation in a λ-calculus.

$$\begin{array}{lcl} \mathbf{case}\ \mathrm{inl}^B\, M\ \mathbf{of}\ \mathrm{inl}\, u \Rightarrow N_1 \mid \mathrm{inr}\, w \Rightarrow N_2 & \longrightarrow_R & [M/u]N_1 \\ \mathbf{case}\ \mathrm{inr}^A\, M\ \mathbf{of}\ \mathrm{inl}\, u \Rightarrow N_1 \mid \mathrm{inr}\, w \Rightarrow N_2 & \longrightarrow_R & [M/w]N_2 \end{array}$$

$$M : A \vee B \quad \longrightarrow_E \quad \mathbf{case}\ M\ \mathbf{of}\ \mathrm{inl}\, u \Rightarrow \mathrm{inl}^B\, u \mid \mathrm{inr}\, w \Rightarrow \mathrm{inr}^A\, w$$

The substitution of a deduction for a hypothesis is represented by the substitution of a proof term for a variable.

Negation. This is similar to implication. Since the premise of the rule is parametric in p the corresponding proof constructor must bind a propositional variable p, indicated by μ^p. Similarly, the elimination construct must record the formula to maintain the property that every valid term proves exactly one proposition. This is indicated as a subscript C to the infix operator "$\cdot$".

$$\frac{\Gamma, u{:}A \vdash M : p}{\Gamma \vdash \mu^p u{:}A.\ M : \neg A}\ \neg\mathrm{I}^{p,u} \qquad\qquad \frac{\Gamma \vdash M : \neg A \qquad \Gamma \vdash N : A}{\Gamma \vdash M \cdot_C N : C}\ \neg\mathrm{E}$$

The reduction performs formula and proof term substitutions.

$$\begin{array}{rcl} (\mu^p u{:}A.\ M) \cdot_C N & \longrightarrow_R & [N/u][C/p]M \\ M : \neg A & \longrightarrow_E & \mu^p u{:}A.\ M \cdot_p u \end{array}$$

Truth. The proof term for $\top$I is written $\langle\,\rangle$.

$$\frac{}{\Gamma \vdash \langle\,\rangle : \top}\ \top\mathrm{I}$$

Of course, there is no reduction rule. The expansion rule reads

$$M : \top \quad \longrightarrow_E \quad \langle\,\rangle$$

Falsehood. Here we need to annotate the proof term abort with the formula being proved to avoid ambiguity.

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathrm{abort}^C\, M : C}\ \bot\mathrm{E}$$

Again, there is no reduction rule, only an expansion rule.

$$M : \bot \quad \longrightarrow_E \quad \mathrm{abort}^{\bot}\, M$$

In summary, we have

$$
\begin{array}{llll}
\text{Terms} & M & ::= & u & \textit{Hypotheses} \\
& & \mid & \langle M_1, M_2\rangle \mid \mathrm{fst}\, M \mid \mathrm{snd}\, M & \textit{Conjunction} \\
& & \mid & \lambda u{:}A.\ M \mid M_1\, M_2 & \textit{Implication} \\
& & \mid & \mathrm{inl}^A\, M \mid \mathrm{inr}^A\, M & \textit{Disjunction} \\
& & \mid & (\,\textbf{case}\ M\ \textbf{of}\ \mathrm{inl}\, u_1 \Rightarrow M_1 \mid \mathrm{inr}\, u_2 \Rightarrow M_2) & \\
& & \mid & \mu^p u{:}A.\ M \mid M_1 \cdot_A M_2 & \textit{Negation} \\
& & \mid & \langle\,\rangle & \textit{Truth} \\
& & \mid & \mathrm{abort}^A\, M & \textit{Falsehood}
\end{array}
$$

and the reduction rules

$$
\begin{array}{rcl}
\mathrm{fst}\,\langle M, N\rangle & \longrightarrow_R & M \\
\mathrm{snd}\,\langle M, N\rangle & \longrightarrow_R & N \\
(\lambda u{:}A.\ M)\, N & \longrightarrow_R & [N/u]M \\
\textbf{case}\ \mathrm{inl}^B\, M\ \textbf{of}\ \mathrm{inl}\, u \Rightarrow N_1 \mid \mathrm{inr}\, w \Rightarrow N_2 & \longrightarrow_R & [M/u]N_1 \\
\textbf{case}\ \mathrm{inr}^A\, M\ \textbf{of}\ \mathrm{inl}\, u \Rightarrow N_1 \mid \mathrm{inr}\, w \Rightarrow N_2 & \longrightarrow_R & [M/w]N_2 \\
(\mu^p u{:}A.\ M) \cdot_C N & \longrightarrow_R & [N/u][C/p]M \\
\textit{no rule for truth} & & \\
\textit{no rule for falsehood} & &
\end{array}
$$

The expansion rules are given below.

$$
\begin{array}{rcl}
M : A \wedge B & \longrightarrow_E & \langle \mathrm{fst}\, M, \mathrm{snd}\, M\rangle \\
M : A \supset B & \longrightarrow_E & \lambda u{:}A.\ M\, u \\
M : A \vee B & \longrightarrow_E & \textbf{case}\ M\ \textbf{of}\ \mathrm{inl}\, u \Rightarrow \mathrm{inl}^B\, u \mid \mathrm{inr}\, w \Rightarrow \mathrm{inr}^A\, w \\
M : \neg A & \longrightarrow_E & \mu^p u{:}A.\ M \cdot_p u \\
M : \top & \longrightarrow_E & \langle\,\rangle \\
M : \bot & \longrightarrow_E & \mathrm{abort}^{\bot}\, M
\end{array}
$$

We can now see that the formulas act as types for proof terms. Shifting to the usual presentation of the typed λ-calculus we use τ and σ as symbols for types, and $\tau \times \sigma$ for the product type, $\tau \to \sigma$ for the function type, $\tau + \sigma$ for the disjoint sum type, 1 for the unit type and 0 for the empty or void type. Base types b remain unspecified, just as the basic propositions of the propositional calculus remain unspecified. Types and propositions then correspond to each other as indicated below.

$$
\begin{array}{rcl}
\text{Types} & \tau & ::= b \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid 1 \mid 0 \\
\text{Propositions} & A & ::= p \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid A_1 \vee A_2 \mid \top \mid \bot
\end{array}
$$

We omit here the negation type which is typically not used in functional programming and thus does not have a well-known counterpart. We can think of $\neg A$ as corresponding to $\tau \to 0$, where τ corresponds to A. We now summarize and restate the rules above, using the notation of types instead of propositions (omitting only the case for negation). Note that contexts Γ now declare variables with their types, rather than hypothesis labels with their proposition.

$$\Gamma \rhd M : \tau \quad \textit{Term } M \textit{ has type } \tau \textit{ in context } \Gamma$$

$$\frac{\Gamma \rhd M : \tau \qquad \Gamma \rhd N : \sigma}{\Gamma \rhd \langle M, N \rangle : \tau \times \sigma}\ \mathsf{pair}$$

$$\frac{\Gamma \rhd M : \tau \times \sigma}{\Gamma \rhd \mathrm{fst}\, M : \tau}\ \mathsf{fst} \qquad \frac{\Gamma \rhd M : \tau \times \sigma}{\Gamma \rhd \mathrm{snd}\, M : \sigma}\ \mathsf{snd}$$

$$\frac{\Gamma, u{:}\tau \rhd M : \sigma}{\Gamma \rhd (\lambda u{:}\tau.\ M) : \tau \to \sigma}\ \mathsf{lam} \qquad \frac{u : \tau \text{ in } \Gamma}{\Gamma \rhd u : \tau}\ \mathsf{var}$$

$$\frac{\Gamma \rhd M : \tau \to \sigma \qquad \Gamma \rhd N : \tau}{\Gamma \rhd M\, N : \sigma}\ \mathsf{app}$$

$$\frac{\Gamma \rhd M : \tau}{\Gamma \rhd \mathrm{inl}^{\sigma}\, M : \tau + \sigma}\ \mathsf{inl} \qquad \frac{\Gamma \rhd N : \sigma}{\Gamma \rhd \mathrm{inr}^{\tau}\, N : \tau + \sigma}\ \mathsf{inr}$$

$$\frac{\Gamma \rhd M : \tau + \sigma \qquad \Gamma, u{:}\tau \rhd N_1 : \nu \qquad \Gamma, w{:}\sigma \rhd N_2 : \nu}{\Gamma \rhd (\ \mathbf{case}\ M\ \mathbf{of}\ \mathrm{inl}\, u \Rightarrow N_1 \mid \mathrm{inr}\, w \Rightarrow N_2) : \nu}\ \mathsf{case}$$

$$\frac{}{\Gamma \rhd \langle\,\rangle : 1}\ \mathsf{unit} \qquad \frac{\Gamma \rhd M : 0}{\Gamma \rhd \mathrm{abort}^{\nu}\, M : \nu}\ \mathsf{abort}$$

2.5 Exercises

Exercise 2.1 Prove the following by natural deduction using only intuitionistic rules when possible. We use the convention that $\supset$, $\wedge$, and $\vee$ associate to the right, that is, $A \supset B \supset C$ stands for $A \supset (B \supset C)$. $A \equiv B$ is a syntactic abbreviation for $(A \supset B) \wedge (B \supset A)$. Also, we assume that $\wedge$ and $\vee$ bind more tightly than $\supset$, that is, $A \wedge B \supset C$ stands for $(A \wedge B) \supset C$. The scope of a quantifier extends as far to the right as consistent with the present parentheses. For example, $(\forall x.\ P(x) \supset C) \wedge \neg C$ would be disambiguated to $(\forall x.\ (P(x) \supset C)) \wedge (\neg C)$.

1. $\vdash A \supset B \supset A$.
2. $\vdash A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$.
3. (Peirce's Law). $\vdash ((A \supset B) \supset A) \supset A$.
4. $\vdash A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$.
5. $\vdash A \supset (A \wedge B) \vee (A \wedge \neg B)$.
6. $\vdash (A \supset \exists x.\ P(x)) \equiv \exists x.\ (A \supset P(x))$.
7. $\vdash ((\forall x.\ P(x)) \supset C) \equiv \exists x.\ (P(x) \supset C)$.

8. $\vdash \exists x.\, \forall y.\, (P(x) \supset P(y))$.

Exercise 2.2 We write $A \vdash B$ if B follows from hypothesis A and $A \dashv\vdash B$ for $A \vdash B$ and $B \vdash A$. Which of the following eight parametric judgments are derivable intuitionistically?

1. $(\exists x.\, A) \supset B \dashv\vdash \forall x.\, (A \supset B)$
2. $A \supset (\exists x.\, B) \dashv\vdash \exists x.\, (A \supset B)$
3. $(\forall x.\, A) \supset B \dashv\vdash \exists x.\, (A \supset B)$
4. $A \supset (\forall x.\, B) \dashv\vdash \forall x.\, (A \supset B)$

Provide natural deductions for the valid judgments. You may assume that the bound variable x does not occur in B (items 1 and 3) or A (items 2 and 4).

Exercise 2.3 Show that the three ways of extending the intuitionistic proof system for classical logic are equivalent, that is, the same formulas are deducible in all three systems.

Exercise 2.4 Assume we had omitted disjunction and existential quantification and their introduction and elimination rules from the list of logical primitives. In the classical system, give a definition of disjunction and existential quantification (in terms of other logical constants) and show that the introduction and elimination rules now become *admissible rules of inference*. A rule of inference is *admissible* if any deduction using the rule can be transformed into one without using the rule.

Exercise 2.5 Assume we would like to design a system of natural deduction for a simple temporal logic. The main judgment is now "*A is true at time t*" written as

$$A \mathbin{@} t.$$

1. Explain how to modify the given rules for natural deduction to this more general judgment and show the rules for implication and universal quantification.
2. Write out introduction and elimination rules for the temporal operator $\bigcirc A$ which should be true if A is true at the next point in time. Denote the "next time after t" by $t + 1$.
3. Show the local reductions and expansions which show the local soundness and completness of your rules.
4. Write out introduction and elimination rules for the temporal operator $\Box A$ which should be true if A is true at all times.
5. Show the local reductions and expansions.

Exercise 2.6 Design introduction and elimination rules for the connectives

1. $A \equiv B$, usually defined as $(A \supset B) \wedge (B \supset A)$,
2. $A \mid B$ (exclusive or), usually defined as $(A \wedge \neg B) \vee (\neg A \wedge B)$,

without recourse to other logical constants or operators. Also show the corresponding local reductions and expansions. For each of the following proposed connectives, write down appropriate introduction and eliminations rules and show the local reductions and expansion or indicate that no such rule may exist.

3. $A \overline{\wedge} B$ for $\neg(A \wedge B)$,
4. $A \overline{\vee} B$ for $\neg(A \vee B)$,
5. $A \overline{\supset} B$ for $\neg(A \supset B)$,
6. $+A$ for $\neg\neg A$,
7. $\exists^* x.\ A$ for $\neg\forall x.\ \neg A$,
8. $\forall^* x.\ A$ for $\neg\exists x.\ \neg A$,
9. $A \Rightarrow B \mid C$ for $(A \supset B) \wedge (\neg A \supset C)$.

Exercise 2.7 A given introduction rule does not necessarily uniquely determine matching elimination rules and vice versa. Explore if the following alternative rules are also sound and complete.

1. Replace the two elimination rules for conjunction by

$$\frac{A \wedge B \ \mathit{true} \qquad \begin{array}{c}\dfrac{}{A\ \mathit{true}}\,u \quad \dfrac{}{B\ \mathit{true}}\,w \\ \vdots \\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\,\wedge\mathrm{E}^{u,w}$$

2. Add the following elimination rule for truth.

$$\frac{\top\ \mathit{true} \qquad C\ \mathit{true}}{C\ \mathit{true}}\,\top\mathrm{E}$$

3. Add the following introduction rule for falsehood.

$$\frac{p\ \mathit{true}}{\bot\ \mathit{true}}\,\bot\mathrm{I}^{p}$$

Consider if any other of the standard connectives might permit alternative introduction or elimination rules which preserve derivability.

Exercise 2.8 For each of 14 following proposed entailments either write out a proof term for the corresponding implication or indicate that it is not derivable.

1. $A \supset (B \supset C) \dashv\vdash (A \wedge B) \supset C$
2. $A \supset (B \wedge C) \dashv\vdash (A \supset B) \wedge (A \supset C)$
3. $A \supset (B \vee C) \dashv\vdash (A \supset B) \vee (A \supset C)$
4. $(A \supset B) \supset C \dashv\vdash (A \vee C) \wedge (B \supset C)$
5. $(A \vee B) \supset C \dashv\vdash (A \supset C) \wedge (B \supset C)$
6. $A \wedge (B \vee C) \dashv\vdash (A \wedge B) \vee (A \wedge C)$
7. $A \vee (B \wedge C) \dashv\vdash (A \vee B) \wedge (A \vee C)$

Exercise 2.9 The de Morgan laws of classical logic allow negation to be distributed over other logical connectives. Investigate which directions of the de Morgan equivalences hold in intuitionistic logic and give proof terms for the valid entailments.

1. $\neg(A \wedge B) \dashv\vdash \neg A \vee \neg B$
2. $\neg(A \vee B) \dashv\vdash \neg A \wedge \neg B$
3. $\neg(A \supset B) \dashv\vdash A \wedge \neg B$
4. $\neg(\neg A) \dashv\vdash A$
5. $\neg\top \dashv\vdash \bot$
6. $\neg\bot \dashv\vdash \top$
7. $\neg\forall x.\, A \dashv\vdash \exists x.\, \neg A$
8. $\neg\exists x.\, A \dashv\vdash \forall x.\, \neg A$

Exercise 2.10 An alternative approach to negation is to introduce another judgment, A *is false*, and develop a system of evidence for this judgment. For example, we might say that $A \wedge B$ is false if either A is false or B is false. Similarly, $A \vee B$ is false if both A and B are false. Expressed as inference rules:

$$\frac{A \mathit{\ false}}{A \wedge B \mathit{\ false}} \qquad \frac{B \mathit{\ false}}{A \wedge B \mathit{\ false}} \qquad \frac{A \mathit{\ false} \qquad B \mathit{\ false}}{A \vee B \mathit{\ false}}$$

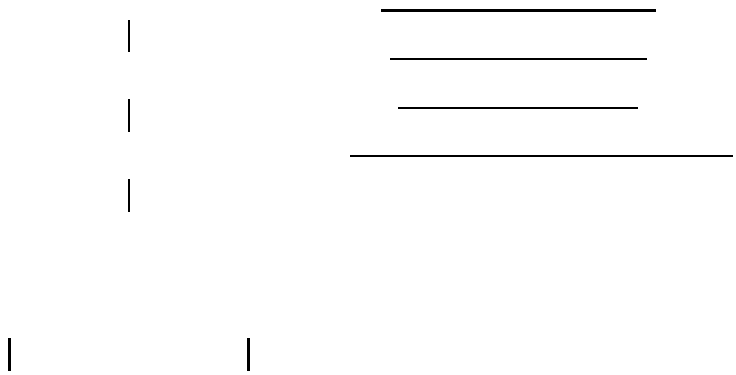
1. Write out a complete set of rules defining the judgment A *false* for the conjunction, implication, disjunction, truth, and falsehood.
2. Verify local soundness and completeness of your rules, if these notions make sense.

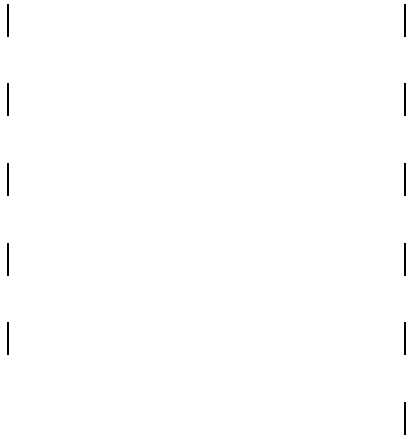
3. Now we define that $\neg A$ *true* if A *false*. Complete the set of rules and verify soundness and completeness if appropriate.

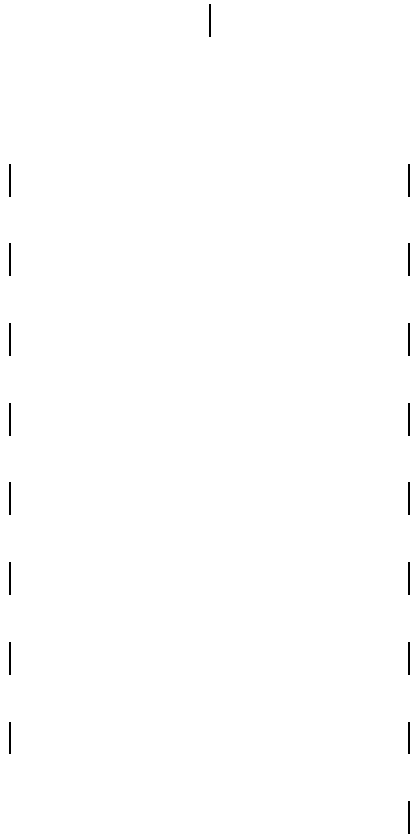
4. Does your system satisfy that every proposition A is either true or false? If so, prove it. Otherwise, show a counterexample.

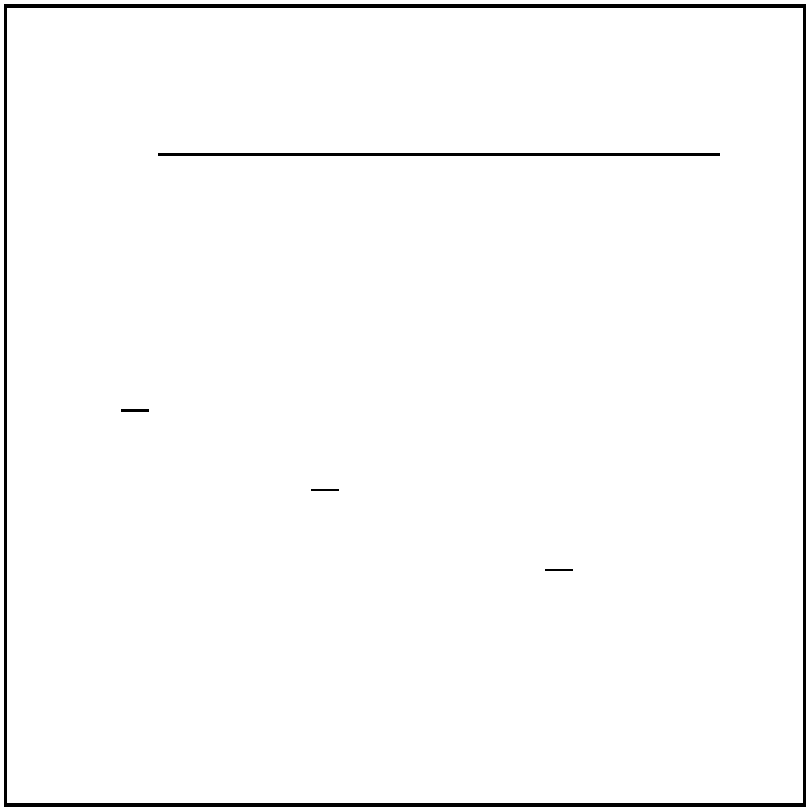
5. Compare this notion of negation with the standard notion in intuitionistic logic.

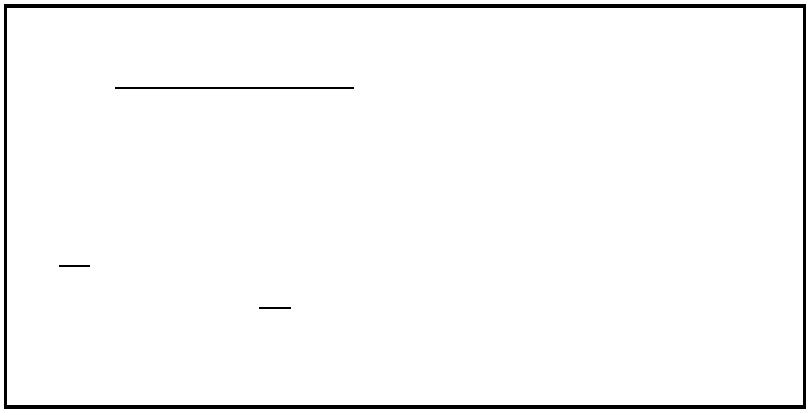
6. Extend your system to include universal and existential quantification (if possible) and discuss its properties.

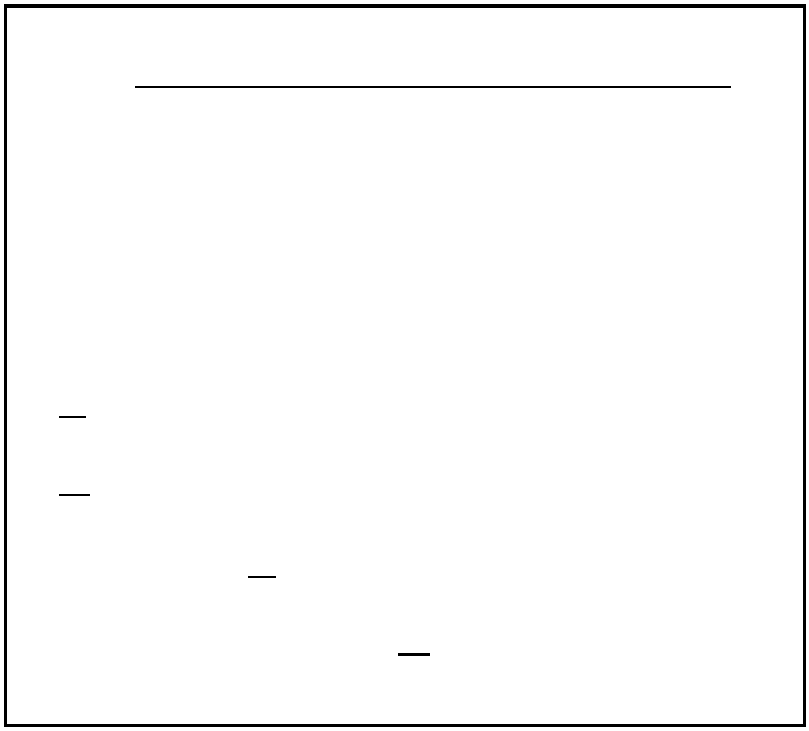


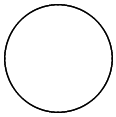


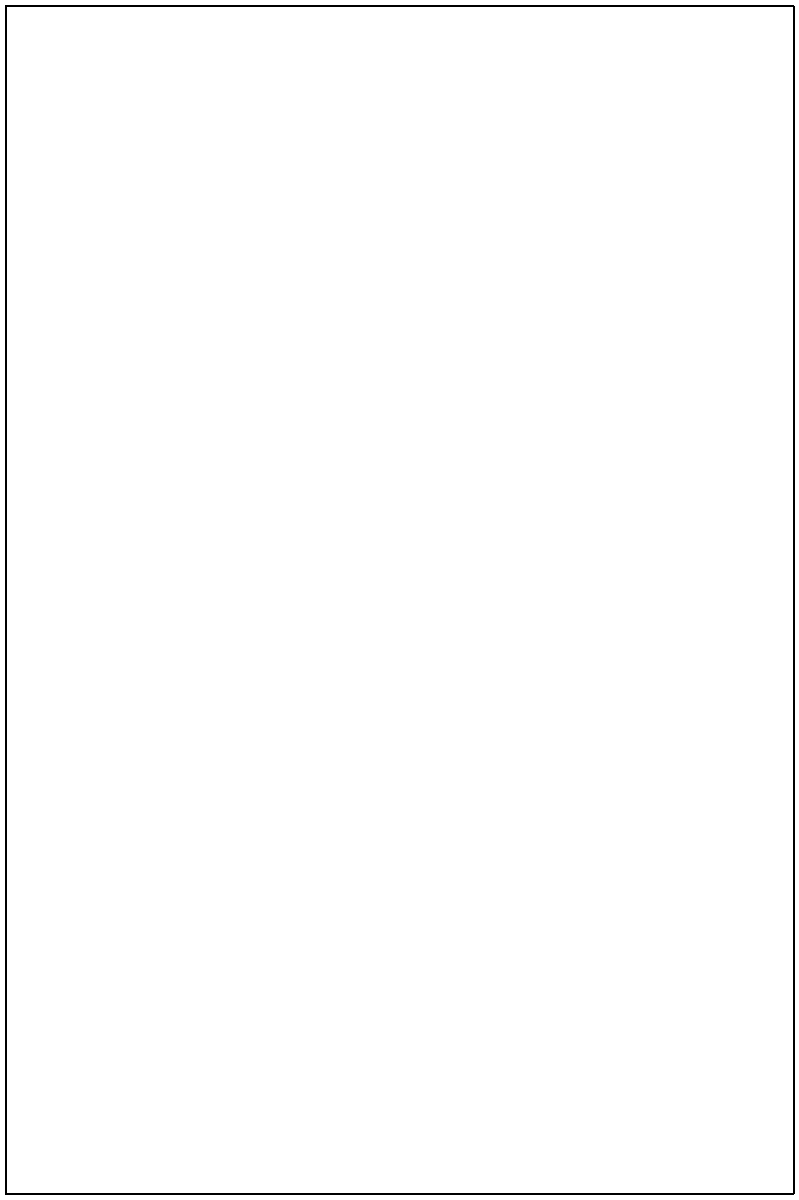


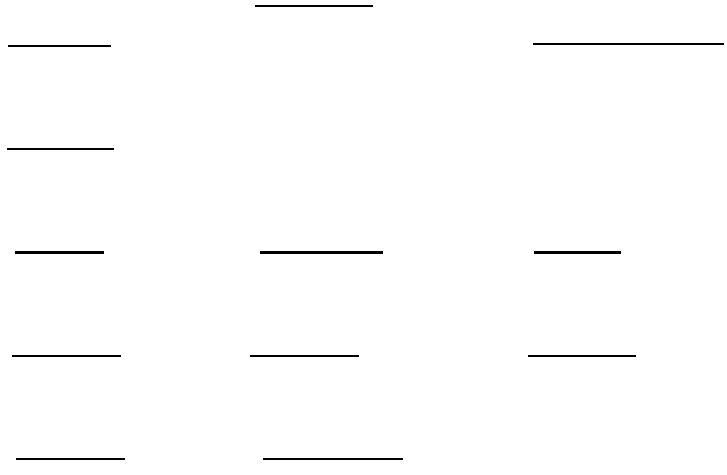


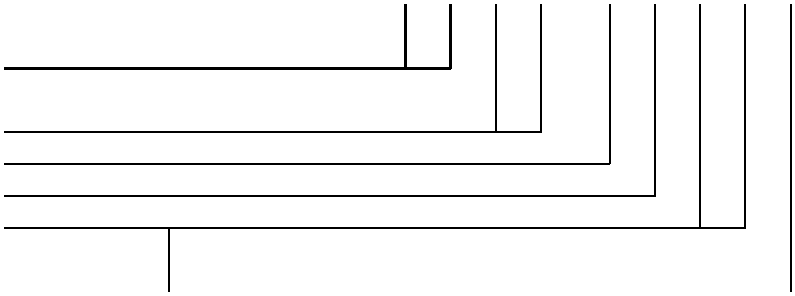












Logique et Analyse,
28 : 1985
119-136 June-Sept 1985

Clare Dixon

THE TABLEAU METHOD FOR TEMPORAL LOGIC: AN OVERVIEW

Pierre WOLPER

Abstract: An overview of the tableau decision method for propositional temporal logic is presented. The method is described in detail for linear time temporal logic. It is then discussed how it can be applied to other variants of temporal logic like branching time temporal logic and extensions of linear time temporal logic. Finally, applications of temporal logics to computer science are reviewed.

1. *Introduction*

Temporal logic (TL) has been studied as a branch of logic for several decades. It was developed as a logical framework to formalize reasoning about time, temporal relations such as "before" or "after", and related concepts such as tenses. TL is closely related to the modal logic of necessity ([HC68]) which attempts to formalize the notions of possible and necessary truth. As temporal logic can in fact be viewed as a special case of modal logic, its origins can also be traced to those of that theory. Tableau decision procedures for temporal and modal logic have been known for some time. An account of earlier work on temporal logic can be found in either the book of Prior ([Pr67]) or of Rescher and Urquhart ([RU71]).

In [Pn77], it was suggested that temporal logic could be a useful tool to formalize reasoning about the execution sequence of programs and especially of concurrent programs. In that approach, the sequence of states a machine goes through during a computation is viewed as the temporal sequence of worlds described by TL. Since then, several researchers have been using TL to state and prove properties of concurrent programs (*e.g.* [GPSS80], [La80], [OL80], [BP80], [MP81], [CES83], [MP83], [MW84]), protocols (*e.g.* [Ha80], [HO81], [SM81], [SM82], [Vo82], [SS82], [SMV83], [SPE84], and hardware (*e.g.* [Bo82], [MO81], [HMM83], [Mo83]).

In this paper, we first define the propositional version of linear time temporal logic. We then describe the tableau decision procedure for PTL. We also review other variants of temporal logics and state results about their decision problems. And finally, we discuss the use of temporal logic in computer science.

2. *Propositional Temporal Logic*

We define here the propositional version of temporal logic (PTL). We have chosen one of the most common versions appearing in the computer science literature. In contrast to the temporal logics studied in philosophical logic it contains only temporal operators dealing with the "future" and no operators concerning the "past".

Syntax: PTL *formulas* are built from

- A set $\mathscr{P}$ of atomic propositions: $p_1, p_2, p_3, \ldots$
- Boolean connectives: $\wedge$, $\neg$
- Temporal operators: $\bigcirc$ ("next"), $\Diamond$ ("eventually"), U ("until").

The formation rules are:

- An atomic proposition $p \in \mathscr{P}$ is a formula.
- If f_1 and f_2 are formulas, so are
$$f_1 \wedge f_2,\ \neg f_1,\ \bigcirc f_1,\ \Diamond f_1,\ f_1 U f_2.$$

We use $\Box$ ("always") as an abbreviation for $\neg\Diamond\neg$. We also use $\vee$ and $\supset$ as the usual abbreviations, and parentheses to resolve ambiguities.

Semantics: A *structure* for a PTL formula (with set $\mathscr{P}$ of atomic propositions) is a triple $\mathscr{A} = (S, N, \pi)$ where

- S is a finite or enumerable set of states.
- N: $(S \to S)$ is a total successor function that for each state gives a unique next state.
- π: $(S \to 2^{\mathscr{P}})$ assigns truth values to the atomic propositions of the language in each state.

For a structure $\mathscr{A}$ and a state $s \in S$ we have

$$\begin{aligned}
&\langle \mathscr{A}, s\rangle \models p \quad \text{iff} \quad p \in \pi(s)\\
&\langle \mathscr{A}, s\rangle \models f_1 \wedge f_2 \quad \text{iff} \quad \langle \mathscr{A}, s\rangle \models f_1 \text{ and } \langle \mathscr{A}, s\rangle \models f_2\\
&\langle \mathscr{A}, s\rangle \models \neg f \quad \text{iff} \quad \text{not } \langle \mathscr{A}, s\rangle \models f\\
&\langle \mathscr{A}, s\rangle \models \bigcirc f \quad \text{iff} \quad \langle \mathscr{A}, N(s)\rangle \models f
\end{aligned}$$

In the following definitions, we denote by $N^i(s)$ the i^{th} state in the sequence

$$s, N(s), N(N(s)), N(N(N(s))), \ldots$$

of successsors of a state s.

$$\begin{aligned}
&\langle \mathscr{A}, s\rangle \models \Diamond f \quad \text{iff} \quad (\exists i \geq 0)(\langle \mathscr{A}, N^i(s)\rangle \models f)\\
&\langle \mathscr{A}, s\rangle \models f_1 U f_2 \quad \text{iff} \quad (\exists i \geq 0)(\langle \mathscr{A}, N^i(s)\rangle \models f_2 \wedge\\
&\qquad\qquad \forall j\,(0 \leq j < i \supset \langle \mathscr{A}, N^j(s)\rangle \models f_1))
\end{aligned}$$

An *interpretation* $\mathscr{I} = \langle \mathscr{A}, s_0\rangle$ for PTL consists of a structure $\mathscr{A}$ and an initial state $s_0 \in S$. We will say that an interpretation $\mathscr{I} = \langle \mathscr{A}, s_0\rangle$ *satisfies* a formula f iff $\langle \mathscr{A}, s_0\rangle \models f$. Since an interpretation $\mathscr{I}$ uniquely determines a sequence

$$\sigma = s_0, N(s_0), N^2(s_0), N^3(s_0), \ldots$$

we will often say "the sequence σ satisfies a formula" instead of "the interpretation $\mathscr{I}$ satisfies a formula". The satisfiability problem for PTL is, given a formula f, to determine if there is some interpretation that satisfies f (*i.e.*, a *model* of f). In the next section, we describe the tableau method for the satisfiability problem of PTL.

Examples:

1) The formula

$$p$$

is satisfied by all sequences in which p is true in the first state.

2) The formula

$$\Box(p \supset \bigcirc q)$$

is satisfied by all sequences where each state in which p is true is followed by a state in which q is true.

3) The formula

$$\Box(p \supset \bigcirc(\neg q \, U \, r))$$

is satisfied by all sequences where if p is true in a given state, then, from the next state on, q is always false until the first state where r is true.

4) The formula

$$\Box \Diamond p$$

is satisfied by all sequences in which p is true infinitely often.

5) The formula

$$\Box p \wedge \Diamond \neg p$$

is not satisfied by any sequence.

6) The formula

$$\Diamond p \supset (\neg p \, U \, p)$$

is satisfied by all sequences (it is valid).

3. *The Tableau Method for PTL*

The tableau method for PTL is an extension of the tableau method for propositional logic (see [Sm68]). Boolean connectives are handled exactly as for propositional logic and temporal connectives are handled by decomposing them into a requirement on the "current state" and a requirement on "the rest of the sequence". This decomposition of the temporal connectives makes the tableau into a state by state search for a model of the formula being considered. The decomposition rules for the temporal operators are based on the following identities:

$$\Diamond f \equiv f \vee \bigcirc \Diamond f \tag{1}$$

$$f_1 \, U \, f_2 \equiv (f_2 \vee (f_1 \wedge \bigcirc(f_1 \, U \, f_2))). \tag{2}$$

Given that decomposing formulas using identities (1-2) does not make them smaller, the tableau might be infinite. However, we will insure that the tableau is finite by identifying the nodes of the tableau that are labeled by the same set of formulas. Note that in the tableau method

for propositional logic this is not necessary as the tableau is always a finite tree. Another difference is that obtaining a tableau with no propositional inconsistencies for a formula f is not a guarantee that f is satisfiable. Indeed, formulas of the form $\Diamond f_1$ or $(f_1 \, U \, f_2)$ which we call *eventualities* might not be satisfied. The problem is that, for instance for a formula $\Diamond f_1$, the tableau rule based on (1) will allow us to continuously postpone the point at which f_1 is satisfied. This corresponds to always choosing the $\bigcirc \Diamond f$ disjunct in (1). We will thus have to add an extra step to the tableau method which will eliminate nodes containing eventualities that are not satisfiable.

More precisely, to test a PTL formula f for satisfiability, the tableau method constructs a directed graph. Each node n of the graph is labeled by a set of formulas T_n. Initially, the graph contains exactly one node, labeled by $\{f\}$. To describe the construction of the graph, we distinguish between elementary and non-elementary formulas. Elementary formulas are those whose main connective is $\bigcirc$ (we call these $\bigcirc$-formulas) or that are either atomic propositions or negations of atomic propositions. The construction of the graph proceeds by using the following decomposition rules which map each non-elementary formula f into a set Σ of sets S_i of formulas f_i:

$$\begin{array}{ll}
\neg\neg f & \rightarrow \{\{f\}\} \\
\neg\bigcirc f & \rightarrow \{\{\bigcirc \neg f\}\} \\
f_1 \wedge f_2 & \rightarrow \{\{f_1, f_2\}\} \\
\neg(f_1 \wedge f_2) & \rightarrow \{\{\neg f_1\}, \{\neg f_2\}\} \\
\Diamond f & \rightarrow \{\{f\}, \{\bigcirc \Diamond f\}\} \\
\neg\Diamond f & \rightarrow \{\{\neg f, \neg\bigcirc \Diamond f\}\} \\
f_1 \, U \, f_2 & \rightarrow \{\{f_2\}, \{f_1, \bigcirc(f_1 \, U \, f_2)\}\} \\
\neg(f_1 \, U \, f_2) & \rightarrow \{\neg f_2, \neg f_1 \vee \neg\bigcirc(f_1 \, U \, f_2)\}
\end{array}$$

During the construction, to avoid decomposing the same formula twice, we will mark the formulas to which a decomposition rule has been applied (we don't simply discard them as we will need them when checking if eventualities are satisfied). Once the graph is constructed, we eliminate unsatisfiable nodes.

The graph construction proceeds as follows:

1) Start with a node labeled by $\{f\}$ where f is the formula to be tested. We will call f the *initial formula* and the corresponding node the

initial node. Then repeatedly apply steps 2) and 3). In these steps, when we say "create a son of node n labeled by a set of formulas T", we mean create a node if the graph does not already contain a node labeled by T. If it does, we just create an edge from n to the already existing node.

2) If a node n labeled by T_n contains an unmarked non-elementary formula f and the tableau rule for f is $f \rightarrow \{S_i\}$, then, for each S_i, create a son of n labeled by $(T_n - \{f\}) \cup S_i \cup \{f^*\}$ where f^* is f marked.

3) If a node n contains only elementary and marked formulas, then create a son of n labeled by the $\bigcirc$-formulas $\in T_n$ with their outermost $\bigcirc$ removed.

A node containing only elementary or marked formulas will be called a *state*. And, a node that is either the initial node or the immediate son of a state will be called a *pre-state*.

Given the form of the tableau rules, the formulas labeling the nodes of the graph are subformulas or negations of subformulas of the initial formula or such formulas preceded by $\bigcirc$. The number of these formulas is equal to $4l$, where l is the length of the initial formula. The number of nodes in the graph is then at most equal to the number of sets of such formulas, that is 2^{4l}.

At this point, to decide satisfiability, we have to eliminate the unsatisfiable nodes of the graph. We repeatedly apply the following three rules.

E1: If a node contains both a proposition p and its negation $\neg p$, eliminate that node.

E2: If all the successors of a node have been eliminated, eliminate that node.

E3: If a node which is a pre-state contains a formula of the form $\Diamond f$ or $f_1 \, U \, f_2$ that is not satisfiable (see below), eliminate that node.

To determine if a formula $\Diamond f$ or $f_1 \, U \, f_2$ is satisfiable, one uses the following rule:

F1: A formula $\Diamond f_2$ or $f_1 \, U \, f_2$ is satisfiable in a pre-state, if there is a path in the tableau leading from that pre-state to a node containing the formula f_2.

The decision procedure ends after all unsatisfiable nodes have been

eliminated. If the initial node has been eliminated, then the initial formula is unsatisfiable, if not it is satisfiable.

Example:

Consider the formula $\Box p \wedge \Diamond \neg p$. The tableau obtained for this formula by the algorithm we have just described is the one appearing in figure 1.

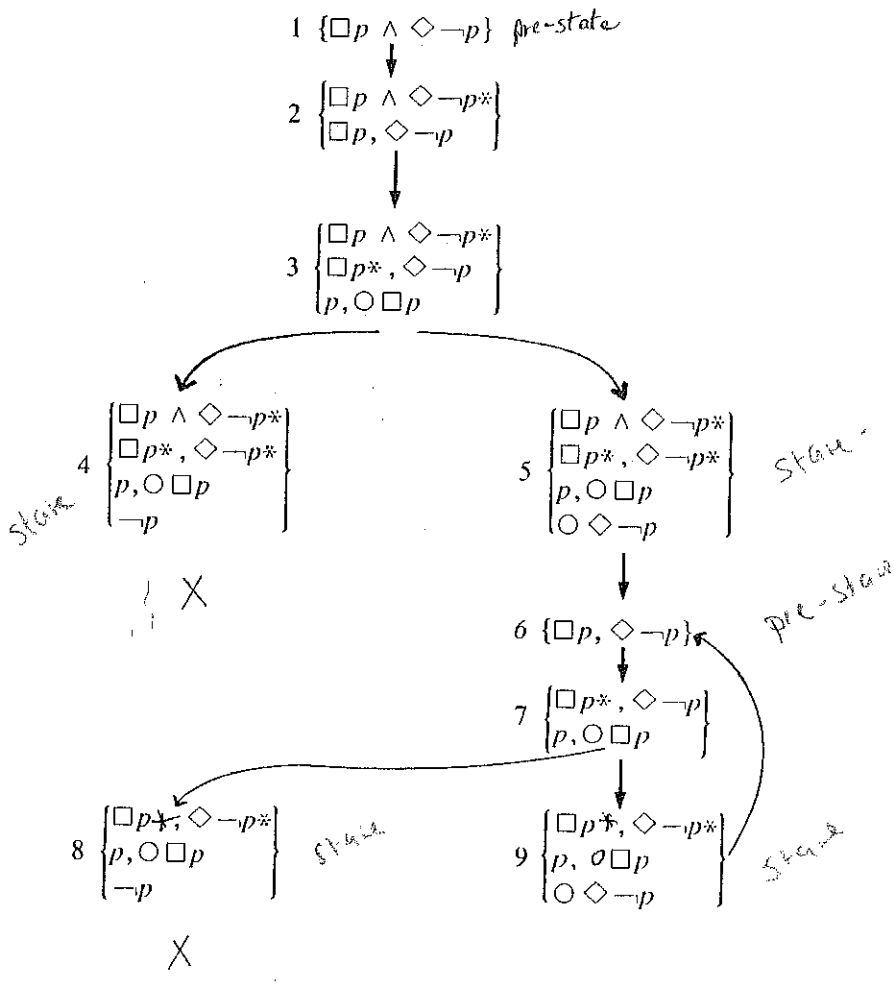


Figure 1

In this tableau, the initial node is 1; the states are 4, 5, 8 and 9; and the pre-states are 1 and 6. Nodes 4 and 8, contain a proposition (p) and its negation, they are thus unsatisfiable and are eliminated by rule E1. Node 6 is a pre-state and contains an eventuality formula ($\Diamond \neg p$) that is not satisfiable as there is no path leading from 6 to a node containing $\neg p$ (node 8 is eliminated). The other nodes are then eliminated by rule E2 and the formula is found to be unsatisfiable.

It is easy to see that the decision procedure requires time and space exponential in the length l of the initial formula. Actually, it is possible to test a PTL formula for satisfiability using only polynomial space. The satisfiability problem for PTL is in fact complete in PSPACE. For a discussion of the complexity of PTL, see [SC82] and [WVS83]. The correctness of the tableau method we have just described is established by the following theorem:

Theorem 1: An PTL formula f is satisfiable iff the initial node of the graph generated by the tableau decision procedure for that formula is not eliminated.

Proof:

a) If the initial node is eliminated, then f is unsatisfiable. (Completeness)

We prove by induction that if a node in the tableau labeled by $\{f_1, \ldots, f_s\}$ is eliminated, then $\{f_1, \ldots, f_s\}$ is unsatisfiable.

Case 1: The node was eliminated by rule E1. It thus contains a proposition and its negation and is unsatisfiable.

Case 2: The node is eliminated by rule E2 and is not a state. The sons of that node were created using a tableau rule $f \rightarrow \{S_i\}$. It is easy to check that for each of these tableau rules, f is satisfiable iff at least one of the S_i is satisfiable. As all the successor nodes have been eliminated, they all contain unsatisfiable sets of formulas and the initial node contains the unsatisfiable formula f.

Case 3: The node is eliminated by rule E2 and is a state. Thus, the set of all the O-formulas in the node is unsatisfiable and so is the set of all formulas in the node.

Case 4: The node was eliminated by using rule E3. Hence, there is an eventuality in the node that is not satisfiable on any path in the tableau. As any model corresponds to some path in the tableau, the eventuality is unsatisfiable and so is the set of all formulas in the node.

b) If the initial node is not eliminated, then f is satisfiable. (Soundness)

To prove this, we have to show that if the initial node is not eliminated, there is a model of the initial formula. First notice that except for satisfying evenualities, a path through the tableau starting with the initial node defines a model of the initial formula. We thus only have to show that we can construct a path through the tableau on which all eventualities are satisfied. It can be done as follows.

For each pre-state in the graph, unwind a path from that pre-state such that all the eventualities it contains are satisfied on that path. This is done by satisfying the eventualities one by one. If one of the eventualities is selected, it is possible to find a path on which that eventuality is satisfied and whose last node is a pre-state (if not the node would have been eliminated). Now, given the tableau rules, the eventualities that are not satisfied will appear in the last node of that path and hence the path can be extended to satisfy a second eventuality. By repeating this construction, one obtains a path on which all eventualities are satisfied. Once all these paths are constructed, we link them together. The model obtained has the following form:

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots . s_j \rightarrow \ldots . s_m .$$

A complete axiomatization of PTL can be obtained from the tableau method in the usual way. The only point worthy of interest is the axiom corresponding to the elmination rule E3. That axiom is basically an induction axiom for the $\Diamond$ operator. [Wo83] contains a complete axiomatization of PTL and a proof of completeness.

4. *Other Temporal Logics*

Extensions of Linear Time Temporal Logic

A simple property like *even* (p), meaning that p is true in every even state of a sequence and might be true or false in odd states is not expressible in the temporal logic we have discussed in the previous section. This led to the definition of an extended temporal logic (ETL) in [Wo83]. The extended temporal logic is based on the observation that one can extend PTL with operators corresponding to arbitrary right-linear grammars. The operators O, $\Diamond$, and U of PTL are in fact

special cases of this more general class of operators. One of the important features of ETL is that it also has a tableau decision procedure similar to the one we have described for PTL (see [Wo83]). In [WVS83], several alternative definitions of ETL are considered and their expressiveness and complexity are characterized.

Quantified Temporal Logic

Another way to extend linear time propositional temporal logic is to allow quantification of propositional variables. The resulting logic, quantified propositional temporal logic (QPTL) is decidable, but unfortunately is of non-elementary complexity. The known decision procedures are by reduction to SnS or by using automata theoretic techniques ([Si83], [Wo82]). Interestingly, the expressiveness of QPTL and of ETL is the same [WVS83].

Branching Time Temporal Logics

In the interpretations of linear time temporal logic, each state has exactly one successor. That is the reason for the name "linear time". If one deals with a situation where there are several possible futures (in computer science, this is the case if one deals with a non-deterministic program), the "linear time" assumption no longer holds. This led to the development of branching time temporal logics (BTL) that are interpreted over structures where each state can have several successors.

Formulas of branching time temporal logic are similar to formulas of linear time temporal logic with the addition of *path quantifiers*. Path quantifiers ($\forall$ and $\exists$) are used to specify to which paths the temporal logic formula applies. For example, $\forall \Diamond p$ is true in a state, if on all paths from that state there is a state satisfying p. Depending on how the path quantifiers and temporal logic formulas are allowed to interact, one defines a variety of branching time temporal logics.

One of the simplest of the branching time temporal logics is the logic *UB* described in [BMP81]. In that logic, path quantifiers and temporal operators are required to always appear together. In other words, *UB* can be viewed as PTL where each of the temporal operators $\bigcirc$, $\Diamond$, and U is replaced by two operators, $\forall\bigcirc$ and $\exists\bigcirc$, $\forall\Diamond$ and $\exists\Diamond$, $\forall U$ and $\exists U$. The logic *UB* also has a tableau decision procedure similar to the one described here for PTL, though slightly more complicated as

the logic is interpreted over branching rather than linear structures. *UB* also has a simple complete axiomatization. However, the complexity of the decision problem for *UB* is EXPTIME rather than PSPACE as it is for the linear time temporal logics. In [EH82], one can find a thorough description of tableau-like decision procedures for branching time temporal logic.

At the other end of the spectrum is the logic *CTL** described in [EH83]. In *CTL**, one allows path quantifiers to apply to arbitrary linear time temporal logic formulas. For instance $\forall(\Box\Diamond p \wedge q\, U(\bigcirc r))$ is a *CTL** formula. *CTL** is strictly more expressive than *UB*. Also, it is possible to define several logics that are in between *CTL** and *UB* as far as expressiveness. [EH83] discusses all these logics and compares their expressiveness. As far as decision procedures, the tableau method that we presented here does not extend naturally to *CTL**. At this point the only decision procedures that are known for *CTL** are based on automata theoretic methods and require time triply or quadruply exponential (see [ES84], [VW83] and [VW84] for a description of these decision procedures). No simple complete axiomatization is known for *CTL**. Also, no precise caracterization of its complexity is known. The best lower bound obtained is exponential, whereas the best upperbound is triply exponential.

Interval Temporal Logic

A variation of temporal logic that has appeared recently is interval temporal logic. Interval temporal logic is a variant of linear time temporal logic that allows explicit description of intervals. Two variants are currently in existence. One described in [SMV83] was developed as a more convenient higher level extension of PTL for the specification and verification of protocols. Its formulas combine the description of an interval and a temporal logic statement concerning that interval. For instance the formula $[I]\,\Box p$ states that the first time the interval I appears, it satisfies $\Box p$ (*i.e.*, all its states satisfy p). It is not more expressive than PTL and there is a translation from its formulas to PTL formulas. This gives a decision procedure for the logic though it is no more a tableau decision procedure closely linked to the syntax of the logic [Pl83]. This interval temporal logic has no known simple complete axiomatization.

A second interval temporal logic is described in [HMM83] and

[Mo83]. It was designed with the description of hardware as a goal. In this logic, all statements are about intervals. Its fundamental operations are O which here maps an interval into its tail (the same interval with the first state removed) and concatenation of intervals (;). These two simple constructs make it into a very powerful language. Unfortunately, it is undecidable in the general case and, the only known decidable subset is of non-elementary complexity (see [Mo83]).

Probabilistic Temporal Logic

Yet another variant of temporal logic is probabilistic temporal logic. Its development has been motivated by the appearance of probabilistic algorithms. As branching time temporal logic, probabilistic temporal logic is interpreted over structures in which states have more than one successor. The difference is that a probability is associated with each transition. The formulas of the logic then state that a given linear time temporal logic formula holds on a set of paths that has probability one. Three different variants of probabilistic temporal logic are described in [LS82]. For each of these a complete axiomatization is given. In [KL83] double exponential decision procedures are given for these logics.

First Order Temporal Logic

Up to this point we have been talking about propsitional temporal logics. However, first order temporal logics are often used when for instance stating properties of programs. Unfortunately, though axiomatizations for first order temporal logics have been proposed (*e.g.*, [Ma81]), they are not complete.

5. *Use of Temporal Logic in Computer Science*

In the last few years, temporal logic has been used in several different areas of computer science. The main ones are the following.

Stating and Proving Properties of Concurrent Programs

This was the initial motivation when temporal logic was introduced to the computer science community in [Pn77]. When reasoning about a concurrent program, it is not sufficient to deal with its input output behavior, one has to consider the entire computation sequence. As temporal logic is geared towards describing sequences, it appeared well suited for this problem.

In this approach, one views the execution sequence of a program as the sequence of states TL describes and one can state properties of that sequence. As TL formulas do not allow us to explicitely represent the program, it has to be encoded in a set of statements that basically represent the allowable transitions in each state. This approach has been further developed in [MP81], and [MP83].

Related methods for specifying and proving the correctness of concurrent programs are described in [OL82] and [La83]. In [OL82] a proof method called *proof lattices* was introduced.

A method to check that finite state programs satisfy some temporal logic specifications was proposed in [CES83]. The idea is that a finite state program can be viewed as a structure over which temporal logic formulas can be interpreted. The problem of checking that the program satisfies a given temporal formula is then equivalent to the problem of checking that the structure corresponding to that program is a model of the formula. In [CES83], it was pointed out that if one uses the branching time logic *UB*, this problem is in polynominal time, which leads to attractive algorithms. Similar ideas were developed in [QS82].

Another application related to the one we are now describing is the study of fairness conditions and properties. When several processes are running concurrently, the outcome of executing the program can depend on how ressources are allocated to the various processes. For example, if ressources are allocated evenly to all processes, the program might terminate whereas if one of the processes does not receive any ressources the program might get blocked. This had led to specifying *fairness conditions* on the execution of concurrent programs. These fairness conditions require a somewhat even distribution of ressources among the various processes. A large number of different conditions have been proposed. Temporal logic has appeared to be a useful tool for stating and reasoning about these conditions (see [LPS81], [QS82b], and [Pn83]).

Synthesis of Concurrent Programs

A direct use of the decision procedure we described in this paper has been the synthesis of the synchronization part of concurrent programs. If one assumes that the various parts of a concurrent program only interact through a finite number of signals, then their interaction can be specified in propositional temporal logic. Now, if one applies the tableau decision procedure to this specification, one obtains a graph that can be viewed as a program satisfying those specifications. Indeed, all executions of the program (paths through the graph) satisfy the specification (if one ensures that eventualities are satisfied). This approach was developed in [Wo82] and [MW84] using a linear time temporal logic and in [CE81] using a branching time temporal logic. A more informal approach to synthesis from temporal logic specifications appears in [RK80].

Specification and Correctness of Protocols

Communication protocols are another area where temporal logic has been applied. Protocols can often be hard to implement correctly and analyse. This is due to the fact that they are often quite intricate and can exhibit unexpected behaviors. This has led to a lot of interest in formal methods to specify and reason about protocols. Among these methods temporal logic has played an important role. Again it is its ability to describe sequences (*e.g.*, the sequence of communications a protocol performs) that has made it attractive for this application (see [Hai80], [HO81], [SM81], [SS82], [SMV83], [SPE84]).

Hardware

The development of always larger integrated circuits has made the need for tools to specify and reason about such circuits more and more necessary. Several researchers have tried to apply temporal logic to this problem ([Bo82], [MO81], [HMM83[, [Mo83]). The technique of model checking introduced in [CES83] for concurrent programs was applied to circuits in [CM84].

AT&T Bell Laboratories
Murray Hill, NJ 07974

Pierre WOLPER

REFERENCES

[BMP81] M. Ben-Ari, Z. Manna, A. Pnueli, "The Temporal Logic of Branching Time", *Eighth ACM Symposium on Principles of Programming Languages*, Williamsburg, VA, January 1981, pp. 164-176.

[Bo82] G.V. Bochmann, "Hardware Specification with Temporal Logic: An Example", *IEEE Transactions on Computers*, C-31(3), March 1982, pp. 223-231.

[BP80] M. Ben-Ari, A. Pnueli, "Temporal Logic Proofs of Concurrent Programs", Technical Report, Department of Mathematical Sciences, Tel-Aviv University, 1980.

[CE81] E.M. Clarke, E.A. Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic", in *Logics of Programs*, Lecture Notes in Computer Science vol. 131, Springer-Verlag, Berlin, 1982, pp. 52-71.

[CES83] E.M. Clarke, E.A. Emerson, A.P. Sistla, "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, Austin, January 1984, pp. 117-126.

[CM84] E. Clarke, S. Mishra, "Automatic Verification of Asynchronous Circuits", in *Logics of Programs*, Springer-Verlag Lecture Notes in Computer Science, Vol. 164, Berlin, 1984, pp. 101-115.

[EC80] E.A. Emerson, E.M. Clarke, "Characterizing Correctness Properties of Parallel Programs as Fixpoints", *Proc. 7th Int. Colloquium on Automata, Languages and Programming*, Lecture notes in Computer Science vol. 85, Springer-Verlag, Berlin, 1981, pp. 169-181.

[EH82] E.A. Emerson, J.Y. Halpern, "Decision Procedures and Expressiveness in the Temporal Logic of Branching Time", *Proceedings of the 14th Symposium on Theory of Computing*, San Francisco, CA, May 1982, pp. 169-180.

[EH83] E.A. Emerson, J.Y. Halpern, "Sometimes and Not Never Revisited: On Branching Versus Linear Time", *Proceedings of the 10th Symposium on Principles of Programming Languages*, Austin, January 1983, pp. 127-140.

[ES84] E.A. Emerson, A.P. Sistla, "Deciding Branching Time Logic", *Proc. 16th ACM Symposium on Theory of Computing*, Washington, May 1984, pp. 14-24.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah and J. Stavi, "The Temporal Analysis of Fairness", *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 163-173.

[Hai80] B.T. Hailpern, "Verifying Concurrent Processes Using Temporal Logic", Ph.D. Thesis, Stanford University, 1980.

[HC68] G.E. Hughes, M.J. Cresswell, *An Introduction to Modal Logic*, Methuen and Co, London, 1968.

[HMM83] J. Halpern, Z. Manna, B. Moszkowski, "A Hardware Semantics Based on temporal Intervals", *Proc. 10th International Colloquium on Automata Languages and Programming*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1983.

[HO81] B.T. Hailpern, S. Owicki, "Modular Verification of Computer Communication Protocols", Research Report RC 8726, IBM T.J. Watson Research Center, March 81.

[KL83] S. Kraus, D. Lehman, "Decision Procedures for Time and Chance", *Proc. 24th IEEE Symposium on Foundations of Computer Science*, Tucson, November 1983, pp. 202-209.

[Lam80] L. Lamport, "Sometimes is Sometimes Not Never", *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 174-185.

[LPS81] D. Lehman, A. Pnueli, J. Stavi, "Impartiality, Justice and Fairness: The Ethics of Concurrent Termination", *Proc. 8th International Colloquium on Automata languages and Programming*, Lecture Notes in Computer Science, Springer-Verlag, Berlin 1981, pp. 264-277.

[LS82] D. Lehman, S. Shelah, "Reasoning with Time and Chance", *Information and Control*, Vol. 53, 1982, pp. 165-198.

[Ma81] Z. Manna, "Verification of Sequential Programs: Temporal Axiomatization", *Theoretical Foundations of Programming Methodology* (F.L. Bauer, E.W. Dijkstra, C.A.R. Hoare, eds.), NATO Scientific Series, D. Reidel Pub. Co., Holland, 1981.

[Mo83] B. Moszkowski, "Reasoning about Digital Circuits", Ph.D. Thesis, Department of Computer Science, Stanford University, July 1983.

[MO81] Y. Malachi and S. Owicki "Temporal Specifications of Self-Timed Systems" in *VLSI Systems and Computations*, H.T. Kung, Bob Sproul, and Guy Steele editors, Computer Science Press 1981, pp. 203-212.

[MP81] Z. Manna, A. Pnueli, "Verification of Concurrent Programs: the Temporal Framework", *The Correctness Problem in Computer Science* (R.S. Boyer and J.S. Moore, eds.), International Lecture Series in Computer Science, Academic Press, London, 1981, pp. 215-273.

[MP83] Z. Manna, A. Pnueli, "How to Cook a Temporal Proof System for your Pet Language, *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, Austin, January 1984, pp. 141-154.

[MW84] Z. Manna, P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 1, January 1984, pp. 68-93.

[OL82] S. Owicki, L. Lamport, "Proving Liveness Properties of Concurrent Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, 1982, pp. 455-496.

[P183] D. Plaisted, "An Intermediate-Level Language for Obtaining Decision Procedures for a Class of Temporal Logics", Technical Report, Computer Science Laboratory, SRI, 1983.

[Pn77] A. Pnueli, "The Temporal Logic of Programs", *Proceedings of the Eighteenth Symposium on Foundations of Computer Science*, Providence, RI, November 1977, pp. 46-57.

[Pn83] A. Pnueli, "On the Extremely Fair Treatment of Probabilistic Algorithms", *Proc. 15th ACM Symposium on Theory of Computing*, Boston, April 1983, pp. 278-289.

[Pr67] A. Prior, *Past, Present and Future*, Oxford University Press, 1967.

[QS82] J.P. Queille, J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR", Lecture Notes in Computer Science, Vol. 137 Springer-Verlag, Berlin, 1982, pp. 337-351.

[QS82b] J.P. Queille, J. Sifakis, "A Temporal Logic to Deal with Fairness in Transition Systems", *Proc. IEEE Symposium on foundations of Computer Science*, Chicago, 1982, pp. 217-225.

[RK80] K. Ramamritham, R.M. Keller, "Specification and Synthesis of Synchronizers", *Proceedings International Symposium on Parallel Processing*, August 1980, pp. 311-321.

[RU71] N. Rescher, A. Urquart, *Temporal Logic*, Springer-Verlag, 1971.

[SC82] A.P. Sistla, E.M. Clarke, "The Complexity of Propositional Linear Temporal Logic", *Proceedings of the 14th ACM Symposium on Theory of Computing*, San Francisco, CA, May 1982, pp. 159-168.

[Si83] A.P. Sistla, "Theoretical Issues in the Design and Verification of Distributed Systems", Ph.D. Thesis, Harvard University, 1983.

[Sm68] R.M. Smullyan, *First Order Logic*, Springer-Verlag, Berlin, 1968.

[SM81] R.L. Schwartz, P.M. Melliar-Smith, "Temporal Logic Specification of Distributed Systems", *Proceedings of the Second International Conference on Distributed Systems*, Paris, France, April 1981.

[SM82] R.L. Schwartz, P.M. Melliar-Smith, "From State Machines to Temporal Logic: Specification Methods for Protocol Standards", *IEEE Transactions on Communications*, December 1982.

[SMV83] R.L. Schwartz, P.M. Melliar-Smith, F.H. Vogt, "An Interval Logic for Higher-Level Temporal Reasoning", *Proc. 2nd ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983, pp. 173-186.

[SPE84] D.E. Shasha, A.Pnueli, W. Ewald, "Temporal Verification of Carrier-Sense Local Area Network Protocols", *Proc. 11th ACM Symposium on Principles of Programming Languages*, Salt Lake City, January 1984, pp. 54-65.

[SS82] K. Sabnani, M. Schwartz, "Verification of a Multidestination Protocol Using Temporal Logic", in *Protocol Specification Testing and Verification*, C. Sunshine ed., North-Holland, 1982, pp. 21-42.

[Vo82] F.H. Vogt, "Event-Based Temporal Logic Specification of Services and Protocols", in *Protocol Specification, Testing and Verification*, North-Holland Publishing, 1982.

[VW83] M.Y. Vardi, P. Wolper, "Yet Another Process Logic", in *Logics of Programs*, Springer-Verlag Lecture Notes in Computer Science, Vol. 164, Berlin, 1984, pp. 501-512.

[VW84] M.Y. Vardi, P. Wolper, "Automata Theoretic Techniques for Modal Logics of Programs", *Proc. 16th ACM Symposium on Theory of Computing*, Washington, May 1984, pp. 446-456.

[Wo82] P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", Ph.D. Thesis, Stanford University, August 1982.

[Wo83] P. Wolper, "Temporal Logic Can Be More Expressive", *Information and Control*, Vol. 56, Nos. 1-2, 1983, pp. 72-99.

[WVS83] P. Wolper, M.Y. Vardi, A.P. Sistla, "Reasoning about Infinite Computation Paths", *Proc. 24th IEEE Symposium on Foundations of Computer Science*, Tucson, 1983, pp. 185-194.

COMPUTATION OF TEMPORAL OPERATORS

Max MICHEL

Abstract: This paper exhibits a strong correspondance between linear temporal logic and sequential machines with infinite input. The method is effective and modular. It enables us to synthetise processes from (generalised) temporal logic formulae and solve logic problems by algorithms applied to machines. It increases our understanding of the way temporal operators work.

1) *Introduction*

Nowadays parallelism is modelled in quite different manners. In Temporal Logic, presented by Pnueli [PNU], it is possible to express the behaviours of parallel programs and to make proofs; it is well fitted to describe, for instance, termination, fairness, absence of deadlocks.

Other people prefere to start from the solid ground of formal language theory and wish to express the problems of concurrency in terms of language and automata (Arnold, Nivat [A-N], [NIV]: they talk of "process languages" and it is then possible to express different properties such as being normal or central).

There exists a connection between these two approaches: it is known that some parts of linear temporal logic can be encoded into second order monadic arithmetics; and Büchi [BUC] gave a decision procedure for this logic by establishing a correspondence between formulae and finite state automata with infinite input. But the methods and the produced automata were rather difficult to use and to understand.

Our method explains more precisely the relationship between a certain temporal logic and automata: as a matter of fact, from a semantic point of view, the operators or the formulae of discrete propositional linear temporal logic can be seen as automata with outputs or nondeterministic sequential machines: they take as input

Automata and Logic

Igor Walukiewicz
Warsaw University[1]

April 2002[2]

Contents

[1]LaBRI, Domaine Universitaire, btiment A30, 351, cours de la Libration, 33405 Talence Cedex, FRANCE. e-mail:igw@labri.fr; www: http://www.labri.fr/∼igw

[2]These are notes for EFF Summer School, July 2001, with minor modifications.

1 Introduction

The connections between automata theory and logic are long and fruitful. Good examples of this are Büchi's proof of decidability of monadic second-order (MSO) theory of infinite words and Rabin's proof of decidability of MSO theory of infinite binary trees. This last theory is one of the strongest known decidable logical theories. Recently, automata play a prominent role in understanding of many logical formalisms used in Computer Aided Verification. Of particular interest to us here will be various versions of the μ-calculus.

An equivalence between MSOL, automata and the μ-calculus is an important and useful property. MSOL gives the guarantee of expressive power as the MSOL properties are by definition closed under Boolean operations and quantification. Automata are the main technical tool in analyzing the logic and in particular for inexpressibility results. They are also crucial in the algorithmic problems. The μ-calculus offers a logical formalism which is usually of much smaller complexity than MSOL. Still, it is a usable formalism in which many interesting properties can be formulated succinctly. Sometimes, as in the case of graphs, the μ-calculus gives a recursive syntax for an interesting, but not recursive, subset of MSOL.

In these notes we will show many situations in which the three formalisms are equivalent. We start with classical equivalences between MSOL, automata and the μ-calculus over words and trees. The tools we develop allow us then to study and compare hierarchies in all of the formalisms. Finally, we describe extensions of the classical results in three directions. We consider, a more general, relational setting, obtaining so called guarded logics. We also discuss the extensions to trace models and to real-time models.

These notes are not intended to be a survey of the results in the area. Some important aspects, as for example the relations with first-order logic, are omitted here. We refer the reader to excellent surveys and books [52, 53, 50, 7].

2 Finite words

Let Σ be a *finite alphabet*. A *finite word* over Σ is a sequence $w = a_0 \dots a_n$ of letters from Σ, or equivalently a function from $\{0, \dots, n\}$ to Σ. We use $|w|$ for the length of w, i.e., $n+1$ in this case. We use $\mathrm{dom}(w)$ for the domain of w, i.e., $\{0, \dots, n\}$. The empty word is denoted by ε. We write Σ^* for the set of all finite words over Σ.

The word w as above can be represented as a relational structure:

$$\mathcal{M}_w = \langle \mathrm{dom}(w), \leq, (P_a^w)_{a \in \Sigma} \rangle$$

where $\leq$ is the standard linear order on $\mathrm{dom}(w)$ and P_a^w are unary predicates with the interpretation: $P_a^w = \{i \in \mathrm{dom}(w) : w(i) = a\}$.

2.1 First-order and monadic second-order logics

The relational structures as above can be described in first-order or second-order logics which we are going to define now. Let $Var_1 = \{x, y, \dots\}$ be the set of *first order variables*. The set of first-order formulas is build from *atomic formulas* of the form:

$$x \leq y, \quad P_a(x) \qquad \text{for every } a \in \Sigma$$

using the connectives $\vee$, $\neg$ and the quantifier $\exists$. A *sentence* is a formula without free variables.

The meaning of a formula in a model of a form $\mathcal{M}_w$ is defined in a standard way. In particular the variables range over positions in w. If $p_1, \dots, p_n \in \mathrm{dom}(w)$ are positions in w and $\varphi(x_1, \dots, x_n)$ is a formula then $\mathcal{M}_w, p_1, \dots, p_n \vDash \varphi(x_1, \dots, x_n)$ means that φ holds in $\mathcal{M}_w$ when each x_i is interpreted as the position p_i. Other connectives as $\wedge$, $\Rightarrow$ and universal quantifier $\forall$ are definable in the usual way. We admit the empty model, $\mathcal{M}_\varepsilon$, as interpretation. In this model all existential sentences $\exists x.\varphi$ are false.

The *language defined by a sentence* φ is the set of words:

$$L(\varphi) = \{w \in \Sigma^* : \mathcal{M}_w \vDash \varphi\}$$

For example the sentence $\forall x.P_a(x) \Rightarrow (\exists y.\ x \leq y \wedge P_b(y))$ defines the language of words where after each letter a there is a letter b; in case $\Sigma = \{a, b\}$ these are the words ending in b. A language $L \subseteq \Sigma^*$ is *FO-definable* if there is a first-order sentence φ such that $L = L(\varphi)$.

Monadic second-order logic is an extension of first-order logic with quantification over sets of elements. Let $Var_2 = \{X, Y, \dots\}$ be the set of *second-order variables*. The syntax of monadic second order logic extends first-order

logic with atomic formulas $X(y)$ and quantification $\exists Y$ over second order variables. To interpret such formulas we need now valuations of the form:

$$V : (\mathit{Var}_1 \to \mathrm{dom}(w)) \times (\mathit{Var}_2 \to \mathcal{P}(\mathrm{dom}(w)))$$

assigning to each first-order variable an element of the domain and to each second-order variable a set of elements in the domain. The term "monadic refers" to the fact that second order variables range over sets of elements and not over relations of higher arity.

As before we write $L(\varphi)$ for a set of words defined by a MSOL sentence φ. For example the sentence: $\exists X.\ \big(\forall y.\ X(y) \Leftrightarrow \neg X(y+1)\big) \wedge \forall z.\ P_b(z) \Rightarrow X(z)$ expresses the fact that b appears only on odd or only on even positions in the word. Here we use $X(y+1)$ as a shorthand for saying that the position $y+1$ belongs to X. Relation $+1(y, z)$ can be defined by a first-order formula:

$$y < z \wedge \forall u.\ u \leq y \vee z \leq u$$

When proving something by induction on the structure of MSOL formulas it will be convenient to have a small set of connectives and atomic formulas. Consider the set of formulas given by the grammar

$$\varphi ::= X \subseteq Y \mid X \subseteq P_a \mid \mathrm{Succ}(X, Y) \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \exists X.\varphi \qquad (1)$$

In these formulas there are no first-order variables and there are two new binary predicates $\subseteq$ and Succ. As for the meaning of such formulas in a structure $\mathcal{M}_w$ and valuation $V : \mathit{Var}_2 \to \mathcal{P}(\mathrm{dom}(w))$ we have:

- $\mathcal{M}_w, V \vDash X \subseteq Y$ if $V(X) \subseteq V(Y)$,
- $\mathcal{M}_w, V \vDash X \subseteq P_a$ if $V(X) \subseteq P_a^w$,
- $\mathcal{M}_w, V \vDash \mathrm{Succ}(X, Y)$ if $V(X)$ and $V(Y)$ are singletons p_1 and p_2, respectively, and $p_1 + 1 = p_2$.

It should be clear that both $\subseteq$ and Succ are definable in MSOL. The converse is also not difficult:

Lemma 1 Every MSOL formula without free first-order variables is equivalent to a formula generated by the grammar (1).

Proof
First, we define singleton sets. Then, we simulate first-order variables and first-order quantification using singletons. □

2.2 Automata

A finite automaton is a tuple:

$$\mathcal{A} = \langle Q, \Sigma, q^0 \in Q, \delta \subseteq Q \times \Sigma \times Q, F \subseteq Q \rangle$$

where: Q is a finite set of states, Σ is the input alphabet, q^0 is the initial state, δ is the transition relation, and F is the set of final states.

A *run* of the automaton on a word $w = a_0 \dots a_n$ is a sequence $q_0, \dots, q_{n+1}$ such that: $q_0 = q^0$ is the initial state, and $(q_i, a_i, q_{i+1}) \in \delta$ for all $i = 0, \dots, n$. A run is *successful* if $q_{n+1} \in F$. A word w is *accepted* by $\mathcal{A}$ when there is a successful run of $\mathcal{A}$ on w. The *language recognized by* $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words accepted by $\mathcal{A}$.

Definition 2 A language $L \subseteq \Sigma^*$ is *regular* iff it is the language recognized by some automaton.

It is well known that the class of regular languages is closed under:

- *sum*: if L_1 and L_2 are regular then $L_1 \cup L_2$ is regular
- *intersection*: if L_1 and L_2 are regular then $L_1 \cap L_2$ is regular,
- *complement*: if L is a regular language over Σ then $\Sigma^* \setminus L$ is regular,
- *projection*: if L is a regular language over $\Sigma = \{0,1\} \times \Sigma'$ then $\pi_2(L) = \{\pi_2(w) \in \Sigma' : w \in L\}$ is regular; where we write $\pi_2(w)$ for a word $a_0 \dots a_n \in \Sigma'$ whenever $w = (b_0, a_0)(b_1, a_1) \dots (b_n, a_n) \in \Sigma^*$.

The following theorem gives the connection between the languages accepted by automata and those defined in MSOL.

Theorem 3 (Büchi, Elgot)
A language of finite words is definable by a MSOL sentence iff it is the language recognized by some finite automaton. The translations in both directions are effective.

Proof
Let $\mathcal{A} = \langle Q, \Sigma, q^0, \delta, F \rangle$ be a finite automaton recognizing L. We need to write a formula $\varphi_{\mathcal{A}}$ which holds in a model $\mathcal{M}_w$ iff $w \in L$. Assume that the set of states is $Q = \{q_0, \dots, q_n\}$, and $q_0 = q^0$ is the initial state.

The formula $\varphi_{\mathcal{A}}$ says that there exist sets $S_0, \dots, S_{|Q|}$ such that:

1. the sets form a partition of the domain;

2. the first element of the domain belongs to S_0, and the last to some S_k such that $q_k \in F$;

3. for every element m different from the last, if $m \in S_i$ and $m+1 \in S_j$, for some i and j, then $(q_i, a, q_j) \in \delta$; where a is the unique letter such that $P_a^w(m)$ holds.

It should be clear that all these requirements can be formulated in MSOL and that the resulting formula expresses the existence of an accepting run of $\mathcal{A}$. This shows the right to left implication of the theorem.

For the implication from the left to the right we are going to construct an automaton for every MSOL formula. By Lemma 1 we can do this by induction on the reduced syntax of MSOL given by (1).

A small complication here is that we are going to translate formulas with free second-order variables. Our inductive translation will be simpler if we fix a set of variables $\{X_1, \ldots, X_n\}$ and provide the translation for formulas using only these variables. This is not a loss of generality as the set is arbitrary.

A formula with free variables in $\{X_1, \ldots, X_n\}$ defines the set of pairs $(\mathcal{M}, V)$ consisting of a model and a valuation in which the formula is satisfied. Such a pair can be coded as a word over the alphabet $\Sigma \times \mathcal{P}(\{1, \ldots, n\})$ where a position m is labelled with (a, S) iff a is the label of m in $\mathcal{M}$ and $S = \{i \in \{1, \ldots, n\} : m \in V(X_i)\}$.

By induction on the syntax of the formula φ we construct an equivalent automaton $\mathcal{A}_\varphi$, i.e., such that $\mathcal{A}_\varphi$ accepts the word representations of exactly those pairs $(\mathcal{M}, V)$ for which $\mathcal{M}, V \vDash \varphi$ holds.

The automaton for an atomic formula of the form $X_i \subseteq X_j$ is very simple. It checks that for all the letters (a, S) appearing in the word we have that whenever $i \in S$ then $j \in S$. The constructions of automata for $X_i \subseteq P_a$ and $\mathrm{Succ}(X_i, X_j)$ are also straightforward.

Consider the induction step. If $\varphi = \varphi_1 \vee \varphi_2$ then $L(\mathcal{A}_\varphi) = L(\mathcal{A}_{\varphi_1}) \cup L(\mathcal{A}_{\varphi_2})$. Similarly negation of a formula corresponds to complementation of the language, and existential quantification corresponds to projection. Hence the inductive step follows from well known constructions on finite automata. $\square$

Let us present a small application of this theorem. An easy pumping argument shows that the language $\{a^n b^n : n \in \mathbb{N}\}$ is not recognizable by any finite automaton, hence not definable in MSOL. We will use this fact to show that existence of Hamiltonian cycle is not expressible in MSOL over graphs. This logic is exactly the same as MSOL we have described above but with $\leq$ relation replaced by E relation interpreted as the edge relation of the graph.

A balanced bipartite graph is a graph whose set of vertices can be divided into two sets of the same size such that there are no edges between vertices in the same set. First, we show that there is no MSOL formula defining balanced bipartite graphs. Suppose to the contrary that ψ defines such graphs. We show that then the language $\{a^n b^n : n \in \mathbb{N}\}$ would be definable in MSOL which is impossible.

A word $a^m b^n$ defines a bipartite graph $K_{m,n}$ with vertices $\{(a,i) : i = 1,\ldots,m\} \cup \{(b,j) : j = 1,\ldots,n\}$ and edges connecting each a vertex with each b vertex. By our assumption we have $K_{m,n} \models \psi$ iff $n = m$. Replace each occurrence of $E(x,y)$ in ψ by the formula $(P_a(x) \Leftrightarrow P_b(y))$. Call the resulting formula $\widehat{\psi}$. An easy induction argument shows that $K_{n,m} \models \psi$ iff $a^n b^m \models \widehat{\psi}$. Hence $\widehat{\psi} \wedge \forall x, y.\ P_a(x) \wedge P_b(y) \Rightarrow x \leq y$ defines $\{a^n b^n : n \in \mathbb{N}\}$ which is impossible.

So, there is no MSOL formula ψ over graphs such that $K_{m,n} \models \psi$ iff $m = n$. In $K_{m,n}$ there is a Hamiltonian cycle iff $m = n$. The graphs of the form $K_{m,n}$ are definable in MSOL, hence there cannot be a MSOL formula defining Hamiltonicity.

2.3 Complexity

Let us shortly summarize the complexity results for MSOL and automata on finite words.

The *emptiness problem* is to decide whether a given automaton accepts some word. It is easy to see that the problem is equivalent to the reachability problem in finite graphs, hence it is NLogspace-complete. Indeed, given an automaton we can construct a graph with states of the automaton as nodes and an edge whenever there is a transition on some letter between the states. The automaton accepts some word iff there is a path in the graph from the initial state to a final state.

The *universality problem* is to decide whether $L(\mathcal{A}) = \Sigma^*$ for a given automaton $\mathcal{A}$. This problem is Pspace-complete [46]. A Pspace algorithm for the problem is to determinize the automaton and look for a word that is not accepted. The deterministic automaton may be of exponential size but we never keep the whole of it in memory; we just calculate its states on demand. For Pspace hardness one shows that the language of words that are not computations of a given $O(n)$ space bounded Turing machine can be recognized by a small nondeterministic automaton.

The *satisfiability problem* for MSOL over finite words is to decide whether for a given formula φ there is a word w such that φ holds in $\mathcal{M}_w$. By Theorem 3 the problem is decidable. The problem is nonelementary even for

first-order logic [35, 49].

Maybe it is worth to clarify the use of the term nonelementary here. Elementary functions were introduced by Grzegorczyk [24]. These are functions obtained from some basic functions by operations of limited summation and limited multiplication. Consider the function $Tower(n, k)$ defined by:

$$Tower(n, 0) = n \qquad Tower(n, k + 1) = 2^{Tower(n,k)}$$

Grzegorczyk has shown that every elementary function in one argument is bounded by $\lambda n.\, Tower(n, c)$ for some fixed c. Hence, the term nonelementary refers to a function that grows faster than any such function. In particular for the case of FOL over finite words it is known that the complexity of the satisfiability problem is of order $Tower(n, cn)$ for some constant c [15].

3 Infinite words

An *infinite word* (or ω-*word*) over an alphabet Σ is an infinite sequence $w = a_0 a_1 \ldots$, or equivalently a function from $\mathbb{N}$ to Σ. We use Σ^ω for the set of infinite words over Σ. An infinite word w can be presented as a relational structure:

$$\mathcal{M}_w = \langle \mathbb{N}, \leq, P_a^w \rangle$$

where $\leq$ is the standard order on $\mathbb{N}$ and $P_a^w(i)$ holds iff $w(i) = a$. A more precise term for an infinite word is ω-word. Of course one can consider words for bigger ordinals than ω, but we will not do it here. Hence, we will use the two terms interchangeably.

The notions of first-order and second-order definability extend smoothly from finite to infinite words. A sentence φ now defines a set $\{w \in \Sigma^\omega : \mathcal{M}_w \models \varphi\}$ of infinite words.

The extension of automata recognizability to infinite words requires more work. An ω-automaton has the form:

$$\mathcal{A} = \langle Q, \Sigma, q^0 \in Q, \delta \subseteq Q \times \Sigma \times Q, Acc \subseteq Q^\omega \rangle$$

where Q is the finite set of states, Σ is the alphabet, q^0 is the initial state, δ is the transition relation, and Acc defines which infinite sequences are accepting. Of course, if we want our automata to be finite, we need some finitary ways to describe the set Acc. These are discussed below.

A run of $\mathcal{A}$ on a word w is a sequence $r : \mathbb{N} \to Q$ such that $r(0) = q^0$, and $(r(i), w(i), r(i+1)) \in \delta$ for all $i \in \mathbb{N}$. A word w is *accepted* by $\mathcal{A}$ iff there is a run r of $\mathcal{A}$ on w such that $r \in Acc$. The *language recognized by* $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$.

We are now going to define several standard ways of describing the set Acc. Each of these ways leads to a different notion of automaton. All of these ways refer to the states appearing infinitely often in the run. Hence we introduce the notation:

$$\mathrm{In}(r) = \{q \in Q : r(i) = q \text{ for infinitely many } i\}$$

The most frequently used acceptance conditions are the following requirements on the set $\mathrm{In}(r)$:

Büchi condition is specified by a set $F \subseteq Q$. We have
$Acc = \{r : \mathrm{In}(r) \cap F \neq \emptyset\}$.

Muller condition is specified by a set $\mathcal{F} \subseteq \mathcal{P}(Q)$. We have
$Acc = \{r : \mathrm{In}(r) \in \mathcal{F}\}$.

Rabin condition is specified by a set $\{(R_1, G_1), \ldots, (R_k, G_k)\}$, where $R_i, G_i \subseteq Q$. We have
$Acc = \{r : \exists i.\ \mathrm{In}(r) \cap R_i = \emptyset \text{ and } \mathrm{In}(r) \cap G_i \neq \emptyset\}$

Streett condition is also specified by a set $\{(R_1, G_1), \ldots, (R_k, G_k)\}$, where $R_i, G_i \subseteq Q$. We have
$Acc = \{r : \forall i.\ \mathrm{In}(r) \cap R_i = \emptyset \text{ or } \mathrm{In}(r) \cap G_i \neq \emptyset\}$

Mostowski condition is specified by a function $\Omega : Q \to \mathbb{N}$. We have
$Acc = \{r : \min(\Omega(\mathrm{In}(r))) \text{ is even}\}$

Rabin condition is sometimes called "pairs condition"; Streett condition is called "complement pairs condition"; Mostowski condition is called "parity condition".

Automata are named after acceptance conditions so we have Büchi automata, Muller automata, etc.

Example: Suppose that $Q = \{q_1, q_2, q_3\}$. We will show how to express with different types of conditions the fact that q_1 appears only finitely often and q_2 appears infinitely often. This property cannot be expressed with Büchi conditions. The property is expressed with the Muller condition $\{\{q_2\}, \{q_2, q_3\}\}$. The Rabin condition for the property is $\{(\{q_1\}, \{q_2\})\}$. The equivalent Streett condition is $\{(\{q_1\}, \emptyset), (\emptyset, \{q_2\})\}$. Finally, the Mostowski condition for the property is given by the function $\Omega(q_i) = i$ for $i = 1, 2, 3$. □

Fact 4 For every Büchi condition there is an equivalent Mostowski condition. Every Mostowski condition has equivalent Rabin and Streett conditions. Every Rabin or Streett condition has an equivalent Muller condition.

Proof
A Büchi condition $F \subseteq Q$ is equivalent to a Mostowski condition $\Omega : Q \to \{0, 1\}$, where $\Omega(q) = 0$ iff $q \in F$. A Mostowski condition $\Omega : Q \to \{0, \ldots, k\}$ is equivalent to a Rabin condition $\{(R_i, G_i) : i = 0, \ldots, k/2\}$ where $R_i = \{q : \Omega(q) < 2i\}$ and $G_i = \{q : \Omega(q) = 2i\}$. The translation to Streett condition is similar. It is obvious that any condition can be translated to a Muller condition. □

Fact 5 Mostowski and Muller conditions are closed under negation. The negation of a Rabin condition is a Streett condition and vice versa.

Proof
The complement of a Muller condition $\mathcal{F} \subseteq \mathcal{P}(Q)$ is $\overline{\mathcal{F}} = \mathcal{P}(Q) \setminus \mathcal{F}$. The

complement of a Mostowski condition $\Omega : Q \to \mathbb{N}$ is given by $\overline{\Omega}(q) = \Omega(q) + 1$. The complement of a Rabin condition $\{(R_1, G_1), \ldots, (R_n, G_n)\}$ is $\{(G_1, R_1), \ldots, (G_n, R_n)\}$ interpreted as a Streett condition. □

Fact 6 Nondeterministic Büchi-, Muller-, Rabin-, Streett-, and Mostowski-automata all recognize the same class of ω-languages.

Proof
Every acceptance condition is a special form of Muller acceptance condition. Hence it is enough to show how to translate Muller automata to Büchi automata. Roughly, a Büchi automaton nondeterministically picks a set S from the Muller condition and checks that S is precisely the set of states appearing infinitely often. □

This fact allows us to formulate the definition:

Definition 7 A language $L \in \Sigma^\omega$ is *regular* if it is the language recognized by some nondeterministic Büchi automaton.

Unlike the case of finite words it is not true that every automaton can be determinized. Büchi automata cannot be determinized as the following example shows. Let $L_a \in \{a, b\}^\omega$ be the set of words containing only finitely many occurrences of a. It is easy to construct a nondeterministic automaton for the language. This automaton just guesses a position and checks that after this position there is no occurrence of a.

Fact 8 There is no deterministic Büchi automaton recognizing L_a.

Proof
Suppose conversely that $\mathcal{A} = \langle Q, \Sigma, q^0, \delta : Q \times \Sigma \to Q, F \rangle$ is a Büchi automaton for the language. Take the word b^ω. There is an accepting run of $\mathcal{A}$ on this word. Let i_1 be the position on which a state from F appears in the run. Consider now the word $b^{i_1}ab^\omega$. It is also accepted, and as the automaton is deterministic, the run is the same up to position i_1. Take a position $i_2 > i_1$ on which a state from F appears. Repeat this process $n = |Q| + 1$ times. We get a word $b^{i_1}ab^{i_2}a \ldots b^{i_n}ab^\omega$ which is accepted by $\mathcal{A}$ and such that the run of $\mathcal{A}$ on this word has accepting states at positions i_j for $j = 1, \ldots, n + 1$. By the choice of n there are two positions, say i_k and i_l where the same state appears. We have that there is an accepting run of $\mathcal{A}$ on the word $b^{i_1}ab^{i_2}a \ldots b^{i_k - 1}(bab^{i_{k+1}}a \ldots b^{i_l - 1})^\omega$ □

There is a deterministic automaton with Mostowski conditions for L_a. This is an automaton that signals 1 when it reads a and 2 when it reads b.

By the definition of the Mostowski condition this automaton accepts iff it signals 1 only finitely often.

This is not a coincident that there is a deterministic Mostowski automaton for L_a. The following important fact shows that deterministic automata with all but Büchi conditions have the same expressive power. In the next section we will see that they are equivalent to nondeterministic automata.

Theorem 9 (Mostowski [36])
For every deterministic Muller automaton there is an equivalent deterministic Mostowski automaton.

Proof
Take a Muller automaton $\mathcal{A} = \langle Q, \Sigma, 1, \delta, \mathcal{F} \rangle$ and assume that $Q = \{1, \ldots, n\}$. The states of the Mostowski automaton $\mathcal{A}'$ will be permutations of Q with a distinguished position. Such a permutation is called *last appearance record* (LAR) and the distinguished position is called *hit position*. So the set of LARs is:

$$Q' = \mathrm{Perm}(\{1, \ldots, n\}) \times \{1, \ldots, n\}$$

The idea is that a LAR keeps the order between the last occurrences of states up to the present point. That is, if a state q appears before q' in the permutation then the last occurrence of q up to the present position is before the last occurrence of q'. In particular the present state is on the last place of the permutation. The hit position shows what was the position of the present state in the previous permutation.

More formally, if the original automaton moves from a state l to a state l' then the simulating automaton $\mathcal{A}'$ is in a state $(i_1, \ldots, i_n, h)$, with $i_n = l$, and changes it to $(i_1, \ldots, i'_k, \ldots, i'_{n-1}, l', k)$ where k is the position of l' in the sequence (i.e. $i_k = l$) and $i'_j = i'_{j+1}$ for $k \leq j < n$.

Let $r = q_0, q_1, \ldots$ be a run of the original automaton $\mathcal{A}$. Suppose that $F = \mathrm{In}(r)$ is the set of states appearing infinitely often in the run. Let us analyze the run of the automaton $\mathcal{A}'$ from the state $(n, \ldots, 1, 1)$. We will say that a state is an F-state if it is of the form $(i_1, \ldots, i_k, \ldots, i_n, h)$ with $\{i_k, \ldots, i_n\} = F$ and $h \geq k$.

Assume first that there is a position m such that only states from F appear after m and the state of $\mathcal{A}'$ at m is an F-state. It is easy to see from the definition of $\mathcal{A}'$ that after m all the states will be F-states. Moreover, the hit position will be k at some position after m. This is because the state i_k is going to appear after m. So, this argument really shows that the hit position will be k infinitely often after m.

Now, let us see why we are bound to reach such a position m as assumed in the previous paragraph. Let m_1 be a position after which no state outside

F appears. Let $(i_1, \ldots, i_k, \ldots, i_n, h)$ be a state of $\mathcal{A}'$ at this position. Take a position m_2 such that between m_1 and m_2 all the states from F appeared at least once. We claim that $m_2 + 1$ is the required m. By the definition of the transition relation of $\mathcal{A}'$ in the permutation at position m_2 all the states from F occur after the states not in F. As the state q_{m_2} is from F the state of $\mathcal{A}'$ at the position m_2 will have the hit position $\geq k$.

So we have shown that if $q_0, q_1, \ldots$ is a run of $\mathcal{A}$ and F is the set of states appearing infinitely often on it then in the corresponding run of $\mathcal{A}'$ almost all states are F-states and the hit position is equal $k = n - |F| + 1$ infinitely often.

Now, we define the Mostowski acceptance condition on states of $\mathcal{A}'$. We put

$$\Omega(i_1, \ldots, i_n, h)) = \begin{cases} 2h & \text{if } \{i_h, \ldots, i_n\} \in \mathcal{F} \\ 2h+1 & \text{if } \{i_h, \ldots, i_n\} \notin \mathcal{F} \end{cases}$$

We claim that $\mathcal{A}'$ is equivalent to $\mathcal{A}$. Let F be a set of states appearing infinitely often on the run of $\mathcal{A}$. By the above considerations almost all states on the run of $\mathcal{A}'$ are F-states and the hit position is equal $k = n - |F| + 1$ infinitely often. Hence priority appearing infinitely often on the run of $\mathcal{A}'$ is either $2(n - |F| + 1)$ or $2(n - |F| + 1) + 1$. It is even iff $F \in \mathcal{F}$. □

It is natural to ask what is the complexity of translating form one form of automaton to the next. From the above we can deduce:

Corollary 10 For every deterministic Muller, Streett or Rabin automaton with n states there is a deterministic Mostowski automaton with $\mathcal{O}(2^{n \log(n)})$ states.

A summary of the results on this subject is presented in [32]. In particular the bound in the corollary is essentially optimal.

3.1 Closure properties of ω-automata

It is easy to see that regular ω-languages are closed under sum. The construction is exactly the same as in the case of automata on finite words. This also true for the closure under projection. The closure under intersection is not that immediate. The construction from the case of finite words needs to be modified because now there is no last letter on which two runs can be synchronized.

Differences between finite and infinite words show up acutely in the case of closure under complement. In the case of automata on finite words a simple powerset construction is enough. In the case of ω-automata a very

refined version of a powerset construction is required. Below, instead of complementation we consider the stronger property of determinization.

Theorem 11 (McNaughton)
For every Büchi automaton there is an equivalent deterministic Rabin automaton.

An immediate corollary of this result is the equivalence between MSOL and Büchi automata. Recall that by the results of the previous section all but deterministic Büchi automata are equivalent to (nondeterministic) Büchi automata.

Corollary 12 (Büchi) A language of ω-words is MSOL definable iff it is the language recognized by some Büchi automaton. The translations in both directions are effective.

Proof
The construction of a formula for a given automaton is almost the same as in the case of finite words. The proof in the other direction is also very similar. We build an automaton by induction on the syntax of a given formula. It is easy to construct automata for atomic formulas. As noted above, Büchi automata are closed under sum and projection. The closure under complementation follows from Theorem 11 and Fact 6 saying that every Rabin automaton can be converted into a nondeterministic Büchi automaton. $\square$

Actually, in 1962 when Büchi proved the above result he did not use determinization construction. The determinization construction was given in 1966 by McNaughton. At that time there was no notion of Rabin automaton. He has shown the determinization theorem for Muller automata. We follow here the construction given by Safra in 1988 [45]. This construction gives better complexity bounds than the original one. Later we will describe how to modify the construction to get an automaton with Mostowski conditions. The proof of the determinization theorem will take the rest of this subsection.

Let us start by examining why the standard subset construction does not work. Take a Büchi automaton:

$$\mathcal{A} = \langle Q, \Sigma, q_0, \delta, F \rangle$$

The states of the powerset automaton $\mathcal{A}^P$ are nonempty subsets of Q. We call these states *macro-states*. The transition function δ^P of this automaton is defined by $\delta^P(S, a) = \{q' : \exists q \in S.\ (q, a, q') \in \delta\}$. So $\mathcal{A}^P$ is a deterministic automaton and there is a run of $\mathcal{A}^P$ on w iff there is a run of $\mathcal{A}$ on w. The problem comes when we want to define an acceptance condition. A first

attempt can be to define a Büchi acceptance condition F^P by taking all the macro-states S containing a state from F. The resulting automaton may accept too much. The problem is presented in part (a) of Figure 1. The big bubbles represent macro-states. The arrows show the transition relation of the original automaton. We assume that $F = \{q_f\}$. In case (a) there is no accepting run of the original automaton because there is no way to prolong a run from an accepting state.

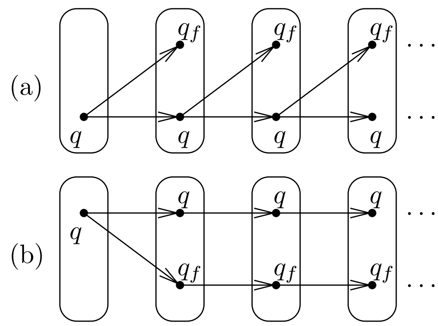


Figure 1: Problems with the powerset construction.

An alternative construction can extend powerset construction with recording some information about past of each state. As this will be a binary information we will use the metaphor of painting states. In the initial macro-state no state is painted. A state q' of a macro-state is painted green if $q' \in F$ or it is a successor of some green state, i.e., q' is obtained from q on the previous position and q was green in the previous macro-state. If the powerset automaton reaches a state with all the components painted green then it signals acceptance and removes the paint from all the components. Then the whole process repeats. An easy application of König's lemma shows that if such powerset automaton signals acceptance infinitely often then there is an accepting run of the original automaton. Unfortunately, as example (b) on Figure 1 shows, there may be an accepting run of the original automaton but the powerset automaton will not be a able to signal acceptance. The problem here is that apart from an accepting run we have a run that does not go through q_f at all. Hence at each macro-state there is a not painted state. If we were allowed to guess, than we would guess that we should restrict the above construction only to q_f and its successors. This would remove the upper part of the run from the considerations and the powerset automaton would accept. Because we are to construct a deterministic automaton we are not allowed to guess. What we will do is to consider all the possible guesses at the same time. We will need a clever way of keeping all these guesses

together so that the sates do not get to big. This is the role of Safra trees defined below.

An ordered tree $(t, \leq)$ is a finite-tree with a partial order $\leq$ relating two nodes of the tree iff they are siblings, i.e., they have a common father. This ordering defines *to the left* relation on nodes of the tree: $left(u, v)$ holds if there are two siblings $u' \neq v'$ such that $u' \leq v'$ and u is a descendant of u' and v is a descendant of v' (we allow for $u = u'$ or $v = v'$).

A *Safra tree* is an ordered tree labelled with nonempty subsets of Q and colors white or green. The tree must also satisfying some coherence conditions. Formally such a tree is a quadruple

$$\tau = (t, \leq, \lambda : t \to \mathcal{P}(Q), c : t \to \{\text{white}, \text{green}\})$$

The conditions are:

1. the label of the father is a proper superset of the sum of the labels of the sons,

2. the labels of two nodes which are not ancestral are disjoint.

The intuition is that the label of the root represents a macro-state from the powerset automaton. The rest of the tree describes a decomposition of the macro-state.

Before defining the transition function on Safra-trees, consider the lemma pointing out properties of the structure of Safra-trees and giving the bound on their size.

Lemma 13 There are at most $|Q|$ nodes in a Safra tree.

Proof
By condition 2, if a state q belongs to a label of a node then it can belong to the label of at most one son of the node. Hence there is a uniquely determined lowest node containing q. By condition 1, every node is the lowest node for some state. Hence, there cannot be more nodes than states. □

The deterministic transition function $\widehat{\delta}$ transforms a given tree τ on a given input a into a tree obtained by the following sequence of actions:

1. Set the color of all the nodes to white,

2. For every node v, create a new node v' with $\lambda(v') = \lambda(v) \cap F$. Make v' a son of v that is bigger than all the other sons of v.

3. For every node v, define the labeling $\lambda'(v) = \bigcup_{q \in \lambda(v)} \delta(q, a)$. (So we apply the successor function of the automaton.)

4. For every node v, define the labeling $\lambda''(v) = \lambda'(v) \setminus \bigcup_{left(v_1,v)} \lambda'(v_1)$. (So we delete a state from the label if it appears somewhere to the left.)

5. Remove all the nodes with empty λ'' labels.

6. For every node v which λ'' label is equal to the sum of the λ'' labels of its sons, remove the descendants of v and color v green.

The resulting tree $\tau' = \widehat{\delta}(\tau, a)$ has the vertices of τ plus the vertices added in step 2 minus the vertices removed in step 6. The labeling is λ'' and the colouring is defined by the last step.

We will show later that $\mathcal{A}$ accepts a word iff there is the sequence of Safra trees constructed according to $\widehat{\delta}$ and a vertex v which is deleted finitely often and which is coloured green infinitely often.

Before defining a Safra automaton we need to solve a small technical problem. In step 2 we add new vertices. We need to bound the number of vertices that can be added in order to have a finite number of Safra trees. We do this by recycling the vertices that are deleted in step 6. As we have observed before there can be at most $|Q|$ vertices in a Safra tree. Hence, we can take a pool of $2|Q|$ vertices. In 2 we take vertices from the pool and in step 6 we put them back. This way we have:

Lemma 14 There are $2^{\mathcal{O}(|Q|\log(|Q|))}$ Safra-trees.

The Safra automaton is $\widehat{\mathcal{A}} = \langle \widehat{Q}, \Sigma, \widehat{q}^0, \widehat{\delta}, \widehat{Acc} \rangle$ where $\widehat{Q}$ is the set of Safra trees over Q; $\widehat{q}^0$ is the tree consisting only of root labelled with $\{q^0\}$; and $\widehat{Acc}$ contains all the sequences such that there is a vertex which is removed only finitely often and lights green infinitely often on the sequence.

We now show that $L(\widehat{\mathcal{A}}) \subseteq L(\mathcal{A})$. Let $\tau_0, \tau_1, \ldots$ be an accepting run of $\widehat{\mathcal{A}}$ on some word w. Let v be a vertex that lights green infinitely often in the run and is not removed after some position m. Take a position i where v is green and a position $j > i$ such that v is not green between i and j. An easy induction shows that if $q' \in \lambda_j(v')$ for some v' a son of v in τ_j then there is $q \in \lambda_i(v)$ and a run of $\mathcal{A}$ on $w_i w_{i+1} \ldots w_j$ from q to q' going through some state in F. Call a run green if it goes through F. Let $i_1 < i_2 < \cdots$ be the sequence of positions after m where v lights green. From the above we have that for every $j = 1, 2, \ldots$ and every $q' \in \lambda_{i_{j+1}}(v)$ there is $q \in \lambda_{i_j}(v)$ and a green run of $\mathcal{A}$ on $w_{i_j} w_{i+1} \ldots w_{i_{j+1}}$ from q to q'. By Königs lemma we can find and infinite sequence $q_1, q_2, \ldots$ such that for every $j = 1, 2, \ldots$ there is a green run of $\mathcal{A}$ on $w_{i_j} w_{i+1} \ldots w_{i_{j+1}}$ form q_j to q_{j+1}. This together with the observation that there is a run of $\mathcal{A}$ from q^0 to q_1 on $w_0 \ldots w_{i_1}$ gives us an accepting run of $\mathcal{A}$ on w.

Next, we show that $L(\mathcal{A}) \subseteq L(\widehat{\mathcal{A}})$. Let q_0, q_1 be an accepting run of $\mathcal{A}$ on w. Consider the unique run $\tau_0, \tau_1, \ldots$ of $\widehat{\mathcal{A}}$ on w. By the definition of the transition function we have that $q_i \in \lambda_i(v_0)$ where v_0 is the root of all the Safra trees and λ_i is the labeling from the tree τ_i. If the root lights green infinitely often then the run of $\widehat{\mathcal{A}}$ is accepting and we are done. If not then let m_0 be the position where $q_{m_0} \in F$ and after which the root does not light green, Hence q_{m_0} appears in some son of v_0. An easy induction shows that for every $i > m_0$ the state q_i will appear in some son of v_0. It may move from one son to some other but this other son must be smaller in the tree ordering. As there is a bounded number of smaller sons of the root, there must be a son v_1 where the run of $\mathcal{A}$ stays forever, i.e., $q_i \in \lambda_i(v_1)$ for all positions bigger than some m_1. If v_1 lights green infinitely often then we are done as v_1 is not removed after m_1. If not then we repeat the reasoning. This way we obtain a path $v_0, v_1, \ldots$ This path cannot be infinite because Safra trees have bounded size. Hence, there must be v_j such that for all positions i bigger than some m_j we have $q_i \in \lambda_i(v_j)$ and v_j lights green infinitely often.

Finally, it remains to show that $\widehat{Acc}$ is a Rabin acceptance condition. Recall that Safra trees were constructed over the fixed set $\{v_1, \ldots, v_{2|Q|}\}$ of vertices. For each $i = 1, \ldots, 2|Q|$ we take a pair (R_i, G_i) where R_i are all the Safra trees without v_i and G_i are all the Safra trees where v_i is coloured green. Then $\widehat{Acc}$ is expressed by the Rabin condition $\{(R_1, G_1), \ldots, (R_{2|Q|}, G_{2|Q|})\}$. A more efficient strategy of vertex recycling would allow us to manage with a pool of $|Q|$ instead of $2|Q|$ vertices. This would reduce the number of pairs in the Rabin condition to $|Q|$.

Corollary 15 For every Büchi automaton with n states there is an equivalent deterministic Rabin automaton with $2^{\mathcal{O}(n \log(n))}$ states and n pairs in the acceptance condition.

This construction can be combined with LAR construction from Theorem 9 to obtain a deterministic Mostowski automaton. A direct application of the theorem would give a Mostowski automaton of doubly exponential size and with big values of function Ω. A closer look shows that it is enough to keep LARs of vertices and not of the whole Safra trees. As there are n vertices, there are $2^{\mathcal{O}(n \log(n))}$ LARs. So the resulting Mostowski automaton is of not much bigger size than the Rabin automaton.

Corollary 16 Every Büchi automaton with n states is equivalent to a deterministic Mostowski automaton with $2^{\mathcal{O}(n \log(n))}$ states and the range of the acceptance condition contained in $\{0, \ldots, 2n\}$.

3.2 Complexity

Let us shortly discuss the complexity of some problems for MSOL and automata on ω-words.

Checking emptiness of an ω-automaton is NLOGSPACE-complete. The lower bound follows from the case of automata on finite words. The upper bound is a modification of the reachability algorithm.

Checking universality is PSPACE-complete for automata on finite words. It is also PSPACE-complete for ω-automata of any the discussed kinds. The argument is essentially the same as for finite words. One constructs a deterministic automaton equivalent to the given one. The states of the automaton are calculated on demand.

As there is an effective translation from MSOL formulas to ω-automata, it follows that the satisfiability problem for MSOL over ω-words is decidable. The complexity of the problem is nonelementary. The lower bound follows from the satisfiability problem of FOL over finite words.

4 Infinite trees

In this section we extend the concept of automaton even further. We consider automata running on full infinite binary trees. We will show that these automata enjoy the same closure properties as word automata. This will allow us to get the equivalence with MSOL over trees.

The *full binary tree* is the set $\{0,1\}^*$ of finite words over a two element alphabet. The root of the tree is the empty word ε. A node $w \in \{0,1\}^*$ has the left son $w0$ and the right son $w1$. A Σ-*labelled full binary tree* is a function $t : \{0,1\} \to \Sigma$. We use Trees(Σ) for the set of all Σ-labelled binary trees.

Similarly as for words, Σ-labelled trees can be represented as relational structures. A tree $t : \{0,1\} \to \Sigma$ is represented by:

$$\mathcal{M}_t = \langle \{0,1\}^*, \leq, s_0, s_1, (P_a)_{a\in\Sigma} \rangle$$

where $u \leq v$ holds if u is a prefix of v and s_0 and s_1 are the binary left and right son relations respectively. As before the predicates $(P_a)_{a\in\Sigma}$ code the labeling function t.

With such representations of trees we can say that a set of trees L is *monadic second-order (first-order) definable* iff $L = \{t : \mathcal{M}_t \vDash \varphi\}$ for some monadic second-order (respectively first-order) formula.

The idea of extending automata to trees is simple. Before, being in some position in a word the automaton had to decide which state to assume in the successor position. Now, as every node in a tree has two successors, the automaton splits and sends one copy of itself to each of the successors. Formally an automaton on trees is:

$$\mathcal{A} = \langle Q, \Sigma, q^0, \delta \subseteq Q \times \Sigma \times Q \times Q, Acc \rangle$$

where all the components but the transition relation δ are the same as for automata on words. Depending on whether Acc is Büchi-, Muller-, etc. condition we call $\mathcal{A}$ Büchi-, Muller-, etc. automaton.

A run of $\mathcal{A}$ on t is a function $r : \{0,1\}^* \to Q$ labeling nodes of the tree with states so that the root is labelled by the initial state, i.e., $r(\varepsilon) = q^0$; and for every node w and its sons $w0$ and $w1$ we have that

$$(r(w), t(w), r(w0), r(w1)) \in \delta.$$

A run r is *accepting* iff for every path P of the tree the sequence of states appearing on the path belongs to Acc. To put it more formally a path is a sequence of nodes $v_0, v_1, \ldots$ such that $v_0 = \varepsilon$ and v_{i+1} is a son of v_i, for every

i. A run r is accepting iff for every such path the sequence $r(v_0), r(v_1), \ldots$ belongs to Acc. A tree is *accepted by* $\mathcal{A}$ iff there is an accepting run of $\mathcal{A}$ on it. The *language recognized by* $\mathcal{A}$ is the set of trees accepted by $\mathcal{A}$.

Example: We show a Büchi automaton for the language $L_a^\infty \in \text{Trees}(\{a, b\})$ of trees having a path with infinitely many a's. The states of the automaton are $q_a, q_b, \top$. The transition relation is given by:

$$\begin{aligned}\delta(q_*, a) =& \{(q_a, \top), (\top, q_a)\}\\ \delta(q_*, b) =& \{(q_b, \top), (\top, q_b)\}\\ \delta(\top, *) =& \{(\top, \top)\}\end{aligned}$$

where in the above $*$ stands for either a or b. The Büchi acceptance condition is $F = \{q_a, \top\}$. Clearly the automaton accepts every tree from the state $\top$. It is easy to see that any run from q_a consists of a single path labelled with states q_a, q_b, and the rest of the tree labelled with $\top$. There are infinitely many states q_a on this path iff there are infinitely many vertices labelled by a. Hence the automaton accepts a tree iff it can find a path with infinitely many a's on it. $\square$

An important difference with the case of words is that Büchi conditions are weaker than the other types of conditions. The following fact was shown by Rabin [43] (see also [52])

Fact 17 The class of languages accepted by Büchi tree automata is not closed under complement. The complement of the language L_a^∞ from the example above is not recognizable by a nondeterministic Büchi automaton.

This is closely connected to the fact that deterministic Büchi automata over words are weaker than other types of deterministic word automata. A tree automaton can be seen as a composition of a guessing automaton, that is a tree automaton without any acceptance conditions, and a checking automaton that is a deterministic word automaton checking for every path if on this path the run guessed by the guessing automaton is accepting. This idea is elaborated in the proof of the following fact.

Fact 18 The classes of tree languages accepted by Mostowski, Rabin, Streett and Muller conditions are the same.

Proof
Let $\mathcal{A} = \langle Q, \Sigma, q^0, \delta \subseteq Q \times \Sigma \times Q \times Q, Acc\rangle$ be a tree automaton. Let $\mathcal{B} = \langle Q^b, \Sigma^b, q_b^0, \delta^b : Q^b \times \Sigma^b \to Q^b, Acc^b\rangle$ a deterministic word automaton over the alphabet $\Sigma^b = Q$ recognizing the language Acc. That is $\mathcal{B}$ reads

infinite sequences of states of $\mathcal{A}$ and accepts a sequence iff it satisfies the acceptance condition Acc. Consider the composition of the two automata:

$$\mathcal{A} \otimes \mathcal{B} = \{Q^{\otimes} = Q \times Q_b,\ \Sigma,\ (q^0, q_b^0),\ \delta^{\otimes} : Q^{\otimes} \times \Sigma \times Q^{\otimes} \times Q^{\otimes},\ Acc^{\otimes}\}$$

where

- $((q_l, q_l^b), (q_r, q_r^b)) \in \delta^{\otimes}((q, q^b), a)$ if $(q_l, q_r) \in \delta(q, a)$ and $q_l^b = \delta^b(q, q_l)$, $q_r^b = \delta^b(q, q_r)$;
- Acc^b contains those sequences $(q_1, q_1^b)(q_2, q_2^b)\ldots$ which projection on the second components $q_1^b, q_2^b, \ldots$ is in Acc_b.

It is not difficult to see that $\mathcal{A} \otimes \mathcal{B}$ accepts the same language as $\mathcal{A}$. The difference between the two automata is that $\mathcal{A} \otimes \mathcal{B}$ has the acceptance condition of the same type as $\mathcal{B}$.

By Theorem 9 we can take $\mathcal{B}$ to be Mostowski, Rabin, Streett or Muller automaton. Then $\mathcal{A} \otimes \mathcal{B}$ will be an automaton equivalent to $\mathcal{A}$ and of the same type as $\mathcal{B}$. □

Hence we can use, say, Rabin conditions to define the notion of regularity. Later we will see that this is a robust and interesting notion.

Definition 19 A tree language $L \subseteq \text{Trees}(\Sigma)$ is *regular* iff there is a Rabin automaton recognizing L.

4.1 Closure properties of tree automata

As in the preceding sections our goal is to show that the class of regular languages coincides with those definable in MSOL. For this we need to check the closure properties of regular tree languages. Essentially the same constructions as for word automata show that the class of regular tree languages is closed under sum and projection. As in the case of infinite words, it is the closure under complement that brings the biggest problems.

Theorem 20 (Rabin)
Regular tree languages are closed under complementation. For every Rabin tree automaton $\mathcal{A}$ one can effectively construct a Rabin automaton accepting the complement of $L(\mathcal{A})$.

Once we prove this theorem we get the equivalence between automata and MSOL. The proof of the corollary below is very similar to the case of words.

Corollary 21 A tree language is definable by a MSOL sentence iff it is the language recognized by some Rabin automaton. The translations in both directions are effective.

In the rest of the section we will sketch the proof of the complementation theorem. Observe that it does not say that Rabin automata can be determinized. Indeed the following easy fact shows that there is no hope for determinization.

Fact 22 Let $L_a^{\exists} \subseteq \text{Trees}(\{a, b\})$ be the set of trees having at least one vertex labelled with a. The language $L_a^{\exists}$ is not recognizable by a deterministic tree automaton with any of the considered acceptance conditions.

Although determinization of tree automata is not possible, still the determinization result for word automata is essential in the complementation proof for tree automata. Except for this result we will need an important fact from the theory of infinite games. We will now define games abstractly and then make the connection to tree automata.

A *game* $G = \langle V, V_0, V_1, E \subseteq V \times V, Acc_G \subseteq V^\omega \rangle$ is a bipartite labelled graph with the partition (V_0, V_1) of the set of vertices V. We say that a vertex v' is a *successor* of a vertex v if $E(v, v')$ holds. The set Acc_G is used to determine the winner in a play of the game.

A *play* from some vertex $v_0 \in V_0$ proceeds as follows: first player 0 chooses a successor v_1 of v_0, then player 1 chooses a successor v_2 of v_1, and so on ad infinitum unless one of the players cannot make a move. If a player cannot make a move he looses. The result of an infinite play is an infinite path $v_0, v_1, v_2, \ldots$ This *path is winning* for player 0 if it belongs to Acc_G. The play from vertices of V_1 is defined similarly but this time player 1 starts.

A *strategy* σ for player 0 is a function assigning to every sequence of vertices $\vec{v}$ ending in a vertex v from V_0 a successor vertex $\sigma(\vec{v}) \in V_1$. A strategy is *memoryless* iff $\sigma(\vec{v}) = \sigma(\vec{w})$ whenever $\vec{v}$ and $\vec{w}$ end in the same vertex. A *strategy is winning* iff it guarantees a win for player 0 whenever he follows the strategy. Similarly we define a strategy for player 1.

In our application to tree automata we will be interested only in games with Acc_G given by Mostowski conditions. Such a condition is determined by a function $\Omega : V \to \mathbb{N}$ with the additional requirement that the image of Ω is finite. In the case of finite automata we did not have to make this finiteness assumption because the set of states was finite by definition. Here we do not assume that the set V of vertices is finite.

Definition 23 A game with Mostowski conditions is given by a labelled graph $\langle V, V_0, V_1, E \subseteq V \times V, \Omega : V \to \mathbb{N} \rangle$ such that the image of Ω is a finite

set.

The following is the main theorem about games with this kind of conditions. The idea of a strategy with bounded memory was introduced by Gurevich and Harrington [25]. They have shown a bounded memory theorem for Rabin conditions. The simplification for Mostowski conditions was proved independently by Emerson and Jutla [19] and by Mostowski [37].

Theorem 24 (Memoryless determinacy)
Let G be a game with Mostowski conditions. From every node of G one of the players has a memoryless winning strategy.

We will not present the proof of this theorem here; nice expositions of the proof can be found among others in [53, 56].

Now we can make the connection between tree automata and games. For a given automaton $\mathcal{A}$ and a given tree t we will define an acceptance game $G_{\mathcal{A},t}$. Player 0 will have a strategy in this game iff $t \in L(\mathcal{A})$. This way $t \notin L(\mathcal{A})$ is equivalent to player 1 having a strategy in $G_{\mathcal{A},t}$. By the above theorem, in this situation there is a memoryless strategy for player 1. The existence of such a strategy can be checked by a finite automaton. This will be the automaton accepting the complement of $L(\mathcal{A})$.

Fix an automaton $\mathcal{A}$ and a tree t. We define the game $G_{\mathcal{A},t}$. The idea of the game is quite simple. Player 0 will try to show that $\mathcal{A}$ accepts t. So, in each position of the game which is a current vertex of the tree and the current state of the automaton player 0 will chose a transition of the automaton. Player 1 will try to show that the choices suggested by player 0 are not correct. To this end he will point to direction, left or right, asking player 0 to provide the evidence for the respective subtree. The result of such play is an infinite path. Player 0 is the winner if the the sequence of states on this path satisfies the acceptance condition of $\mathcal{A}$, otherwise player 1 wins. Formally the game $G_{\mathcal{A},t}$ is defined by:

- the set V_0 of vertices for player 0 is $\{0,1\}^* \times Q$,
- the set V_1 of vertices for player 1 is $\{0,1\}^* \times (Q \times Q)$,
- from each vertex $(v, q) \in V_0$, for each transition $(q, a, q_0, q_1) \in \delta$ with $t(v) = a$ we have an edge to $(v, (q_0, q_1))$,
- from each vertex $(v, (q_0, q_1)) \in V_1$ we have edges to $(v0, q_0)$ and $(v1, q_1)$,
- the acceptance condition Acc_G consists of the sequences

$$(v_0, q_0)(v_0, m_0)(v_1, q_1)(v_1, m_1)\ldots$$

such that the sequence $q_0 q_1 \ldots$ is in Acc, i.e., it belongs to the acceptance condition of the automaton. (Here we use letters m_i to denote the elements of $Q \times Q$.)

Directly from the definition of the game we have that there is one to one correspondence between accepting runs of $\mathcal{A}$ on t and winning strategies for player 0 in $G_{\mathcal{A},t}$.

Lemma 25 $t \in L(\mathcal{A})$ iff in $G_{\mathcal{A},t}$ player 0 has a winning strategy from the position (ε, q^0), i.e., the position consisting of the root of the tree and the initial state of $\mathcal{A}$.

Hence, by Theorem 24, $t \notin L(\mathcal{A})$ iff player 1 has a winning strategy in $G_{\mathcal{A},t}$. We will now construct an automaton $\mathcal{B}$ which accepts a tree t iff player 1 has a memoryless strategy in the game $G_{\mathcal{A},t}$. This way we will have that $L(\mathcal{B})$ accepts the complement of $L(\mathcal{A})$.

A memoryless strategy for player 1 is a function $\sigma_1 : V_1 \to V_0$ which for each vertex $(v, (q_0, q_1)) \in V_1$ chooses either $(v0, q_0)$ or $(v1, q_1)$. An important point is that such a function can be coded as a labeling function

$$f : \{0,1\}^* \to Moves_1$$

where $Moves_1$ is the finite set $((Q \times Q) \to \{0,1\})$.

Consider infinite words over the alphabet $\Sigma' = \Sigma \times Moves_1 \times \{0,1\}$. Such a word $\xi = ((a_i, f_i, d_i))_{i \in \mathbb{N}}$ describes a path $\varepsilon, d_0, d_0 d_1, \ldots$ in the tree. Letter a_i is the label of the vertex $d_0 \ldots d_{i-1}$, and f_i is the strategy of player 1 in this vertex. A *play staying* ξ is a sequence $(v_0, q_0)(v_0, m_0)(v_1, q_1)(v_1, m_1)\ldots$ such that

- $v_0 = \varepsilon$ and $q_0 = q^0$,
- $m_i \in \delta(q_i, a_i)$,
- $f_i(m_i) = d_i$ and $q_{i+1} = q^{d_i}$ where $m_i = (q^0, q^1)$.

That is the strategy always suggests the directions along the path determined by ξ.

Let $\mathcal{C}$ be a nondeterministic automaton accepting precisely those words ξ over Σ' which have a play staying in ξ that is winning for player 0. It is quite easy to construct such an automaton from the description above. (It can nondeterministically guess the witnessing play.) By Theorem 16 there is a deterministic Rabin automaton $\overline{\mathcal{C}}$ accepting the complement of this language. That is $\overline{\mathcal{C}}$ accepts those words ξ for which all the plays staying ξ are winning for player 1.

Consider a tree t and a strategy function f. If f is not winning for player 1 then there is a play which is winning for player 0 when player 1 uses the strategy defined by f. This play proceeds along some path of the tree. Hence automaton $\mathcal{C}$ would accept description of such a path. If f is winning then none of the paths is accepted by $\mathcal{C}$. Hence every path is accepted by $\overline{\mathcal{C}}$. So f is a winning strategy for player 1 iff each path of $t \times f$ (i.e. the tree t labelled additionally with the values of f) is accepted by $\overline{\mathcal{C}}$.

The tree automaton $\mathcal{B}$ accepting the complement of $\mathcal{A}$ consists of two automata. The first guesses the value of strategy a function f_1 in each vertex. The second runs $\overline{\mathcal{C}}$ on every path of the tree. Automaton $\mathcal{B}$ accepts iff $\overline{\mathcal{C}}$ accepts on all the paths.

4.2 Complexity

The complexity of decision problems for tree automata is more subtle than for word automata.

Recall that the emptiness problem, is to decide if there is a tree accepted by a given automaton. For Büchi automata the problem is PTIME-complete. For Rabin automata it is NP-complete [18]. As Streett conditions are negations of Rabin conditions, the emptiness problem for these automata is co-NP-complete. The exact complexity of the emptiness problem for Mostowski automata is not known. The problem is in NP and co-NP [17]. Determining whether the problem is in PTIME is one of the main open problems in the area.

The universality problem is EXPTIME-complete even for automata on finite trees [46]. The EXPTIME-completeness result carries over to all kinds of automata discussed in this section.

The satisfiability problem for MSOL on binary trees is nonelementary. The decidability of the problem follows from the effective translation to Rabin automata. The lower bound is inherited from that for first-order logic over finite words.

5 The μ-calculus and alternating automata

In the previous sections we have described very classical equivalences between monadic second-order logic and automata. Here we will present another logical formalism equivalent to the two. This will be the μ-calculus, an extension of modal logic with fixpoint operators. We will show a very direct connection between the μ-calculus and alternating automata. These are an extension of nondeterministic automata with universal moves, in the same way as alternating Turing machines are an extension of nondeterministic Turing machines.

5.1 Syntax and semantics of the μ-calculus

Here we will introduce the μ-calculus over binary trees. Later, we will be also interested in the μ-calculus over words and arbitrary graphs. These variations can be easily obtained by changing the set of modalities.

Let $Var_2 = \{X, Y, \dots\}$ be the set of second order variables. Let $\{P_a : a \in \Sigma\}$ be the set of propositional letters. The syntax of the μ-calculus is given by the following grammar:

$$X \mid P_a \mid \neg\alpha \mid \alpha \vee \beta \mid \langle 0\rangle\alpha \mid \langle 1\rangle\alpha \mid \mu X.\alpha(X)$$

where in the last construct we require that X appears only positively (under even number of negations) in $\alpha(X)$. In this construct μ binds X. This has the same consequences for substitution as in the case of quantifiers in first-order logic (free variables of the formula being substituted should not be captured by the binders of the formula into which we substitute). We write $\alpha[\beta/X]$ for the result of substituting the formula β for the variable X in the formula α.

A binary tree is represented as a structure $\mathcal{M} = \langle \{0,1\}^*, (P_a^{\mathcal{M}})_{a\in\Sigma}\rangle$. Previously we had the successor relations s_1 and s_2 in the signature. Now we do not need them as we do not have them in the syntax of the logic. The meaning of a sentence is a set of nodes of a tree. To define the meaning of a formula with free variables we need a valuation $V : Var_2 \to \mathcal{P}(\{0,1\}^*)$ assigning to each variable a set of nodes of the tree. The meaning $[\![\alpha]\!]_V^{\mathcal{M}}$ of a formula α in a tree $\mathcal{M}$ and valuation V is defined inductively as follows:

- $[\![X]\!]_V^{\mathcal{M}} = V(X)$
- $[\![P_a]\!]_V^{\mathcal{M}} = P_a^{\mathcal{M}}$
- $[\![\neg\alpha]\!]_V^{\mathcal{M}} = \{0,1\}^* \setminus [\![\alpha]\!]_V^{\mathcal{M}}$

- $[\![\alpha \vee \beta]\!]_V^{\mathcal{M}} = [\![\alpha]\!]_V^{\mathcal{M}} \cup [\![\beta]\!]_V^{\mathcal{M}}$
- $[\![\langle 0\rangle \alpha]\!]_V^{\mathcal{M}} = \{v : v0 \in [\![\alpha]\!]_V^{\mathcal{M}}\}$
- $[\![\langle 1\rangle \alpha]\!]_V^{\mathcal{M}} = \{v : v1 \in [\![\alpha]\!]_V^{\mathcal{M}}\}$
- $[\![\mu X.\alpha(X)]\!]_V^{\mathcal{M}} = \bigcap\{S \subseteq \{0,1\}^* : [\![\alpha]\!]_{V[S/X]}^{\mathcal{M}} \subseteq S\}$

So, the meaning of $\mu X.\alpha(X)$ is the least fixpoint of an operator assigning to a set S the set $[\![\alpha(X)]\!]_{V[S/X]}^{\mathcal{M}}$. By our assumption on the positivity of X, this operator is monotone, i.e., $[\![\alpha(X)]\!]_{V[S_1/X]}^{\mathcal{M}} \subseteq [\![\alpha(X)]\!]_{V[S_2/X]}^{\mathcal{M}}$ if $S_1 \subseteq S_2$. Hence, the least fixpoint always exists in the complete lattice of sets of tree nodes.

If α is a sentence then its meaning does not depend on the valuation. We will then write $[\![\alpha]\!]^{\mathcal{M}}$ or even $[\![\alpha]\!]$ if $\mathcal{M}$ is clear from the context. We will sometimes write $\mathcal{M}, v \vDash \alpha$ instead of $v \in [\![\alpha]\!]^{\mathcal{M}}$. A sentence α defines the language of trees

$$L(\alpha) = \{t : \mathcal{M}_t, \varepsilon \vDash \alpha\}$$

So a formula defines a set of trees for which it holds in the root node.

The greatest fixpoint operator, denoted $\nu X.\alpha(X)$ is definable using the least fixpoint by

$$\nu X.\alpha(X) = \neg\mu X.\neg\alpha(\neg X)$$

We will use σ to mean μ or ν.

Let us look at some example properties expressible in the μ-calculus. The formula $\mu X.P_a \vee \langle 0\rangle X \vee \langle 1\rangle X$ holds in the root of a tree whenever there is a node labelled by a. If there is such a node v then the path from ε to v belongs to the least fixpoint of the operator defined by the formula (any fixed point should contain v and should be closed under father relation). If there is no node labelled by a then the empty set is a fixpoint.

The formula $\nu X.P_b \wedge (\langle 0\rangle X \vee \langle 1\rangle X)$ holds in the root of a tree if there is an infinite path from ε every node of which is labelled with b. Indeed, if there is such a path then it is a fixpoint of the operator. Other way around, if S is a fixpoint of the operator and $\varepsilon \in S$ then ε is labeled by b and ε has a successor $v \in S$. Hence, inductively we can construct an infinite path with all the vertices labelled by b. Observe that if in the above formula we replaced the greatest fixpoint with the least, obtaining $\mu X.P_b \wedge (\langle 0\rangle X \vee \langle 1\rangle X)$, then we would get an unsatisfiable formula.

Sometimes it will be convenient to work with formulas in *positive normal form*, i.e., formulas were negations occur only before propositional constants and variables. The next lemma says that for this we need to add conjunction and the greatest fixpoint operators to the syntax.

Lemma 26 Every formula of the μ-calculus is equivalent to a formula in a positive normal form, possibly using conjunction and the greatest fixpoint operator.

Proof
A formula in a positive normal form can be obtained by repetitive uses of de Morgan laws and the equivalences:

$$\neg\langle 0\rangle\alpha \equiv \langle 0\rangle\neg\alpha \qquad \neg\langle 1\rangle\alpha \equiv \langle 1\rangle\neg\alpha$$
$$\neg\, X.\alpha(X) \equiv \nu X.\neg\alpha(\neg X)$$

□

5.2 Alternating automata

Alternating automata model extends nondeterministic automata with a notion of universal moves. Here we will present alternating tree automata postponing presentation of variations for words and graphs to later sections.

An alternating tree automaton is a tuple:

$$\mathcal{A} = \langle Q, Q_\exists, Q_\forall, \Sigma, q^0, \delta : Q \times \Sigma \to \mathcal{P}(Q \times \{0, 1, \varepsilon\}), Acc\rangle$$

There are two differences with nondeterministic automata. First, the set Q of states is partitioned into existential and universal states, $Q_\exists$ and $Q_\forall$ respectively. Next, the transition relation is a function and ε-transitions are allowed. If an automaton is in an existential state q and in a vertex labelled by a then it chooses a transition from $\delta(q, a)$ which it is going to execute. If q is universal then the automaton has to execute all the transitions from $\delta(q, a)$. To execute a transition (q', d) in a vertex v means to go to the vertex vd and change the state to q'. So, if $d = \varepsilon$ then the automaton stays in v, if $d = 0$ it moves to the left son of v.

It is the simplest to formalize the notion of a run and acceptance of alternating automata in terms of games. Given a tree t we define the *acceptance game* $G_{\mathcal{A},t}$ which is very similar to the one defined for nondeterministic automata. We have:

- the set V_0 of vertices for player 0 is $\{0, 1\}^* \times Q_\exists$,
- the set V_1 of vertices for player 1 is $\{0, 1\}^* \times Q_\forall$,
- from each vertex (v, q) and $(q', d) \in \delta(q, t(v))$ there is an edge to (vd, q').

- the acceptance condition Acc consists of the sequences

$$(v_0, q_0)(v_1, q_1)\dots$$

such that the sequence $q_0 q_1 \dots$ is in Acc, i.e., it belongs to the acceptance condition of the automaton.

We say that $\mathcal{A}$ *accepts* a tree t iff player 0 has a winning strategy in the game $G_{\mathcal{A},t}$. The *language recognized by* $\mathcal{A}$ is the set of trees accepted by $\mathcal{A}$.

Example: An automaton expressing that there is a descendant labelled by a can have two states $q, \top$ and the transition function:

$$(q, b) \mapsto \{(q, 0), (q, 1)\} \qquad (q, a) \mapsto \{(\top, \varepsilon)\}$$

where $\top$ is a state from which every tree is accepted. We make q an existential state and put $\Omega(q) = 1$. When such an automaton starts from the state q in a vertex labelled by a then it goes to the accepting state $\top$. If the vertex is labelled by b then it goes with the state q to one of its successors and the process repeats. It cannot repeat like this for ever because then we would have a path consisting only of q states and such a path is not accepting. Hence in order to accept the automaton must finally meet a vertex labelled by a. If we changed the status of q from existential to universal then the automaton would accept from q precisely those trees which have a on every path. $\square$

The following lemma shows that the complementation is easy for alternating automata. It is also easy to show that alternating automata are closed under disjunction and conjunction. Hence it seems that they may be a better candidate to use in the proof of Rabin's theorem than nondeterministic automata where we need a lot of work to show the closure under complement. Unfortunately, it is difficult to show that alternating automata are closed under projection. One essentially needs to convert an alternating automaton into a nondeterministic automaton. This in turn is as complicated as the closure under complement for nondeterministic automata.

Lemma 27 For every Mostowski alternating automaton $\mathcal{A}$ there is a dual Mostowski automaton $\overline{\mathcal{A}}$, such that $L(\overline{\mathcal{A}}) = \mathrm{Trees}(\Sigma) \setminus \mathcal{A}$. The size of $\overline{\mathcal{A}}$ is the same as the size of $\mathcal{A}$.

Proof
Let $\mathcal{A} = \langle Q, Q_\exists, Q_\forall, \Sigma, q^0, \delta, \Omega \rangle$ be an alternating automaton as above. The dual automaton is

$$\overline{\mathcal{A}} = \langle Q, Q_\forall, Q_\exists, \Sigma, q^0, \delta, \overline{\Omega} \rangle$$

where $\overline{\Omega}(q) = \Omega(q) + 1$ for every $q \in Q$. Hence, $\overline{\mathcal{A}}$ has the same set of states and the same transition function as $\mathcal{A}$. The difference is that the roles of existential and universal states are interchanged. Also the acceptance condition is negated.

To see that $L(\overline{\mathcal{A}})$ is the complement of $L(\mathcal{A})$ consider a tree $t \in \text{Trees}(\Sigma)$ and the acceptance games $G_{\mathcal{A},t}$, $G_{\overline{\mathcal{A}},t}$. The graphs of these games are the same but every position for player 0 in $G_{\mathcal{A},t}$ becomes a position for player 1 in $G_{\overline{\mathcal{A}},t}$ and vice versa. Moreover, for every infinite play in these games, the play is winning for player 0 in $G_{\mathcal{A},t}$ iff it is winning for player 1 in $G_{\overline{\mathcal{A}},t}$. Hence, player 0 has a winning strategy in $G_{\mathcal{A},t}$ iff player 1 has a winning strategy in $G_{\overline{\mathcal{A}},t}$ (this is just the same strategy). So, $t \in L(\mathcal{A})$ iff $t \notin L(\overline{\mathcal{A}})$. □

5.3 From the μ-calculus to alternating automata

The translation from the μ-calculus to alternating automata is quite direct. The interesting and not obvious part of the translation is the construction of acceptance conditions. The presented translation is based on the ideas from [51, 48]

Let us fix a sentence α in a positive normal form and such that each variable is bound at most once in α. Let $cl(\alpha)$ stand for the set of subformulas of α. As every variable is bound at most once, we can use β_X for the unique subformula such that $\sigma X.\beta_X \in cl(\alpha)$. If σ is μ then we call X a *μ-variable*, otherwise we call X a *ν-variable*. We construct an alternating automaton:

$$\mathcal{A}^\alpha = \langle Q^\alpha, Q^\alpha_\exists, Q^\alpha_\forall, \Sigma, \alpha, \delta^\alpha, Acc^\alpha \rangle$$

where $Q^\alpha = cl(\alpha) \cup \{\top, \bot\}$ is the set of subformulas of α with two additional states. The intended meaning of the additional states is that the automaton should accept everything from $\top$ and it should accept nothing from $\bot$. The initial state is the formula α itself. The partition of states is such that $Q_\exists$ contains all disjunctions (formulas of the form $\beta_1 \vee \beta_2$) and $Q_\forall$ contains the rest. When we will define transition function, it will be clear that from states other than disjunctions and conjunctions there is no choice. So it is irrelevant whether we put these states in $Q_\exists$ or $Q_\forall$.

The transition function is defined by:

- $\delta(P_a, a) = \{(\top, \varepsilon)\}$,
- $\delta(P_a, b) = \{(\bot, \varepsilon)\}$ for $a \neq b$,
- $\delta(X, a) = \{(\beta_X, \varepsilon)\}$,

- $\delta(\beta_1 \vee \beta_2, a) = \{(\beta_1, \varepsilon), (\beta_2, \varepsilon)\}$ and the same for conjunction,
- $\delta(\langle 0\rangle\beta, a) = \{(\beta, 0)\}$ and similarly for 1,
- $\delta(\sigma X.\beta, a) = (\beta, \varepsilon)$,

Hence, the automaton just decomposes the formula and, in the case of modal formulas, it proceeds in an appropriate direction. In the case of a proposition P_a it checks the labeling of the current node. If the label is a then it accepts the rest of the tree, otherwise it rejects. In the case of a variable, it replaces it by its fixpoint definition. We have not shown the obvious transitions from $\top$ and $\bot$.

Before defining the acceptance condition we need one more notion.

Definition 28 The *dependence order* on bound variables of α is the smallest partial order such that $X <_\alpha Y$ if X occurs free in $\sigma Y.\beta_Y$. The *alternation depth* of a variable X, denoted adepth(X), is the maximal number of alternations between μ and ν-variables in a chain $X <_\alpha Z_1 <_\alpha \cdots <_\alpha Z_k$.

For example, in the formula $\mu X.(\nu Y.\langle 0\rangle Y) \wedge \langle 1\rangle X$ the alternation depths of both X and Y are 0. This is because X is not smaller than Y in the dependence order as it does not occur free in the Y subformula. On the other hand, in $\mu X.(\nu Y.\langle 1\rangle X \wedge \langle 0\rangle Y)$ the alternation depth of X is 1.

The acceptance condition Acc_α is the Mostowski condition $\Omega_\alpha : Q_\alpha \to \mathbb{N}$ defined by

$$\Omega_\alpha(\beta) = \begin{cases} 2 * (\max_d - \text{adepth}(X)) & \text{if } \beta = X \text{ is a } \nu\text{-variable} \\ 2 * (\max_d - \text{adepth}(X)) + 1 & \text{if } \beta = X \text{ is a } \mu\text{-variable} \\ M & \text{otherwise} \end{cases}$$

where $\max_d$ is the maximal alternation depth of a variable from $cl(\alpha)$, and M is the maximal value of Ω_α for variables. The intention is that meeting a ν-variable brings us closer to acceptance while meeting a μ-variable takes us further away. The values of Ω_α for formulas other than variables do not matter.

The correctness of the construction is stated in the next theorem.

Theorem 29
For every tree t represented by a structure $\mathcal{M}_t$: $\mathcal{M}_t, \varepsilon \vDash \alpha$ iff $t \in L(\mathcal{A}_\alpha)$.

In the rest of the subsection we will sketch the proof of the theorem. The main point is to understand the behaviour of fixpoints. We do this by introducing fixpoint approximations. This will allow to use an induction argument on approximations.

Definition 30 An *approximation* of a formula $\mu X.\beta(X)$ is $\mu^\tau X.\beta(X)$ for some ordinal τ. The meaning of such an approximation is defined by:

$$[\![\mu^0 X.\beta(X)]\!]_V^{\mathcal{M}} = \emptyset \qquad [\![\mu^{\tau+1} X.\beta(X)]\!] = [\![\beta(X)]\!]^{\mathcal{M}}_{V[[\![\mu^\tau X.\beta(X)]\!]^{\mathcal{M}}_V/X]}$$
$$[\![\mu^\tau X.\beta(X)]\!]_V^{\mathcal{M}} = \bigcup_{\tau'<\tau} [\![\mu^{\tau'} X.\beta(X)]\!]_V^{\mathcal{M}} \quad \text{if } \tau \text{ is a limit ordinal}$$

Similarly we define approximations $\nu^\tau X.\beta(X)$ of ν-formulas.

Approximations are not themselves formulas of the μ-calculus. They are extensions of the syntax needed in the inductive argument showing correctness of our automaton. Recall that by Knaster-Tarski theorem we have:

$$[\![\mu X.\beta(X)]\!]_V^{\mathcal{M}} = \bigcup_{\tau \in Ord} [\![\mu^\tau X.\beta(X)]\!]_V^{\mathcal{M}}$$

Here the sum is over all ordinals, but it is enough to stop at the first ordinal whose cardinality is bigger than the cardinality of $\mathcal{M}$.

Definition 31 Let $X_1, \ldots, X_k$ be all the variables from α listed in the order respecting $<_\alpha$ relation (with smaller variables first). For a sequence of ordinals $\vec{\tau} = (\tau_1, \ldots, \tau_k)$ and a formula $\gamma \in cl(\alpha)$ we define:

$$\langle\!|\gamma|\!\rangle^\mu_{\vec{\tau}} = \gamma[\sigma'_k X_k.\beta_k/X_k] \ldots [\sigma'_1 X_1.\beta_1/X_1]$$

where $\sigma'_m = \nu$ if X_m is a ν-variable and $\sigma'_m = \mu^{\tau_m}$ otherwise. We define $\langle\!|\gamma|\!\rangle^\nu_{\vec{\tau}}$ similarly but now $\sigma'_m = \nu^{\tau_m}$ if X_m is a ν-variable and $\sigma'_m = \mu$ otherwise.

So $\langle\!|\gamma|\!\rangle^\mu_{\vec{\tau}}$ is the result of replacing sequentially every free ν-variable by a fixpoint formula and every μ variable by its approximation.

Definition 32 If $v \in [\![\gamma]\!]_V^{\mathcal{M}}$ for some $\gamma \in cl(\alpha)$ then we define the *μ-signature*, $\mathrm{Sig}(\gamma, v)$, of γ in the vertex v of the model $\mathcal{M}$ to be the least in the lexicographical ordering tuple of ordinals $\vec{\tau}$ such that $v \in [\![\langle\!|\gamma|\!\rangle^\mu_{\vec{\tau}}]\!]_V^{\mathcal{M}}$.

If $v \notin [\![\gamma]\!]_V^{\mathcal{M}}$ then the *ν-signature* of γ, $\mathrm{Sig}^\nu(\gamma, v)$, is the least in the lexicographical ordering tuple of ordinals $\vec{\tau}$ such that $v \notin [\![\langle\!|\gamma|\!\rangle^\nu_{\vec{\tau}}]\!]_V^{\mathcal{M}}$.

Using Knaster-Tarski theorem one can check that μ and ν-signatures always exists when the conditions of the definition are satisfied. Having signatures we can formulate the signature decrease lemma which is the main tool in proving correctness of our automaton.

Lemma 33 (Signature decrease) For every vertex v, whenever the left hand-sides are defined we have:

- $\mathrm{Sig}(\beta_1 \wedge \beta_2, v) = \max(\mathrm{Sig}(\beta_1, v), \mathrm{Sig}(\beta_2, v))$.
- $\mathrm{Sig}(\beta_1 \vee \beta_2, v) = \mathrm{Sig}(\beta_1, v)$ or $\mathrm{Sig}(\beta_1 \vee \beta_2, v) = \mathrm{Sig}(\beta_2, v)$.
- $\mathrm{Sig}(\langle 0 \rangle \beta, v) = \mathrm{Sig}(\beta, v0)$ and similarly for 1.
- $\mathrm{Sig}(\nu X.\beta(Y), v) = \mathrm{Sig}(\beta(Y), v)$.
- $\mathrm{Sig}(\mu X_i.\beta(X_i), v)$ is the same as $\mathrm{Sig}(\beta(X_i), v)$ on the first $i-1$ positions.
- If Y is a ν-variable, $\mathrm{Sig}(Y, v) = \mathrm{Sig}(\beta_Y(Y), v)$.
- If X is a μ-variable, $\mathrm{Sig}(X_i, v)$ is bigger than $\mathrm{Sig}(\beta_{X_i}(X_i), v)$ and the difference is at position i.

Similarly for ν-signatures but with interchanged roles of μ with ν, and conjunction with disjunction.

Using the signature decrease lemma we can show that if $\mathcal{M}_t, \varepsilon \vDash \alpha$ then $t \in L(\mathcal{A}_\alpha)$. For this we show that there is a winning strategy for player 0 in the acceptance game $G(\mathcal{A}_\alpha, t)$. The only choice for player 0 in this game is in the case of a disjunction $\beta_1 \vee \beta_2$. In a position $(v, \beta_1 \vee \beta_2)$ he should choose (v, β_1) if $\mathrm{Sig}(\beta_1 \vee \beta_2, v) = \mathrm{Sig}(\beta_1, v)$ and (v, β_2) otherwise.

To see that such a strategy is winning for player 0 assume conversely that there is a play $(v_1, \gamma_1)(v_2, \gamma_2)\ldots$ on which some odd priority p is the least priority appearing infinitely often. This means that on this play we infinitely often meet the μ-variable X_l where $l = (p-1)/2$. Let m be a step of the play after which no priority smaller than p appears. In particular it means that after m we there are no variables with indices smaller than l. By the signature decrease lemma, the signatures of positions of the play after m never increase on the first l positions. They decrease every time we meet X_l. But this is impossible as the lexicographic order on l-tuples of ordinals is a well ordering. Hence, such a play cannot exist, and the strategy we have defined is winning for player 1.

The proof of the theorem in the other direction is very similar. We show that if $\mathcal{M}_t, \varepsilon \nvDash \alpha$ then $t \notin L(\mathcal{A}_\alpha)$. For this we present a winning strategy for player 1 in $G(\mathcal{A}_\alpha, t)$. Player 1 has a choice only in case of conjunction. In a position $(v, \beta_1 \wedge \beta_2)$ he should choose (v, β_1) if $\mathrm{Sig}^\nu(\beta_1 \vee \beta_2, v) = \mathrm{Sig}^\nu(\beta_1, v)$ and (v, β_2) otherwise.

5.4 From alternating automata to the μ-calculus

In the translation from alternating automata to the μ-calculus it will be convenient to use vectorial syntax as an intermediate step. Hence, we start

with a definition of μ-formulas in vectorial form. As it will turn out later, every μ-calculus formula is equivalent to the one in this form.

Let $n \in \mathbb{N}$. An n-array μ-calculus formula has the form

$$\sigma_1 \begin{pmatrix} X_1^1 \\ \vdots \\ X_n^1 \end{pmatrix} \ldots \sigma_m \begin{pmatrix} X_1^m \\ \vdots \\ X_n^m \end{pmatrix} . \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}$$

where m is some integer; $\sigma_1, \ldots, \sigma_m$ are fixpoint operators; and $\alpha_1, \ldots, \alpha_n$ are formulas of the ordinary (scalar) μ-calculus without fixpoint operators.

The semantics of such a formula in a tree $\mathcal{M}$ and a valuation V is a set of n-tuples of vertices of $\mathcal{M}$. For a vectorial formula without fixpoints we have

$$[\![(\alpha_1, \ldots, \alpha_n)]\!]_V^{\mathcal{M}} = [\![\alpha_1]\!]_V^{\mathcal{M}} \times \cdots \times [\![\alpha_n]\!]_V^{\mathcal{M}}$$

For a fixpoint we have

$$[\![\mu \vec{X}.\vec{\beta}]\!]_V^{\mathcal{M}} = \bigcap \{\vec{S} \subseteq (\{0,1\}^*)^n : [\![\vec{\beta}]\!]_{V[\vec{S}/\vec{X}]}^{\mathcal{M}} \subseteq \vec{S}\}$$

So, this is a fixpoint of an operator over sets of n-tuples of vertices. The greatest fixpoint is defined similarly.

Finally, we introduce the projection operation. If $\vec{\beta}$ is an n-array formula then we write $(\vec{\beta})\downarrow_1$ for the value of the first component. So, $[\![(\vec{\beta})\downarrow_1]\!]_V^{\mathcal{M}}$ is the first component of $[\![\vec{\beta}]\!]_V^{\mathcal{M}}$.

A first useful observation is that vectorial μ-calculus is not more powerful than the ordinary one

Lemma 34 For every vectorial formula $\vec{\beta}$ there is a formula α of the ordinary (scalar) μ-calculus such that for every tree $\mathcal{M}$ and valuation V we have: $[\![(\vec{\beta})\downarrow_1]\!]_V^{\mathcal{M}} = [\![\alpha]\!]_V^{\mathcal{M}}$.

Proof
The proof is a rather tedious application of Bekic principle:

$$\left(\sigma \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} . \begin{pmatrix} \alpha_1(X_1, X_2) \\ \alpha_2(X_1, X_2) \end{pmatrix}\right)\downarrow_1 = \ \sigma X_1.\ \alpha_1(X_1, \sigma X_2.\ \alpha_2(X_1, X_2))$$

that is, the meaning of the first component of the formula on the left is the same as the meaning of the formula on the right. This principle holds for every complete lattice; hence in all interpretations we consider in these notes. □

Let $\mathcal{A}$ be an alternating automaton with a Mostowski acceptance condition given by a function $\Omega : Q \to \mathbb{N}$. Let $q_1, \ldots, q_n$ be an ordering of the states of $\mathcal{A}$ such that $\Omega(q_i) \leq \Omega(q_j)$ for every $i < j$. The vectorial formula corresponding to $\mathcal{A}$ is

$$\alpha_{\mathcal{A}} = \sigma_1 \vec{X}_1 \ldots \sigma_n \vec{X}_n . \begin{pmatrix} \bigvee_{a \in \Sigma} (P_a \wedge \lceil \delta(q_1, a) \rceil) \\ \vdots \\ \bigvee_{a \in \Sigma} (P_a \wedge \lceil \delta(q_n, a) \rceil) \end{pmatrix}$$

where σ_i is μ if $\Omega(q_i)$ is odd and σ_i is ν otherwise. We also need to explain what $\lceil \delta(q_i, a) \rceil$ stand for. Recall that $\delta(q_i, a)$ is a subset of $Q \times \{0, 1, \varepsilon\}$. For a pair $(q_j, d) \in Q \times \{0, 1, \varepsilon\}$ we put:

$$\lceil (q_j, d) \rceil = \begin{cases} \langle 0 \rangle X_j^j & \text{if } d = 0 \\ \langle 1 \rangle X_j^j & \text{if } d = 1 \\ X_j^j & \text{if } d = \varepsilon \end{cases}$$

then we put

$$\lceil \delta(q_i, a) \rceil = \begin{cases} \bigvee \{ \lceil (q', d') \rceil : (q', d') \in \delta(q_i, a) \} & \text{if } q_i \in Q_\exists \\ \bigwedge \{ \lceil (q', d') \rceil : (q', d') \in \delta(q_i, a) \} & \text{if } q_i \in Q_\forall \end{cases}$$

Theorem 35
For every tree t: $t \in L(\mathcal{A})$ iff $\mathcal{M}_t, \varepsilon \vDash (\alpha_{\mathcal{A}}) \downarrow_1$

The proof is similar to the one for the translation form formulas to automata. It also uses the signature decrease lemma.

5.5 The μ-calculus and alternating automata over graphs

Originally [31] the μ-calculus was defined over arbitrary directed graphs and not just binary trees. In this setting, instead of $\langle 0 \rangle$ and $\langle 1 \rangle$ modalities it has one modality $\langle \cdot \rangle$. The models are Σ-labelled graphs

$$G = \langle V, E \subseteq V \times V, \lambda : V \to \Sigma \rangle$$

Such a graph can be represented as a structure $\mathcal{M}_G = \langle V, E, (P_a)_{(a \in \Sigma)} \rangle$. The meaning of the modality is:

- $[\![\langle \cdot \rangle \alpha]\!]_V^{\mathcal{M}} = \{v : \exists v' .\ E(v, v') \text{ and } v' \in [\![\alpha]\!]_V^{\mathcal{M}}\}$

The rest of the clauses is the same as for the μ-calculus over binary trees. A new thing in the present situation is that the modality is "nondeterministic", i.e., there are several possible v' in the semantical clause above. Using negation we can define the dual modality $[\cdot]\alpha = \neg \langle \cdot \rangle \neg \alpha$. So we have:

- $[\![[\cdot]\alpha]\!]_V^{\mathcal{M}} = \{v : \forall v'.\ E(v,v') \Rightarrow v' \in [\![\alpha]\!]_V^{\mathcal{M}}\}$

We can also extend the notion of alternating automata to Σ-labelled graphs. Such an automaton has a form:

$$\mathcal{A} = \langle Q, \Sigma, Q_\exists, Q_\forall, q^0, \delta : Q \times \Sigma \to \mathcal{P}(Q \times \{\langle\cdot\rangle, [\cdot], \varepsilon\}), Acc\rangle$$

The only difference with the automaton on binary trees is in the transition function. Before we had 0 and 1 and now we have $\langle\cdot\rangle$ and $[\cdot]$. At first it may seem that $\langle\cdot\rangle$ should be enough, but we do not have negation in automata so we would have no means to express $[\cdot]$. Before we did not need negation of $\langle 0\rangle$ because over binary trees it was expressible using $\langle 0\rangle$ itself.

The acceptance of such automata is defined using games almost the same way as before. Given a Σ labelled graph $\mathcal{M}$ we have the game $G_{\mathcal{A},\mathcal{M}}$:

- the set V_0 of vertices for player 0 is $\{0,1\}^* \times (Q_\exists \cup Q \times \{\langle\cdot\rangle, \varepsilon\})$,
- the set V_1 of vertices for player 1 is $\{0,1\}^* \times (Q_\forall \cup Q \times \{[\cdot]\})$,
- from a vertex (v,q), for each $(q',d') \in \delta(q,\lambda(v))$ there is an edge to $(v,(q',d'))$
- form a vertex $(v,(q',\varepsilon))$ there is an edge to (v,q'),
- form a vertex $(v,(q',\langle\cdot\rangle))$ or $(v,(q',[\cdot]))$ there is an edge to (v',q') for every successor v' of v.
- the acceptance condition Acc_G consists of the sequences

  $$(v_0,q_0),(v_0,(q_1,d_1)),(v_1,q_1),(v_1,(q_2,d_2)),(v_2,q_2),\ldots$$

  such that the sequence $q_0q_1\ldots$ is in Acc, i.e., it belongs to the acceptance condition of the automaton.

The difference with the game for binary trees is that now we have an additional round. If a game reaches a position $(v,(q',\langle\cdot\rangle))$ then player 0 chooses the successor of v. In a position $(v,(q',[\cdot]))$ the choice is made by player 1.

Very similar translations to the previous ones show

Theorem 36
The μ-calculus over Σ-labelled graphs is equivalent to the alternating automata. The translations in both directions are effective.

5.6 Relation to MSOL: binary trees

An interesting question is to compare the μ-calculus and alternating automata with MSOL. Given all the facts on automata and MSOL that we have seen till now, it is not difficult to show that over words and trees the formalisms are the same.

Consider the following translation of the μ-calculus over binary trees into MSOL:

$$\begin{aligned} P_a &\rightsquigarrow P_a(x) \\ X &\rightsquigarrow X(x) \\ \alpha \vee \beta &\rightsquigarrow \varphi_\alpha(x) \vee \varphi_\beta(x) \\ \neg\alpha &\rightsquigarrow \neg\varphi_\alpha(x) \\ \langle 0\rangle\alpha &\rightsquigarrow \exists y.s_0(x,y) \wedge \varphi_\alpha(y) \\ \mu X.\alpha(X) &\rightsquigarrow \forall Z.(\forall y.\ \varphi_\alpha(Z,y) \Rightarrow Z(y)) \Rightarrow Z(x) \end{aligned}$$

This translation produces a MSOL formula with one free first order variable x and the same free second-order variables as in the starting formula. It is not difficult to prove by induction on a formula α that for every tree t, vertex v of t and valuation $V : Var_2 \to \mathcal{P}(\{0,1\}^*)$ of second order variables we have:

$$v \in [\![\alpha]\!]_V^{\mathcal{M}_t} \quad \text{iff} \quad \mathcal{M}_t, V \vDash \varphi_\alpha(v)$$

This shows that whenever a set of trees is definable by a μ-calculus formula then it is definable by a MSOL formula.

The translation in the other direction goes through nondeterministic automata. We know that for every MSOL formula over binary trees there is an equivalent nondeterministic automaton. A nondeterministic automaton is a special case of alternating automaton. For every alternating automaton there is an equivalent μ-calculus formula. Summarizing we get:

Corollary 37 Over binary trees MSOL, the μ-calculus, alternating automata and nondeterministic automata have the same expressive power. This equivalence holds also over finite and infinite words.

From the presented translations it follows that MSOL is the most succinct of the formalism. For a formula of the μ-calculus or for an automaton there is an equivalent MSOL formula of a linear size. It can be shown that every translation the other way must produce results of nonelementary size. As we have seen, the translations between alternating automata and vectorial μ-calculus are linear. The known translations to scalar μ-calculus produce an exponential blowup. The translation from alternating to nondeterministic automata has an exponential lower and upper bound.

5.7 Relation to MSOL: graphs

An interesting question is whether we can relate the μ-calculus and MSOL over graphs. We cannot hope to have exact equivalence. As we will see μ-calculus sentences cannot distinguish between bisimilar models while sentences MSOL sometimes can. Still, we get a surprisingly strong connection.

A *bisimulation* between two Σ-labelled graphs $G_1 = \langle V_1, E_1, \lambda_1 \rangle$ and $G_2 = \langle V_2, E_2, \lambda_2 \rangle$ is a relation $R \subseteq V_1 \times V_2$ such that whenever $(v_1, v_2) \in R$ then:

- $\lambda(v_1) = \lambda(v_2)$,
- for each successor v_1' of v_1 there is a successor v_2' of v_2 with $(v_1', v_2') \in R$,
- the same with v_1 and v_2 interchanged.

Fact 38 Every μ-calculus sentence is invariant under bisimulation. That is if $\mathcal{M}, v \vDash \alpha$ and there is a bisimulation on $\mathcal{M} \times \mathcal{M}'$ relating v ad v' then $\mathcal{M}, v' \vDash \alpha$.

Proof
The proof is quite easy for alternating automata. We get the thesis using the correspondence from Theorem 36. □

Fact 39 There is an MSOL sentence which is not invariant under bisimulation.

Proof
Just consider a sentence saying that a model is a tree and its root has exactly two successors. □

So the most we can expect is that every MSOL sentence that is invariant under bisimulation is equivalent to a μ-calculus sentence. This is indeed the case. Before stating this theorem let us examine the power of MSOL on trees of arbitrary degree (i.e., not only binary).

A graph is a tree iff to every node there is a unique finite path from the distinguished vertex called the root. A counting alternating automaton is an extension of an alternating automaton with the ability to count the number of successors. It is of the form:

$$\mathcal{A} = \langle Q, \Sigma, Q_\exists, Q_\forall, q^0, \delta : Q \times \Sigma \to \mathcal{P}(Q \times \{\varepsilon, \langle n \rangle, [n] : n \in \mathbb{N}\}), Acc \rangle$$

The intuitive meaning of $(q, \langle n \rangle)$ move is that the automaton needs to find n distinct successors to which it should go with the state q. The meaning for $(q, [n])$ is that it should go to all but n successors with the state q. We have [54]

Theorem 40
Counting alternating automata characterize the expressive power of MSOL on trees of arbitrary degree.

Clearly counting alternating automata can distinguish between bisimilar structures. From every counting alternating automaton we can produce an ordinary one by changing each $\langle n \rangle$ and $[n]$ into $\langle \cdot \rangle$ and $[\cdot]$ respectively. An interesting thing is that if the counting automaton happen to accept a bisimulation invariant language then a slight improvement of this simple translation would produce an equivalent non-counting alternating automaton. It in turn can be translated in to a μ-calculus formula. This is roughly the way to show the following correspondence [30].

Theorem 41
A bisimulation invariant property of graphs is MSOL definable iff it is μ-calculus definable.

5.8 Complexity

We briefly summarize the complexity of some decision problems for the μ-calculus and alternating automata.

The satisfiability problem for the μ-calculus is to decide whether a given formula has a model. The problem is EXPTIME-complete [51, 18]. For the lower bound one can reduce the problem of universality of tree automata. For the upper bound one can use the translation to alternating automata.

The emptiness problem for alternating automata is EXPTIME-complete. The EXPTIME algorithm is to translate the automaton into a nondeterministic automaton and check the emptiness of the result. An important point here is that although the automaton grows exponentially in the translation, the acceptance conditions do not [39, 18].

Another important problem is the *model checking* problem: "For a finite graph G and a μ-calculus formula α decide if α holds in the given vertex of G". The problem is equivalent to the emptiness problem for Mostowski alternating tree automata over one letter alphabet [17]. The later problem is in turn equivalent to the emptiness problem for nondeterministic Mostowski tree automata. Hence, its complexity is in NP$\cap$CO-NP but no deterministic polynomial time algorithm is known.

6 Hierarchies

As we have seen, the connections between MSOL, the μ-calculus and alternating automata are quite strong. Here we will refine the connections by considering relations between the hierarchies. For each of the three formalisms there are some "obvious" ways of defining hierarchies. For MSOL we can consider the quantifier alternation hierarchy. For the μ-calculus this will be the fixpoint alternation hierarchy. For alternating automata we can measure the complexity of the acceptance condition.

6.1 Definitions of the hierarchies

MSOL First, we define the hierarchy for MSOL. As it will turn out, a notion of weak quantification is relevant here. A weak quantifier is a quantifier ranging over finite sets. We write a formula $\exists^w X.\ \varphi(X)$ with the meaning that there is a finite set S for which $\varphi(S)$ holds. Over binary trees finiteness is definable so addition of weak quantification does not extend the power of the logic over this model. Let WMSOL be the fragment of MSOL using only first-order and weak second order quantification.

Consider MSOL sentences of the form $Q_1 X_1 \dots Q_n X_n.\ \varphi$ where φ is a WMSOL formula and $Q_1 \dots Q_n$ are second-order quantifiers. Define the classes $\Sigma_0^M = \Pi_0^M$ to be exactly WMSOL. Next, for each $i \in \mathbb{N}$ the level Σ_{i+1}^M is the set of formulas $\exists X_1 \dots \exists X_n.\varphi$ with $\varphi \in \Pi_i^M$. Similarly Π_{i+1}^M consists of formulas $\forall X_1 \dots \forall X_n.\varphi$ with $\varphi \in \Sigma_i^M$.

Definition 42 A language L of words, trees or graphs is in Σ_n^M if there is a Σ_n^M formula defining this language. Otherwise it is Σ_n^M*-unfeasible*. Similarly for Π_n^M classes.

μ-calculus The definition of the μ-calculus hierarchy is easier to formulate for the vectorial μ-calculus. The formulas of levels Σ_0^μ and Π_0^μ are of the form $\vec{\alpha}$ for some vector of formulas without a fixpoint operator. Σ_{i+1}^μ formulas are of the form $\mu\vec{X}.\vec{\alpha}$ for some Π_i^μ formula $\vec{\alpha}$. Π_{i+1}^μ formulas are of the form $\nu\vec{X}.\vec{\alpha}$ for some Σ_i^μ formula $\vec{\alpha}$.

Equivalently the definition of the hierarchy for the μ-calculus can be formulated using the scalar syntax. Then $\Sigma_0^\mu = \Pi_0^\mu$ is the set of formulas without fixpoints. Σ_{i+1}^μ is the closure of Π_0^μ under conjunction, disjunction, substitutions and application of the least fixpoint operator μ (i.e. $\mu X.\alpha \in \Sigma_{i+1}^\mu$ if $\alpha \in \Sigma_{i+1}^\mu$). Similarly for Π_{i+1}^μ but now the class is closed under applications of the greatest fixpoint operator.

An *alternation free* fragment of the μ-calculus is the closure of $\Sigma_1^\mu \cup \Pi_1^\mu$ under substitutions.
Remark: Observe that closure under substitutions implies the closure under Boolean operations.
Remark: Here we use the term alternation with a different meaning than in the case of alternating automata.

Definition 43 A language L of words, trees or graphs is in Σ_n^μ if there is a Σ_n^μ formula defining this language (alternatively there is a Σ_n^μ vectorial formula α such that $(\alpha)\downarrow_1$ defines L). Otherwise it is Σ_n^μ-*unfeasible*. Similarly for Π_n^μ classes.

Automata Finally, we define the hierarchy for automata. The hierarchy is based on the size of acceptance conditions. It is the easiest to define it for Mostowski acceptance conditions although it was originally formulated for Rabin conditions (cf. [41])

Automata of level Σ_n^a have the range of the acceptance function Ω contained in the set $\{1, \ldots, n\}$. Automata of level Π_n^a have the range of the acceptance function in $\{0, \ldots, n-1\}$. In the case of $n = 1$ automata from Σ_1^a would accept nothing. To remedy this we assume that automata have a special state $\top$ from which they accept every tree. Such a state is definable in all but Σ_1^a automata.

Remark: The range of the function Ω defining a Mostowski acceptance condition can be always scaled down, so that the smallest number in the range is 0 or 1. Just observe that subtracting 2 from every value of Ω does not change the semantics of the automaton (provided we do not get negative values). Similarly we can cut out any gaps in the range of Ω. So we can always make the image of Ω to be an interval starting from 0 or 1.

A *weak automaton* is an automaton with weak acceptance conditions [38, 33]. These are like Büchi, Rabin, etc. conditions but on the set of all the states appearing in the run and not only on the set of states appearing infinitely often. So for example a weak Büchi condition $F \subseteq Q$ defines a set of runs going through some state from F at least once.

Fact 44 Every weak alternating automaton is a Σ_1^a and a Π_1^a automaton.

The above definitions did not depend on whether an automaton in question is deterministic, nondeterministic or alternating. Hence we have defined not one but actually three hierarchies.

Definition 45 A language L of words, trees or graphs is in deterministic, nondeterministic or alternating Σ_n^a if there is a Σ_n^a deterministic, nondeterministic or alternating automaton recognizing this language. Otherwise it is Σ_n^a-*unfeasible*. Similarly for Π_n^a classes.

6.2 Connecting fixpoint alternation and index hierarchies

Looking closer at the translations between the μ-calculus and alternating automata from Sections 5.3 and 5.4 we can see that they preserve the levels of the hierarchy.

Theorem 46
For every n, $\Sigma_n^\mu = \Sigma_n^a$ *and* $\Pi_n^\mu = \Pi_n^a$.

The hierarchy of nondeterministic tree automata is different. There are Σ_2^a languages that are arbitrary high in the hierarchy of nondeterministic automata. The hierarchy of nondeterministic automata has a corresponding fixpoint hierarchy for formulas with a very limited use of conjunction. These relations are in depth discussed in [41].

We also cannot expect that the MSOL hierarchy coincides with the fixpoint alternation hierarchy. It should be clear that existence of a run of an automaton can be expressed by a formula quite low in the monadic hierarchy. So over words or trees the MSOL hierarchy collapses. The level on which it collapses depends on the model. We will examine it in more detail below.

6.3 The case of words

In the case of infinite words most of the hierarchies collapse on the first level. This is mainly due to the fact that there are deterministic devices capturing the power of MSOL on infinite words.

Fact 47 WMSOL=MSOL over infinite words.

Proof
Let φ be a MSOL formula and let $\mathcal{A}_\varphi$ be a deterministic parity automaton accepting exactly the models of φ. We write a WMSOL formula expressing the fact that there is an accepting run of $\mathcal{A}_\varphi$.

Let $w \in \Sigma^\omega$ be some word. Because $\mathcal{A}_\varphi$ is deterministic, if $\mathcal{A}_\varphi$ has arbitrary long finite prefixes of runs on w then it has a unique infinite run on w. Hence, to state an existence of a run of $\mathcal{A}_\varphi$ it is enough to say that for every position there is a run up to this position.

A run is accepting iff there is some even priority p which appears infinitely often on the run and every smaller priority appears only finitely often on the run. This property can be expressed by saying that there is some position x such that for all positions y after x: (i) the prefix of the run up to y does not end in state of priority smaller than p; and (ii) there is $z > y$ with the run having a state of priority p at position z.

All these facts can be formulated in WMSOL because they refer only to finite prefixes of the model. □

Fact 48 The hierarchy of nondeterministic word automata collapses on the level Π_1^a. Every word automaton is equivalent to a weak alternating automaton.

Proof
The first statement is just rephrasing of the fact that every automaton over words is equivalent to a Büchi nondeterministic automaton.

The second fact is the rephrasing of the above considerations for WMSOL. Let $\mathcal{A}$ be a deterministic parity automaton. We want to find an equivalent weak automaton $\mathcal{B}$. First part of $\mathcal{B}$ just simulates $\mathcal{A}$. All the states in this part have priority 1, so $\mathcal{B}$ cannot stay in this part if it is going to accept a word. It can go to another part when it decides that from that moment no priority smaller than p is going to appear and p is going to appear infinitely often. To check this $\mathcal{B}$ simulates $\mathcal{A}$ over states of priority $\geq p$. It also uses universal branching to check that after every position a state of priority p eventually occurs. □

Corollary 49 Every μ-calculus sentence is equivalent over words to a sentence from the alternation-free fragment.

The picture changes if we restrict ourselves to deterministic automata. We present here the analysis from [42]. Later we will see that very similar examples show up in the case of trees.

Theorem 50
The Σ_n^a hierarchy for deterministic word automata is strict.

To see the examples of the strictness consider for each $n \in \mathbb{N}$ an alphabet $\Sigma_n = \{0, \ldots, n\}$. Then we define the languages:

$$M_n = \{w \in \Sigma_n^\omega : \liminf_{n\to\infty} w(n) \text{ is even}\}$$
$$N_n = \{w \in \Sigma_n^\omega : \liminf_{n\to\infty} w(n) \text{ is odd}\}$$

So, M_n consists of words where the smallest number appearing infinitely often is even, and for words in N_n this number is odd.

It is easy to see that M_n can be recognized by a Σ_n^a deterministic automaton and N_n can be recognized by a Π_n^a deterministic automaton. The proof that there are no simpler automata follows from a more general lemma presented below. It shows a connection between the Mostowski index of an ω-word language and the shape of a deterministic Mostowski automaton recognizing the language. Roughly speaking, it says that in the graph of an automaton recognizing a "hard" language there must be a subgraph, called a flower, "witnessing" this hardness.

Definition 51 Let $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \Omega \rangle$ be a deterministic Mostowski automaton on words. The *graph of* $\mathcal{A}$ is the graph obtained by taking Q as the set of vertices and adding an edge from q to q' whenever $\langle q, a, q' \rangle \in \delta$, for some letter a.

A *path* in a graph is a sequence of vertices $v_1, \ldots, v_j$, such that, for every $i = 1, \ldots, j-1$ there is an edge from v_i to v_{i+1} in the graph. A *maximal strongly connected component* of a graph is a maximal subset of vertices of the graph, such that, for every two vertices v_1, v_2 in the subset there is a path from v_1 to v_2 and from v_2 to v_1.

For an integer k, a *k-loop* in $\mathcal{A}$ is a path $v_1, \ldots, v_j$ in the graph of $\mathcal{A}$ with $v_1 = v_j$, $j > 1$ and $k = \min\{\Omega(v_i) : i = 1, \ldots, j\}$. Observe that a k-loop must necessarily go through at least one edge.

Given integers m and n, a state $q \in Q$ is a *m-n-flower* in $\mathcal{A}$ if for every $k \in \{m, \ldots, n\}$ there is, in the graph of $\mathcal{A}$, a k-loop containing q.

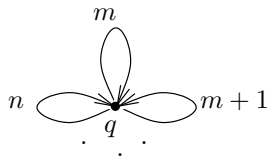


Figure 2: m-n-flower

Definition 52 We say that a language $L \subseteq \Sigma^\omega$ *admits* an m-n-flower if there exists a deterministic Mostowski automaton $\mathcal{A}$, such that, $L = L(\mathcal{A})$ and $\mathcal{A}$ has an m-n-flower q for some q not a useless state in $\mathcal{A}$ (i.e. q occurring in some accepting run of $\mathcal{A}$).

The delicate point about the above definition is that it talks about any deterministic automaton for the language. Intuitively we are interested in the

minimal automaton for the language, but for automata on infinite words the notion of minimality is not very useful to work with. In particular it is not known how to minimize such automata (except for the brute-force algorithm checking all automata up to a given size).

Lemma 53 (Flower Lemma) For every $n \in \mathbb{N}$ and $L \subseteq \Sigma^\omega$: (1) if L is Σ^a_{n+1}-unfeasible then L admits a $2i$-$(2i+n)$-flower, for some i; (2) if L is Π^a_n-unfeasible then L admits a $(2i+1)$-$(2i+1+n)$-flower, for some i.

Corollary 54 The problem of establishing the index of the language accepted by a deterministic automaton $\mathcal{A}$ with a Mostowski condition can be solved in time $\mathcal{O}(|\mathcal{A}|^2)$.

6.4 The case of trees

The case of trees is even more interesting than the case of words. The fixpoint alternation and automata hierarchies are infinite over binary trees. The MSOL hierarchy collapses on Σ^M_2 level. But even here we have an intriguing correspondence between lower levels of the MSOL hierarchy and the fixpoint alternation hierarchy.

The fixpoint alternation hierarchy in infinite

The main result of this section is the proof of strictness of the fixpoint alternation hierarchy. The strictness of the hierarchy over graphs was proved by Bradfield [10]. For binary trees the result was independently shown by Bradfield [11] and Arnold [5]. We present here the beautiful proof of Arnold.

The example languages showing the strictness of the hierarchy reflect closely the acceptance games of alternating automata. For a number $n \in \mathbb{N}$ consider the alphabet $\Sigma_n = \{c_i, d_i : i = 1, \ldots, n\}$. A tree over Σ_n represents a game. In vertices labelled by d_i player 0 chooses a successor, in vertices labelled by c_i player 1 makes a choice. The result of a play in such a game is a path in the tree. We look at the subscripts of the letters c and d and consider those subscripts which appear infinitely often on the path. Player 0 wins if the minimal subscript is even. Hence, a tree over Σ_n defines a game with Mostowski winning conditions given by subscripts of the letters from the alphabet.

With the interpretation of trees over Σ_n as games we can define two families of languages:

$$\begin{aligned} M_n &= \{t \in \mathrm{Trees}(\Sigma_n) : \text{player 0 has a winning strategy on } t\} \\ N_n &= \{t \in \mathrm{Trees}(\Sigma_n) : \text{player 1 has a winning strategy on } t\} \end{aligned}$$

The following easy lemma gives an upper bound on the complexity of these languages.

Lemma 55 Languages M_n and N_n can be recognized by nondeterministic Σ_n^a and Π_n^a automata respectively.

The theorem we want to prove now says that there are no simpler automata, even alternating ones, recognizing these languages.

Theorem 56 (Bradfield, Arnold)
Language N_n cannot be recognized by an alternating Σ_n^a automaton. Similarly, M_n cannot be recognized by an alternating Π_n^a automaton.

For the proof we need a way of coding runs of automata as game trees. Let $\mathcal{A} = \langle Q, Q_\exists, Q_\forall, \Sigma_n, q^0, \delta, Acc\rangle$ be an alternating automaton. We can assume without a loss of generality that for every $q \in Q$ and $a \in \Sigma$ the set $\delta(q,a)$ has precisely two elements. If not then we can split bigger sets and add some auxiliary states connecting them. Sets having zero or one element can be extended with dummy states.

Acceptance of a tree t by $\mathcal{A}$ was defined using the game $G_{\mathcal{A},t}$. By our assumption on the values of δ, every vertex of this game has a degree 2. Still $G_{\mathcal{A},t}$ is usually not a tree. What we are after is an unwinding of this game into a tree such that each node is labelled with c_i or d_i depending on the player this node belongs to and the priority of the node. Formally, we define a function $run_\mathcal{A} : Q \times \mathrm{Trees}(\Sigma_n) \to \mathrm{Trees}(\Sigma_n)$ to be the unique function satisfying that for every $q \in Q$, whenever $\delta(q, t(a)) = \{(q_0, d_0), (q_1, d_1)\}$ then:

$$run_\mathcal{A}(q,t) = \begin{cases} c_i(run_\mathcal{A}(q_0, t|_{d_0}), run_\mathcal{A}(q_1, t|_{d_1})) & \text{for } q \in Q_\forall \\ d_i(run_\mathcal{A}(q_0, t|_{d_0}), run_\mathcal{A}(q_1, t|_{d_1})) & \text{for } q \in Q_\exists \end{cases}$$

We write $c_i(t_0, t_1)$ to denote the tree with root labelled c_i and the trees t_0, t_1 as the left and right subtrees respectively. We use $t|_d$ to denote the subtree rooted in the vertex d. So, $t|_0$ denotes the left subtree of the root and $t|_\varepsilon = t$. In the above definition a small catch is that the value of the function δ is an unordered pair and the definition of $run_\mathcal{A}$ depends on the order in this pair. We assume that we have some unambiguous way of ordering such pairs.

The function we are interested in is $run^0_\mathcal{A} : \mathrm{Trees}(\Sigma_n) \to \mathrm{Trees}(\Sigma_n)$ defined by $run^0_\mathcal{A}(t) = run_\mathcal{A}(q^0, t)$ where q^0 is the initial state of $\mathcal{A}$. The following lemma is just a reformulation of the definition of acceptance.

Lemma 57 For every tree $t \in \mathrm{Trees}(\Sigma_n)$, $t \in L(\mathcal{A})$ iff $run^0_\mathcal{A}(t) \in M_n$.

The proof of the theorem uses a tree which is a fixpoint of the function $run^0_{\mathcal{A}}$. The existence of such a fixpoint is guaranteed by Banach theorem which we now recall. We begin by with the usual ultrametric distance on $\mathrm{Trees}(\Sigma_n)$.

Definition 58 The distance between two trees $t, t' \in \mathrm{Trees}(\Sigma_n)$ is defined inductively as follows. If $t = t'$ then $dist(t, t') = 0$. If the labels of the roots of t and t' are different then $dist(t, t') = 1$, otherwise,

$$dist(t, t') = 1/2 \max(dist(t|_0, t'|_0), dist(t|_1, t'|_1))$$

A mapping $f : \mathrm{Trees}(\Sigma_n) \to \mathrm{Trees}(\Sigma_n)$ is called *contracting* if there is a constant $c < 1$ such that for every $t, t' \in \mathrm{Trees}(\Sigma_n)$ we have:

$$dist(f(t), f(t')) \leq c\, dist(t, t')$$

Lemma 59 The mapping $run^0_{\mathcal{A}} : \mathrm{Trees}(\Sigma_n) \to \mathrm{Trees}(\Sigma_n)$ is contracting.

A metric space is *complete* if every Cauchy sequence of elements of the space has a limit. It is not difficult to check that Trees_Σ with *dist* metric is complete.By Banach theorem the function $run^0_{\mathcal{A}}$ has a unique fixpoint. We use this fixpoint to prove the theorem.

Proof (of Theorem 56)
For the first statement, suppose that N_n is recognized by an alternating Σ^a_n automaton $\mathcal{A}$. Take the fixpoint t of the $run^0_{\mathcal{A}}$ mapping. We have:

$$t \in N_n \quad \text{iff} \quad t \in L(\mathcal{A}) \quad \text{iff} \quad run^0_{\mathcal{A}}(t) \in M_n \quad \text{iff} \quad t \in M_n$$

The second equivalence follows from Lemma 57. The third holds just because $run^0_{\mathcal{A}}(t) = t$.

For the second statement, suppose that M_n is recognized by an alternating Π^a_n automaton $\mathcal{A}$. Language N_n is the complement of M_n. Hence, it is recognized by $\overline{\mathcal{A}}$, the dual of $\mathcal{A}$ (see Lemma 27). But, $\overline{\mathcal{A}}$ is a Σ^a_n automaton. Contradiction with the previous paragraph. □

The languages M_n, N_n showing strictness of the hierarchy use alphabets depending on n. It is possible to show the strictness for the languages over a two element alphabet. Big alphabets can be coded by sequences over two letters. Then, one can show that if the coding of N_n were accepted by a Σ^a_n alternating automaton then N_n also would be accepted by an automaton of this kind.

Relation to the MSOL hierarchy

The MSOL hierarchy over binary trees also collapses but on a higher level than over words.

Lemma 60 Every MSOL formula over trees is equivalent to a Σ_2^M formula.

Proof
It is enough to show that for every nondeterministic tree automaton $\mathcal{A}$ there is a Σ_2^M formula $\varphi_{\mathcal{A}}$ expressing the fact that $\mathcal{A}$ has an accepting run. The formula is of the form

$$\exists X_1 \dots X_n . \forall P.\ Run(X_1, \dots, X_n) \wedge (Path(P) \Rightarrow Accepting(X_1, \dots, X_n, P))$$

So we use existential quantification to guess a run and then universal quantification to quantify over paths of the tree. The facts that $X_1, \dots, X_n$ define a run and that that the run is accepting on a path P can be expressed in first-order logic. □

There is a nice correspondence between lower levels of the MSOL hierarchy and the fixpoint alternation hierarchy.

Theorem 61
Σ_1^M properties are exactly Π_2^μ properties. $\Sigma_0^M = \Pi_0^M$ properties are exactly the properties expressible in the alternation free fragment of the μ-calculus. Moreover, $\Sigma_0^M = \Sigma_1^M \cap \Pi_1^M$.

Recall that Π_2^μ tree languages are precisely the languages definable by Büchi tree automata. By Fact 17 we know that Σ_1^M is strictly smaller than Σ_2^M. As the set of Σ_2^M tree languages is closed under complement, also Π_1^M is strictly smaller that Σ_2^M. For the similar reasons $\Sigma_0^M = \Pi_0^M$ is strictly included in Σ_1^M and Π_1^M.

6.5 The case of graphs

The hierarchies over graphs are also very interesting. There are still many open questions in this setting. The first important difference is that finiteness is not definable is MSOL over graphs. So the standard definition of the hierarchy changes. Now the $\Sigma_0^M = \Pi_0^M$ level consists of first-order formulas. The rest of the monadic hierarchy is defined the same way as before. This is the way we will understand the monadic hierarchy in this section.

The important difference with the previous cases is that here the monadic hierarchy is infinite over graphs [34].

Theorem 62 (Matz, Schweikardt, Thomas)
The MSOL hierarchy over finite graphs is strict.

The strictness of the fixpoint alternation hierarchy over binary trees implies the strictness of this hierarchy over (finite) graphs. A natural question

to ask is what are the relations between the hierarchies. It turns out that we cannot hope for a complete correspondence [29].

Fact 63 There are bisimulation invariant graph properties that are arbitrary high in the fixpoint alternation hierarchy but on Σ_2^M level of the monadic hierarchy.

It is an interesting open question whether the monadic hierarchy is strict for bisimulation invariant properties. In other words, whether one can translate the μ-calculus into some fixed level of the monadic hierarchy.

A whole new spectrum of problems opens when one considers a modification of the monadic hierarchy called *closed monadic hierarchy*. Roughly, in the closed monadic hierarchy we can use first-order quantifiers for free. See [1] for the introduction to this hierarchy. In [6] the closed monadic hierarchy over trees is discussed.

7 Guarded logic

The goal of this section is to extend the results on MSOL and the μ-calculus to a more general relational setting. Consider a translation of modal logic (i.e. the μ-calculus without fixpoints) to first-order logic:

$$\begin{aligned} Z &\rightsquigarrow Z(x) \\ P_a &\rightsquigarrow P_a(x) \\ \langle\cdot\rangle\alpha &\rightsquigarrow \exists y.\ E(x,y) \wedge \varphi_\alpha(y) \\ \alpha \vee \beta &\rightsquigarrow \varphi_\alpha(x) \vee \varphi_\beta(x) \end{aligned}$$

The translation gives for a modal formula α a formula $\varphi_\alpha(x)$ with one free variable x, s.t, for every labelled graph $\mathcal{M} = \langle V, E, (P_a)_{a\in\Sigma}\rangle$ we have $\| \alpha \|_V^{\mathcal{M}} = \{s : \mathcal{M}, V \vDash \varphi_\alpha(s)\}$

The set of formulas obtained from the translation is called the *modal fragment* of FOL. These formulas have several special properties. They use only monadic relations except for the edge relation. They use only two variables. The quantification pattern is very specific. First-order logic is undecidable, but the modal fragment is decidable in exponential time (because the μ-calculus is). The question is: "what makes the modal fragment so special?"

Here we show that it is the quantification patterns that are important. The idea of having the same quantification pattern as in the modal fragment is captured by the definition below. The main extension is that we put no restrictions on arity of relations. In the definition we use bold letters for vectors of variables.

Definition 64 The *guarded fragment* GF [4] of first-order logic is defined inductively as follows:

1. Every relational atomic formula belongs to GF.
2. GF is closed under propositional connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$.
3. If $\boldsymbol{x}, \boldsymbol{y}$ are tuples of variables, $\alpha(\boldsymbol{x}, \boldsymbol{y})$ is a positive atomic formula and $\psi(\boldsymbol{x}, \boldsymbol{y})$ is a formula in GF such that $\text{free}(\psi) \subseteq \text{free}(\alpha) = \boldsymbol{x} \cup \boldsymbol{y}$, then the formulas

$$\begin{aligned} &\exists \boldsymbol{y}(\alpha(\boldsymbol{x}, \boldsymbol{y}) \wedge \psi(\boldsymbol{x}, \boldsymbol{y})) \\ &\forall \boldsymbol{y}(\alpha(\boldsymbol{x}, \boldsymbol{y}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y})) \end{aligned}$$

belong to GF.

Here free(ψ) denotes the set of free variables of ψ. An atom $\alpha(\boldsymbol{x}, \boldsymbol{y})$ that relativizes a quantifier as in rule (3) is the *guard* of the quantifier. Notice that the guard must contain *all* the free variables of the formula in the scope of the quantifier.

Note that first-order quantification over individual free variable is always admissible in GF, since singletons are guarded:

$$\exists x.\ \varphi(x) \ \equiv\ \exists x.\ x = x \wedge \varphi(x)$$

The important point here is that $\varphi(x)$ cannot have any other free variable but x.

Clearly all modal formulas are translatable to GF. But there are other formulas in GF too. For example backwards modalities are expressible in GF:

$$\langle\ \rangle^{-1}\alpha \ \rightsquigarrow\ \exists y.\ R(y,x) \wedge \varphi_\alpha(y)$$

We can also have very strange modalities like

$$\langle\circlearrowleft\rangle\alpha \ \rightsquigarrow\ \exists y.\ R(x,y) \wedge R(y,x) \wedge \varphi_\alpha(y)$$

GF seems not to be able to express all of temporal logic over $(\mathbb{N}, \leq)$. Indeed, the straightforward translation of (ψ **until** ϕ) into first-order logic

$$\exists y(x \leq y \wedge \phi(y) \wedge \forall z((x \leq z \wedge z < y) \rightarrow \psi(z))$$

is not guarded in the sense of Definition 64. However, the quantifier $\forall z$ in this formula is guarded in a weaker sense, which lead van Benthem [8] to the following generalization of GF.

Definition 65 The *loosely guarded fragment* LGF is defined similarly to GF, but the quantifier-rule is relaxed as follows:

(3)' If $\psi(\boldsymbol{x}, \boldsymbol{y})$ is in LGF, and $\alpha(\boldsymbol{x}, \boldsymbol{y}) = \alpha_1 \wedge \cdots \wedge \alpha_m$ is a conjunction of atoms, then

$$\exists \boldsymbol{y}((\alpha_1 \wedge \cdots \wedge \alpha_m) \wedge \psi(\boldsymbol{x}, \boldsymbol{y}))$$
$$\forall \boldsymbol{y}((\alpha_1 \wedge \cdots \wedge \alpha_m) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y}))$$

belong to LGF, provided that free(ψ) $\subseteq$ free(α) $= \boldsymbol{x} \cup \boldsymbol{y}$ and for every quantified variable $y \in \boldsymbol{y}$ and every variable $z \in \boldsymbol{x} \cup \boldsymbol{y}$ there is at least one atom α_j that contains both y and z.

In the translation of (ψ **until** ϕ) described above, the quantifier $\forall z$ is loosely guarded by $(x \leq z \wedge z < y)$ since z coexists with both x and y in some conjunct of the guard. On the other side, the transitivity axiom

$$\forall xyz(Exy \wedge Eyz \rightarrow Exz)$$

is not in LGF. The conjunction $Exy \wedge Eyz$ is not a proper guard of $\forall xyz$ since x and z do not coexist in any conjunct. Indeed, adding transitivity statement to GF makes the fragment undecidable [21].

Notation. We will use the notation $(\exists \boldsymbol{y} \,.\, \alpha)$ and $(\forall \boldsymbol{y} \,.\, \alpha)$ for relativized quantifiers, i.e., we write guarded formulas in the form $(\exists \boldsymbol{y} \,.\, \alpha)\psi(\boldsymbol{x}, \boldsymbol{y})$ and $(\forall \boldsymbol{y} \,.\, \alpha)\psi(\boldsymbol{x}, \boldsymbol{y})$. When this notation is used, then it is always understood that α is indeed a proper guard as specified by condition (3) or (3)'.

The following theorem says that LGF is not much more difficult than modal logic. The theorem refers to the width of a formula. This is a maximal number of free variables in any subformula of the formula.

Theorem 66 (Grädel [21])
The satisfiability problem for LGF is 2EXPTIME*-complete. It is* EXPTIME*-complete for formulas of bounded width.*

The reason for this doubly exponential complexity is just the fact that the formulas have unbounded width. Given that even a single predicate of arity n over a domain of just two elements leads to 2^{2^n} possible types already on the atomic level, the double exponential lower complexity bound is hardly a surprise. When the width is bounded, the complexity of LGF is just slightly bigger than that of the modal logic (which is PSPACE-complete).

The next step is to add fixpoints to LGF without loosing decidability. We follow [23].

Definition 67 The guarded fixpoint logics μGF and μLGF are obtained by adding to GF and LGF, respectively, the following rules for constructing fixed-point formulas:

> Let W be a k-ary relation variable and let $\boldsymbol{x}$ be a k-tuple of distinct variables. Further, let $\psi(W, \boldsymbol{x})$ be a guarded formula where W appears only positively and not in guards. Moreover we require that all the free variables of $\psi(W, \boldsymbol{x})$ are contained in $\boldsymbol{x}$. For such a formula $\psi(W, \boldsymbol{x})$ we can build a formula
>
> $$[\mathsf{LFP}\, W\boldsymbol{x} \,.\, \psi](\boldsymbol{x})$$

The part in square brackets, i.e. $[\mathsf{LFP}\, W\boldsymbol{x} \,.\, \psi]$ is called *fixpoint predicate*.

The semantics of fixpoint formulas is the usual one: Given a structure $\mathcal{M}$ and a valuation V for the free second-order variables in ψ, other than W, the formula $\psi(W, \boldsymbol{x})$ defines an operator on k-ary relations $W \subseteq M^k$, namely

$$\psi^{\mathcal{M},V}(W) := \{\boldsymbol{a} \in M^k : \mathcal{M}, V \models \psi(W, \boldsymbol{a})\}.$$

Since W occurs only positively in ψ, this operator is monotone (i.e., $W \subseteq W'$ implies $\psi^{\mathcal{M},V}(W) \subseteq \psi^{\mathcal{M},V}(W')$) and therefore has a least fixed point $\mathrm{LFP}(\psi^{\mathcal{M},V})$. Now, the semantics of least fixed point formulas is defined by

$$\mathcal{M}, V \models [\mathsf{LFP}\, W\boldsymbol{x}\,.\,\psi(W, \boldsymbol{x})](\boldsymbol{a}) \quad \text{iff} \quad \boldsymbol{a} \in \mathrm{LFP}(\psi^{\mathcal{M},V})$$

Similarly as in the μ-calculus the greatest fixpoint is definable by:

$$[\mathsf{GFP}\, W\boldsymbol{x}\,.\,\psi(W, \boldsymbol{x})](\boldsymbol{a}) \;\equiv\; \neg[\mathsf{LFP}\, W\boldsymbol{x}\,.\,\neg\psi(\neg W, \boldsymbol{x})](\boldsymbol{a})$$

Observe that we do not allow to use fixed point predicates in guards. Otherwise guarded quantification would be as powerful as unrestricted quantification. Indeed, for every k, we can define the universally true k-ary relation by the fixed point predicate $[\mathsf{GFP}\, U^k x_1 \cdots x_k\,.\,\mathit{true}]$ (where *true* stands for any tautology). Using these predicates as guards one could obtain unrestricted quantification. Also the use of the fixed point variable W as a guard inside the formula defining it as a least or greatest fixed point, or the use of additional first-order variables as parameters in fixed point formulas would lead to an undecidable logic.

Despite all of these restrictions on constructing fixpoints, we can still translate the μ-calculus to GF. We extend the translation of modal logic to GF given at the beginning of the section:

$$\mu Z.\alpha(Z) \;\rightsquigarrow\; [\mathsf{LFP}\, Z(y).\, \varphi_\alpha(y)](x)$$

Contrary to both GF and the μ-calculus, guarded fixed-point logic does not have the finite model property. An infinity axiom is a satisfiable sentence that does not have a finite model.

Proposition 68 Guarded least fixpoint logic (even with only two variables, without nested fixed points and without equality) contains infinity axioms.

Proof
Consider the formulas

$$\exists xy.\; Fxy$$
$$(\forall xy\,.\,Fxy)\exists xFyx$$
$$(\forall xy\,.\,Fxy)[\mathsf{LFP}\, Wx\,.\,(\forall y\,.\;Fyx)Wy](x)$$

The first two formulas say that a model should contain an infinite F-path and the third formula says that F is well-founded, thus, in particular, acyclic. Therefore every model of these formulas is infinite. On the other side, the formulas are clearly satisfiable, for instance by $(\mathbb{N}, <)$. □

Even though we can express more than in the μ-calculus, the complexity of μLGF stays essentially the same.

Theorem 69
The satisfiability problem for μLGF is 2EXPTIME*-complete. It is* EXPTIME*-complete for formulas of bounded width.*

Note that this is the same complexity as for guarded first-order sentences, so, if theoretical complexity is concerned, we do not pay any penalty for fixpoints. Fortunately, in most practical applications, formulas have only bounded width. In particular, for a fixed finite vocabulary all guarded formulas have bounded width. For example, the translation of the μ-calculus into μGF uses at most binary relations and leads to formulas of width two.

Knowing the complexity of guarded fragments it would be nice to understand the expressive power of the logic. Theorem 41 characterizes the expressive power of the μ-calculus by MSOL properties invariant under bisimulation. Of course we cannot directly compare μGF with MSOL as the signatures of the logics are different (μGF contains relations of higher arity). We rather define a fragment of second-order logic which we call guarded fragment or GSO for short. Then we define guarded bisimulation which will relate tuples of elements of two structures and not just single elements as bisimulation did. Finally, we show that GSO sentences that are guarded bisimulation invariant are exactly μGF sentences. The results summarized below come from [22].

Definition 70 Let $\mathcal{M}$ be a structure over a signature Sig and with the universe M. A tuple $(m_1, \ldots, m_k)$ is *guarded* iff there is a relation R and elements $m'_1, \ldots, m'_l$ such that $R(m'_1, \ldots, m'_l)$ holds in $\mathcal{M}$ and $\{m_1, \ldots, m_k\} \subseteq \{m'_1, \ldots, m'_l\}$. A relation $S \subseteq M^n$ is guarded if it consists of guarded tuples.

Definition 71 *Guarded second-order logic* (GSO), is an extension of first-order logic with second-order quantifiers ranging over guarded relations.

Lemma 72 SO is strictly more expressive than GSO. In particular GSO collapses to MSOL over words in the case when words are represented as structures with a successor relation instead of linear ordering.

In order to define guarded bisimulation it will be useful to have a notion of *partial isomorphism*. The bisimulation relation relates single elements. These elements are required to have the same labeling. Now we want to relate tuples of elements and we want to say that the tuples satisfy the same relations. This is precisely what partial isomorphism is saying. Formally, a partial isomorphism between structures $\mathcal{M}_1$ and $\mathcal{M}_2$ is a bijective function $f : X \to Y$ for some $X \subseteq \mathcal{M}_1$ and $Y \subseteq \mathcal{M}_2$. It must satisfy the condition that for every relation symbol R and a tuple of elements $a_1, \ldots, a_k \in X$: relation $R^{\mathcal{M}_1}(a_1, \ldots, a_k)$ holds iff $R^{\mathcal{M}_2}(f(a_1), \ldots, f(a_k))$ holds.

Definition 73 *Guarded bisimulation* between two structures $\mathcal{M}_1$, $\mathcal{M}_2$ of signature Sig is a non-empty set I of partial isomorphisms from $\mathcal{M}_1$ to $\mathcal{M}_2$ such that for every $f : X \to Y$ in I the following conditions hold:

- for every guarded set $X' \subseteq \mathcal{M}_1$ there is in I a partial isomorphism $g : X' \to Y'$ such that f and g agree on $X \cap X'$.
- for every guarded set $Y' \subseteq \mathcal{M}_2$ there is in I a partial isomorphism $g : X' \to Y'$ such that f^{-1} and g^{-1} agree on $Y \cap Y'$.

Two tuples of elements $(a_1, \ldots, a_n) \in \mathcal{M}_1$ and $(b_1, \ldots, b_n) \in \mathcal{M}_2$ are *guarded bisimilar* if there is $f \in I$ mapping a_i to b_i for all $i = 1, \ldots, n$.

Definition 74 A formula $\varphi(x_1, \ldots, x_n)$ is *invariant under bisimulation* if it cannot distinguish between guarded bisimilar tuples, i.e., if $\mathcal{M}_1 \models \varphi(a_1, \ldots, a_n)$ and $(a_1, \ldots, a_n)$ is guarded bisimilar to a tuple $(b_1, \ldots, b_n) \in \mathcal{M}_2$ then $\mathcal{M}_2 \models \varphi(b_1, \ldots, b_n)$.

The following theorem from [22] ties together the expressive power of GSO and μGF.

Theorem 75 (Grädel, Hirsch, Otto)
Every formula of GSO invariant under guarded bisimulation is equivalent to a μGF formula.

8 Traces

Infinite words, which are linear orders on *events*, are often used to model executions of systems. Infinite *traces*, which are partial orders on events, can be used to model concurrent systems when we do not want to put some arbitrary ordering on actions occurring concurrently. The idea is that if we have two actions, say a and b, occurring concurrently then we do not want to model this neither as a word ab nor as ba. A more faithful representation of what happened is a partial order with two events a, b and no ordering between them.

A *trace alphabet* is a pair (Σ, D) where Σ is a finite set of *actions* (i.e. letters) and $D \subseteq \Sigma \times \Sigma$ is a reflexive and symmetric *dependence relation*. Intuitively if $(a, b) \in D$ then a and b share some resource, so their occurrences should be ordered. On the other hand, if $(a, b) \notin D$ then there is no reason to order occurrences of these actions.

A *trace* or *dependence graph* is a labelled graph

$$G = \langle E, R \subseteq E \times E, \lambda : E \to \Sigma \rangle$$

such that R is a partial order on E and the following conditions are satisfied:

(T1) $\forall e \in E.$ $\quad \{e' : R(e', e)\}$ is a finite set.
(T2) $\forall e, e' \in E.$ $\quad (\lambda(e), \lambda(e')) \in D \Rightarrow R(e, e') \vee R(e', e).$
(T3) $\forall e, e' \in E.$ $\quad R(e, e') \Rightarrow (\lambda(e), \lambda(e')) \in D \vee$
$\qquad\qquad\qquad\qquad\qquad\qquad \exists e''.\ R(e, e'') \wedge R(e'', e') \wedge e \neq e'' \neq e'.$

The nodes of a dependence graph are called *events*. An *a-event* is an event $e \in E$ which is labelled by a, i.e., $\lambda(e) = a$. We say that e is *before* e' iff $R(e, e')$ holds. In this case we also say that e' is *after* e.

The first condition of the definition of dependence graphs says that the past of each event (the set of the events before the event) is finite. The second one postulates that events labelled by dependent letters are ordered. The third, says that the order is induced by the order between dependent letters.

Below we describe a variation on the representation of dependence graphs. This variation will be important when defining the μ-calculus over traces.

Definition 76 A *Hasse diagram* of a trace $G = \langle E, R, \lambda \rangle$ is a labelled graph $\langle E, R_H, \lambda \rangle$ where R_H is the smallest relation needed to determine R, i.e., the reflexive and transitive closure of R_H is R, and if $R_H(e, e')$ holds then there is no e'' different from e and e' such that $R_H(e, e'')$ and $R_H(e'', e')$ hold.

Büchi theorem tells us that for the class of finite or infinite words (dependence graphs for alphabets where all the letters are mutually dependent) the properties definable by MSOL are exactly the languages recognizable by automata. This characterization carries through to traces with an appropriate modification of the notion of automata.

MSOL logic over traces is just MSOL logic over dependence graphs considered as labelled graphs. Observe that a Hasse diagram of a trace is MSOL definable in a dependence graph, i.e., there is a MSOL formula defining R_H from R. Also dependence graph is MSOL definable in a Hasse diagram of a trace. Hence, MSOL definability over dependence graphs and over Hasse diagrams are the same thing.

Another way of defining traces is to consider linearizations of traces. A *linearization of a trace* $\langle E, R, \lambda\rangle$ is an injective function $f : E \to \mathbb{N}$ such that if $R(e, e')$ holds then $f(e) \leq f(e')$. We can identify a linearization f with an ω-word $\lambda(f^{-1}(0))\lambda(f^{-1}(1))\ldots$. It is easy to see that an infinite word over Σ defines the unique trace of which it is a linearization. This defines a trace equivalence over ω-words: $w \sim w'$ if they define the same trace. A language L is *trace consistent* if whenever $w \in L$ and $w \sim w'$ then $w' \in L$. Hence, a way to define a trace language is to define the set of its linearizations, i.e., a trace consistent language.

Yet another way to describe a set of traces is to use automata working directly on traces. It will be easier here to describe their behaviour on linearizations of traces. From their definition it will be clear that they only accept trace consistent sets of ω-words.

Suppose we have some number k off *processes* and consider function $loc : \Sigma \to \mathcal{P}(\{1, \ldots, k\})$ assigning to each letter a set of processes. Intuitively these are the processes that are needed to read input a. The distribution of letters should reflect our dependence alphabet (Σ, D) in a sense that $(a, b) \in D$ iff $loc(a) \cap loc(b) \neq \emptyset$. Intuitively, the two letters are dependent if for reading them some common process is needed. An *asynchronous automaton* is a tuple:

$$\mathcal{A} = \langle Q_1, \ldots, Q_k, \Sigma, q^0, (\delta_a)_{a\in\Sigma}, \mathcal{F}\rangle$$

satisfying the following conditions

- the global set space $Q = \prod_{i=1}^{k} Q_i$ is the product of local finite states spaces Q_i,
- $q^0 \in Q$,
- for each $a \in \Sigma$, relation $\delta_a \subseteq Q \times Q$ such that if

$$((q_1, \ldots, q_k), a, (q'_1, \ldots, q'_k)) \in \delta_a$$

and $i \notin loc(a)$ then $q_i = q_i'$ and for every $\widehat{q} \in Q_i$:

$$((q_1, \ldots, q_{i-1}, \widehat{q}, \ldots, q_k), a, (q_1', \ldots, \widehat{q}, \ldots, q_k')) \in \delta_a$$

- $\mathcal{F} = \{(F_1^\omega, F_1), \ldots, (F_k^\omega, F_k)\} \subseteq \mathcal{P}(Q) \times \mathcal{P}(Q)$ defines the acceptance condition.

Hence the transition relation for a letter a is allowed to examine and change only the components of the automaton that are in $loc(a)$. A run of $\mathcal{A}$ on a ω-word $w \in \Sigma^\omega$ is defined as for ordinary finite automata over states Q. For a run $r : \mathbb{N} \to Q$ and $p \in \{1, \ldots, k\}$ we define $\mathrm{Inf}_p(r) = \mathrm{Inf}(r) \cap Q_p$ to be the set of sates form Q_p that appear infinitely often in the run. A run r is *accepting* if:

- $\mathrm{Inf}_p(r) \cap F_p^\omega \neq \emptyset$ for every p such that a letter b with $p \in loc(b)$ appears infinitely often in w, and
- $\mathrm{Inf}_p(r) \cap F_p \neq \emptyset$ for every other p.

So the acceptance condition is a Büchi condition but it can also tell whether a process p was active infinitely often or not. Observe that if no letter b with $p \in loc(b)$ appears infinitely often in w then in any run on w the p-th component of the state vector is ultimately constant.

A trace is *accepted* by $\mathcal{A}$ if one of its linearizations is accepted by $\mathcal{A}$. The *language recognized by* $\mathcal{A}$ is the set of traces accepted by $\mathcal{A}$.

The following theorem summarizing many results on traces can be found in [16, 20, 40].

Theorem 77
Fix a trace alphabet. For a set L of traces the following are equivalent:

- *L is definable by a MSOL formula.*
- *L is recognizable by an asynchronous automaton.*
- *The set of linearizations of traces in L is a recognizable language of infinite words.*

In the case of words we had also a characterization of regular languages by the μ-calculus. As traces are just labelled graphs we can evaluate the μ-calculus directly on them, but we have a small problem when we want to make it precise what it means that a set of traces is definable by a μ-calculus formula. In the case of words we said that these are the words where the formula holds on the first position. In the case of traces we may have several

minimal events. To overcome this problem we assume that in our traces we have always the least element $\perp$ labelled with a special letter also denoted by $\perp$. This letter is dependent on every other letter in Σ.

If G is a trace that has the least event $\perp$ and α is a μ-calculus sentence then we write $G \vDash \alpha$ to mean that $G, \perp \vDash \alpha$. Sentence α defines the set of traces $\{G : G \vDash \alpha\}$.

The μ-calculus over traces does not have sufficient expressive power. Let us see an example showing that there are even first-order definable properties of traces which are not expressible in the μ-calculus.

We claim that no μ-calculus sentence can distinguish between the following two Hasse diagrams of traces presented in Figure 3. In the left graph

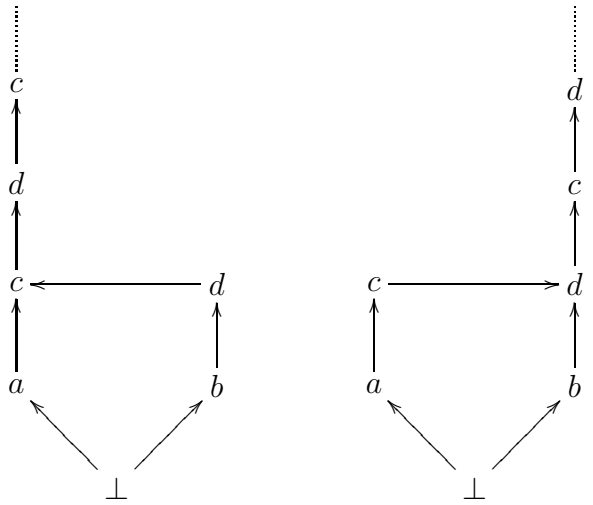


Figure 3: Indistinguishable traces

the dots stand for the sequence $(dc)^\omega$ and in the right graph for $(cd)^\omega$. In this example the trace alphabet $(\{\perp, a, b, c, d\}, D)$ where D is the smallest symmetric and reflexive relation containing the pairs $\{(a, c), (b, d), (c, d)\} \cup \{\perp\} \times \{a, b, c, d\}$. The two Hasse diagrams are bisimilar, hence indistinguishable by μ-calculus formulas. Still, a first order formula saying that the first d is before the first c is satisfied in the left graph but not in the right. The figure shows Hasse diagrams, but also dependence graphs of these two traces are bisimilar.

The above example indicates that some extension of the μ-calculus is needed. To see such an extension consider a *concurrency relation* in a trace G defined by $co(e, e')$ if neither $R(e, e')$ nor $R(e', e)$ holds. Then we can extend the μ-calculus with a proposition $co(a)$ for every $a \in \Sigma$. The semantics is

- $G, e \vDash co(a)$ if there is e' such that $co(e, e')$ and $\lambda(e') = a$.

Let μ^{co} be the extension of the μ-calculus with $co(a)$ propositions.

In the μ^{co}-calculus we can distinguish the two traces from Figure 3. The formula $\langle\cdot\rangle(P_a \wedge co(d))$ says that there is a successor of the least event, this successor is labelled by a and has a concurrent event labelled by d. The formula is true in the left graph but not in the right. The theorem below [55] says that this is not a coincident.

Theorem 78 (Expressive completeness)
The μ^{co} calculus is equivalent in expressive power to MSOL over traces.

Moreover the satisfiability problem for the μ^{co}-calculus is relatively easy.

Theorem 79
The satisfiability problem for the μ^{co}-calculus is PSPACE-complete.

9 Real-time

In real-time systems we have an interaction between continuous behaviour of some physical components and finite, discrete control. A standard example of such a system is a steam boiler, that needs to keep water warm by switching a heater on and off. To describe behaviour of such a device it seems natural to model time flow by nonnegative real numbers. So, now we will be interested in properties of *signals* which are functions from reals to $\{0, 1\}$. Equivalently a signal is a monadic predicate $P \subseteq \mathbb{R}^+$.

Intuitively, not all predicates can model a behaviour of a physical device. It is implausible to expect a device which is on when the time is a rational number and which is off when the time is an irrational number. To exclude such predicates one sometimes postulates *non-Zeno* assumption. A predicate is non-Zeno if on every bounded interval it changes the value only finitely many times. An equivalent, convenient definition is the following:

Definition 80 A predicate $P \subseteq \mathbb{R}^+$ is *non-Zeno* if there is an unbounded sequence $0 = \tau_0 < \tau_1 < \tau_2 < \cdots$ of reals such that for every $i \in \mathbb{N}$: either $(\tau_i, \tau_{i+1}) \subseteq P$ or $(\tau_i, \tau_{i+1}) \cap P = \emptyset$. We write $\mathcal{P}_{NZ}(\mathbb{R}^+)$ for the set of non-Zeno predicates on $\mathbb{R}^+$.

In this section we will describe logics and automata for real-time properties. As we will see the situation here is much less satisfactory than in the cases we have discussed till now.

9.1 FOL and MSOL over reals

The signature of all the logics will be the same. It consists of a binary predicate symbol $\leq$ and unary predicate symbols $P_1, P_2, \ldots$ We will consider three classes of models:

- $\mathcal{N}$ is the class of models of the form $\mathcal{M} = \langle \mathbb{N}, \mathcal{P}(\mathbb{N}), \leq, P_1^{\mathcal{M}}, \ldots \rangle$
- $\mathcal{R}$ is the class of models of the form $\mathcal{M} = \langle \mathbb{R}^+, \mathcal{P}(\mathbb{R}^+), \leq, P_1^{\mathcal{M}}, \ldots \rangle$
- $\mathcal{R}_{NZ}$ is the class of models of the form $\mathcal{M} = \langle \mathbb{R}^+, \mathcal{P}_{NZ}(\mathbb{R}^+), \leq, P_1^{\mathcal{M}}, \ldots \rangle$

The second element of the structure defines the range of second order variables and interpretation of predicates. In particular in the case of $\mathcal{R}_{NZ}$ all predicates must be non-Zeno. In all three cases $\leq$ is interpreted as the standard relation on numbers.

The semantics of first-order and monadic second-order logics over these classes of structures is standard. In particular the second component of

the structures plays no role in the semantics of first-order logic. In case of MSOL the range of second order variables is restricted to the elements of the second component of the structures. So, in case of $\mathcal{R}_{NZ}$ it means that we can quantify only over non-Zeno subsets of $\mathbb{R}^+$.

As we have seen in Section 3, MSOL over $\mathcal{N}$ is decidable and has characterizations in terms of automata and fixpoint calculi. Interestingly, the decidability of $\mathcal{R}_{NZ}$ can be reduced to the decidability of MSOL theory of the binary tree [44]. This gives

Theorem 81 (Rabin)
MSOL theory of $\mathcal{R}_{NZ}$ is decidable.

The picture for the class $\mathcal{R}$ is different (cf. [47, 13])

Theorem 82 (Shelah)
MSOL theory of $\mathcal{R}$ is undecidable. FOL theory of $\mathcal{R}$ is decidable.

Even in MSOL over reals we cannot express properties like "after no more than 5 units of time the heater switches off". To express such a property it seems to be a good idea to add $+1$ predicate to the classes of models considered above. So let $\mathcal{N}^{+1}$, $\mathcal{R}^{+1}$ and $\mathcal{R}^{+1}_{NZ}$ stand for the classes of models extended with a binary predicate $+1(x, y)$ saying that $x + 1 = y$.

Unfortunately, it is not difficult to see that an unrestricted use of $+1$ predicate makes all the considered logics over reals undecidable.

Fact 83 FOL over $\mathcal{R}^{+1}$ or $\mathcal{R}^{+1}_{NZ}$ is undecidable.

We finish this subsection with a proposal of limiting the use of $+1$ predicate so that the decidability is regained. The logic L_1 [28] is an extension of FOL without $+1$ predicates by the following construction:

> If $\varphi(x)$ is a L_1 formula and x is the only free variable in $\varphi(x)$ then
>
> $$(\exists x)^{\leq x_0+1}_{>x_0}\ \varphi(x) \qquad \text{and} \qquad (\exists x)^{\leq x_0}_{>x_0-1}\ \varphi(x)$$
>
> are formulas of L_1. The variable x_0 is the only free variable in these formulas

The semantics is as the syntax suggests:

> $\mathcal{M}, V \vDash (\exists x)^{\leq x_0+1}_{>x_0}\ \varphi(x)$ iff there is t such that $\mathcal{M}, V[t/x] \vDash \varphi(x)$ and $V(x_0) < t < V(x_0) + 1$.

We can also define ML_1 as an extension of L_1 with monadic quantification.

Theorem 84 (Hirshfeld, Rabinovich)
Both L_1 and ML_1 are decidable over $\mathcal{R}_{NZ}^{+1}$.

The complexity of FOL over $\mathcal{R}$, and hence also of L_1, is nonelementary. The bound follows, as usual, from the complexity of FOL over finite words. In [27] a decidable extension of L_1 is presented.

9.2 Real-time automata

Here we want to present an automata model for specifying real-time properties. Unfortunately the model is not closed under complement. We will also discuss some restrictions of this model.

Models over $\mathbb{N}$ are represented by ω-words. Models over $\mathbb{R}^+$ are represented by *timed words*. Unfortunately the representation is not as good as in the case of $\mathbb{N}$.

A *timed word* over an alphabet Σ is an infinite sequence $(a_0, \tau_0), (a_1, \tau_1), \ldots$ over $\Sigma \times \mathbb{R}^+$ such that , $\tau_i \leq \tau_{i+1}$ for every $i \in \mathbb{N}$. The idea is that the first components describe the events that occur and the second the times at which they occur.

There is no hope to have one-to-one correspondence between timed words and models from $\mathcal{R}$. We have a better chance with non-Zeno models from $\mathcal{R}_{NZ}$. There are several ways of coding a model $\mathcal{M} \in \mathcal{R}_{NZ}$ as a timed word. To have one-to-one correspondence between non-Zeno models and timed words it is necessary to put some restrictions on timed words. One of them is a progress requirement which says that for every $t \in \mathbb{R}^+$ there should be i with $\tau_i > t$. We will not discuss such a coding in detail because there is no standard coding and a coding is not important for the results on timed automata that we are going to present.

Let $\mathcal{Z}$ be a set of clocks (variables ranging over $\mathbb{R}^+$). Consider clock constraints given by the grammar:

$$CC(\mathcal{Z}) := x \leq c \mid c \leq x \mid c < x \mid x < c \mid CC(\mathcal{Z}) \wedge CC(\mathcal{Z})$$

where $x \in \mathcal{Z}$ is a clock and $c \in \mathbb{N}$ is a constant. A *clock interpretation* is a function $V : \mathcal{Z} \to \mathbb{R}^+$. The satisfaction relation $V \vDash \alpha$ for a clock constraint α is defined in a expected way. For a set of clocks $\mathcal{Y} \subseteq \mathcal{Z}$, let $V[\mathcal{Y} := 0]$ be the clock interpretation which is identical to V on clocks not in $\mathcal{Y}$ and equal to 0 on clocks in $\mathcal{Y}$. For every $t \in \mathbb{R}^+$, the interpretation $V + t$ gives the value $V(x) + t$ for every clock x.

A *timed automaton* is a tuple:

$$\mathcal{A} = \langle Q, \Sigma, \mathcal{Z}, q^0, \delta \subseteq Q \times \Sigma \times CC(\mathcal{Z}) \times \mathcal{P}(\mathcal{Z}) \times Q, Acc \subseteq Q^\omega \rangle$$

where Q is a finite set of states, Σ is a finite alphabet, $\mathcal{Z}$ is a finite set of clocks, q^0 is the initial state, and Acc is the set of accepting runs as in the case of ordinary ω-word automata. The transition relation is slightly complicated. Additionally to the usual components it has a clock constraint to be satisfied and a set of clocks to be reset.

A *configuration* of a timed automaton is a pair (q, V) consisting of a state of $\mathcal{A}$ and a valuation of the clocks. For a fixed automaton we define three relations on configurations:

- $(q, V) \xrightarrow{t} (q, V')$ for $t \in \mathbb{R}^+$ and $V' = V + t$,
- $(q, V) \xrightarrow{a} (q', V')$ if there is $(q, a, \alpha, \mathcal{Y}, q') \in \delta$ with $V \vDash \alpha$ and $V' = V[\mathcal{Y} := 0]$.
- $(q, V) \xrightarrow[t]{a} (q', V')$ if $(q, V) \xrightarrow{t} (q, V'')$ and $(q, V'') \xrightarrow{a} (q', V')$ for some V''.

A run of $\mathcal{A}$ on a timed word $(a_0, \tau_0), (a_1, \tau_1), \ldots$ is a sequence:

$$(q_0, V_0) \xrightarrow[t_0]{a_0} (q_1, V_1) \xrightarrow[t_1]{a_1} (q_2, V_2) \xrightarrow[t_2]{a_2} \cdots$$

such that: $q_0 = q^0$ is the initial state; V_0 is the initial valuation assigning 0 to every clock; $t_0 = \tau_0$ and $t_{i+1} = \tau_{i+1} - \tau_i$. A run is *accepting* if the sequence of states from the run is in Acc. The language recognized by $\mathcal{A}$ is the set of timed words accepted by $\mathcal{A}$.

Example: Consider a one letter alphabet $\Sigma = \{a\}$. Let L_1 be the language of timed words $(a, \tau_0), (a, \tau_1), \ldots$ such that $\tau_i + 1 = \tau_j$ for some i and j. So, we require that there are two occurrences of the letter with exactly one time unit difference. The language is recognized by the automaton in Figure 4. The initial state of the automaton is q_0. The acceptance condition Acc consists of all the sequences with infinitely many occurrences of q_2. $\square$

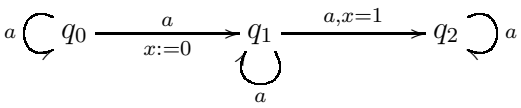


Figure 4: Timed automaton recognizing L_1.

The main result about timed automata is the decidability of the emptiness problem [2].

Theorem 85 (Alur, Dill)
The following problem is PSPACE complete: given $\mathcal{A}$ decide if $\mathcal{A}$ accepts some timed word.

Strangely enough the universality problem, i.e., whether an automaton accepts all timed words, is highly undecidable.

Theorem 86 (Alur, Dill)
The universality problem is Π_1^1*-complete.*

This suggests that there is a difficulty with complementing timed automata. Indeed we have:

Fact 87 There is no timed automaton recognizing the complement of the language L_1 from the example above.

One solution to the problem with complement is to restrict to deterministic timed automata. These are automata that from any state at any moment have at most one transition for each letter in the alphabet. Unfortunately the class of languages recognized by deterministic timed automata is not closed under projection. Deterministic automata are discussed in [2].

In [3] a notion of event-time automaton is proposed. In this model we have clocks associated with letters of the alphabet. With each letter a we have a clock x_a telling how much time elapsed since the last occurrence of a and a clock y_a telling in how much time the next a will occur. So y_a is a kind of prophecy clock. The important difference with the ordinary timed automata is that there are no explicit ways of resetting clocks. We have:

Theorem 88 (Alur, Fix, Henzinger)
Deterministic and nondeterministic versions of event-time automata have the same expressive power. Every event clock automaton is equivalent to some standard clock automaton. The emptiness problem for event clock automata is decidable in PSPACE.

Unfortunately, event-clock automata are not closed under projection. Still [26] shows a correspondence between a (hierarchical) extension of event-timed automata model and some monadic second-order logic over timed sequences.

There are numerous other extensions/modifications of timed automata model, for some recent papers see [12, 14, 9]. These variations give us better understanding of the situation for reals but they also show that we do not yet have the same set of canonical notions as in the case of words or trees.

References

[1] M. Ajtaj, R. Fagin, and L. Stockmeyer. The colsure of monadic NP. In *STOC'98*, pages 309–318, 1998.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur, L. Fix, and T. Henzinger. A determinizable class of timed automata. *Theoretical Computer Science*, 204, 1997.

[4] H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. Technical report, ILLC Research Report ML-96-03, 1996. 59 pages.

[5] A. Arnold. The mu-calculus alternation-depth hierarchy is strict on binary trees. *RAIRO–Theoretical Informatics and Applications*, 33:329–339, 1999.

[6] A. Arnold, G. Lenzi, and J. Marcinkowski. The hierarchy inside closed monadic Σ_1 collapses on the iinfinite binary tree. In *LICS'01*, 2001. To appear.

[7] A. Arnold and D. Niwiski. *The Rudiments of the Mu-Calculus*, volume 146 of *Studies in Logic*. North-Holand, 2001.

[8] J. Benthem. Dynamic bits and pieces. Technical report, University of Amsterdam, 1997. IILC research report.

[9] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In *CAV'2000*, volume 1855 of *LNCS*, pages 464–479, 2000.

[10] J. Bradfield. The modal mu-calculus alternation hierarchy is strict. *Theoretical Computer Science*, 195:133–153, 1997.

[11] J. Bradfield. Fixpoint alternation: Arithmetic, transition systems, and the binary tree. *RAIRO–Theoretical Informatics and Applications*, 33:341–356, 1999.

[12] B. Brard, V. Diekert, P. Gastin, , and A. Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2):145–182, 1998.

[13] J. Burgess and Y. Gurevich. The decision problem for linear temporal logic. *Notre Dame Journal of Formal Logic*, 26:115–128, 1985.

[14] C. Choffrut and M. Goldwurm. Timed automata with periodic clock constraints. *Journal of Automata, Languages and Combinatorics*, 5:371–404, 2000.

[15] K. J. Compton and C. W. Henson. A uniform method for proving lower bounds on the computational complexity of logical theories. *Annals of Pure and Applied Logic*, 48:1–79, 1990.

[16] W. Ebinger. Logical definability of trace languages. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 382–390. World Scientific, 1995.

[17] E. A. Emerson, C. Jutla, and A. Sistla. On model-checking for fragments of μ-calculus. In *CAV'93*, volume 697 of *LNCS*, pages 385–396, 1993.

[18] E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs. In *29th FOCS*, 1988.

[19] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. FOCS 91*, pages 368–377, 1991.

[20] P. Gastin and A. Petit. Asynchronous cellurar automata for infinite traces. In *ICALP '92*, volume 623 of *LNCS*, pages 583–594, 1992.

[21] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 1999. to appear.

[22] E. Grädel, C. Hirsch, and M. Otto. Back and forth between guarded and modal logics. In *LICS'00*, pages 217–228, 2000.

[23] E. Grädel and I. Walukiewicz. Guarded fixpoint logic. In *LICS'99*, pages 45–55, 1999.

[24] A. Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 4:1–45, 1953.

[25] Y. Gurevich and L. Harrington. Trees, automata and games. In *14th ACM Symp. on Theory of Computations*, pages 60–65, 1982.

[26] T. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In *ICALP 97: Automata, Languages, and Programming*, volume 1443 of *LNCS*, pages 580–591. 1998.

[27] Y. Hirshfeld and A. Rabinovich. A framework for decidable metrical logics. In *ICALP 99*, volume 1664 of *LNCS*, pages 422–432, 1999.

[28] Y. Hirshfeld and A. Rabinovich. Quantitative temporal logic. In *CSL'99*, volume 1683 of *LNCS*, pages 172–187, 1999.

[29] D. Janin and G. Lenzi. Relating levels of the mu-calculus hierarchy and levels of the monadic hierarchy. In *LICS'01*, 2001. To appear.

[30] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR'96*, volume 1119 of *LNCS*, pages 263–277, 1996.

[31] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[32] C. Löding. Optimal bounds for the transformation of omega-automata. In *FSTTCS'99*, volume 1738 of *LNCS*, pages 97–109, 1999.

[33] C. Löding and W. Thomas. Alternating automata and logics over infinite words. In *IFIP TCS 2000*, volume 1872 of *LNCS*, pages 521–535, 2000.

[34] O. Matz, N. Schweikardt, and W. Thomas. The monadic quantifier alternation hierarchy over grids and graphs. Information and Computation (to appear), 2000.

[35] A. Meyer. Weak monadic second order theory of one successor is not elementary. In *Lecture Notes in Mathematics*, volume 453, pages 132–154. Springer-Verlag, 1975.

[36] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In A. Skowron, editor, *Fifth Symposium on Computation Theory*, volume 208 of *LNCS*, pages 157–168, 1984.

[37] A. W. Mostowski. Games with forbidden positions. Technical Report 78, University of Gdansk, 1991.

[38] A. W. Mostowski. Hierarchies of weak automata and week monadic formulas. *Theoretical Computer Science*, 83:323–335, 1991.

[39] D. Muller and P. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[40] P. Niebert. A ν-calculus with local views for sequential agents. In *MFCS '95*, volume 969 of *LNCS*, pages 563–573, 1995.

[41] D. Niwiński. Fixed point characterization of infinite behaviour of finite state systems. *Theoretical Computer Science*, 189:1–69, 1997.

[42] D. Niwiński and I. Walukiewicz. Relating hierarchies of word and tree automata. In *STACS'98*, volume 1373 of *LNCS*. Springer-Verlag, 1998.

[43] M. Rabin. Weakly definable relations and special automata. In Y.Bar-Hillel, editor, *Mathematical Logic in Foundations of Set Theory*, pages 1–23. 1970.

[44] M. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*. Elsevier, 1977.

[45] S. Safra. On the complexity of ω-automata. In *29th IEEE Symp. on Foundations of Computer Science*, 1988.

[46] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19:424–437, 1990.

[47] S. Shelah. The monadic second order theory of order. *Annals of Mathematics*, 102:379–419, 1975.

[48] C. S. Stirling. Modal and temporal logics. In S.Abramsky, D.Gabbay, and T.Maibaum, editors, *Handbook of Logic in Comuter Science*, pages 477–563. Oxford University Press, 1991.

[49] L. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Department of Electrical Engineering MIT, 1974.

[50] H. Straubing. *Finite Automata, Formal Logic and Circuit Complexity*. Birkhäuser, 1994.

[51] R. S. Streett and E. A. Emerson. An automata theoretic procedure for the propositional mu-calculus. *Information and Computation*, 81:249–264, 1989.

[52] W. Thomas. Automata on infinite objects. In J. van Leeuven, editor, *Handbook of Theoretical Computer Science Vol.B*, pages 133–192. Elsevier, 1990.

[53] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer-Verlag, 1997.

[54] I. Walukiewicz. Monadic second order logic on tree-like structures. In *STACS '96*, volume 1046 of *LNCS*, pages 401–414, 1996. Full version to appear in Theoretical Computer Science.

[55] I. Walukiewicz. Local logics for traces. Technical Report RS-00-2, BRICS, Aarhus University, 2000. Extended version to appear in Journal of Automata, Languages and Combinatorics.

[56] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinte trees. *Theoretical Computer Science*, 200:135–183, 1998.

Automata: From Logics to Algorithms

Moshe Y. Vardi[1], Thomas Wilke[2]

[1] Department of Computer Science
Rice University
6199 S. Main Street
Houston, TX 77005-1892, U.S.A.
vardi@cs.rice.edu

[2] Institut für Informatik
Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4
24118 Kiel, Germany
wilke@ti.informatik.uni-kiel.de

Abstract

We review, in a unified framework, translations from five different logics—monadic second-order logic of one and two successors (S1S and S2S), linear-time temporal logic (LTL), computation tree logic (CTL), and modal μ-calculus (MC)—into appropriate models of finite-state automata on infinite words or infinite trees. Together with emptiness-testing algorithms for these models of automata, this yields decision procedures for these logics. The translations are presented in a modular fashion and in a way such that optimal complexity bounds for satisfiability, conformance (model checking), and realizability are obtained for all logics.

1 Introduction

In his seminal 1962 paper [Büc62], Büchi states: "Our results [...] may therefore be viewed as an application of the theory of finite automata to logic." He was referring to the fact that he had proved the decidability of the monadic-second order theory of the natural numbers with successor function by translating formulas into finite automata, following earlier work by himself [Büc60], Elgot [Elg61], and Trakthenbrot [Tra62]. Ever since, the approach these pioneers were following has been applied successfully in many different contexts and emerged as a major paradigm. It has not only brought about a number of decision procedures for mathematical theories, for instance, for the monadic second-order theory of the full binary tree [Rab69], but also efficient algorithms for problems in verification, such as a highly useful algorithm for LTL model checking [VW86a].

We are grateful to Detlef Kähler, Christof Löding, Oliver Matz, and Damian Niwiński for comments on drafts of this paper.

The "automata-theoretic paradigm" has been extended and refined in various aspects over a period of more than 40 years. On the one hand, the paradigm has led to a wide spectrum of different models of automata, specifically tailored to match the distinctive features of the logics in question, on the other hand, it has become apparent that there are certain automata-theoretic constructions and notions, such as determinization of automata on infinite words [McN66], alternation [MS85], and games of infinite duration [Büc77, GH82], which form the core of the paradigm.

The automata-theoretic paradigm is a common thread that goes through many of Wolfgang Thomas's scientific works. In particular, he has written two influential survey papers on this topic [Tho90a, Tho97].

In this paper, we review translations from five fundamental logics, monadic second-order logic of one successor function (S1S), monadic second-order logic of two successor functions (S2S), linear-time temporal logic (LTL), computation tree logic (CTL), and the modal μ-calculus (MC) into appropriate models of automata. At the same time, we use these translations to present some of the core constructions and notions in a unified framework. While adhering, more or less, to the chronological order as far as the logics are concerned, we provide modern translations from the logics into appropriate automata. We attach importance to present the translations in a modular fashion, making the individual steps as simple as possible. We also show how the classical results on S1S and S2S can be used to derive first decidability results for the three other logics, LTL, CTL, and MC, but the focus is on how more refined techniques can be used to obtain good complexity results.

While this paper focuses on the translations from logics into automata, we refer the reader to the excellent surveys [Tho90a, Tho97] and the books [GTW02, PP03] for the larger picture of automata and logics on infinite objects and the connection with games of infinite duration.

Basic Notation and Terminology

Numbers. In this paper, the set of natural numbers is denoted ω, and each natural number stands for the set of its predecessors, that is, $n = \{0, \ldots, n-1\}$.

Words. An alphabet is a nonempty finite set, a word over an alphabet A is a function $n \to A$ where $n \in \omega$ for a finite word and $n = \omega$ for an infinite word. When $u\colon n \to A$ is a word, then n is called its length and denoted $|u|$, and, for every $i < n$, the value $u(i)$ is the letter of u in position i. The set of all finite words over a given alphabet A is denoted A^*, the set of all infinite words over A is denoted A^ω, the empty word is denoted ε, and A^+ stands for $A^* \setminus \{\varepsilon\}$.

When u is a word of length n and $i, j \in \omega$ are such that $0 \leq i, j < n$, then $u[i, j] = u(i) \dots u(j)$, more precisely, $u[i, j]$ is the word u' of length $\max\{j - i + 1, 0\}$ defined by $u'(k) = u(i + k)$ for all $k < |u'|$. In the same fashion, we use the notation $u[i, j)$. When u denotes a finite, nonempty word, then we write $u(*)$ for the last letter of u, that is, when $|u| = n$, then $u(*) = u(n - 1)$. Similarly, when u is finite or infinite and $i < |u|$, then $u[i, *)$ denotes the suffix of u starting at position i.

Trees. In this paper, we deal with trees in various contexts, and depending on these contexts we use different types of trees and model them in one way or another. All trees we use are directed trees, but we distinguish between trees with unordered successors and n-ary trees with named successors.

A tree with unordered siblings is, as usual, a tuple $\mathscr{T} = (V, E)$ where V is a nonempty set of vertices and $E \subseteq V \times V$ is the set of edges satisfying the usual properties. The root is denoted $\mathrm{root}(\mathscr{T})$, the set of successors of a vertex v is denoted $\mathrm{succs}_{\mathscr{T}}(v)$, and the set of leaves is denoted $\mathrm{lvs}(\mathscr{T})$.

Let n be a positive natural number. An n-ary tree is a tuple $\mathscr{T} = (V, \mathrm{suc}_0, \dots, \mathrm{suc}_{n-1})$ where V is the set of vertices and, for every $i < n$, suc_i is the ith successor relation satisfying the condition that for every vertex there is at most one ith successor (and the other obvious conditions). Every n-ary tree is isomorphic to a tree where V is a prefix-closed nonempty subset of n^* and $\mathrm{suc}_i(v, v')$ holds for $v, v' \in V$ iff $v' = vi$. When a tree is given in this way, simply by its set of vertices, we say that the tree is given in implicit form. The full binary tree, denoted $\mathscr{T}_{bin}$, is 2^* and the full ω-tree is ω^*. In some cases, we replace n in the above by an arbitrary set and speak of D-branching trees. Again, D-branching trees can be in implicit form, which means they are simply a prefix-closed subset of D^*.

A branch of a tree is a maximal path, that is, a path which starts at the root and ends in a leaf or is infinite. If an n-ary tree is given in implicit form, a branch is often denoted by its last vertex if it is finite or by the corresponding infinite word over n if it is infinite.

Given a tree $\mathscr{T}$ and a vertex v of it, the subtree rooted at v is denoted $\mathscr{T}{\downarrow}v$.

In our context, trees often have vertex labels and in some rare cases edge labels too. When L is a set of labels, then an L-labeled tree is a tree with a function l added which assigns to each vertex its label. More precisely, for trees with unordered successors, an L-labeled tree is of the form (V, E, l) where $l\colon V \to E$; an L-labeled n-ary tree is a tuple $(V, \mathrm{suc}_0, \dots, \mathrm{suc}_{n-1}, l)$ where $l\colon V \to E$; an L-labeled n-ary tree in implicit form is a function $t\colon V \to L$ where $V \subseteq n^*$ is the set of vertices of the tree; an L-labeled D-branching tree in implicit form is a function $t\colon V \to L$ where $V \subseteq D^*$ is the set of vertices of the tree. Occasionally, we also have more than one vertex labeling or edge labelings, which are added as other components to

the tuple.

When $\mathscr{T}$ is an L-labeled tree and u is a path or branch of $\mathscr{T}$, then the labeling of u in $\mathscr{T}$, denoted $l^{\mathscr{T}}(u)$, is the word w over L of the length of u and defined by $w(i) = l(u(i))$ for all $i < |u|$.

Tuple Notation. Trees, graphs, automata and the like are typically described as tuples and denoted by calligraphic letters such as $\mathscr{T}$, $\mathscr{G}$, and so on, possibly furnished with indices or primed. The individual components are referred to by $V^{\mathscr{T}}$, $E^{\mathscr{G}}$, $E^{\mathscr{T}'}$, ... The ith component of a tuple $t = (c_0, \ldots, c_{r-1})$ is denoted $\mathrm{pr}_i(t)$.

2 Monadic Second-Order Logic of One Successor

Early results on the close connection between logic and automata, such as the Büchi–Elgot–Trakhtenbrot Theorem [Büc60, Elg61, Tra62] and Büchi's Theorem [Büc62], center around monadic second-order logic with one successor relation (S1S) and its weak variant (WS1S). The formulas of these logics are built from atomic formulas of the form $\mathrm{suc}(x, y)$ for first-order variables x and y and $x \in X$ for a first-order variable x and a set variable (monadic second-order variable) X using boolean connectives, first-order quantification ($\exists x$), and second-order quantification for sets ($\exists X$). The two logics differ in the semantics of the set quantifiers: In WS1S quantifiers only range over finite sets rather than arbitrary sets.

S1S and WS1S can be used in different ways. First, one can think of them as logics to specify properties of the natural numbers. The formulas are interpreted in the structure with the natural numbers as universe and where suc is interpreted as the natural successor relation. The most important question raised in this context is:

Validity. Is the (weak) monadic second-order theory of the natural numbers with successor relation decidable? (Is a given sentence valid in the natural numbers with successor relation?)

A slightly more general question is:

Satisfiability. Is it decidable whether a given (W)S1S formula is satisfiable in the natural numbers?

This is more general in the sense that a positive answer for closed formulas only already implies a positive answer to the first question. Therefore, we only consider satisfiability in the following.

Second, one can think of S1S and WS1S as logics to specify the behavior of devices which get, at any moment in time, a fixed number of bits as input and produce a fixed number of bits as output (such as sequential circuits), see Figure 1. Then the formulas are interpreted in the same structure as above, but for every input bit and for every output bit there will be exactly one free set variable representing the moments in time where the respective

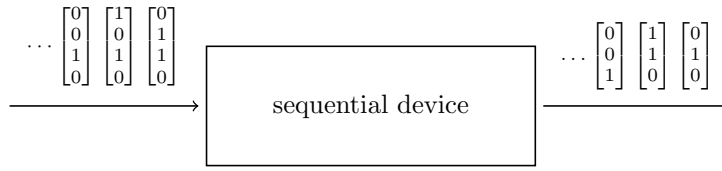


FIGURE 1. Sequential device

bit is true. (The domain of time is assumed discrete; it is identified with the natural numbers.) A formula will then be true for certain input-output pairs—coded as variable assignments—and false for the others.

For instance, when we want to specify that for a given device with one input bit, represented by the set variable X, and one output bit, represented by Y, it is the case that for every other moment in time where the input bit is true the output bit is true in the subsequent moment in time, we can use the following formula:

$$\exists Z(\text{“}Z \text{ contains every other position where } X \text{ is true”} \wedge \forall x(x \in Z \rightarrow \text{“the successor of } x \text{ belongs to } Y\text{”})).$$

That the successor of x belongs to Y is expressed by $\forall y(\mathrm{suc}(x, y) \rightarrow y \in Y)$. That Z contains every other position where X is true is expressed by the conjunction of the three following conditions, where we assume, for the moment, that the “less than” relation on the natural numbers is available:

- Z is a subset of X, which can be stated as $\forall x(x \in Z \rightarrow x \in X)$,
- If X is nonempty, then the smallest element of X does not belong to Z, which can be stated as $\forall x(x \in X \wedge \forall y(y < x \rightarrow \neg y \in X) \rightarrow \neg x \in Z)$.
- For all $x, y \in X$ such that $x < y$ and such that there is no element of X in between, either x or y belongs to Z, which can be stated as

$$\forall x \forall y(x \in X \wedge y \in X \wedge x < y \wedge \forall z(x < z \wedge z < y \rightarrow \neg z \in X) \rightarrow (x \in Z \leftrightarrow \neg y \in Z)).$$

To conclude the example, we need a formula that specifies that x is less than y. To this end, we express that y belongs to a set which does not contain x but with each element its successor:

$$\exists X(\neg x \in X \wedge \forall z \forall z'(z \in X \wedge \mathrm{suc}(z, z') \rightarrow z' \in X) \wedge y \in X).$$

The most important questions that are raised with regard to this usage of (W)S1S are:

Conformance. Is it decidable whether the input-output relation of a given device satisfies a given formula?

Realizability. Is it decidable whether for a given input-output relation there exists a device with the specified input-output relation (and if so, can a description of this device be produced effectively)?

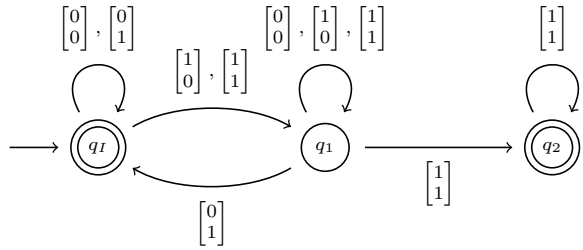
Obviously, it is important what is understood by "device". For instance, Church, when he defined realizability in 1957 [Chu60], was interested in boolean circuits. We interpret device as "finite-state device", which, on a certain level of abstraction, is the same as a boolean circuit.

In this section, we first describe Büchi's Theorem (Section 2.1), from which we can conclude that the first two questions, satisfiability and conformance, have a positive answer. The proof of Büchi's Theorem is not very difficult except for a result about complementing a certain type of automaton model for infinite words, which we then establish (Section 2.2). After that we prove a result about determinization of the same type of automaton model (Section 2.3), which serves as the basis for showing that realizability is decidable, too. The other ingredient of this proof, certain games of infinite duration, are then presented, and finally the proof itself is given (Section 2.4).

2.1 Büchi's Theorem

The connection of S1S and WS1S to automata theory, more precisely, to the theory of formal languages, is established via a simple observation. Assume that φ is a formula such that all free variables are set variables among $V_0, \ldots, V_{m-1}$, which we henceforth denote by $\varphi = \varphi(V_0, \ldots, V_{m-1})$. Then the infinite words over $[2]_m$, the set of all column vectors of height m with entries from $\{0, 1\}$, correspond in a one-to-one fashion to the variable assignments $\alpha\colon \{V_0, \ldots, V_{m-1}\} \to 2^\omega$, where 2^M stands for the power set of any set M. More precisely, for every infinite word $u \in [2]_m^\omega$ let α_u be the variable assignment defined by $\alpha_u(V_j) = \{i < \omega\colon u(i)_{[j]} = 1\}$, where, for every $a \in [2]_m$, the expression $a_{[j]}$ denotes entry j of a. Then α_u ranges over all variable assignments as u ranges over all words in $[2]_m^\omega$. As a consequence, we use $u \models \varphi$, or, when "weak quantification" (only finite sets are considered) is used, $u \models^{\mathrm{w}} \varphi$ rather than traditional notation such as $\mathfrak{N}, \alpha \models \varphi$ (where $\mathfrak{N}$ stands for the structure of the natural numbers). Further, when φ is a formula as above, we define two formal languages of infinite words depending on the type of quantification used:

$$\mathscr{L}(\varphi) = \{u \in [2]_m^\omega\colon u \models \varphi\}, \qquad \mathscr{L}^{\mathrm{w}}(\varphi) = \{u \in [2]_m^\omega\colon u \models^{\mathrm{w}} \varphi\}.$$



The initial state has an incoming edge without origin; final states are shown as double circles.

FIGURE 2. Example for a Büchi automaton

We say that φ defines the language $\mathscr{L}(\varphi)$ and weakly defines the language $\mathscr{L}^{\mathrm{w}}(\varphi)$. Note that, for simplicity, the parameter m is not referred to in our notation.

Büchi's Theorem states that the above languages can be recognized by an appropriate generalization of finite-state automata to infinite words, which we introduce next. A Büchi automaton is a tuple

$$\mathscr{A} = (A, Q, Q_I, \Delta, F)$$

where A is an alphabet, Q is a finite set of states, $Q_I \subseteq Q$ is a set of initial states, $\Delta \subseteq Q \times A \times Q$ is a set of transitions of $\mathscr{A}$, also called its transition relation, and $F \subseteq Q$ is a set of final states of $\mathscr{A}$. An infinite word $u \in A^\omega$ is accepted by $\mathscr{A}$ if there exists an infinite word $r \in Q^\omega$ such that $r(0) \in Q_I$, $(r(i), u(i), r(i+1)) \in \Delta$ for every i, and $r(i) \in F$ for infinitely many i. Such a word r is called an accepting run of $\mathscr{A}$ on u. The language recognized by $\mathscr{A}$, denoted $\mathscr{L}(\mathscr{A})$, is the set of all words accepted by $\mathscr{A}$.

For instance, the automaton in Figure 2 recognizes the language corresponding to the formula

$$\forall x(x \in V_0 \to \exists y(x < y \wedge y \in V_1)),$$

which says that every element from V_0 is eventually followed by an element from V_1. In Figure 2, q_I is the state where everything is fine; q_1 is the state where the automaton is waiting for an element from V_1 to show up; q_2 is used when from some point onwards all positions belong to V_0 and V_1. Nondeterminism is used to guess that this is the case.

Büchi's Theorem can formally be stated as follows.

Theorem 2.1 (Büchi, [Büc62])**.**

1. There exists an effective procedure that given a formula $\varphi = \varphi(V_0, \ldots, V_{m-1})$ outputs a Büchi automaton $\mathscr{A}$ such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\varphi)$.

2. There exists an effective procedure that given a Büchi automaton $\mathscr{A}$ over an alphabet $[2]_m$ outputs a formula $\varphi = \varphi(V_0, \ldots, V_{m-1})$ such that $\mathscr{L}(\varphi) = \mathscr{L}(\mathscr{A})$.

The proof of part 2 is straightforward. The formula which needs to be constructed simply states that there exists an accepting run of $\mathscr{A}$ on the word determined by the assignment to the variables V_i. One way to construct φ is to write it as $\exists X_0 \ldots \exists X_{n-1} \psi$ where each set variable X_i corresponds exactly to one state of $\mathscr{A}$ and where ψ is a first-order formula (using $<$ in addition to suc) which states that the X_i's encode an accepting run of the automaton (the X_i's must form a partition of ω and the above requirements for an accepting run must be satisfied): 0 must belong to one of the sets X_i representing the initial states; there must be infinitely many positions belonging to sets representing final states; the states assumed at adjacent positions must be consistent with the transition relation.

The proof of part 1 is more involved, although the proof strategy is simple. The desired automaton $\mathscr{A}$ is constructed inductively, following the structure of the given formula. First-order variables, which need to be dealt with in between, are viewed as singletons. The induction base is straightforward and two of the three cases to distinguish in the inductive step are so, too: disjunction on the formula side corresponds to union on the automaton side and existential quantification corresponds to projection. For negation, however, one needs to show that the class of languages recognized by Büchi automata is closed under complementation. This is not as simple as with finite state automata, especially since deterministic Büchi automata are strictly weaker than nondeterministic ones, which means complementation cannot be done along the lines known from finite words.

In the next subsection, we describe a concrete complementation construction.

Büchi's Theorem has several implications, which all draw on the following almost obvious fact. Emptiness for Büchi automata is decidable. This is easy to see because a Büchi automaton accepts a word if and only if in its transition graph there is a path from an initial state to a strongly connected component which contains a final state. (This shows that emptiness can even be checked in linear time and in nondeterministic logarithmic space.)

Given that emptiness is decidable for Büchi automata, we can state that the first question has a positive answer:

Corollary 2.2 (Büchi, [Büc62])**.** Satisfiability is decidable for S1S.

Proof. To check whether a given S1S formula $\varphi = \varphi(V_0, \ldots, V_{m-1})$ is satisfiable one simply constructs the Büchi automaton which is guaranteed to exist by Büchi's Theorem and checks this automaton for emptiness. Q.E.D.

Observe that in the above corollary we use the term "satisfiability" to denote the decision problem (Given a formula, is it satisfiable?) rather than the question from the beginning of this section (Is it decidable whether ...). For convenience, we do so in the future too: When we use one of the terms satisfiability, conformance, or realizability, we refer to the corresponding decision problem.

For conformance, we first need to specify formally what is meant by a finite-state device, or, how we want to specify the input-output relation of a finite-state device. Remember that we think of a device as getting inputs from $[2]_m$ and producing outputs from $[2]_n$ for given natural numbers m and n. So it is possible to view an input-output relation as a set of infinite words over $[2]_{m+n}$. To describe an entire input-output relation of a finite-state device we simply use a nondeterministic finite-state automaton. Such an automaton is a tuple

$$\mathscr{D} = (A, S, S_I, \Delta)$$

where A is an alphabet, S is a finite set of states, $S_I \subseteq S$ is a set of initial states, and $\Delta \subseteq S \times A \times S$ is a transition relation, just as with Büchi automata. A word $u \in A^\omega$ is accepted by $\mathscr{D}$ if there exists $r \in S^\omega$ with $r(0) \in S_I$ and $(r(i), u(i), r(i+1)) \in \Delta$ for every $i < \omega$. The set of words accepted by $\mathscr{D}$, denoted $\mathscr{L}(\mathscr{D})$, is the language recognized by $\mathscr{D}$. Observe that $\mathscr{L}(\mathscr{D})$ is exactly the same as the language recognized by the Büchi automaton which is obtained from $\mathscr{D}$ by adding the set S as the set of final states.

Conformance can now be defined as follows: Given an S1S formula $\varphi = \varphi(X_0, \ldots, X_{m-1}, Y_0, \ldots, Y_{n-1})$ and a finite-state automaton $\mathscr{D}$ with alphabet $[2]_{m+n}$, determine whether $u \models \varphi$ for all $u \in \mathscr{L}(\mathscr{D})$.

There is a simple approach to decide conformance. We construct a Büchi automaton that accepts all words $u \in \mathscr{L}(\mathscr{D})$ which do not satisfy the given specification φ, which means we construct a Büchi automaton which recognizes $\mathscr{L}(\mathscr{D}) \cap \mathscr{L}(\neg\varphi)$, and check this automaton for emptiness. Since Büchi's Theorem tells us how to construct an automaton $\mathscr{A}$ that recognizes $\mathscr{L}(\neg\varphi)$, we only need a construction which, given a finite-state automaton $\mathscr{D}$ and a Büchi automaton $\mathscr{A}$, recognizes $\mathscr{L}(\mathscr{A}) \cap \mathscr{L}(\mathscr{D})$. The construction depicted in Figure 3, which achieves this, is a simple automata-theoretic product. Its correctness can be seen easily.

Since we already know that emptiness is decidable for Büchi automata, we obtain:

The product of a Büchi automaton $\mathscr{A}$ and a finite-state automaton $\mathscr{D}$, both over the same alphabet A, is the Büchi automaton denoted $\mathscr{A} \times \mathscr{D}$ and defined by

$$\mathscr{A} \times \mathscr{D} = (A, Q \times S, Q_I \times S_I, \Delta, F \times S)$$

where

$$\Delta = \{((q,s), a, (q',s')) \colon (q,a,q') \in \Delta^{\mathscr{A}} \text{ and } (s,a,s') \in \Delta^{\mathscr{D}}\}.$$

FIGURE 3. Product of a Büchi automaton with a finite-state automaton

Corollary 2.3 (Büchi, [Büc62])**.** Conformance is decidable for S1S.

From results by Stockmeyer and Meyer [SM73, Sto74], it follows that the complexity of the two problems from Corollaries 2.2 and 2.3 is nonelementary, see also [Rei01].

Another immediate consequence of Büchi's Theorem and the proof of part 2 as sketched above is a normal form theorem for S1S formulas. Given an arbitrary S1S formula, one uses part 1 of Büchi's Theorem to turn it into an equivalent Büchi automaton and then part 2 to reconvert it to a formula. The proof of part 2 of Büchi's Theorem is designed in such a way that a formula will emerge which is of the form $\exists V_0 \ldots \exists V_{n-1} \psi$ where ψ is without second-order quantification but uses $<$. Such formulas are called existential S1S formulas.

Corollary 2.4 (Büchi-Thomas, [Büc62, Tho82])**.** Every S1S formula is equivalent to an existential S1S formula, moreover, one existential set quantifier is sufficient.

To conclude this subsection we note that using the theory of finite automata on finite words only, one can prove a result weaker than Büchi's Theorem. In the statement of this theorem, automata on finite words are used instead of Büchi automata and the weak logic is used instead of the full logic. Moreover, one considers only variable assignments for the free set variables that assign finite sets only. The latter is necessary to be able to describe satisfying assignments by finite words. Such a result was obtained independently by Büchi [Büc60], Elgot [Elg61], and Trakhtenbrot [Tra62], preceding Büchi's work on S1S.

2.2 Complementation of Büchi Automata

Büchi's original complementation construction, more precisely, his proof of the fact that the complement of a language recognized by a Büchi automaton can also be recognized by a Büchi automaton, as given in [Büc62], follows an

algebraic approach. Given a Büchi automaton $\mathscr{A}$, he defines an equivalence relation on finite words which has

1. only a finite number of equivalence classes and
2. the crucial property that $UV^\omega \subseteq \mathscr{L}(\mathscr{A})$ or $UV^\omega \cap \mathscr{L}(\mathscr{A}) = \varnothing$ for all its equivalence classes U and V.

Here, UV^ω stands for the set of all infinite words which can be written as $uv_0v_1v_2\ldots$ where $u \in U$ and $v_i \in V$ for every $i < \omega$. To complete his proof Büchi only needs to show that

(a) each set UV^ω is recognized by a Büchi automaton,

(b) every infinite word over the given alphabet belongs to such a set, and

(c) the class of languages recognized by Büchi automata is closed under union.

To prove (b), Büchi uses a weak variant of Ramsey's Theorem; (a) and (c) are easy to see. The equivalence relation Büchi defines is similar to Nerode's congruence relation. For a given word u, he considers

(i) all pairs (q, q') of states for which there exists a path from q to q' labeled u and

(ii) all pairs (q, q') where, in addition, such a path visits a final state,

and he defines two nonempty finite words to be equivalent if they agree on these pairs. If one turns Büchi's "complementation lemma" into an actual complementation construction, one arrives at a Büchi automaton of size $2^{\theta(n^2)}$ where n denotes the number of states of the given Büchi automaton.

Klarlund [Kla91] and Kupferman and Vardi [KV01] describe complementation constructions along the following lines. Given a Büchi automaton $\mathscr{A}$ and a word u over the same alphabet, they consider the run DAG of $\mathscr{A}$ on u, which is a narrow DAG which contains exactly the runs of $\mathscr{A}$ on u. Vertices in this run DAG are of the form (q, i) with $q \in Q$ and $i \in \omega$ and all runs where the ith state is q visit this vertex. They show that u is not accepted by $\mathscr{A}$ if and only if the run DAG can be split into at most $2n$ alternating layers of two types where within the layers of the first type every vertex has proper descendants which are labeled with nonfinal states and where within the layers of the second type every vertex has only a finite number of descendants (which may be final or nonfinal). This can easily be used to construct a Büchi automaton for the complement: It produces the run DAG step by step, guesses for each vertex to which layer it belongs, and checks that its guesses are correct. To check the requirement for the

layers of the second type, it uses the Büchi acceptance condition. The size of the resulting automaton is $2^{\theta(n \log n)}$. Optimizations lead to a construction with $(0.97n)^n$ states [FKV06], while the best known lower bound is $(0.76n)^n$, established by Yan [Yan06]. For practical implementations of the construction by Kupferman and Vardi, see [GKSV03].

In Section 2.2.2, we describe a complementation construction which is a byproduct of the determinization construction we explain in Section 2.3. Both constructions are based on the notion of reduced acceptance tree, introduced by Muller and Schupp [MS95] and described in what follows.

2.2.1 Reduced Acceptance Trees

Recall the notation and terminology with regard to binary trees introduced in Section 1.

Let $\mathscr{A}$ be a Büchi automaton as above, u an infinite word over the alphabet A. We consider a binary tree, denoted $\mathscr{T}_u$, which arranges all runs of $\mathscr{A}$ on u in a clever fashion, essentially carrying out a subset construction that distinguishes between final and nonfinal states, see Figure 4 for a graphical illustration.

The tree $\mathscr{T}_u = (V_u, l_u)$ is a 2^Q-labeled tree in implicit form defined inductively as follows.

(i) $\varepsilon \in V_u$ and $l_u(\varepsilon) = Q_I$.

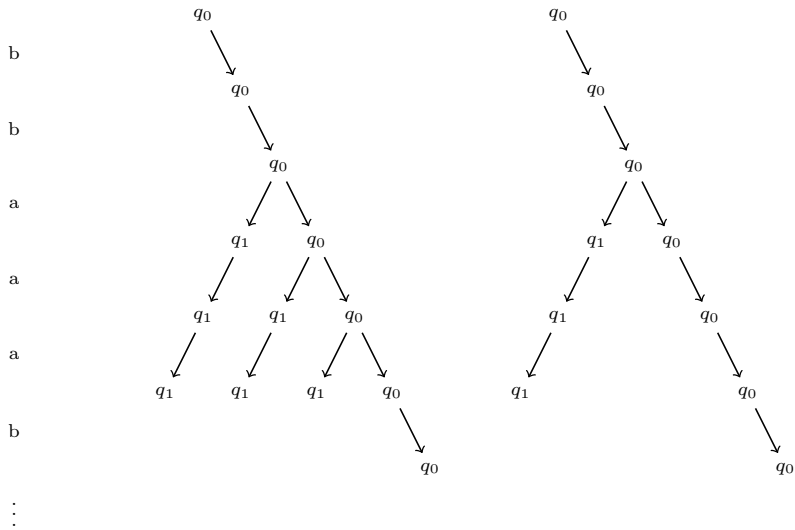
(ii) Let $v \in V_u$, $Q' = l_u(v)$, $a = u(|v|)$, and $Q'' = \bigcup\{\Delta(q,a)\colon q \in Q'\}$. Here and later, we use $\Delta(q,a)$ to denote $\{q' \in Q\colon (q,a,q') \in \Delta\}$.

- If $Q'' \cap F \neq \varnothing$, then $v0 \in V_u$ and $l_u(v0) = Q'' \cap F$.
- If $Q'' \setminus F \neq \varnothing$, then $v1 \in V_u$ and $l_u(v1) = Q'' \setminus F$.

The resulting tree is called the run tree of u with respect to $\mathscr{A}$.

A partial run of $\mathscr{A}$ on u is a word $r \in Q^+ \cup Q^\omega$ satisfying $r(0) \in Q_I$ and $(r(i), u(i), r(i+1)) \in \Delta$ for all i such that $i+1 < |r|$. A run is an infinite partial run.

Every partial run r of $\mathscr{A}$ on u determines a path b in the run tree: The length of b is $|r| - 1$ and $b(i) = 0$ if $r(i+1) \in F$ and $b(i) = 1$ otherwise, for $i < |r| - 1$. We write $r{\downarrow\downarrow}$ for this path and call it the 2-projection of r. Clearly, if r is an accepting run of $\mathscr{A}$ on u, then $r{\downarrow\downarrow}$ has infinitely many left turns, where a left turn is a vertex which is a left successor. Conversely, if b is an infinite branch of $\mathscr{T}_u$, then there exists a run r of $\mathscr{A}$ on u such that $r{\downarrow\downarrow} = b$, and if b has infinitely many left turns, then r is accepting. This follows from Kőnig's lemma.



Depicted are the run tree and the reduced run tree of the automaton to the right for the given word. Note that in the trees the labels of the vertices should be sets of states, but for notational convenience we only list their elements.

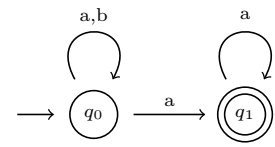


FIGURE 4. Run tree and reduced run tree

From this, we can conclude:

Remark 2.5. An infinite word u is accepted by a Büchi automaton $\mathscr{A}$ if and only if its run tree has a branch with an infinite number of left turns. We call such a branch an acceptance witness.

The tree $\mathscr{T}_u$ has two other interesting properties, which we discuss next. The first one is that $\mathscr{T}_u$ has a "left-most" acceptance witness, provided there is one at all. This acceptance witness, denoted b_u, can be constructed as follows. Inductively, assume $b_u(i)$ has already been defined for all $i < n$ in a way such that there is an acceptance witness with prefix $b' = b_u(0) \dots b_u(n-1)$. If there is an acceptance witness with prefix $b'0$, we set $b_u(n) = 0$. Otherwise, there must be an acceptance witness with prefix $b'1$, and we set $b_u(n) = 1$. Clearly, this construction results in an acceptance witness. One can easily prove that b_u is the left-most acceptance witness in the sense that it is minimal among all acceptance witnesses with respect to the lexicographical ordering (but we do not need this here).

The second interesting property says something about the states occurring to the left of b_u. We say a state q is persistent in a vertex v of a branch b of $\mathscr{T}_u$ if there is a run r of $\mathscr{A}$ on u such that $r{\downarrow\downarrow} = b$ and $q \in r(|v|)$, in other words, q is part of a run whose 2-projection contains v. A word $v \in \{0,1\}^*$ is said to be left of a word $w \in \{0,1\}^*$, denoted $v <_{\text{lft}} w$, if $|v| = |w|$ and $v <_{\text{lex}} w$, where $<_{\text{lex}}$ denotes the lexicographical ordering. The crucial property of b_u is:

Lemma 2.6. Let u be an infinite word accepted by a Büchi automaton $\mathscr{A}$, w a vertex on the left-most acceptance witness b_u, and q a state which is persistent in w on b_u. Then $q \notin l_u(v)$ for every $v \in V_u$ such that $v <_{\text{lft}} w$.

Proof. Assume that w is a vertex on b_u and that $v \in V_u$ is left of w, let $n = |v|$ $(= |w|)$. For contradiction, assume q is persistent in w on b_u and $q \in l_u(v) \cap l_u(w)$. Since $q \in l_u(v)$, we know there is a partial run r of $\mathscr{A}$ on u with $r{\downarrow\downarrow} = v$ and $r(n) = q$.

Since q is persistent in w on b_u there exists a run r' of $\mathscr{A}$ on u such that $r'{\downarrow\downarrow} = b_u$ and $r'(n) = q$. Then $r'[n,\infty)$ is an uninitialized run of $\mathscr{A}$ on $u[n,\infty)$ starting with q, where an uninitialized run is one where it is not required that the first state is the initial state. This implies that $r'' = rr'(n,\infty)$ is a run of $\mathscr{A}$ on u. Moreover, $r(i) = r''(i)$ for all $i \geq n$, which implies $r''{\downarrow\downarrow}$ is an acceptance witness, too. Let c be the longest common prefix of $r''{\downarrow\downarrow}$ and b_u. We know that $c0 \leq_{\text{prf}} r''{\downarrow\downarrow}$ and $c1 \leq_{\text{prf}} b_u$, which is a contradiction to the definition of b_u—recall that $r''{\downarrow\downarrow}$ is an acceptance witness. Q.E.D.

The above fact can be used to prune $\mathscr{T}_u$ in such a way that it has finite width, but still contains an acceptance witness if and only if u is accepted by $\mathscr{A}$. We denote the pruned tree by $\mathscr{T}'_u = (V'_u, l'_u)$ and call it the reduced acceptance tree. Informally, $\mathscr{T}'_u$ is obtained from $\mathscr{T}_u$ by keeping on each level only the first occurrence of a state, reading the level from left to right, see Figure 4. Formally, the reduced acceptance tree is inductively defined as follows.

(i) $\varepsilon \in V'_u$ and $l'_u(\varepsilon) = Q_I$.

(ii) Let $v \in V'_u$, $Q' = l'_u(v)$, $a = u(|v|)$, and $Q'' = \bigcup\{\Delta(q,a)\colon q \in Q'\}$, just as above. Assume $l'_u(w)$ has already been defined for $w <_{\text{lft}} v0$ and let $\bar{Q} = \bigcup\{l'_u(w)\colon w \in V'_u \text{ and } w <_{\text{lft}} v0\}$.

- If $Q'' \cap F \setminus \bar{Q} \neq \varnothing$, then $v0 \in V'_u$ and $l'_u(v0) = Q'' \cap F \setminus \bar{Q}$.
- If $Q'' \setminus (F \cup \bar{Q}) \neq \varnothing$, then $v1 \in V'_u$ and $l'_u(v1) = Q'' \setminus (F \cup \bar{Q})$.

As a consequence of Lemma 2.6, we have:

Corollary 2.7. Let $\mathscr{A}$ be a Büchi automaton and u an infinite word over the same alphabet. Then $u \in \mathscr{L}(\mathscr{A})$ iff $\mathscr{T}'_u$ contains an acceptance witness.

Since $\mathscr{T}'_u$ is a tree of width at most $|Q|$, it has at most $|Q|$ infinite branches. So u is not accepted by $\mathscr{A}$ if and only if there is some number n such that $b(i)$ is not a left turn for all infinite branches b of $\mathscr{T}'_u$. This fact can be used to construct a Büchi automaton for the complement language, as will be shown in what follows.

2.2.2 The Complementation Construction

Let n be an arbitrary natural number and $v_0 <_{\text{lft}} v_1 <_{\text{lft}} \dots <_{\text{lft}} v_{r-1}$ be such that $\{v_0, \dots, v_{r-1}\} = \{v \in V'_u \colon |v| = n\}$, that is, $v_0, \dots, v_{r-1}$ is the sequence of all vertices on level n of $\mathscr{T}'_u$, from left to right. We say that $l'_u(v_0) \dots l'_u(v_{r-1})$, which is a word over the alphabet 2^Q, is slice n of $\mathscr{T}'_u$.

It is straightforward to construct slice $n+1$ from slice n, simply by applying the transition relation to each element of slice n and removing multiple occurrences of states just as with the construction of $\mathscr{T}'_u$. Suppose $Q_0 \dots Q_{r-1}$ is slice n and $a = u(n)$. Let $Q'_0, \dots, Q'_{2r-1}$ be defined by

$$Q'_{2i} = \Delta(Q_i, a) \cap F \setminus \bar{Q}_i, \qquad Q'_{2i+1} = \Delta(Q_i, a) \setminus (F \cup \bar{Q}_i),$$

where $\bar{Q}_i = \bigcup_{j<2i} Q'_j$. Further, let $j_0 < j_1 < \dots < j_{s-1}$ be such that $\{j_0, \dots, j_{s-1}\} = \{j < 2r \colon Q'_j \neq \varnothing\}$. Then $Q'_{j_0} \dots Q'_{j_{s-1}}$ is slice $n+1$ of $\mathscr{T}'_u$. This is easily seen from the definition of the reduced run tree.

We say that a tuple $U = Q_0 \dots Q_{r-1}$ is a slice over Q if $\varnothing \neq Q_i \subseteq Q$ holds for $i < r$ and if $Q_i \cap Q_j = \varnothing$ for all $i, j < r$ with $i \neq j$. The sequence $Q'_{j_0} \dots Q'_{j_{s-1}}$ from above is said to be the successor slice for U and a and is denoted by $\delta_{\text{slc}}(Q_0 \dots Q_{r-1}, a)$.

The automaton for the complement of $\mathscr{L}(\mathscr{A})$, denoted $\mathscr{A}^{\mathsf{C}}$, works as follows. First, it constructs slice after slice as it reads the given input word. We call this the initial phase. At some point, it guesses

(i) that it has reached slice n or some later slice, with n as described right after Corollary 2.7, and

(ii) which components of the slice belong to infinite branches.

The rest of its computation is called the repetition phase. During this phase it carries out the following process, called verification process, over and over again. It continues to construct slice after slice, checking that (i) the components corresponding to vertices on infinite branches all continue to the right (no left turn anymore) and (ii) the components corresponding to the other branches die out (do not continue forever). The newly emerging components corresponding to branches which branch off to the left from the vertices on the infinite branches are marked. As soon as all branches

supposed to die out have died out, the process starts all over again, now with the marked components as the ones that are supposed to die out.

To be able to distinguish between components corresponding to infinite branches, branches that are supposed to die out, and newly emerging branches, the components of the slice tuples are decorated by *inf*, *die*, or *new*. Formally, a decorated slice is of the form $(Q_0 \dots Q_{r-1}, f_0 \dots f_{r-1})$ where $Q_0 \dots Q_{r-1}$ is a slice and $f_i \in \{\mathit{inf}, \mathit{die}, \mathit{new}\}$ for $i < r$. A decorated slice where $f_i \neq \mathit{die}$ for all $i < r$ is called final.

The definition of the successor of a decorated slice is slightly more involved than for ordinary slices, and such a successor may not even exist. Assume a decorated slice as above is given, let V stand for the entire slice and U for its first component (which is an ordinary slice). Let the Q'_j's and j_i's be defined as above. The successor slice of V with respect to a, denoted $\delta_d(V, a)$, does not exist if there is some $i < r$ such that $Q'_{2i+1} = \varnothing$ and $f_i = \mathit{inf}$, because this means that a branch guessed to be infinite and without left turn dies out. In all other cases, $\delta_d(V, a) = (\delta_{\mathrm{slc}}(U, a), f'_{j_0} \dots f'_{j_{s-1}})$ where the f'_j's are defined as follows, depending on whether the automaton is within the verification process (V is not final) or at its end (V is final):

Slice V is not final. Then $f'_{2i} = f'_{2i+1} = f_i$ for every $i < r$, except when $f_i = \mathit{inf}$. In this case, $f'_{2i} = \mathit{new}$ and $f_{2i+1} = f_i$.

Slice V is final. Then $f'_{2i} = f'_{2i+1} = \mathit{die}$ for every $i < r$, except when $f_i = \mathit{inf}$. In this case, $f'_{2i+1} = \mathit{inf}$ and $f'_{2i} = \mathit{die}$.

These choices reflect the behavior of the automaton as described above.

To describe the transition from the first to the second phase formally, assume U is a slice and $a \in A$. Let $\Delta_s(U, a)$ contain all decorated slices $(\delta_{\mathrm{slc}}(U, a), f_0 \dots f_{s-1})$ where $f_i \in \{\mathit{inf}, \mathit{die}\}$ for $i < s$. This reflects that the automaton guesses that certain branches are infinite and that the others are supposed to die out. The full construction of $\mathscr{A}^{\mathsf{C}}$ as outlined in this section is described in Figure 5. A simple upper bound on its number of states is $(3n)^n$.

Using L^{C} to denote the complement of a language, we can finally state:

Theorem 2.8. Let $\mathscr{A}$ be a Büchi automaton with n states. Then $\mathscr{A}^{\mathsf{C}}$ is a Büchi automaton with $(3n)^n$ states such that $\mathscr{L}(\mathscr{A}^{\mathsf{C}}) = \mathscr{L}(\mathscr{A})^{\mathsf{C}}$.

2.3 Determinization of Büchi Automata

As noted above, determinstic Büchi automata are strictly weaker than nondeterministic ones in the sense that there are ω-languages that can be recognized by a nondeterministic Büchi automaton but by no deterministic Büchi automaton. (Following classical terminology, a Büchi automaton is called deterministic if $|Q_I| = 1$ and there is a function $\delta\colon Q \times A \to Q$ such that

Let $\mathscr{A}$ be a Büchi automaton. The Büchi automaton $\mathscr{A}^{\mathsf{C}}$ is defined by

$$\mathscr{A}^{\mathsf{C}} = (A, Q^{\mathrm{s}} \cup Q^{\mathrm{d}}, Q_I, \Delta', F')$$

where the individual components are defined as follows:

$$\begin{aligned} Q^{\mathrm{s}} &= \text{set of slices over } Q, \\ Q^{\mathrm{d}} &= \text{set of decorated slices over } Q, \\ F' &= \text{set of final decorated slices over } Q, \end{aligned}$$

and where for a given $a \in A$ the following transitions belong to Δ':

- $(U, a, \delta_{\mathrm{slc}}(U, a))$ for every $U \in Q^{\mathrm{s}}$,
- (U, a, V) for every $U \in Q^{\mathrm{s}}$ and $V \in \Delta_s(U, a)$,
- $(V, a, \delta_d(V, a))$ for every $V \in Q^{\mathrm{d}}$, provided $\delta_d(V, a)$ is defined.

FIGURE 5. Complementing a Büchi automaton

$\Delta = \{(q, a, \delta(q, a)) \colon a \in A \wedge q \in Q\}$.) It turns out that this is due to the weakness of the Büchi acceptance condition. When a stronger acceptance condition—such as the parity condition—is used, every nondeterministic automaton can be converted into an equivalent deterministic automaton.

The determinization of Büchi automata has a long history. After a flawed construction had been published in 1963 [Mul63], McNaughton, in 1966 [McN66], was the first to prove that every Büchi automaton is equivalent to a deterministic Muller automaton, a model of automata on infinite words with an acceptance condition introduced in Muller's work. In [ES84b, ES84a], Emerson and Sistla described a determinization construction that worked only for a subclass of all Büchi automata. Safra [Saf88] was the first to describe a construction which turns nondeterministic Büchi automata into equivalent deterministic Rabin automata—a model of automata on infinite words with yet another acceptance condition—which has optimal complexity in the sense that the size of the resulting automaton is $2^{\theta(n \log n)}$ and one can prove that this is also a lower bound [Mic88]. In 1995, Muller and Schupp [MS95] presented a proof of Rabin's Theorem via

an automata-theoretic construction which has an alternative determinization construction with a similar complexity built-in; Kähler [Käh01] was the first to isolate this construction, see also [ATW06]. Kähler [Käh01] also showed that based on Emerson and Sistla's construction one can design another determinization construction for all Büchi automata which yields automata with of size $2^{\theta(n \log n)}$, too. In 2006, Piterman [Pit06] showed how Safra's construction can be adapted so as to produce a parity automaton of the same complexity.

The determinization construction described below is obtained by applying Piterman's improvement of Safra's construction to Muller and Schupp's determinization construction. We first introduce parity automata, then continue our study of the reduced acceptance tree, and finally describe the determinization construction.

2.3.1 Parity Automata

A parity automaton is very similar to a Büchi automaton. The only difference is that a parity automaton has a more complex acceptance condition, where every state is assigned a natural number, called priority, and a run is accepting if the minimum priority occurring infinitely often (the limes inferior) is even. States are not just accepting or rejecting; there is a whole spectrum. For instance, when the smallest priority is even, then all states with this priority are very similar to accepting states in Büchi automata: If a run goes through these states infinitely often, then it is accepting. When, on the other hand, the smallest priority is odd, then states with this priority should be viewed as being the opposite of an accepting state in a Büchi automaton: If a run goes through these states infinitely often, the run is not accepting. So parity automata allow for a finer classification of runs with regard to acceptance.

Formally, a parity automaton is a tuple

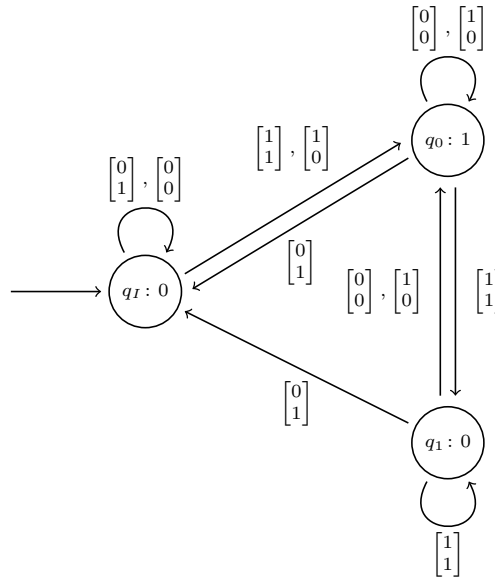
$$\mathscr{A} = (A, Q, Q_I, \Delta, \pi)$$

where A, Q, Q_I, and Δ are as with Büchi automata, but π is a function $Q \to \omega$, which assigns to each state its priority. Given an infinite sequence r of states of this automaton, we write $\mathrm{val}_\pi(r)$ for the limes inferior of the sequence $\pi(r(0)), \pi(r(1)), \ldots$ and call it the value of the run with respect to π. Since Q is finite, the value of each run is a natural number. A run r of $\mathscr{A}$ is accepting if its value is even. In other words, a run r of $\mathscr{A}$ is accepting if there exists an even number v and a number k such that

(i) $\pi(r(j)) \geq v$ for all $j \geq k$ and

(ii) $\pi(r(j)) = v$ for infinitely many $j \geq k$.

Consider, for example, the parity automaton depicted in Figure 6. It recognizes the same language as the Büchi automaton in Figure 2.



The values in the circles next to the names of the states are the priorities.

FIGURE 6. Deterministic parity automaton

As far as nondeterministic automata are concerned, Büchi automata and parity automata recognize the same languages. On the one hand, every Büchi automaton can be viewed as a parity automaton where priority 1 is assigned to every non-final state and priority 0 is assigned to every final state. (That is, the parity automaton in Figure 6 can be regarded as a deterministic Büchi automaton.) On the other hand, it is also easy to see that every language recognized by a parity automaton is recognized by some Büchi automaton: The Büchi automaton guesses a run of the parity automaton and an even value for this run and checks that it is indeed the value of the run. To this end, the Büchi automaton runs in two phases. In the first phase, it simply simulates the parity automaton. At some point, it concludes the first phase, guesses an even value, and enters the second phase during which it continues to simulate the parity automaton but also verifies (i) and (ii) from above. To check (i), the transition relation is restricted appropriately. To check (ii), the Büchi acceptance condition is used. This leads to the construction displayed in Figure 7. The state space has two different types of states: the states from the given Büchi automaton for the first phase and states of the form (q, k) where $q \in Q$ and k is a priority for

Let $\mathscr{A}$ be a parity automaton. The Büchi automaton A^{par} is defined by

$$A^{par} = (A, Q \cup Q \times E, Q_I, \Delta \cup \Delta', \{(q,k)\colon \pi(q) = k\})$$

where $E = \{\pi(q)\colon q \in Q \wedge \pi(q) \bmod 2 = 0\}$ and Δ' contains

- $(q, a, (q', k))$ for every $(q, a, q') \in \Delta$, provided $k \in E$, and
- $((q,k), a, (q', k))$ for every $(q, a, q') \in \Delta$, provided $\pi(q') \geq k$ and $k \in E$.

FIGURE 7. From parity to Büchi automata

the second phase. The priority in the second component never changes; it is the even value that the automaton guesses.

Remark 2.9. Let $\mathscr{A}$ be a parity automaton with n states and k different even priorities. Then the automaton A^{par} is an equivalent Büchi automaton with $(k+1)n$ states.

2.3.2 Approximating Reduced Run Trees

Let $\mathscr{A}$ be a Büchi automaton as above and $u \in A^\omega$ an infinite word. The main idea of Muller and Schupp's determinization construction is that the reduced acceptance tree, $\mathscr{T}'_u$, introduced in Section 2.2.1, can be approximated by a sequence of trees which can be computed by a deterministic finite-state automaton. When these approximations are adorned with additional information, then from the sequence of the adorned approximations one can read off whether there is an acceptance witness in the reduced acceptance tree, which, by Remark 2.5, is enough to decide whether u is accepted.

For a given number n, the nth approximation of $\mathscr{T}'_u$, denoted $\mathscr{T}^n_u$, is the subgraph of $\mathscr{T}'_u$ which consists of all vertices of distance at most n from the root and which are on a branch of length at least n. Only these vertices can potentially be on an infinite branch of $\mathscr{T}'_u$. Formally, $\mathscr{T}^n_u$ is the subtree of $\mathscr{T}'_u$ consisting of all vertices $v \in V'_u$ such that there exists $w \in V'_u$ satisfying $v \leq_{\mathrm{prf}} w$ and $|w| = n$, where $\leq_{\mathrm{prf}}$ denotes the prefix order on words.

Note that from Lemma 2.6 we can conclude:

Remark 2.10. When u is accepted by $\mathscr{A}$, then for every n the prefix of length n of b_u is a branch of $\mathscr{T}_u^n$.

The deterministic automaton to be constructed will observe how approximations evolve over time. There is, however, the problem that, in general, approximations grow as n grows. But since every approximation has at most $|Q|$ leaves, it has at most $|Q| - 1$ internal vertices with two successors—all other internal vertices have a single successor. This means that their structure can be described by small trees of bounded size, and only their structure is important, except for some more information of bounded size. This motivates the following definitions.

A segment of a finite tree is a maximal path where every vertex except for the last one has exactly one successor, that is, it is a sequence $v_0 \dots v_r$ such that

(i) the predecessor of v_0 has two successors or v_0 is the root,

(ii) v_i has exactly one successor for $i < r$, and

(iii) v_r has exactly two successors or is a leaf.

Then every vertex of a given finite tree belongs to exactly one segment.

A contraction of a tree is obtained by merging all vertices of a segment into one vertex. Formally, a contraction of a finite tree $\mathscr{T}$ in implicit form is a tree $\mathscr{C}$ together with a function $c\colon t \to V^{\mathscr{C}}$, the contraction map, such that the following two conditions are satisfied:

(i) For all $v, w \in V^{\mathscr{T}}$, $c(v) = c(w)$ iff v and w belong to the same segment.

When p is a segment of $\mathscr{T}$ and v one of its vertices, we write $c(p)$ for $c(v)$ and we say that $c(v)$ represents p.

(ii) For all $v \in V^{\mathscr{T}}$ and $i < 2$, if $vi \in V^{\mathscr{T}}$ and $c(v) \neq c(vi)$, then $\mathrm{suc}_1^{\mathscr{C}}(c(v), c(vi))$.

Note that this definition can easily be adapted to the case where the given tree is not in implicit form.

We want to study how approximations evolve over time. Clearly, from the nth to the $(n+1)$st approximation of $\mathscr{T}_u'$ segments can disappear, several segments can be merged into one, new segments of length one can emerge, and segments can be extended by one vertex. We reflect this in the corresponding contractions by imposing requirements on the domains of consecutive contractions.

A sequence $\mathscr{C}_0, \mathscr{C}_1, \dots$ of contractions with contraction maps $c_0, c_1, \dots$ is a contraction sequence for u if the following holds for every n:

(i) $\mathscr{C}_n$ is a contraction of the nth approximation of $\mathscr{T}_u'$.

(ii) Let p and p' be segments of $\mathscr{T}_u^n$ and $\mathscr{T}_u^{n+1}$, respectively. If p is a prefix of p' (including $p = p'$), then $c_{n+1}(p') = c_n(p)$ and p' is called an extension of p in $n+1$.

(iii) If p' is a segment of $\mathscr{T}_u^{n+1}$ which consists of vertices not belonging to $\mathscr{T}$, then $c_{n+1}(p') \notin V^{\mathscr{C}_n}$, where $V^{\mathscr{C}_n}$ denotes the set of vertices of $\mathscr{C}_n$.

Since we are interested in left turns, we introduce one further notion. Assume that p and p' are segments of $\mathscr{T}_u^n$ and $\mathscr{T}_u^{n+1}$, respectively, and p is a prefix of p', just as in (ii) above. Let p'' be such that $p' = pp''$. We say that $c_{n+1}(p')$ (which is equal to $c_n(p)$) is left extending in $n+1$ if there is a left turn in p''.

For a graphical illustration, see Figure 8.

We can now give a characterization of acceptance in terms of contraction sequences.

Lemma 2.11. Let $\mathscr{C}_0, \mathscr{C}_1, \ldots$ be a contraction sequence for an infinite word u with respect to a Büchi automaton $\mathscr{A}$. Then the following are equivalent:
(A) $\mathscr{A}$ accepts u.
(B) There is a vertex v such that

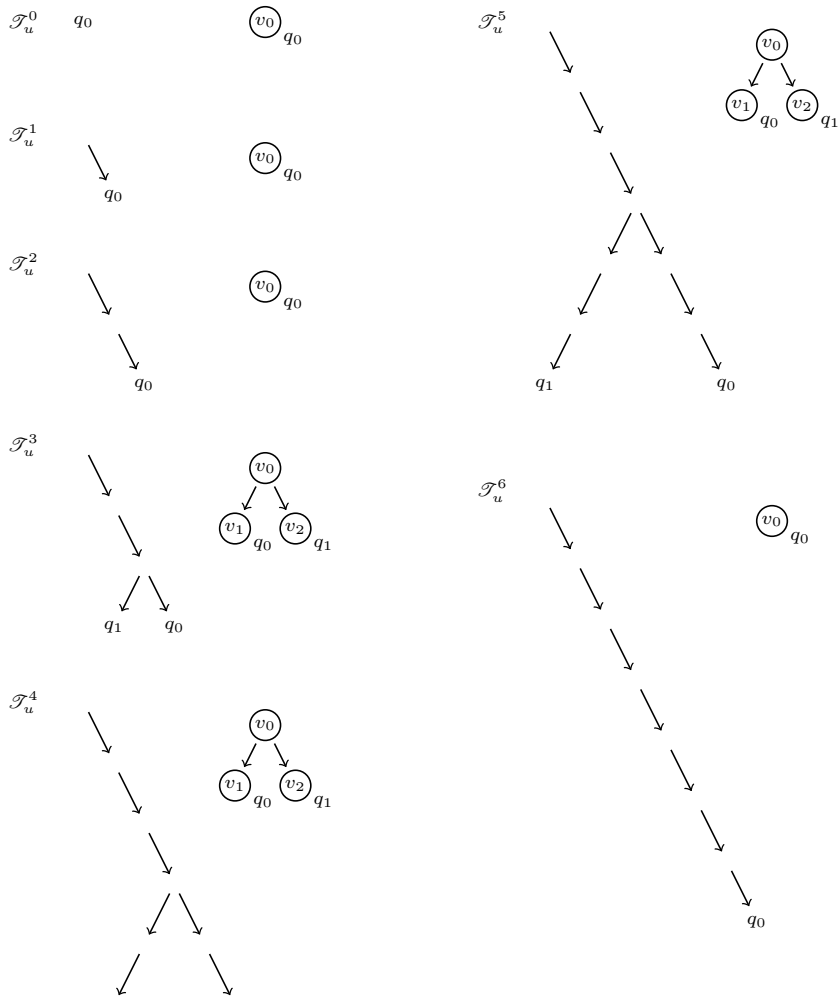
(a) $v \in V^{\mathscr{C}_n}$ for almost all n and

(b) v is left extending in infinitely many n.

Proof. For the implication from (A) to (B), we start with a definition. We say that a segment p of the nth approximation is part of b_u, the left-most acceptance witness, if there are paths p_0 and p_1 such that $b_u = p_0 p p_1$. We say a vertex v represents a part of b_u if there exists i such that for all $j \geq i$ the vertex v belongs to $V^{\mathscr{C}_j}$ and the segment represented by v is part of b_u. Observe that from Remark 2.10 we can conclude that the root of $\mathscr{C}_0$ is such a vertex (where we can choose $i = 0$). Let V be the set of all vertices that represent a part of b_u and assume i is chosen such that $v \in V^{\mathscr{C}_j}$ for all $j \geq i$ and all $v \in V$. Then all elements from V form the same path in every $\mathscr{C}_j$ for $j \geq i$, say $v_0 \ldots v_r$ is this path.

If the segment representing v_r is infinitely often extended, it will also be extended by a left turn infinitely often (because b_u is an acceptance witness), so v_r will be left extending in infinitely many i.

So assume that v_r is not extended infinitely often and let $i' \geq i$ be such that the segment represented by v_r is not extended any more for $j \geq i'$. Consider $\mathscr{C}_{i'+1}$. Let v' be the successor of v_r such that the segment represented by v' is part of b_u, which must exist because of Remark 2.10. Clearly, for the same reason, v' will be part of $V^{\mathscr{C}_j}$ for $j \geq i'+1$, hence $v' \in V$—a contradiction.



Depicted is the beginning of the contraction sequence for $u = bbaaab\ldots$ with respect to the automaton from Figure 4. Note that, just as in Figure 4, we simply write q_i for $\{q_i\}$.

FIGURE 8. Contraction sequence

For the implication from (B) to (A), let v be a vertex as described in (B), in particular, let i be such that $v \in V^{\mathscr{C}_j}$ for all $j \geq i$. For every $j \geq i$, let p^j be the segment represented by v in $\mathscr{C}_j$. Since $p^i \leq_{\mathrm{prf}} p^{i+1} \leq_{\mathrm{prf}} p^{i+2} \leq_{\mathrm{prf}} \ldots$

we know there is a vertex w such that every p^j, for $j \geq i$, starts with w. Since the number of left turns on the p^j's is growing we know there is an infinite path d starting with w such that $p^j \leq_{\mathrm{prf}} d$ for every $j \geq i$ and such that d is a path in $\mathscr{T}'_u$ with infinitely many left turns. The desired acceptance witness is then given by the concatenation of the path from the root to w, the vertex w itself excluded, and d. Q.E.D.

2.3.3 Muller–Schupp Trees

The only thing which is left to do is to show that a deterministic finite-state automaton can construct a contraction sequence for a given word u and that a parity condition is strong enough to express (B) from Lemma 2.11. It turns out that when contractions are augmented with additional information, they can actually be used as the states of such a deterministic automaton. This will lead us to the definition of Muller–Schupp trees.

Before we get to the definition of these trees, we observe that every contraction has at most $|Q|$ leaves, which means it has at most $2|Q| - 1$ vertices. From one contraction to the next in a sequence of contractions, at most $|Q|$ new leaves—and thus at most $|Q|$ new vertices—can be introduced. In other words:

Remark 2.12. For every infinite word u, there is a contraction sequence $\mathscr{C}_0, \mathscr{C}_1, \ldots$ such that $V^{\mathscr{C}_i} \subseteq V$ for every i for the same set V with $3|Q|$ vertices, in particular, $V = \{0, \ldots, 3|Q| - 1\}$ works.

A Muller-Schupp tree for $\mathscr{A}$ is a tuple

$$\mathscr{M} = (\mathscr{C}, l_q, l_l, R, h)$$

where

- $\mathscr{C}$ is a contraction with $V^{\mathscr{C}} \subseteq \{0, \ldots, 3\,|Q| - 1\}$,
- $l_q \colon \mathrm{lvs}(\mathscr{C}) \to 2^Q$ is a leaf labeling,
- $l_l \colon V^{\mathscr{C}} \to \{0, 1, 2\}$ is a left labeling,
- $R \in \{0, \ldots, 3|Q| - 1\}^*$ is a latest appearance record, a word without multiple occurrences of letters, and
- $h \leq |R|$ is the hit number.

To understand the individual components, assume $\mathscr{C}_0, \mathscr{C}_1, \ldots$ is a contraction sequence for u with $V^{\mathscr{C}_n} \subseteq \{0, \ldots, 3|Q| - 1\}$ for every n. (Recall that Remark 2.12 guarantees that such a sequence exists.) The run of the deterministic automaton on u to be constructed will be a sequence $\mathscr{M}_0, \mathscr{M}_1, \ldots$

of Muller-Schupp trees $\mathscr{M}_n = (\mathscr{C}_n, l^n_q, l^n_l, R^n, h^n)$, with contraction map c_n for $\mathscr{C}_n$ and such that the following conditions are satisfied.

Leaf labeling. For every n and every leaf $v \in \mathrm{lvs}(\mathscr{T}^n_u)$, the labeling of v will be the same as the labeling of the vertex of the segment representing the segment of this leaf, that is, $l^n_q(c_n(v)) = l'_u(v)$.

Left labeling. For every n and every $v \in V^{\mathscr{C}_n}$:

(i) if v represents a segment without left turn, then $l_n(v) = 0$,

(ii) if v is left extending in n, then $l^n_l(v) = 2$, and

(iii) $l^n_l(v) = 1$ otherwise.

Clearly, this will help us to verify (b) from Lemma 2.11(B).

Latest appearance record. The latest appearance record R^n gives us the order in which the vertices of $\mathscr{C}_n$ have been introduced. To make this more precise, for every n and $v \in V^{\mathscr{C}_n}$, let

$$d_n(v) = \min\{i \colon v \in V^{\mathscr{C}_j} \text{ for all } j \text{ such that } i \le j \le n\}$$

be the date of introduction of v. Then R^n is the unique word $v_0 \dots v_{r-1}$ over $V^{\mathscr{C}_n}$ without multiple occurrences such that

- $\{v_0, \dots, v_{r-1}\} = V^{\mathscr{C}_n}$,
- either $d_n(v_j) = d_n(v_k)$ and $v_j < v_k$ or $d_n(v_j) < d_n(v_k)$, for all j and k such that $j < k < r$.

We say that $v \in V^{\mathscr{C}_n}$ has index j if $R^n(j) = v$.

Hit number. The hit number h^n gives us the number of vertices whose index has not changed. Let $R^n = v_0 \dots v_{r-1}$ as above. The value h^n is the maximum number $\le r$ such that $d_n(v_j) < n$ for $j < h$. In other words, the hit number gives us the length of the longest prefix of R^n which is a prefix of R^{n-1}.

We need one more definition before we can state the crucial property of Muller–Schupp trees. Let $\mathscr{M}$ be any Muller–Schupp tree as above and m the minimum index of a vertex with left labeling 2 (it is left extending). If such a vertex does not exist, then, by convention, $m = n$. We define $\pi(\mathscr{M})$, the priority of $\mathscr{M}$, as follows. If $m < h$, then $\pi(\mathscr{M}) = 2m$, and else $\pi(\mathscr{M}) = 2h + 1$.

Lemma 2.13. Let $\mathscr{A}$ be a Büchi automaton, u a word over the same alphabet, and $\mathscr{M}_0, \mathscr{M}_1, \dots$ a sequence of Muller–Schupp trees satisfying the above requirements (leaf labeling, left labeling, latest appearance record, hit number). Let $p^\infty = \mathrm{val}_\pi(\mathscr{M}_0\mathscr{M}_1 \dots)$, that is, the smallest value occurring infinitely often in $\pi(\mathscr{M}_0)\pi(\mathscr{M}_1) \dots$. Then the following are equivalent:

(A) $\mathscr{A}$ accepts u.
(B) p^∞ is even.

Proof. For the implication from (A) to (B), let v be a vertex as guaranteed by (B) in Lemma 2.11. There must be some n and some number i such that $v = R^n(i) = R^{n+1}(i) = \ldots$ and $R^n[0,i] = R^{n+1}[0,i] = \ldots$. This implies $h^j > i$ for all $j \geq n$, which means that if p^j is odd for some $j \geq n$, then $p^j > 2i$. In addition, since v is left extending for infinitely many j, we have $p^j \leq 2i$ and even for infinitely many j. Thus, p^∞ is an even value (less than or equal to $2i$).

For the implication from (B) to (A), assume that p^∞ is even and n is such that $p^j \geq p^\infty$ for all $j \geq n$. Let $n' \geq n$ be such that $p^{n'} = p^\infty$ and let v be the vertex of $\mathscr{C}_{n'}$ which gives rise to $p^{n'}$ (left extending with minimum index). Then $v \in V^{\mathscr{C}_j}$ for all $j \geq n'$ and v has the same index in all these $\mathscr{C}_j$. That is, whenever $p^j = p^\infty$ for $j \geq n'$, then v is left extending. So (B) from Lemma 2.11 is satisfied and we can conclude that u is accepted by $\mathscr{A}$. Q.E.D.

2.3.4 The Determinization Construction

In order to arrive at a parity automaton, we only need to convince ourselves that a deterministic automaton can produce a sequence $\mathscr{M}_0, \mathscr{M}_1, \ldots$ as above. We simply describe an appropriate transition function, that is, we assume a Muller–Schupp tree $\mathscr{M}$ and a letter a are given, and we describe how $\mathscr{M}'$ is obtained from $\mathscr{M}$ such that if $\mathscr{M} = \mathscr{M}_n$ and $a = u(n)$, then $\mathscr{M}' = \mathscr{M}_{n+1}$. This is, in principle, straightforward, but it is somewhat technical. One of the issues is that during the construction of $\mathscr{M}'$ we have trees with more than $3\,|Q|$ vertices. This is why we assume that we are also given a set W of $2\,|Q|$ vertices disjoint from $\{0, \ldots, 3\,|Q| - 1\}$.

A Muller-Schupp tree $\mathscr{M}'$ is called an a-successor of $\mathscr{M}$ if it is obtained from $\mathscr{M}$ by applying the following procedure.

(i) Let $V_{new} = \{0, \ldots, 3\,|Q| - 1\} \setminus V^{\mathscr{C}}$.

(ii) To each leaf v, add a left and right successor from W.

Let $w_0, \ldots, w_{2r-1}$ be the sequence of these successors in the order from left to right.

(iii) For $i = 0$ to $r - 1$, do:

(a) Let v be the predecessor of w_{2i} and $Q' = l(w_0) \cup \cdots \cup l(w_{2i-1})$.

(b) Set $l_q(w_{2i}) = \Delta(l_q(v), a) \cap F \setminus Q'$ and $l_q(w_{2i+1}) = \Delta(l_q(v), a) \setminus (F \cup Q')$.

(c) Set $l_q(w_{2i}) = 2$ and $l_q(w_{2i+1}) = 0$.

(iv) Remove the leaf labels from the old leaves, that is, make l_q undefined for the predecessors of the new leaves. Mark every leaf which has label $\varnothing$. Recursively mark every vertex whose two successors are marked. Remove all marked vertices.

(v) Replace every nontrivial segment by its first vertex, and set its left labeling to

  (a) 2 if one of the other vertices of the segment is labeled 1 or 2,

  (b) 0 if each vertex of the segment is labeled 0, and

  (c) 1 otherwise.

(vi) Replace the vertices from W by vertices from V_{new}.

(vii) Let R_0 be obtained from R by removing all vertices from $V^{\mathscr{C}} \setminus V^{\mathscr{C}'}$ from R and let R_1 be the sequence of all elements from $V^{\mathscr{C}'} \setminus V^{\mathscr{C}}$ according to the order $<$ on V. Then $R' = R_0 R_1$.

(viii) Let $h' \leq |R|$ be the maximal number such that $R(i) = R'(i)$ for all $i < h'$.

The full determinization construction is given in Figure 9. Summing up, we can state:

Theorem 2.14. (McNaughton-Safra-Piterman, [Büc62, Saf88, Pit06]) Let $\mathscr{A}$ be a Büchi automaton with n states. Then $\mathscr{A}^{\mathrm{det}}$ is an equivalent deterministic parity automaton with $2^{\theta(n \log n)}$ states and $2n+1$ different priorities.

Proof. The proof of the correctness of the construction described in Figure 9 is obvious from the previous analysis. The claim about the size of the resulting automaton can be established by simple counting arguments. Q.E.D.

The previous theorem enables to determine the expressive power of WS1S:

Corollary 2.15. There exists an effective procedure that given an S1S formula $\varphi = \varphi(V_0, \ldots, V_{m-1})$ produces a formula ψ such that $\mathscr{L}(\varphi) = \mathscr{L}^{\mathrm{w}}(\psi)$. In other words, every S1S formula is equivalent to a WS1S formula.

Sketch of proof. Given such a formula φ, one first uses Büchi's Theorem to construct a Büchi automaton $\mathscr{A}$ such that $\mathscr{L}(\varphi) = \mathscr{L}(\mathscr{A})$. In a second step, one converts $\mathscr{A}$ into an equivalent deterministic parity automaton $\mathscr{B}$, using the McNaughton–Safra–Piterman Theorem. The subsequent step is the crucial one. Assume $Q' = \{q_0, \ldots, q_{n-1}\}$ and, for every $u \in [2]^\omega_m$, let r_u be the (unique!) run of $\mathscr{B}$ on u. For every $i < n$, one constructs a formula $\psi_i = \psi_i(x)$ such that $u, j \models \psi_i(x)$ if and only if $r_u(j) = q_i$ for $u \in [2]^\omega_m$

Let $\mathscr{A}$ be a Büchi automaton. The deterministic parity automaton $\mathscr{A}^{det}$ is defined by

$$\mathscr{A}^{det} = (A, M, \mathscr{M}_I, \delta, \pi)$$

where

- M is the set of all Muller–Schupp trees over Q,
- $\mathscr{M}_I$ is the Muller–Schupp tree with just one vertex and leaf label Q_I,
- δ is such that $\delta(\mathscr{M}, a)$ is an a-successor of $\mathscr{M}$ (as defined above), and
- π is the priority function as defined for Muller–Schupp trees.

FIGURE 9. Determinization of a Büchi automaton

and $j \in \omega$. These formulas can be built as in the proof of part 2 of Büchi's Theorem, except that one can restrict the sets X_i to elements $\leq j$, so weak quantification is enough. Finally, the formulas $\psi_i(x)$ are used to express acceptance. Q.E.D.

2.4 The Büchi–Landweber Theorem

The last problem remaining from the problems listed at the beginning of this section is realizability, also known as Church's problem [Chu60, Chu63]. In our context, it can be formalized more precisely as follows.

For letters $a \in [2]_m$ and $b \in [2]_n$, we define $a^\frown b \in [2]_{m+n}$ by $(a^\frown b)_{[i]} = a_{[i]}$ for $i < m$ and $a^\frown b_{[i]} = b_{[i-m]}$ for i with $m \leq i < m+n$. Similarly, when u and v are words of the same length over $[2]_n^\infty$ and $[2]_n^\infty$, respectively, then $u^\frown v$ is the word over $[2]_{m+n}$ with the same length defined by $(u^\frown v)(i) = u(i)^\frown v(i)$ for all $i < |u|$. Realizability can now be defined as follows: Given a formula

$$\varphi = \varphi(X_0, \ldots, X_{m-1}, Y_0, \ldots, Y_{n-1}),$$

determine whether there is a function $f\colon [2]_m^+ \to [2]_n$ such that $u^\frown v \models \varphi$ holds for every $u \in [2]_m^\omega$ and $v \in [2]_n^\omega$ defined by $v(i) = f(u[0, i])$.

Using the traditional terminology for decision problems, we say that φ is an instance of the realizability problem, f is a solution if it has the desired property, and φ is a positive instance if it has a solution.

Observe that the function f represents the device that produces the output in Figure 1: After the device has read the sequence $a_0 \dots a_r$ of bit vectors (with m entries each), it outputs the bit vector $f(a_0 \dots a_r)$ (with n entries).

In the above definition of realizability, we do not impose any bound on the complexity of f. In principle, we allow f to be a function which is not computable. From a practical point of view, this is not very satisfying. A more realistic question is to ask for a function f which can be realized by a finite-state machine, which is a tuple

$$\mathscr{M} = (A, B, S, s_I, \delta, \lambda)$$

where A is an input alphabet, B is an output alphabet, S is a finite set of states, $s_I \in S$ the initial state, $\delta\colon S \times A \to S$ the transition function, and $\lambda\colon S \to B$ the output function. To describe the function realized by $\mathscr{M}$ we first define $\delta^*\colon A^* \to S$ by setting $\delta(\varepsilon) = s_I$ and $\delta^*(ua) = \delta(\delta^*(u), a)$ for all $u \in A^*$ and $a \in A$. The function realized by $\mathscr{M}$, denoted $f_{\mathscr{M}}$, is defined by $f_{\mathscr{M}}(u) = \lambda(\delta(s_I, u))$ for every $u \in A^+$.

A solution f of an instance of the realizability problem is called a finite-state solution if it is realized by a finite-state machine.

Finite-state realizability is the variant of realizability where one is interested in determining whether a finite-state solution exists. We later see that there is no difference between realizability and finite-state realizability.

Several approaches have been developed to solving realizability; we follow a game-based approach. It consists of the following steps: We first show that realizability can be viewed as a game and that solving realizability means deciding who wins this game. We then show how the games associated with instances of the realizability problem can be reduced to finite games with a standard winning objective, namely the parity winning condition. Finally, we use known results on finite games with parity winning conditions to prove the desired result.

2.4.1 Game-Theoretic Formulation

There is a natural way to view the realizability problem as a round-based game between two players, the environment and the (device) builder. In each round, the environment first provides the builder with an input, a vector $a \in [2]_m$, and then the builder replies with a vector $b \in [2]_n$, resulting in a combined vector $a^\frown b$. In this way, an infinite sequence of vectors is constructed, and the builder wins the play if this sequence satisfies the given S1S formula. Now, the builder has a winning strategy in this game if

and only if the instance of the realizability problem we are interested in is solvable.

We make this more formal in what follows. A game is a tuple

$$\mathscr{G} = (P_0, P_1, p_I, M, \Omega)$$

where P_0 is the set of positions owned by Player 0, P_1 is the set of positions owned by Player 1 (and disjoint from P_0), $p_I \in P_0 \cup P_1$ is the initial position, $M \subseteq (P_0 \cup P_1) \times (P_0 \cup P_1)$ is the set of moves, and $\Omega \subseteq P^\omega$ is the winning objective for Player 0. The union of P_0 and P_1 is the set of positions of the game and is denoted by P.

A play is simply a maximal sequence of positions which can be obtained by carrying out moves starting from the initial position, that is, it is a word $u \in P^+ \cup P^\omega$ such that $u(0) = p_I$, $(u(i), u(i+1)) \in M$ for every $i < |u|$, and if $|u| < \omega$, then there is no p such that $(u(*), p) \in M$. This can also be thought of as follows. Consider the directed graph (P, M), which is called the game graph. A play is simply a maximal path through the game graph (P, M) starting in p_I.

A play u is a win for Player 0 if $u \in \Omega \cup P^* P_1$, else it is a win for Player 1. In other words, if a player cannot move he or she loses right away.

A strategy for Player α are instructions for Player α how to move in every possible situation. Formally, a strategy for Player α is a partial function $\sigma\colon P^* P_\alpha \to P$ which

(i) satisfies $(u(*), \sigma(u)) \in M$ for all $u \in \mathrm{dom}(\sigma)$ and

(ii) is defined for every $u \in P^* P_\alpha \cap p_I P^*$ satisfying $u(i+1) = \sigma(u[0,i])$ for all $i < |u| - 1$ where $u(i) \in P_\alpha$.

Observe that these conditions make sure that a strategy is defined when Player α moves according to it. A play u is consistent with a strategy σ if $u(i+1) = \sigma(u[0,i])$ for all i such that $u(i) \in P_\alpha$. A strategy σ is called a winning strategy for Player α if every play consistent with σ is a win for Player α. We then say that Player α wins the game.

The analogue of a finite-state solution is defined as follows. A strategy σ for Player α is finite-memory if there exists a finite set C, called memory, an element $m_I \in C$, the initial memory content, a function $\mu\colon C \times P \to C$, called update function, and a function $\xi\colon C \times P_\alpha \to P$ such that $\sigma(u) = \xi(\mu^*(u), u(*))$ for every $u \in \mathrm{dom}(\sigma)$, where μ^* is defined as δ^* above. That is, the moves of Player α depend on the current memory contents and the current position.

An even stronger condition than being finite-state is being memoryless. A strategy σ is memoryless if it is finite-state for a memory C which is a singleton set. As a consequence, if σ is memoryless, then $\sigma(up) = \sigma(u'p)$

Let $\varphi = \varphi(X_0, \ldots, X_{m-1}, Y_0, \ldots, Y_{n-1})$ be an S1S formula. The game $\mathscr{G}[\varphi]$ is defined by

$$\mathscr{G}[\varphi] = ([2]_m, \{p_I\} \cup [2]_n, p_I, M, \Omega)$$

where p_I is some initial position not contained in $[2]_m \cup [2]_n$ and

$$M = ([2]_m \times [2]_n) \cup ((\{p_I\} \cup [2]_n) \times [2]_m),$$
$$\Omega = \{p_I u_0 v_0 \ldots : (u_0 \frown v_0)(u_1 \frown v_1) \ldots \models \varphi\}.$$

FIGURE 10. Game for a realizability instance

for all $u, u' \in P^*$ with $up, u'p \in \mathrm{dom}(\sigma)$. So in this case, we can view a strategy as a partial function $P_\alpha \to P$. In fact, we use such functions to describe memoryless strategies.

We can now give the game-theoretic statement of the realizability problem. For an instance φ, consider the game $\mathscr{G}[\varphi]$ described in Figure 10.

Lemma 2.16. Let $\varphi = \varphi(X_0, \ldots, X_{m-1}, Y_0, \ldots, Y_{n-1})$ be an S1S formula. Then the following are equivalent:
(A) The instance φ of the realizability problem is solvable.
(B) Player 0 wins the game $\mathscr{G}[\varphi]$.
Moreover, φ is a positive instance of finite-state realizability if and only if Player 0 has a finite-memory winning strategy in $\mathscr{G}[\varphi]$.

Proof. For the implication from (A) to (B), let $f\colon [2]_m^+ \to [2]_n$ be the solution of an instance φ of the realizability problem. We define a partial function $\sigma\colon p_I([2]_m[2]_n)^*[2]_m \to [2]_n$ by setting $\sigma(p_I a_0 b_1 \ldots b_{r-1} a_r) = f(a_0 \ldots a_r)$ where $a_i \in [2]_m$ for $i \leq r$ and $b_j \in [2]_n$ for $j < r$. It is easy to see that σ is a winning strategy for Player 0 in $\mathscr{G}[\varphi]$. Conversely, a winning strategy σ for Player 0 can easily be transformed into a solution of the instance φ of the realizability problem.

To prove the additional claim, one simply needs to observe that the transformations used in the first part of the proof convert a finite-state solution into a finite-memory strategy, and vice versa. The state set of the finite-state machine used to show that a solution to the realizability problem is finite-state can be used as memory in a proof that the winning strategy constructed above is finite-memory, and vice versa. Q.E.D.

Let $\mathscr{G}$ be a game and $\mathscr{A}$ a deterministic parity automaton with alphabet P such that $\mathscr{L}(\mathscr{A}) = \Omega$. The expansion of $\mathscr{G}$ by $\mathscr{A}$ is the game

$$\mathscr{G} \times \mathscr{A} = (P_0 \times Q, P_1 \times Q, (p_I, q_I), M', \pi')$$

where

$$M' = \{((p,q),(p',\delta(q,p'))) \colon q \in Q \wedge (p,p') \in \Delta\}$$

and $\pi'((p,q)) = \pi(q)$ for all $p \in P$ and $q \in Q$.

FIGURE 11. Product of a game with a deterministic parity automaton

In our definition of game, there is no restriction on the winning objective Ω, but since we are interested in winning objectives specified in S1S, we focus on parity winning conditions—remember that every S1S formula can be turned into a deterministic parity automaton. It will turn out that parity conditions are particularly apt to an algorithmic treatment while being reasonably powerful.

2.4.2 Reduction to Finite Parity Games

A winning objective Ω of a game $\mathscr{G}$ is a parity condition if there is a natural number n and a function $\pi \colon P \to n$ such that $u \in \Omega$ iff $\mathrm{val}_\pi(u) \bmod 2 = 0$ for all $u \in P^\omega$. If this is the case, we replace Ω by π and speak of a parity game.

We next show that if Ω is a winning objective and $\mathscr{A}$ a deterministic parity automaton such that $\mathscr{L}(\mathscr{A}) = \Omega$, then we can "expand" a game $\mathscr{G}$ with winning objective Ω into a parity game, simply by running $\mathscr{A}$ in parallel with the moves of the players. The respective product construction is given in Figure 11.

Lemma 2.17. Let $\mathscr{G}$ be a finite game and $\mathscr{A}$ a deterministic parity automaton such that $\mathscr{L}(\mathscr{A}) = \Omega$. Then the following are equivalent:
(A) Player 0 wins $\mathscr{G}$.
(B) Player 0 wins $\mathscr{G} \times \mathscr{A}$.
Moreover, there exists a finite-memory winning strategy for Player 0 in $\mathscr{G}$ iff there exists such a strategy in $\mathscr{G} \times \mathscr{A}$.

Proof. The proof is straightforward. We transform a winning strategy for Player 0 in $\mathscr{G}$ into a winning strategy for Player 0 in $\mathscr{G} \times \mathscr{A}$ and vice versa.

First, we define u^δ for every $u \in P^*$ to be a word of the same length where the letters are determined by $u^\delta(i) = (u(i), \delta^*(q_I, u[0,i]))$ for every $i < |u|$.

Let $\sigma\colon P^*P_0 \to P$ be a winning strategy for Player 0 in $\mathscr{G}$. We transform this into $\sigma'\colon (P \times Q)^*(P_0 \times Q) \to P \times Q$ by letting $\sigma'(u^\delta) = \sigma(u)$ for every $u \in \mathrm{dom}(\sigma)$. It is easy to check that this defines a strategy and that this strategy is indeed winning.

Given a winning strategy $\sigma'\colon (P \times Q)^*(P_0 \times Q) \to P \times Q$, we define a winning strategy $\sigma\colon P^*P_0 \to P$ for Player 0 simply by forgetting the second component of the position. That is, for every u such that $u^\delta \in \mathrm{dom}(\sigma')$ we set $\sigma(u) = \sigma'(u^\delta)$. Observe that this does not lead to any ambiguities, that is, σ is well-defined, because $\mathscr{A}$ is a deterministic automaton. It is easy to check that this defines a strategy and that this strategy is indeed winning.

If we have a finite-memory strategy σ for $\mathscr{G}$, say with memory C, we can use the same memory C and a modified update function to show that σ' as defined above is finite-state. Conversely, if we have a finite-memory strategy σ', say with memory C, we can use memory $Q \times C$ to show that σ as constructed above is finite-memory, too. Q.E.D.

Corollary 2.18. Let $\varphi = \varphi(X_0, \ldots, X_{m-1}, Y_0, \ldots, Y_{n-1})$ be an instance of the realizability problem for S1S and $\mathscr{A}$ a deterministic parity automaton recognizing $\mathscr{L}(\varphi)$. Then the following are equivalent:
(A) The instance φ of the realizability problem is solvable.
(B) Player 0 wins the game $\mathscr{G} \times \mathscr{A}$.
Moreover, if Player 0 has a finite-memory winning strategy in $\mathscr{G} \times \mathscr{A}$, then φ has a finite-state solution.

Using the fact that it can be determined effectively whether Player 0 wins a finite parity game (see Theorem 2.20 below), we obtain:

Theorem 2.19 (Büchi-Landweber, [BL69])**.** The realizability problem is decidable for S1S.

2.4.3 Background on Games

In this section, we provide background on games, which we already used to solve Church's problem and which we need in various places.

Since plays of games may be infinite, it is not at all clear whether in a given game one of the two players has a winning strategy, that is, whether the game has a winner. When this is the case one says that the game is determined. It is said to be memoryless determined if there exists a memoryless winning strategy.

Theorem 2.20 (Emerson-Jutla-Mostowski, [EJ91b, Mos91])**.** Every parity game is memoryless determined.

That every parity game is determined follows immediately from a result by Martin [Mar75].

For S1S realizability it is enough to know that the winner in a parity game can be effectively determined. In a later section, we need to know more about the computational complexity of this problem, in particular, we need to know how it depends on the number of priorities occurring in a game:

Theorem 2.21 (Jurdziński, [Jur00])**.** Every parity game with n positions, m edges, and at most d different priorities in every strongly connected component of its game graph can be decided in time $O(n + mn^{\lfloor d/2 \rfloor})$ and an appropriate memoryless winning strategy can be computed within the same time bound.

2.5 Notes

Büchi's Theorem has been the blueprint for many theorems characterizing monadic second-order logic by automata. The most important theorem to mention is Rabin's Theorem [Rab69], which extends Büchi's Theorem to the monadic theory of two successor relations and is the subject of the next section. Other early results, besides the Büchi–Elgot–Trakthenbrot theorem and Büchi's Theorem, are a result by Büchi [Büc65] on ordinals and a result by Doner [Don70] (see also Thatcher and Wright [TW68]), which characterizes monadic second-order logic over finite trees in terms of automata and allows to prove that the weak monadic theory of two successor relations is decidable. Later results deal, for instance, with finite and infinite traces (certain partial orders) [Tho90b, EM93], see also [DG], pictures (matrices with letters as entries) [GRST96], see also [GR, MS], and weighted automata [DG05]. In some of these cases, the proofs are much harder than for S1S and Büchi automata.

When establishing a characterization of automata in terms of monadic second-order logic, proving part 2—the description of the behavior of an automaton by a monadic second-order formula—is straightforward very often and leads to existential monadic second-order formulas, just as for S1S. The other direction—from full monadic second-order logic to automata—fails, however, for various automaton models because closure under complementation (negation) cannot be shown. In such cases, a partial result can sometimes nevertheless be obtained by showing that every existential monadic second-order formula can be translated into an automaton. This is, for instance, the case for pictures [GRST96], see also [MS].

Büchi's Theorem characterizes monadic second-order logic in terms of finite-state automata on infinite words. It is only natural to ask whether there are fragments of monadic second-order logics or other logics similar in expressive power to monadic second-order logic that can be characterized in

a comparable fashion. We have already seen that the existential fragment of S1S has the same expressive power as S1S, but one can prove that first-order logic with ordering (and successor) or with successor only is strictly less expressive than S1S. The first of the two logics can be characterized just as in the case of finite words as defining exactly

(i) the star-free languages of infinite words,

(ii) the languages expressible in linear-time temporal logic, and

(iii) the languages of infinite words which are recognized by counter-free automata

[Kam68, Tho79, Per86, Zuc86], the second can be characterized as the weak version of locally threshold testability [Tho82].

Ever since Büchi's seminal work automata on infinite words and formal languages of infinite words have been a major topic in research, motivated both from a mathematical and a computer science perspective. There have been many (successful) attempts to adapt the facts known from classical automata theory and the classical theory of formal languages to the setting with infinite words, for instance, regular expressions were extended to ω-regular expressions and the algebraic theory of regular languages was extended to an algebraic theory of ω-regular languages. But there are also new issues that arise for infinite words, which are essentially irrelevant for finite words. For example, the set of infinite words over a given alphabet can easily be turned into a topological space and it is interesting to study how complex languages are that can be recognized by finite-state automata.

One particularly interesting issue are the different types of acceptance conditions that are available for automata on infinite words. In our exposition, we work with Büchi and parity acceptance, but there are many more acceptance conditions which are suggested and widely used throughout the literature. The most prominent are: Streett [Str82], Rabin [Rab69], and Muller conditions [Mul63]. An important question regarding all these different acceptance conditions is which expressive power they have, depending on whether they are used with deterministic or nondeterministic automata. It turns out that when used with nondeterministic automata all the aforementioned conditions are not more powerful than nondeterministic Büchi automata and when used with deterministic automata they are all as powerful as deterministic parity automata. In other words, each of the three conditions is just as good as the parity condition. Given McNaughton's Theorem, this is not very difficult to show. In almost all cases, asymptotically optimal conversions between the various conditions are known [Saf88]. Recent improvements are due to Yan [Yan06].

It is not only the type of acceptance condition that can be varied, but also the type of "mode". In this section, we have dealt with deterministic and nondeterministic automata. One can either look for

(i) modes in between or

(ii) modes beyond nondeterminism.

As examples for (i) we mention unambiguous automata [Arn83], which are Büchi automata which admit at most one accepting run for each word, and prophetic automata [CPP], which are Büchi automata with the property that there is exactly one run on each word (besides partial runs that cannot be continued), be it accepting or not.

Examples for (ii) are alternating automata on infinite words, which are explained in detail in Section 4. Since they are, in principle, stronger than nondeterministic automata, they often allow for more succinct representations, which is why they have been studied extensively from a practical and complexity-theoretic point of view. Moreover, they can often be used to make automata-theoretic constructions more modular and transparent and help to classify classes of languages. For instance, the Kupferman–Vardi complementation construction for Büchi automata uses what are called weak alternating automata as an intermediate model of automaton, see also [Tho99].

As can be seen from the Büchi–Landweber theorem, games of infinite duration are intimately connected with the theory of automata on infinite words. This becomes even more obvious as soon as alternating automata come into the picture, because they can be viewed as defining families of games in a uniform fashion. These games play a similar role in the theory of automata on infinite trees, as will be explained in the next section. Regardless of this, these games are interesting in their own right and there is an extensive literature on them. One of the major open problems is the computational complexity of parity games. The best upper bounds are that the problem is in UP $\cap$ co-UP, which is a result by Jurdziński [Jur98], that it can be solved by subexponential algorithms, see, for instance, [JPZ06], and polynomial time algorithms when the underlying game graphs belong to certain restricted classes of graphs, see, for instance, [BDHK06].

3 Monadic-Second Order Logic of Two Successors

Büchi's Theorem is a blueprint for Rabin's result on monadic second-order logic of two successors (S2S). The formulas of that logic are built just as S1S formulas are built, except that there are two successor relations and not only one. More precisely, while in S1S the atomic formulas are of the form $x \in X$ and $\mathrm{suc}(x, y)$ only, in S2S the atomic formulas are of the form $x \in X$, $\mathrm{suc}_0(x, y)$, and $\mathrm{suc}_1(x, y)$, where $\mathrm{suc}_0(x, y)$ and $\mathrm{suc}_1(x, y)$ are read as "y is

the left successor of x" and "y is the right successor of x", respectively. S2S formulas are interpreted in the full binary tree $\mathcal{T}_{bin}$.

As a first simple example, we design a formula with one free set variable X which holds true if and only if the set assigned to X is finite. This can be expressed by saying that on every branch there is a vertex such that the subtree rooted at this vertex does not contain any element from X. This leads to:

$$\forall Y(\text{“}Y \text{ is a branch of the binary tree”} \rightarrow \exists y(y \in Y \wedge \forall z(y \leq z \rightarrow \neg z \in X))),$$

where $\leq$ is meant to denote the prefix order on the vertices of $\mathcal{T}_{bin}$. That Y is a branch of $\mathcal{T}_{bin}$ can easily be expressed as a conjunction of several simple conditions:

- Y is not empty, which can be stated as $\exists x(x \in Y)$,
- with each element of Y its predecessor (provided it exists) belongs to Y, which can be stated as $\forall x \forall y(y \in Y \wedge (\mathrm{suc}_0(x, y) \vee \mathrm{suc}_1(x, y)) \rightarrow x \in Y)$, and
- each element of Y has exactly one successor in Y, which can be stated as $\forall x \forall y \forall z(x \in Y \wedge \mathrm{suc}_0(x, y) \wedge \mathrm{suc}_1(x, z) \rightarrow (y \in Y \leftrightarrow z \notin Y))$.

To conclude the example, we define $x \leq y$ by stating that every successor-closed set containing x contains y as well:

$$\forall X(x \in X \wedge \forall z \forall z'(z \in X \wedge (\mathrm{suc}_0(z, z') \vee \mathrm{suc}_1(z, z')) \rightarrow z' \in X) \rightarrow y \in X).$$

Observe that we have a universally quantified set variable in this formula, whereas in Section 2 we use an existentially quantified set variable to define ordering for the natural numbers. In both situations, one can use either type of quantifier.

As a second example, we consider the property that on every branch there are only finitely many elements from X. This can be specified by:

$$\forall Y(\text{“}Y \text{ is a branch of the binary tree”} \rightarrow \exists y(y \in Y \wedge \forall z(x \leq z \wedge z \in Y \rightarrow \neg z \in X))),$$

using the same auxiliary formulas from above.

The most important question about S2S is whether satisfiability is decidable. A positive answer to this question implies decidability of the monadic

second-order theory of the binary tree and a number of related theories as Rabin showed in his 1969 paper [Rab69].

That satisfiability of an S2S formula is decidable can, in principle, be shown in the same way as the analogous statement for S1S: One first proves that every S2S formula can be translated into an equivalent automaton—this time a tree automaton—and then shows that emptiness for the automata involved is decidable. This is the approach that Rabin took in [Rab69], and which we follow here, too.

3.1 Rabin's Theorem

In his original paper [Rab69] Rabin used what we nowadays call Rabin tree automata to characterize S2S. We use the same model of tree automaton but with a simpler acceptance condition, the parity acceptance condition, which we also use in the context of S1S.

It is not clear right away how a tree automaton model should look like, but it turns out that it is reasonable to envision a tree automaton as follows. Starting in an initial state at the root of the tree the automaton splits up into two copies, one which proceeds at the left successor of the root and one which proceeds at the right successor of the root. The states which are assumed at these vertices are determined by the initial state and the label of the root. Then, following the same rules, the copy of the automaton residing in the left successor of the root splits up into two copies which proceed at the left successor of the left successor of the root and the right successor of the left successor of the root, and so on. In this way, every vertex of the tree gets assigned a state, and a tree is accepted if the state labeling of each branch satisfies the acceptance condition.

Formally, a parity tree automaton is a tuple

$$\mathscr{A} = (A, Q, q_I, \Delta, \pi),$$

where A, Q, and π are as with parity (word) automata (see Section 2.3.1), q_I is an initial state instead of a set of initial states, and Δ is a transition relation satisfying $\Delta \subseteq Q \times A \times Q \times Q$. Such an automaton runs on full A-labeled binary trees which are given implicitly. A run of $\mathscr{A}$ on a binary tree $t\colon 2^* \to A$ is a binary tree $r\colon 2^* \to Q$ such that $(r(u), t(u), r(u0), r(u1)) \in \Delta$ for all $u \in 2^*$. It is accepting if for every infinite branch $u \in 2^\omega$ its labeling satisfies the parity condition, that is, if $\mathrm{val}_\pi(r(u(0))r(u(1))\dots) \bmod 2 = 0$.

As an example, consider the set L of all binary trees over $\{0,1\}$ with only finitely many vertices labeled 1 on each branch, which is very similar to the second property discussed above. It is straightforward to construct a parity tree automaton that recognizes L. The main idea is to use two states, q_0 and q_1, to indicate which label has just been read and to use the parity condition to check that on every path there are only finitely many

vertices labeled q_1. In other words, we have $A = \{0, 1\}$, $Q = \{q_I, q_0, q_1\}$, $\Delta = \{(q, a, q_a, q_a) \colon a \in A, q \in Q\}$, $\pi(q_I) = 0$, and $\pi(q_a) = a + 1$ for $a \in A$.

Rabin, in [Rab69], proved a complete analogue of Büchi's theorem. We state Rabin's Theorem using the same notation as in the statement of Büchi's Theorem, which means, for instance, that we write $\mathscr{L}(\mathscr{A})$ for the set of all trees accepted by a parity tree automaton $\mathscr{A}$.

Theorem 3.1 (Rabin, [Rab69])**.**

(i) There exists an effective procedure that given an S2S formula $\varphi = \varphi(V_0, \ldots, V_{m-1})$ outputs a parity tree automaton $\mathscr{A}$ such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\varphi)$.

(ii) There exists an effective procedure that given a parity tree automaton $\mathscr{A}$ over an alphabet $[2]_m$ outputs a formula $\varphi = \varphi(V_0, \ldots, V_{m-1})$ such that $\mathscr{L}(\varphi) = \mathscr{L}(\mathscr{A})$.

To prove part 2 one can follow the same strategy as with S1S: One simply constructs a formula that describes an accepting run of a given parity tree automaton. Proofs of part 2 of Theorem 3.1 can also be carried out as with S1S: One uses a simple induction on the structure of the formula. The induction base and all but one case to be considered in the inductive step are almost straightforward. The difficult step is—just as with Büchi automata—negation. One has to show that the complement of a tree language recognized by a parity tree automaton can be recognized by a parity tree automaton, too. This result, also known as Rabin's complementation lemma, can be proved in different ways. We present a proof which, in spirit, is very similar to what can be found in Büchi's [Büc77] and Gurevich and Harrington's [GH82] work. At its heart, there is a game-theoretic description of acceptance (Section 3.2). The complementation construction itself has the determinization from Theorem 2.14 built in (Section 3.3).

3.2 The Automaton-Pathfinder Game

Let $\mathscr{A}$ be a parity tree automaton as above and $t \colon 2^* \to A$ a binary tree. We consider a parity game where one can think of Player 0 as proving to Player 1 that t is accepted by $\mathscr{A}$, as follows. The game starts at the root of the tree and Player 0 suggests a transition which works at the root of the tree, which means it must start with the initial state and it must show the symbol the root is labeled with. Then Player 1 chooses the left or right successor of the root, say she chooses the left successor. Now it's Player 0's turn again. He must choose a transition which works for the left successor, which means it must start with the state chosen for the left successor in the transition chosen in the previous round and it must show the symbol the left successor is labeled with. Then Player 1 chooses one of the two successors,

Let $\mathscr{A}$ be a parity tree automaton and $t\colon 2^* \to A$ a tree over the same alphabet. The automaton-pathfinder game for $\mathscr{A}$ and t is the parity game $\mathscr{G}[\mathscr{A}, t]$ defined by

$$\mathscr{G}[\mathscr{A}, t] = (2^* \times Q, 2^* \times \Delta, (\varepsilon, q_I), M_0 \cup M_1, \pi')$$

where

- for every word $u \in 2^*$, state $q \in Q$, and $(q, t(u), q_0, q_1) \in \Delta$, the move $((u, q), (u, (q, t(u), q_0, q_1))$ belongs to M_0,
- for every word $u \in 2^*$, transition $(q, t(u), q_0, q_1) \in \Delta$, and $i < 2$, the move $((u, (q, t(u), q_0, q_1), (ui, q_i))$ belongs to M_1, and
- $\pi'((u, q)) = \pi(q)$ for all $u \in 2^*$ and $q \in Q$.

FIGURE 12. Automaton-pathfinder game

and so on. As the play proceeds, a sequence of transitions is constructed. Player 0 wins this play when the respective sequence of the source states of the transitions satisfies the parity condition.

The precise definition of the parity game is given in Figure 12. Observe that for convenience the priority function is only partially defined. This does not cause any problems since there is an infinite number of vertices with priorities assigned to them on every path through the game graph.

Lemma 3.2 (Gurevich-Harrington, [GH82])**.** Let $\mathscr{A}$ be a parity tree automaton and $t\colon 2^* \to A$ a tree over the same alphabet. Then the following are equivalent:
(A) $\mathscr{A}$ accepts t.
(B) Player 0 wins $\mathscr{G}[\mathscr{A}, t]$.

Proof. For the implication from (A) to (B), we show how to convert an accepting run $r\colon 2^* \to Q$ of $\mathscr{A}$ on t into a winning strategy for Player 0 in $\mathscr{G}[\mathscr{A}, t]$. A strategy σ for Player 0 is defined on words of the form

$$u = (\varepsilon, q_0)(\varepsilon, \tau_0)(a_0, q_1)(a_0, \tau_1)(a_0 a_1, q_2) \dots (a_0 \dots a_{n-1}, q_n)$$

with $q_0 = q_I$, $q_i \in Q$ for $i \leq n$, $\tau_i \in \Delta$, and $a_i \in \{0, 1\}$ for $i < n$. For such a word u, we set $v_u = a_0 \dots a_{n-1}$. After the explanations given

Let $\mathscr{A}$ be a parity tree automaton. The emptiness game $\mathscr{G}_\varnothing[\mathscr{A}]$ is defined by

$$\mathscr{G}_\varnothing[\mathscr{A}] = (Q, \Delta, q_I, M_0 \cup M_1, \pi)$$

where

- for $q \in Q$ and $(q, a, q_0, q_1) \in \Delta$, the move $(q, (q, a, q_0, q_1))$ belongs to M_0,
- for every $(q, a, q_0, q_1) \in \Delta$ and $i < 2$, the move $((q, a, q_0, q_1), q_i)$ belongs to M_1.

FIGURE 13. Emptiness game for a parity tree automaton

above on how one should think of the game, it should be clear that we set $\sigma(u) = (u, (q_n, a, q^0, q^1))$ with $q^i = r(v_u i)$ for $i < 2$. It is easy to check that this defines a winning strategy, because every play conform with σ corresponds to a branch of the run r.

Conversely, assume σ is a winning strategy for Player 0 in the above game. Then an accepting run r can be defined as follows. For every partial play u as above which is conform with σ, we set $r(v_u) = q_n$. It is straightforward to check that this defines an accepting run, because every path in r corresponds to a play of $\mathscr{G}[\mathscr{A}, t]$ conform with σ. Q.E.D.

There is a similar parity game—the emptiness game—which describes whether a given parity tree automaton accepts some tree. In this game, when Player 0 chooses a transition, he does not need to take into account any labeling; he simply needs to make sure that the transition is consistent with the previously chosen transition. The full game is described in Figure 13.

With a proof similar to the one of Lemma 3.2, one can show:

Lemma 3.3. Let $\mathscr{A}$ be a parity tree automaton. Then $\mathscr{L}(\mathscr{A}) \neq \varnothing$ if and only if Player 0 wins $\mathscr{G}_\varnothing[\mathscr{A}]$.

Taking Theorem 2.21 into account, we obtain:

Corollary 3.4 (Rabin, [Rab69])**.** The emptiness problem for parity tree automata is decidable.

Rabin proved, in some sense, a stronger result, because he used tree automata with Rabin acceptance condition. As a further consequence, taking Rabin's Theorem into account, we note:

Corollary 3.5 (Rabin, [Rab69])**.** Satisfiability is decidable for S2S.

3.3 Complementation of Parity Tree Automata

We can finally turn to the question of how to arrive at a parity tree automaton for the complement of a set of trees accepted by a given parity tree automaton. We are given a parity tree automaton $\mathscr{A}$ and we want to construct a parity tree automaton which recognizes $\mathscr{L}(\mathscr{A})^C$, where for each tree language L over some alphabet A we write L^C for the set of all trees over A which do not belong to L.

We describe the entire construction as a composition of several simpler constructions. More precisely, we first show that for every tree in the complement there exists a tree over an enhanced alphabet which witnesses its membership to the complement. The second step is to prove that the set of these witnesses can be recognized by a universal parity tree automaton. The third step consists in showing that universal parity tree automaton can be converted into (ordinary nondeterministic) parity tree automata, and the final step shows how to reduce the enhanced alphabet to the real one.

The first key step is to combine the automaton-pathfinder game with memoryless determinacy. To this end, we encode memoryless (winning) strategies for the pathfinder in trees. Observe that a memoryless strategy for the pathfinder in $\mathscr{G}[\mathscr{A}, t]$ for some automaton $\mathscr{A}$ and some tree t is simply a (partial) function $\sigma\colon 2^* \times \Delta \to 2^* \times Q$. Since, by construction of $\mathscr{G}[\mathscr{A}, t]$, we always have $\sigma(u, (q, a, q_0, q_1)) = (ui, q_i)$ for some $i < 2$, we can view such a function as a function $2^* \times \Delta \to 2$, which, in turn, can be viewed as a function $2^* \to 2^\Delta$. The latter is simply a 2^Δ-labeled tree. When we further encode the given tree t in that tree, we arrive at the following notion of complement witness for $(A \times 2^\Delta)$-labeled trees.

Let $\mathscr{A}$ be a parity tree automaton and $t'\colon 2^* \to A \times 2^\Delta$ a tree. For simplicity, we write $t'(u)$ as (a_u, f_u) for every $u \in 2^*$. The tree t' is a complement witness if for every branch $u \in 2^\omega$ the following holds. If $\tau_0\tau_1 \cdots \in \Delta^\omega$ with $\tau_i = (q_i, a_{u[0,i)}, q_i^0, q_i^1)$ is such that $q_0 = q_I$ and $q_{i+1} = q_i^b$ where $b = f_{u[0,i)}(\tau_i)$ for every i, then $\mathrm{val}_\pi(q_0 q_1 \dots) \bmod 2 = 1$, that is, $q_0 q_1 \dots$ is not accepting with respect to π.

After the explanation given above, Theorem 2.20 now yields the lemma below, where we use the following notation. Given a tree $t'\colon 2^* \to A \times B$ for alphabet A and B, we write $\mathrm{pr}_0(t')$ for the tree defined by $\mathrm{pr}_0(t')(u) = \mathrm{pr}_0(t'(u))$ for every $u \in 2^*$, that is, we simply forget the second component of every label.

Lemma 3.6. Let $\mathscr{A}$ be a parity tree automaton and $t\colon 2^* \to A$ a tree over the same alphabet. Then the following are equivalent:
(A) $t \in \mathscr{L}(\mathscr{A})^C$.
(B) There is a complement witness t' for $\mathscr{A}$ such that $\mathrm{pr}_0(t') = t$. Q.E.D.

Using more notation, we can state the above lemma very concisely. First, we extend projection to tree languages, that is, given a tree language L over some alphabet $A \times B$, we write $\mathrm{pr}_0(L)$ for $\{\mathrm{pr}_0(t)\colon t \in L\}$. Second, given a parity tree automaton $\mathscr{A}$, we write $\mathscr{C}(\mathscr{A})$ for the set of all complement witnesses for $\mathscr{A}$. Then Lemma 3.6 simply states:

Remark 3.7. For every parity tree automaton $\mathscr{A}$,

$$\mathscr{L}(\mathscr{A})^{\mathsf{C}} = \mathrm{pr}_0(\mathscr{C}(\mathscr{A})).$$

So, clearly, once we have a parity tree automaton for $\mathscr{C}(\mathscr{A})$, we also have a parity tree automaton for $\mathscr{L}(\mathscr{A})^{\mathsf{C}}$, because we only need to omit the second component from the letters in the transition function to obtain the desired automaton.

It is not straightforward to find a parity tree automaton that recognizes $\mathscr{C}(\mathscr{A})$; it is much easier to show that $\mathscr{C}(\mathscr{A})$ is recognized by a universal parity tree automaton. Such an automaton is a tuple

$$\mathscr{A} = (A, Q, q_I, \Delta, \pi)$$

where A, Q, q_I, and π are as with parity tree automata and $\Delta \subseteq Q \times A \times 2 \times Q$. Let $t\colon 2^* \to A$ be a tree over A. A word $r \in Q^\omega$ is said to be a run for branch $u \in 2^\omega$ if $(r(i), t(u[0,i)), u(i), r(i+1)) \in \Delta$ for every i and $r(0) = q_I$. A tree is accepted if every $r \in Q^\omega$ which is a run for some branch satisfies the parity acceptance condition.

We can now rephrase Lemma 3.6 in terms of the new automaton model. We can express the complement of a tree language recognized by a parity tree automaton as the projection of a tree language recognized by a universal parity tree automaton. The latter is defined in Figure 14. Observe that the runs for the branches in this automaton correspond to the words $\tau_0\tau_1\ldots$ in the definition of complement witness.
We immediately obtain:

Remark 3.8. For every parity tree automaton $\mathscr{A}$,

$$\mathscr{C}(\mathscr{A}) = \mathscr{L}(\mathscr{A}^{\mathrm{cw}}).$$

To complete the description of the complementation procedure, we need to explain how a universal parity tree automaton can be converted into a parity tree automaton. One option for such a construction is depicted in Figure 16. It uses McNaughton's Theorem, namely that every nondeterministic Büchi automaton can be turned into a deterministic parity automaton. The idea is that the tree automaton follows all runs of a given branch at the same time by running a deterministic word automaton in parallel.

Let $\mathscr{A}$ be a parity tree automaton. The universal parity tree automaton $\mathscr{A}^{\mathrm{cw}}$ is defined by

$$\mathscr{A}^{\mathrm{cw}} = (A \times 2^{\Delta}, Q, q_I, \Delta', \pi + 1)$$

where $(q, (a, f), d, q') \in \Delta'$ if there exists $\tau = (q, a, q_0, q_1) \in \Delta$ such that $f(\tau) = d$ and $q_d = q'$, and where $\pi + 1$ stands for the priority function π' defined by $\pi'(q) = \pi(q) + 1$.

FIGURE 14. Universal parity tree automaton for complement witnesses

Let Q be a finite set of states and $\pi\colon Q \to \omega$ a priority function. Let $\mathscr{Q}$ be the alphabet consisting of all binary relations over Q. Then every word $u \in \mathscr{Q}^\omega$ generates a set of infinite words $v \in Q^\omega$, denoted $\langle u \rangle$, defined by

$$\langle u \rangle = \{v \in Q^\omega \colon \forall i((v(i), v(i+1)) \in u(i))\},$$

and called the set of paths through u, because one can think of $\langle u \rangle$ as the set of all infinite paths through the graph which is obtained by "collating" $u(0), u(1), \ldots$. We are interested in a deterministic parity automaton $\mathscr{A}[Q, \pi]$ which checks that all paths through a given u satisfy the given parity condition, that is, which has the following property. For every $u \in \mathscr{Q}^\omega$,

$$u \in \mathscr{L}(\mathscr{A}[Q, \pi]) \qquad \text{iff} \qquad \forall v(v \in \langle u \rangle \to \mathrm{val}_\pi(v) \bmod 2 = 0). \tag{1.1}$$

Using Theorem 2.14, such an automaton, which we call a generic automaton for Q and π, can easily be constructed, as can be seen from Figure 15. Observe that, by construction,

$$u \in \mathscr{L}(\mathscr{C}) \qquad \text{iff} \qquad \exists v(v \in \langle u \rangle \wedge \mathrm{val}_\pi(v) \bmod 2 = 1),$$

for every $u \in \mathscr{Q}^\omega$. We conclude:

Remark 3.9. Let Q be a finite state set and $\pi\colon Q \to \omega$ a priority function. Then 1.1 holds for every $u \in \mathscr{Q}^\omega$.

Given the generic automaton, it is now easy to convert universal tree automata into nondeterministic ones: One only needs to run the generic automaton on all paths. This is explained in detail in Figure 16.

Lemma 3.10. Let $\mathscr{A}$ be a universal parity tree automaton. Then $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}^{nd})$.

Let Q be a finite set of states and $\pi\colon Q \to \omega$ a priority function. Consider the parity word automaton

$$\mathscr{B} = (\mathscr{Q}, Q, Q_I, \Delta, \pi + 1)$$

where $\Delta = \{(q, R, q')\colon (q, q') \in R\}$. Let $\mathscr{C}$ be an equivalent Büchi automaton (Figure 7) and $\mathscr{D}$ a deterministic parity automaton equivalent to $\mathscr{C}$ (Figure 9). The automaton $\mathscr{A}[Q, \pi]$ is defined by

$$\mathscr{A}[Q, \pi] = (\mathscr{Q}, Q^{\mathscr{D}}, q_I^{\mathscr{D}}, \delta^{\mathscr{D}}, \pi + 1).$$

FIGURE 15. Generic automaton for state set and priority function

Proof. For convenience, we write $\mathscr{B}$ for $\mathscr{A}[Q^{\mathscr{A}}, \pi^{\mathscr{A}}]$.

First observe that for every $t\colon 2^* \to A$ there is exactly one run of $\mathscr{A}^{nd}$ on t. This is because Δ is such that for every $s \in S$ and $a \in A$, there is exactly one transition in Δ of the form (s, a, s_0, s_1). For a given t, let r_t denote this run. So in order to determine whether a tree is accepted by $\mathscr{A}^{nd}$, we only need to determine whether r_t is accepting. To this end, we consider a branch $w \in 2^\omega$ of this tree.

By construction of $\mathscr{A}^{nd}$, the labeling of w in r_t is the run of $\mathscr{B}$ on $u = R_0^u R_1^u \dots$ where $R_i^u = R_{t(w[0,i)),w(i)}$. So $\langle u \rangle$ is the set of runs of $\mathscr{A}$ on branch w. In view of Remark 3.9, this implies that w is accepting as a branch of r_t if and only if all runs of $\mathscr{A}$ on w are accepting. From this, the claim of the lemma follows immediately. Q.E.D.

This was also the last missing piece in the construction from a given parity tree automaton to a parity tree automaton for its complement:

Lemma 3.11 (Rabin, [Rab69])**.** There is an effective procedure that turns a given parity tree automaton $\mathscr{A}$ into a parity tree automaton $\mathscr{A}^C$ that recognizes the complement of the language recognized by $\mathscr{A}$. Q.E.D.

3.4 Notes

Rabin's Theorem is important from a mathematical (logical) point of view because it is a very strong decidability result and can as such be used to show the decidability of many theories, see, for instance, Rabin's original paper [Rab69] and the book [BGG97]. A very specific question to ask is how one can prove that the monadic second-order (or first-order) theory

Let $\mathscr{A}$ be a universal parity tree automaton and assume that the generic automaton for $Q^{\mathscr{A}}$ and $\pi^{\mathscr{A}}$ is given as $\mathscr{A}[Q^{\mathscr{A}}, \pi^{\mathscr{A}}] = (\mathscr{Q}^{\mathscr{A}}, S, s_I, \delta, \pi)$. The parity tree automaton $\mathscr{A}^{nd}$ is defined by

$$\mathscr{A}^{\mathrm{nd}} = (A, S, s_I, \Delta, \pi)$$

where for every $a \in A$ and $s \in S$,

$$\tau_{s,a} = (s, a, \delta(s, R_{a,0}), \delta(s, R_{a,1}))$$

with $R_{a,d} = \{(q, q') \colon (q, a, d, q') \in \Delta^{\mathscr{A}}\}$ for $d < 2$ and

$$\Delta = \{\tau_{s,a} \colon a \in A \wedge s \in S\}.$$

FIGURE 16. From universal to nondeterministic parity tree automata

of a certain structure is decidable using the fact that it is decidable for the binary tree. There is a wide spectrum of techniques that have been developed to this end and are explained in detail in [BCL], see also [Cau].

It may seem that the results proved for S1S and automata on infinite words extend to S2S and automata on infinite trees in a straightforward fashion. This is true in many respects, but there are important differences. Most importantly, it is neither true that every tree language recognized by a parity tree automaton can be recognized by a Büchi tree automaton nor is it true that WS2S is equally expressive as S2S. There is, however, an interesting connection between Büchi tree automata and WS2S: a set of trees is definable in WS2S if and only if it is recognized by a Büchi tree automaton and its complement is so, too, which was proved by Rabin [Rab70]. Moreover, being definable in WS2S is equivalent to being recognized by a weak alternating tree automaton [KV99]. It is true though that every S2S formula is equivalent to an existential S2S formula. Also note that the second formula given as example at the beginning of this section is one which cannot be recognized by a Büchi tree automaton, let alone specified in WS2S. Another noticeable difference between automata on infinite words and automata on infinite trees is that unambiguous tree automata are weaker than nondeterministic ones, which is a result due to Niwiński and Walukiewicz [NW]. Its proof was recently simplified considerably by Carayol and Löding [CL07].

The most complicated automata-theoretic building block of our proof of Rabin's theorem is McNaughton's Theorem, the determinization of word automata. It is not clear to which extent McNaughton's Theorem is necessary for the proof of Rabin's Theorem. The proof presented here is based on a translation in the sense that for every S2S formula φ we construct an automaton $\mathscr{A}$ such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\varphi)$ and it makes full use of a determinization construction. There are other proofs, such as the one by Kupferman and Vardi [KV05], which do not rely on the entire construction but only on the fact that there are determinization constructions with a certain bound on the number of states. These constructions, however, yield a slightly weaker result in the sense that they only reduce S2S satisfiability to tree automaton emptiness. In the proof presented here, determinization is used to turn a universal automaton into a nondeterministic one, which could be called a de-universalization construction. It would be interesting to see if one can also go in the reverse direction, that is, whether there is a determinization construction which can be built on a de-universalization construction.

At the end of the previous section, we mentioned that topological questions are interesting in the context of infinite words and automata on infinite words. This is even more true for infinite trees, see [ADNM].

4 Linear-Time Temporal Logic

Although originally introduced in this context, S1S and WS1S have only very rarely been used to specify properties of (finite-state) devices (see [HJJ+95] for a noticeable exception). For S2S, this is even more true; it has almost always been used to obtain decidability for logical theories as pointed out in Section 3. But the ever-increasing number of real computational devices and large-scale production lines of such devices has called for appropriate specification logics. In this section, we consider a logic that was introduced in this regard and show how it can be dealt with using automata theory, in particular, we show how specifically tailored automata can be used to obtain optimal upper bounds for problems such as satisfiability, conformance—in this context called model checking—, and realizability.

4.1 LTL and S1S

Linear-time temporal logic (LTL) is a modal logic designed to specify temporal relations between events occurring over time, designed by Kamp [Kam68] to formally describe temporal relationships expressible in natural language and introduced into computer science by Pnueli [Pnu77] (see also the work by Burstall [Bur74] and Kröger [Krö77]) as an appropriate specification language for systems with nonterminating computations. Nowadays, LTL is widely spread and used in practice.

From a syntactic point of view LTL is propositional logic augmented by temporal operators. LTL formulas are built from tt and propositional variables using negation ($\neg$), disjunction ($\vee$), and the binary temporal operator XU called "strict until" and used in infix notation. For instance, when p is a propositional variable, then $\neg p \wedge \mathsf{tt}\,\mathsf{XU}\,p$ is an LTL formula. When P is a finite set of propositional variables and φ an LTL formula with propositional variables from P, then φ is called a formula over P.

LTL formulas are typically interpreted in infinite words, more precisely, given a finite set P of propositional variables, an LTL formula φ over P, a word $u \in (2^P)^\omega$, and $i \geq 0$, it is defined what it means that φ holds in u at position i, denoted $u, i \models \varphi$:

- $u, i \models \mathsf{tt}$,
- $u, i \models p$ if $p \in u(i)$, for every $p \in P$,
- $u, i \models \neg\varphi$ if $u, i \not\models \varphi$, for every LTL formula φ over P,
- $u, i \models \varphi \vee \psi$ if $u, i \models \varphi$ or $u, i \models \psi$, for LTL formulas φ and ψ over P, and
- $u, i \models \varphi\,\mathsf{XU}\,\psi$ if there exists $j > i$ such that $u, j \models \psi$ and $u, i' \models \varphi$ for all i' such that $i < i' < j$.

So $\varphi\,\mathsf{XU}\,\psi$ means that the formula φ holds true in the future until a point is reached where ψ holds true. For a word u as above and an LTL formula φ we say that φ holds in u, denoted $u \models \varphi$, if $u, 0 \models \varphi$. The language defined by φ is $\mathscr{L}(\varphi) = \{u \in (2^P)^\omega \colon u \models \varphi\}$, where, for convenience, we do not refer to P in the notation.

Clearly, there are many more basic temporal relations than just "until". So, often, other temporal operators are used:

- "next" is denoted X and defined by $\mathsf{X}\varphi = \neg\mathsf{tt}\,\mathsf{XU}\,\varphi$,
- "sometime in the future" is denoted XF and defined by $\mathsf{XF}\varphi = \mathsf{tt}\,\mathsf{XU}\,\varphi$, and
- "always in the future" is denoted XG and defined by $\mathsf{XG}\varphi = \neg\mathsf{XF}\neg\varphi$.

In many situations, it is convenient to include the current point in time, which leads to defining F by $\mathsf{F}\varphi = \varphi \vee \mathsf{XF}\varphi$ and, similarly, G by $\mathsf{G}\varphi = \neg\mathsf{F}\neg\varphi$ as well as U by $\varphi\,\mathsf{U}\,\psi = \psi \vee (\varphi \wedge \varphi\,\mathsf{XU}\,\psi)$.

It is remarkable that Kamp in his 1968 thesis [Kam68] proved that every temporal relation expressible in natural (English) language can be expressed in linear-time temporal logic as defined above. As a yardstick for what is expressible in natural language he used first-order logic, considering formula

with one free variable. To be precise, to obtain his result Kamp also had to add a past version of until, called since. That until by itself is enough to express everything expressible in first-order logic when only sentences are considered was proved by Gabbay, Pnueli, Shelah, and Stavi [GPSS80].

A typical LTL formula is

$$\mathsf{G}(p_r \rightarrow \mathsf{XF}p_a)$$

which expresses that for every occurrence of p_r there is a later occurrence of p_a, or, simply, every "request" is followed by an "acknowledge".

Another, more complicated, example is a formula expressing that competing requests are served in order. We assume that r_0 and r_1 are propositional variables indicating the occurrence of requests and a_0 and a_1 are matching propositional variables indicating the occurrence of acknowledgments. We want to specify that whenever an r_0 request occurs while no r_1 request is pending, then a_1 does not occur before the next occurrence of a_0.

We first specify that starting from an r_0 request there is an a_1 acknowledgment before an a_0 acknowledgment:

$$\alpha = r_0 \wedge (\neg a_0 \,\mathsf{XU}\, (a_1 \wedge \neg a_0)).$$

Next, we observe that there are two different types of situations where an r_0 request can occur while no r_1 request is pending. The first type of situation is when there has been no r_1 request before the r_0 request in question. The second type is when a_1 occurred before the r_0 request in question and in between there has been no r_1 request. For each type of situation, we have a separate disjunct in our formula:

$$\neg(\neg r_1 \,\mathsf{U}\, (\neg r_1 \wedge \alpha)) \vee \neg\mathsf{F}(a_1 \wedge \neg r_1 \,\mathsf{U}\, (\neg r_1 \wedge \alpha)).$$

Clearly, in the context of LTL all the algorithmic problems discussed for S1S—satisfiability, conformance (model checking), and realizability—can be discussed. For instance, we can ask whether a given formula φ over P is satisfiable in the sense that there exists a word $u \in (2^P)^\omega$ such that $u \models \varphi$ or, given φ and a finite-state automaton $\mathscr{D}$ over 2^P, we can ask whether $u \models \varphi$ for all $u \in \mathscr{L}(\mathscr{D})$.

We can show in just one step that all these problems are decidable, namely by showing that every LTL formula is equivalent to an S1S formula; the results from Section 2 then apply. Unfortunately, the decision procedures that one obtains in this way have a nonelementary complexity. We can do better by using specifically tailored automata-theoretic constructions. We first present, however, the translation into S1S and then only turn to better decision procedures.

We start by defining the notion of equivalence we use to express the correctness of our translation. Let $P = \{p_0, \ldots, p_{r-1}\}$ be a set of propositional

variables. Rather than interpreting LTL formulas over P in words over 2^P, we interpret them in words over $[2]_r$, where we think of every letter $a \in 2^P$ as the letter $b \in [2]_r$ with $b_{[j]} = 1$ iff $p_j \in a$ for every $j < r$. We say that an S1S formula $\psi = \psi(V_0, \ldots, V_{r-1})$ is equivalent to an LTL formula φ over P if for every $u \in [2]_r^\omega$ the following holds: $u \models \varphi$ iff $u \models \psi$.

In the proposition below, we make a stronger statement, and this involves the notion of global equivalence, which is explained next. Given a word $u \in [2]_r^\omega$, a position i, and an S1S formula $\psi = \psi(V_0, \ldots, V_{r-1}, x)$ where x is a first-order variable, we write $u, i \models \psi$ if ψ holds true when the set variables are assigned values according to u and x is assigned i. We say that ψ is globally equivalent to an LTL formula φ over P if the following holds: $u, i \models \varphi$ iff $u, i \models \psi$ for every $u \in [2]_r^\omega$ and every i.

Proposition 4.1. Let $P = \{p_0, \ldots, p_{r-1}\}$ be a finite set of propositional variables and x a first-order variable. For every LTL formula φ over P a globally equivalent S1S formula $\tilde{\varphi} = \varphi(V_0, \ldots, V_{r-1}, x)$ can be constructed.

Observe that $\exists x(\forall y \neg \mathrm{suc}(y, x) \wedge \tilde{\varphi})$ will be equivalent to φ.

Proof. A proof can be carried out by a straightforward induction on the structure of φ. When $\varphi = \mathsf{tt}$, we choose $\tilde{\varphi} = (x = x)$, and when $\varphi = p_j$, we take $\tilde{\varphi} = x \in V_j$.

In the inductive step, we distinguish various cases. When $\varphi = \neg\psi$, we can choose $\tilde{\varphi} = \neg\tilde{\psi}$. Similarly, when $\varphi = \psi \vee \chi$, we can choose $\tilde{\varphi} = \tilde{\psi} \vee \tilde{\chi}$. Finally, assume $\varphi = \psi \,\mathsf{XU}\, \chi$. Then we choose

$$\tilde{\varphi} = \exists z(x < z \wedge \tilde{\chi}(V_0, \ldots, V_{r-1}, z) \wedge \forall y(x < y < z \rightarrow \tilde{\psi}(V_0, \ldots, V_{r-1}, y))),$$

which simply reflects the semantics of XU. Q.E.D.

Observe that the above proof even shows that every formula is equivalent to a first-order formula (without set quantification but with ordering), and a slightly more careful proof would show that three first-order variables are sufficient [IK89]. Kamp's seminal result [Kam68] is the converse of the above proposition when first-order logic with ordering is considered instead of S1S.

As a consequence of Proposition 4.1, we can state:

Corollary 4.2. LTL satisfiability, model-checking, and realizability are decidable.

This result is not very satisfying, because in view of [SM73, Sto74] the decision procedures obtained in this way have nonelementary complexity. As it turns out, it is much better to translate LTL directly into Büchi automata and carry out the same constructions we have seen for S1S all over again. The key is a good translation from LTL into Büchi automata.

4.2 From LTL to Büchi Automata

Vardi and Wolper [VW86a, VW94] were the first to describe and advocate a separate translation from LTL into Büchi automata, resulting in essentially optimal bounds for the problems dealt with in Section 2. These bounds were originally achieved by Sistla and Clarke [SC82, SC85], for satisfiability and model checking, and by Pnueli and Rosner [PR89], for realizability.

There are several ways of translating LTL into Büchi automata. We present two translations, a classical and a modern translation: the first one goes from an LTL formula via a generalized Büchi automaton to an ordinary Büchi automaton, while the second one goes via very weak alternating automata.

Both of the constructions we are going to present are based on formulas in positive normal form, which we define next. The operator "release", denoted XR, is defined by $\varphi \mathsf{XR} \psi = \neg(\neg\varphi \mathsf{XU} \neg\psi)$. In a certain sense, $\varphi \mathsf{XR} \psi$ expresses that the requirement of ψ to hold is released by the occurrence of φ. LTL formulas in positive normal form are built starting from tt, ff, p, and $\neg p$ using $\vee$, $\wedge$, XU, and XR, that is, negations are only allowed to occur right in front of propositional variables.

The following identities show that every LTL formula can be transformed into an equivalent LTL formula in positive normal form which is not longer than the given one.

Lemma 4.3. For LTL formulas φ and ψ over a finite set P of propositional variables, $u \in (2^P)^\omega$, and $i \geq 0$, the following holds:

$$\begin{aligned}
u, i &\models \neg\mathsf{tt} && \text{iff} && u, i \models \mathsf{ff}, \\
u, i &\models \neg\neg\varphi && \text{iff} && u, i \models \varphi, \\
u, i &\models \neg(\varphi \vee \psi) && \text{iff} && u, i \models \neg\varphi \wedge \neg\psi, \\
u, i &\models \neg(\varphi \,\mathsf{XU}\, \psi) && \text{iff} && u, i \models \neg\varphi \,\mathsf{XR}\, \neg\psi.
\end{aligned}$$

Proof. A proof can be carried out in a straightforward fashion, using the definition of the semantics of LTL. Q.E.D.

As mentioned above, the other ingredient for our translation are generalized Büchi automata, introduced in [GPVW95]. Such an automaton is a tuple

$$\mathscr{A} = (A, Q, Q_I, \Delta, \mathscr{F})$$

where the first four components are as with ordinary Büchi automata, the only difference is in the last component: $\mathscr{F}$ is a set of subsets of Q, each called an acceptance set of $\mathscr{A}$. A run r is accepting if for every acceptance set $F \in \mathscr{F}$ there exist infinitely many i such that $r(i) \in F$. So generalized Büchi automata can express conjunctions of acceptance conditions in a simple way.

The essential idea for constructing a generalized Büchi automaton equivalent to a given LTL formula is as follows. As the automaton reads a given word it guesses which subformulas are true. At the same time it verifies its guesses. This is straightforward for almost all types of subformulas, for instance, when the automaton guesses that $\neg p$ is true, it simply needs to check that $p \notin a$ if a is the current symbol read. The only subformulas that are difficult to handle are XU-subformulas, that is, subformulas of the form $\psi \,\mathsf{XU}\, \chi$. Checking that such a subformula is true cannot be done directly or in the next position in general because the "satisfaction point" for an XU-formula—the position where χ becomes true—can be in the far future. Of course, by keeping $\psi \,\mathsf{XU}\, \chi$ in the state the automaton can remember the obligation to eventually reach a satisfaction point, but the acceptance condition is the only feature of the automaton which can be used to really check that reaching the satisfaction point is not deferred forever.

The complete construction is described in Figure 17; it uses $\mathrm{sub}(\varphi)$ to denote the set of all subformulas of a formula φ including φ itself. Note that for every XU-subformula $\psi \,\mathsf{XU}\, \chi$ there is a separate acceptance set, which contains all states which do not have an obligation for eventually satisfying this subformula or satisfy it in the sense that χ is an obligation too.

Theorem 4.4 (Gerth-Peled-Vardi-Wolper, [GPVW95])**.** Let P be a finite set of propositional variables and φ an LTL formula over P with n subformulas and k XU-subformulas. Then $\mathscr{A}[\varphi]$ is a generalized Büchi automaton with 2^n states and k acceptance sets such that $\mathscr{L}(\mathscr{A}[\varphi]) = \mathscr{L}(\varphi)$.

Proof. We first show $\mathscr{L}(\mathscr{A}[\varphi]) \subseteq \mathscr{L}(\varphi)$. Let $u \in \mathscr{L}(\mathscr{A}[\varphi])$ and let r be an accepting run of $\mathscr{A}[\varphi]$ on u. We claim that for every i, if $\psi \in r(i)$, then $u, i \models \psi$. The proof is by induction on the structure of ψ. If $\psi = \mathsf{tt}$, $\psi = \mathsf{ff}$, $\psi = p$, or $\psi = \neg p$, then this follows directly from (i) or (ii). If $\psi = \chi \vee \zeta$, the claim follows from the induction hypothesis and (iii). Similarly, the claim holds for a conjunction.

Assume $\psi = \chi \,\mathsf{XR}\, \zeta$. Then (vi) tells us that

(a) $\chi \,\mathsf{XR}\, \zeta, \zeta \in r(j)$ for every $j > i$ or

(b) there exists $j \geq i$ such that $\chi \,\mathsf{XR}\, \zeta, \zeta \in r(i')$ for i' with $i < i' < j$ and $\chi, \zeta \in r(j)$.

From the induction hypothesis and (a), we can conclude that we have $u, i' \models \zeta$ for all $i' > i$, which means $u, i \models \psi$. Similarly, from the induction hypothesis and (b), we can conclude that we have $u, i' \models \zeta$ for all i' such that $i < i' \leq j$ and $u, j \models \chi$, which implies $u, i \models \psi$, too.

Finally, assume $\psi = \chi \,\mathsf{XU}\, \zeta$. From (v), we obtain that

(a) $\chi \,\mathsf{XU}\, \zeta \in r(j)$ for all $j > i$ or

Let P be a finite set of propositional variables and φ an LTL formula over P in positive normal form. The generalized Büchi automaton for φ with respect to P, denoted $\mathscr{A}[\varphi]$, is defined by

$$\mathscr{A}[\varphi] = (2^P, 2^{\mathrm{sub}(\varphi)}, Q_I, \Delta, \mathscr{F})$$

where a triple (Ψ, a, Ψ') with $\Psi, \Psi' \subseteq \mathrm{sub}(\varphi)$ and $a \in 2^P$ belongs to Δ if the following conditions are satisfied:

(i) $\mathsf{ff} \notin \Psi$,

(ii) $p \in \Psi$ iff $p \in a$, for every $p \in P$,

(iii) if $\psi \vee \chi \in \Psi$, then $\psi \in \Psi$ or $\chi \in \Psi$,

(iv) if $\psi \wedge \chi \in \Psi$, then $\psi \in \Psi$ and $\chi \in \Psi$,

(v) if $\psi \,\mathsf{XU}\, \chi \in \Psi$, then $\chi \in \Psi'$ or $\{\psi, \psi \,\mathsf{XU}\, \chi\} \subseteq \Psi'$,

(vi) if $\psi \,\mathsf{XR}\, \chi$, then $\{\psi, \chi\} \subseteq \Psi'$ or $\{\chi, \psi \,\mathsf{XR}\, \chi\} \subseteq \Psi'$,

and where

$$Q_I = \{\Psi \subseteq \mathrm{sub}(\varphi) \colon \varphi \in \Psi\},$$
$$\mathscr{F} = \{F_{\psi \mathsf{XU} \chi} \colon \psi \,\mathsf{XU}\, \chi \in \mathrm{sub}(\varphi)\},$$

with $F_{\psi \mathsf{XU} \chi}$ defined by

$$F_{\psi \mathsf{XU} \chi} = \{\Psi \subseteq \mathrm{sub}(\varphi) \colon \chi \in \mathrm{sub}(\varphi) \text{ or } \psi \,\mathsf{XU}\, \chi \notin \Psi\}.$$

FIGURE 17. From LTL to generalized Büchi automata

(b) there exists j such that $\chi \,\mathsf{XU}\, \zeta \in r(i')$ for all i' with $i < i' < j$ and $\zeta \in r(j)$.

Just as with XR, we obtain $u, i \models \psi$ from the induction hypothesis and (b). So we only need to show that if (a) occurs, we also have (b). Since r is accepting, there is some $\Psi \in F_{\chi \mathsf{XU} \zeta}$ such that $r(j) = \Psi$ for infinitely many j. Assuming (a), we can can conclude $\zeta \in \Psi$, which, by induction hypothesis, means we also have (b).

For the other inclusion, $\mathscr{L}(\varphi) \subseteq \mathscr{L}(\mathscr{A}[\varphi])$, we simply show that for a given u such that $u \models \varphi$ the word r defined by $r(i) = \{\psi \in \mathrm{sub}(\varphi)\colon u, i \models \psi\}$ is an accepting run of $\mathscr{A}[\varphi]$ on u. To this end, we need to show that

(a) r starts with an initial state,

(b) $(r(i), u(i), r(i+1)) \in \Delta$ for all i, and

(c) $r(i) \in F_{\psi \mathsf{XU} \chi}$ for infinitely many i, for every formula $\psi \,\mathsf{XU}\, \chi \in \mathrm{sub}(\varphi)$.

That (a) is true follows from the assumption $u \models \varphi$. Condition (b) is true simply because of the semantics of LTL. To see that (c) is true, let $\psi \,\mathsf{XU}\, \chi \in \mathrm{sub}(\varphi)$. We distinguish two cases. First, assume there exists i such that $u, j \not\models \chi$ for all $j > i$. Then $u, j \not\models \psi \,\mathsf{XU}\, \chi$ for all $j \geq i$, hence $r(j) \in F_{\psi \mathsf{XU} \chi}$ for all $j \geq i$, which is enough. Second, assume there are infinitely many i such that $u, i \models \chi$. Then $\chi \in r(i)$ for the same values of i, which is enough, too. Q.E.D.

Generalized Büchi automata can be converted into equivalent Büchi automata in a straightforward fashion. The idea is to check that every acceptance set is visited infinitely often by visiting these sets one after the other, in a fixed order, and repeating this process over and over again. In Figure 18, a respective construction is described. The second component of the state space is a counter which is used to keep track of the acceptance set to be visited next. When this counter reaches its maximum, every acceptance set has been visited once, and it can be reset.

Remark 4.5. Let $\mathscr{A}$ be a generalized Büchi automaton with n states and k acceptance sets. Then $\mathscr{A}^{BA}$ is an equivalent Büchi automaton with at most $(k+1)n$ states.

Corollary 4.6 (Vardi-Wolper, [VW86a, VW94])**.** There exists an effective procedure that given an LTL formula φ with n states and k XU-subformulas outputs a Büchi automaton $\mathscr{A}$ with at most $(k+1)2^n$ states such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\varphi)$.

4.3 From LTL to Alternating Automata

The above translation from LTL into Büchi automata serves our purposes perfectly. We can use it to derive all the desired results about the complexity of the problems we are interested in, satisfiability, model checking, and realizability, as will be shown in the next subsection. There is, however, a translation using alternating automata, which is interesting in its own right. The motivation behind considering such a translation is to pass from the logical framework to the automata-theoretic framework in an as simple as possible fashion (to be able to apply powerful automata-theoretic tools as early as possible).

Let $\mathscr{A}$ be a generalized Büchi automaton with $\mathscr{F} = \{F_0, \ldots, F_{k-1}\}$. The Büchi automaton $\mathscr{A}^{BA}$ is defined by

$$\mathscr{A}^{BA} = (A, Q \times \{0, \ldots, k\}, Q_I, \Delta', Q \times \{k\})$$

where Δ' contains for every $(q, a, q') \in \Delta$ the following transitions:

- $((q, k), a, (q', 0))$,
- $((q, i), a, (q', i))$ for every $i < k$,
- $((q, i), a, (q', i + 1))$ for every $i < k$ such that $q' \in F_i$.

FIGURE 18. From generalized Büchi to ordinary Büchi automata

Alternating automata are provided with a feature to spawn several copies of themselves while running over a word. Formally, an alternating Büchi automaton is a tuple

$$\mathscr{A} = (P, Q, q_I, \delta, F)$$

where P, Q, and q_I are as usual, F is a Büchi acceptance condition, and δ is a function which assigns to each state q a transition condition, where every transition condition $\delta(q)$ is a positive boolean combination of formulas of the type p and $\neg p$, for $p \in P$, and $\bigcirc q$, for $q \in Q$. More precisely, the set of transition conditions over P and Q, denoted $\mathrm{TC}(P, Q)$, is the smallest set such that

(i) $\mathsf{tt}, \mathsf{ff} \in \mathrm{TC}(P, Q)$,

(ii) $p, \neg p \in \mathrm{TC}(P, Q)$ for every $p \in P$,

(iii) $\bigcirc q \in \mathrm{TC}(P, Q)$ for every $q \in Q$,

(iv) $\gamma \wedge \gamma', \gamma \vee \gamma' \in \mathrm{TC}(P, Q)$ for $\gamma, \gamma' \in \mathrm{TC}(P, Q)$.

A run of such an automaton on a word $u \in (2^P)^\omega$ is a tree $\mathscr{R}$ labeled with elements from $(Q \cup \mathrm{TC}(P, Q)) \times \omega$ such that $l^{\mathscr{R}}(\mathrm{root}(\mathscr{R})) = (q_I, 0)$ and the following conditions are satisfied for every $v \in V^{\mathscr{R}}$, assuming $l^{\mathscr{R}}(v) = (\gamma, i)$:

(i) $\gamma \neq \mathsf{ff}$,

The automaton has states $q_I, q_0, q_1, \ldots, q_{10}$ where q_0 is the only final state and the transition function δ is defined by

- $\delta(q_I) = \bigcirc q_I \vee \bigcirc q_0$,
- $\delta(q_0) = \bigcirc q_0 \wedge ((p \wedge \bigcirc q_1) \vee \neg p)$,
- $\delta(q_i) = \bigcirc q_{i+1}$ for all i such that $0 < i < 10$,
- $\delta(q_{10}) = p$.

FIGURE 19. Example for an alternating automaton

(ii) if $\gamma = p$ for some $p \in P$, then $p \in u(i)$,

(iii) if $\gamma = \neg p$ for some $p \in P$, then $p \notin u(i)$.

(iv) if $\gamma = q$, then v has a successor v' such that $l^{\mathscr{R}}(v') = (\delta(q), i)$,

(v) if $\gamma = \bigcirc q'$, then v has a successor v' such that $l^{\mathscr{R}}(v') = (q', i+1)$,

(vi) if $\gamma = \gamma_0 \wedge \gamma_1$, then v has successors v_0 and v_1 such that $l^{\mathscr{R}}(v_j) = (\gamma_j, i)$ for $j < 2$,

(vii) if $\gamma = \gamma_0 \vee \gamma_1$, then there exists $j < 2$ such that v has a successor v' with $l^{\mathscr{R}}(v') = (\gamma_j, i)$.

An infinite branch b of $\mathscr{R}$ is accepting if there are infinitely many i such that $l^{\mathscr{R}}(b(i)) \in F \times \omega$, in other words, there are infinitely many vertices with a final state in the first component of their labeling. The run is accepting if every infinite branch of it is accepting.

As a simple example, consider the language L_{10} over 2^P where $P = \{p\}$ which contains all words u satisfying the following condition: There exists some number i such that $p \in u(j+10)$ for all $j \geq i$ with $p \in u(j)$. If we wanted to construct a nondeterministic automaton for this language, we could not do with less than 1000 states, but there is a small alternating automaton that recognizes this language. It simply guesses the right positions i and for each such position it spawns off a copy of itself checking that after 10 further steps p holds true again. The details are given in Figure 19.

It is (vi) from above which forces the tree to become a real tree, that is, it requires that a vertex has two successors (unless $\gamma_0 = \gamma_1$). So this is the condition that makes the automaton alternating: For a run to be accepting,

Let φ be an LTL formula in positive normal form over P and Q the set which contains for each $\psi \in \mathrm{sub}(\varphi)$ an element denoted $[\psi]$. The automaton $\mathscr{A}^{alt}[A]$ is defined by

$$\mathscr{A}^{alt}[\varphi] = (P, Q, [\varphi], \delta, F)$$

where

$$\begin{aligned} \delta([\mathsf{tt}]) &= \mathsf{tt}, & \delta([\mathsf{ff}]) &= \mathsf{ff}, \\ \delta([p]) &= p, & \delta([\neg p]) &= \neg p, \\ \delta([\psi \vee \chi]) &= \delta([\varphi]) \vee \delta([\psi]), & \delta([\psi \wedge \chi]) &= \delta([\varphi]) \wedge \delta([\psi]), \end{aligned}$$

$$\begin{aligned} \delta([\psi \,\mathsf{XU}\, \chi]) &= \bigcirc[\chi] \vee (\bigcirc[\psi] \wedge \bigcirc[\psi \,\mathsf{XU}\, \chi]), \\ \delta([\psi \,\mathsf{XR}\, \chi]) &= \bigcirc[\psi] \wedge (\bigcirc[\psi] \vee \bigcirc[\psi \,\mathsf{XR}\, \chi]), \end{aligned}$$

and F contains all the elements $[\psi] \in Q$ where ψ is not a XU-formula.

FIGURE 20. From an LTL formula to an alternating automaton

both alternatives have to be pursued. We can think of the automaton as splitting into two copies.

The translation from LTL to alternating Büchi automata, given in Figure 20, is straightforward as it simply models the semantics of LTL. It exploits the fact that $\psi \mathsf{XU} \chi$ and $\psi \mathsf{XR} \chi$ are equivalent to $\mathsf{X}\chi \vee (\mathsf{X}\psi \wedge \mathsf{X}(\psi \mathsf{XU} \chi))$ and $\mathsf{X}\chi \wedge (\mathsf{X}\psi \vee \mathsf{X}(\psi \,\mathsf{XR}\, \chi))$, respectively. Note that we use the notation $[\psi]$ to distinguish subformulas of φ from transition conditions ($p_0 \wedge p_1$ is different from $[p_0 \wedge p_1]$).

The transition function of $\mathscr{A}^{alt}[\varphi]$ has an interesting property, which we want to discuss in detail. Let $\leq$ be any linear ordering which extends the partial order on Q defined by $[\psi] \leq [\chi]$ if $\psi \in \mathrm{sub}(\chi)$. For every $\psi \in \mathrm{sub}(\varphi)$ and every $[\chi]$ occurring in $\delta([\psi])$, we have $[\chi] \leq [\psi]$. Following Gastin and Oddoux [GO01], we call an automaton satisfying this property a very weak alternating automaton.

The transition function of $\mathscr{A}^{alt}[\varphi]$ has an even stronger structural property, which we explain next. For a given symbol $a \in 2^P$, a transition condition γ, a state $q \in Q$, and a set $Q' \subseteq Q$, we define what it means that Q' is an a-successor of q with respect to γ, denoted $q \rightarrow^{a,\gamma} Q'$. This is defined inductively:

- $q \to^{a,\mathsf{tt}} \varnothing$,
- $q \to^{a,p} \varnothing$ if $p \in a$, and, similarly, $q \to^{a,\neg p} \varnothing$ if $p \notin a$,
- $q \to^{a,\bigcirc q'} \{q'\}$,
- $q \to^{a,\gamma_0 \vee \gamma_1} Q'$ if $q \to^{a,\gamma_0} Q'$ or $q \to^{a,\gamma_1} Q'$,
- $q \to^{a,\gamma_0 \wedge \gamma_1} Q'$ if there exists $Q_0, Q_1 \subseteq Q$ such that $Q' = Q_0 \cup Q_1$, $q \to^{a,\gamma_0} Q_0$, and $q \to^{a,\gamma_1} Q_1$.

Note that $q \to^{a,\gamma} Q'$ has a natural interpretation in terms of runs. If a vertex v of a run is labeled (q,i) and Q' is the set of all states q' such that $(q',i+1)$ is a label of a descendant of v, then $q \to^{a,\gamma} Q'$, provided, of course, that the run is minimal, which we can and will henceforth assume without loss of generality.

We use $q \to^{a} Q'$ as an abbreviation for $q \to^{a,\delta(q)} Q'$. We say a state q is persistent if there exists Q' such that $q \in Q'$ and $q \to^{a} Q'$ for some letter a.

Using the new notation, we can give an equivalent definition of being a very weak alternating automaton. It simply means that there exists a linear ordering $\leq$ on the states of the automaton such that if $q \to^{a} Q'$, then $q' \leq q$ for all $q' \in Q'$.

The automaton $\mathscr{A}^{alt}[\varphi]$ has the following property. For every persistent state q there exists a state q' such that

(i) $q \to^{a} \{q'\}$ for every letter a and

(ii) whenever $q \to^{a} Q'$, then either $q \in Q'$ or $Q' = \{q'\}$.

(Every $q \notin F$ is of the form $[\psi \,\mathsf{XU}\, \chi]$, which means that we can choose $q' = [\chi]$.) We call very weak alternating automata that have this property ultra weak alternating automata and a state as q' above a discharging state for q and denote it by q^{d}.

Lemma 4.7. Let φ be an LTL formula with n subformulas. Then $\mathscr{A}^{alt}[\varphi]$ is an ultra weak alternating automaton with n states such that $\mathscr{L}(\mathscr{A}^{alt}[\varphi]) = \mathscr{L}(\varphi)$.

Proof. We only need to prove its correctness, which we do by an induction on the structure of φ. We start with a simple observation. Let $\mathscr{R}$ be an accepting run of $\mathscr{A}^{alt}[\varphi]$ on u and $v \in V^{\mathscr{R}}$ labeled $([\psi], i)$ for some $\psi \in \mathrm{sub}(\varphi)$. Then $\mathscr{R}{\downarrow}v$ can be turned into an accepting run of $\mathscr{A}^{alt}[\psi]$ on $u[i,*)$ by changing each second component j of a vertex label by $j - i$. Clearly, for this to be true $\mathscr{R}$ needs to be minimal (see above).

For the induction base, first assume $\varphi = \mathsf{tt}$ or $\varphi = \mathsf{ff}$. There is nothing to show. Second, assume $\varphi = p$. Suppose $u \models \varphi$. Then $p \in u(0)$, that is,

the two-vertex tree where the root is labeled $([p], 0)$ and its only successor is labeled $(p, 0)$ is an accepting run of $\mathscr{A}^{\mathrm{alt}}[\varphi]$ on u. Conversely, if $\mathscr{R}$ is a (minimal) run of $\mathscr{A}^{\mathrm{alt}}[\varphi]$ on u, then $\mathscr{R}$ has two vertices labeled $([p], 0)$ and $(p, 0)$, respectively. This implies $p \in u(0)$, which, in turn, implies $u \models \varphi$. An analogous argument applies to $\neg p$.

In the inductive step, first assume $\varphi = \psi_0 \wedge \psi_1$. If there exists an accepting run $\mathscr{R}$ of $\mathscr{A}^{\mathrm{alt}}[\varphi]$ on u, then, because of $\delta([\varphi]) = \delta([\psi_0]) \wedge \delta([\psi_1])$, the root has successors v_0 and v_1 such that $l^{\mathscr{R}}(v_i) = (\delta([\psi_i]), 0)$. For every i, we can turn $\mathscr{R}{\downarrow}v_i$ into an accepting run $\mathscr{R}_i$ of $\mathscr{A}^{\mathrm{alt}}[\psi_i]$ on u by adding a new root labeled $([\psi_i], 0)$. By induction hypothesis, we obtain $u \models \psi_i$ for every i, hence $u \models \varphi$. Conversely, assume $u \models \varphi$. Then $u \models \psi_i$ for $i < 2$, and, by induction hypothesis, there exist accepting runs $\mathscr{R}_i$ of $\mathscr{A}^{\mathrm{alt}}[\psi_i]$ on u for $i < 2$. These runs can be turned into an accepting run of $\mathscr{A}^{\mathrm{alt}}[\varphi]$ on u by simply making their vertex sets disjoint, removing their roots, and adding a new common root labeled $([\varphi], 0)$.

A similar argument applies to formulas of the form $\psi_0 \vee \psi_1$.

Next, assume $\varphi = \psi \,\mathsf{XU}\, \chi$. Suppose $\mathscr{R}$ is an accepting run of $\mathscr{A}^{\mathrm{alt}}[\varphi]$ on u and let v_0 be the root of this run. Also, let $u_i = u[i, *)$ for every i. Then, by definition of accepting run, $l^{\mathscr{R}}(v_0) = ([\psi \,\mathsf{XU}\, \chi], 0)$. From the definition of the transition function we can conclude that v_0 has a successor, say v_1, which is labeled by $(\bigcirc[\chi] \vee (\bigcirc[\psi] \wedge \bigcirc[\psi \mathsf{XU} \chi]), 0)$, which, in turn, has a successor, say v_2, which is labeled by either $(\bigcirc[\chi], 0)$ or $(\bigcirc[\psi] \wedge \bigcirc[\psi \mathsf{XU} \chi], 0)$. In the first case, there is a further successor labeled $([\chi], 1)$ and we obtain $u_1 \models \chi$ from the induction hypothesis, hence, $u \models \varphi$. In the second case, we know there exist successors v_3 and v_3' of v_2 labeled $(\bigcirc[\psi \,\mathsf{XU}\, \chi], 0)$ and $(\bigcirc[\psi], 0)$, respectively, which themselves have successors v_4 and v_4' labeled $([\psi \,\mathsf{XU}\, \chi], 1)$ and $([\psi], 1)$, respectively. By induction hypothesis, we obtain $u_1 \models \psi$. Applying the same arguments as before, we find that either there is a vertex labeled $([\chi], 2)$ or there are vertices v_8 and v_8' labeled $[(\psi \,\mathsf{XU}\, \chi, 2)]$ and $([\psi], 2)$, respectively. In the first case, we get $u \models \varphi$ because we also know $u_1 \models \psi$, whereas in the second case we can again apply the same arguments as before. Continuing in this fashion, we find that the only case which remains is the one where we have an infinite sequence of vertices $v_4, v_8, v_{12}, \ldots$ on the same branch and every vertex with label in $Q \times \omega$ is labeled $([\varphi], i)$, which means that this branch is not accepting—a contradiction.

For the other direction, assume $u \models \varphi$ and use the same notation as before. Then there is some $j > 0$ such that $u_j \models \chi$ and $u_i \models \psi$ for all i with $0 < i < j$. By induction hypothesis, there are accepting runs $\mathscr{R}_i$ for i with $0 < i < j$ of $\mathscr{A}^{\mathrm{alt}}[\psi]$ on u_i and an accepting run $\mathscr{R}_j$ of $\mathscr{A}^{\mathrm{alt}}[\chi]$ on u_j. Assume that $v_1, \ldots, v_j$ are the roots of these trees and assume that their sets of vertices are pairwise disjoint. Then we can construct an accepting run $\mathscr{R}$

for $\mathscr{A}^{alt}[\varphi]$ on u as follows. The vertices of $\mathscr{R}$ are the vertices of the $\mathscr{R}_k$'s and, in addition, the new vertices $w_0, w_0', w_0'', w_0''', \hat{w}_0, w_1, \ldots, w_{j-1}, w_{j-1}', w_{j-1}''$. The labeling is as follows:

- w_i is labeled $([\varphi], i)$ for $i < j$,
- w_i' is labeled $(\bigcirc[\chi] \vee (\bigcirc[\psi] \wedge \bigcirc[\varphi]), i)$ for $i < j$,
- w_i'' is labeled $(\bigcirc[\psi] \wedge \bigcirc[\varphi], i)$ for $i < j - 1$,
- w_i''' is labeled $(\bigcirc[\varphi], i)$ for $i < j - 1$,
- $\hat{w}_i$ is labeled $(\bigcirc[\psi], i)$ for $i < j - 1$, and
- w_j'' is labeled $(\bigcirc[\chi], j - 1)$.

The tree $\mathscr{R}$ has all edges from the $\mathscr{R}_k$'s and, in addition,

- edges such that $w_0 w_0' w_0'' w_0''' \ldots w_{j-1} w_{j-1}' w_{j-1}'' v_j$ is a path and
- edges $(w_i', \hat{w}_i)$ and $(\hat{w}_i, v_i)$ for every $i < j$.

This yields an accepting run of $\mathscr{A}^{alt}[\varphi]$ on u.

Finally, XR can be dealt with in a similar fashion. Q.E.D.

It is not very difficult to translate alternating Büchi automata into nondeterministic Büchi automata, as was shown by Miyano and Hayashi [MH84], but it yields a worse upper bound compared to a translation from ultra weak alternating automata to Büchi automata. This is why we present the latter. Another advantage of this translation is that it can be simplified by going through alternating generalized Büchi automata.

The main idea of the translation from ultra weak alternating automata to (generalized) Büchi automata is to use a powerset construction to keep track of the individual branches of an accepting run of the alternating automaton. There are two technical problems that we face in the translation. First, we need to take care of the vertices in the runs which are not labeled with a state (but with a transition condition), and, second, we need to take care of the acceptance condition. The first problem is similar to removing ε-transitions and the second problem can be solved by using the fact that the automata are ultra weak. The entire construction is described in Figure 21.

Lemma 4.8. Let $\mathscr{A}$ be an ultra weak alternating automaton with n states and k final states. Then $\mathscr{A}^{gBA}$ is an equivalent generalized Büchi automaton with 2^n states and k acceptance sets.

Let $\mathscr{A}$ be an ultra weak alternating automaton over a finite set P of propositional variables. The generalized Büchi automaton for $\mathscr{A}$, denoted $\mathscr{A}^{gBA}$, is defined by

$$\mathscr{A}^{gBA} = (2^P, 2^Q, \{q_I\}, \Delta, \mathscr{F})$$

where

- the transition relation Δ contains a transition (Q', a, Q'') if for every $q \in Q'$ there exists a set Q_q such that $q \rightarrow^{a,\delta(q)} Q_q$ and $\bigcup_{q \in Q'} Q_q \subseteq Q''$ and
- the set $\mathscr{F}$ of acceptance sets contains for every $q \notin F$ the set F_q defined by $\{Q' \subseteq Q \colon q^{\mathrm{d}} \in Q' \text{ or } q \notin Q'\}$.

FIGURE 21. From ultra weak to generalized Büchi automata

Proof. The claim about the number of states and the number of acceptance sets is obvious. We only need to show that the translation is correct.

First, assume $u \in \mathscr{L}(\mathscr{A})$. Then there is an accepting run $\mathscr{R}$ of $\mathscr{A}$ on u (which we assume to be minimal again). We say a vertex $v \in V^{\mathscr{R}}$ is a state vertex if the first component of its label is a state. Let $\mathscr{R}'$ be the tree which is obtained from $\mathscr{R}$ by "removing" the non-state vertices while keeping their edges. Formally, $\mathscr{R}'$ is constructed inductively as follows. We start with the root of $\mathscr{R}$, which is a state vertex by definition. Then, once we have a vertex v of $\mathscr{R}'$, we add all state vertices v' of $\mathscr{R}$ as successors of v to $\mathscr{R}'$ which can be reached from v in $\mathscr{R}$ via a path without state vertices (not counting the first and last vertex).

The tree $\mathscr{R}'$ has the following property. When v is a vertex labeled (q, i) and $\{v_0, \ldots, v_{m-1}\}$ is the set of its successors where v_j is labeled (q_j, i_j), then $q \rightarrow^{u(i)} \{q_0, \ldots, q_{m-1}\}$ and $i_j = i + 1$ for every $j < m$. This is because the definition of $\rightarrow^{a,\gamma}$ simply models the requirements of a run.

Using the above property of $\mathscr{R}'$ we can easily construct a run r of $\mathscr{A}^{gBA}$ on u as follows. We simply let $r(i)$ be the set of all q such that there exists a vertex v in $\mathscr{R}'$ labeled (q, i). By definition of $\mathscr{A}^{gBA}$, this is a run. What remains to be shown is that r is an accepting run.

Assume $q \notin F$ and i is an arbitrary number. We have to show that there exists $j \geq i$ such that $r(j) \in F_q$. If there is some $j \geq i$ such that $q \notin r(j)$, this is true. So assume that $q \in r(j)$ for all $j \geq i$. By construction

of $\mathscr{R}'$ there exists a vertex v_0 in $\mathscr{R}'$ which is labeled (q, i). If one of the successors of v_0 is labeled q^{d} in the first component, then $r(i+1) \in F_q$, which is enough. If, on the other hand, all successors are labeled distinct from q^{d} in their first component, then, since $\mathscr{A}$ is assumed to be ultra weak, one of the successors, say v_1, is labeled q in the first component. We can apply the same argument as before to v_1 now. We find that $r(i+2) \in F_q$ or we find a successor v_2 of v_1 with q in the first component of its label, too. If we continue like this and we do not find $r(j)$ such that $r(j) \in F_q$, we obtain an infinite path $v_0 v_1 \dots$ in $\mathscr{R}'$ where every v_i is labeled q in the first component. This path can be prefixed such it becomes a branch of $\mathscr{R}$, and this branch is not accepting—a contradiction to the assumption that $\mathscr{R}$ is accepting.

For the other direction, assume $u \in (2^P)^\omega$ is accepted by $\mathscr{A}^{gBA}$ and let r be an accepting run of $\mathscr{A}^{gBA}$ on u. For every i and every $q \in r(i)$, let Q_q^i be a set such that $q \to^{u(i),\delta(q)} Q_q^i$ for all $q \in r(i)$ and $\bigcup\{Q_q^i \colon q \in r(i)\} \subseteq r(i)$. By definition of $\mathscr{A}^{gBA}$, such sets exist. For some combinations of q and i there might be several choices for Q_q^i. By convention, if $q^{\mathrm{d}} \in r(i+1)$, we let $Q_q^i = \{q^{\mathrm{d}}\}$, which is a possible choice since $\mathscr{A}$ is assumed to be ultra weak. Using these sets, we construct a tree $\mathscr{R}'$ from r inductively as follows. We start with the root and label it $(q_I, 0)$. If we have a vertex v labeled (q, i), we add a successor to v for every $q' \in Q_q^i$ and label it $(q', i+1)$. By expanding $\mathscr{R}'$ according to the semantics of the transition conditions, we obtain a tree $\mathscr{R}$ which is a run of $\mathscr{A}$ on u. It remains to be shown that this run is accepting. Assume this is not the case. Then, because $\mathscr{A}$ is ultra weak, there is a non-final state q, a branch $v_0 v_1 \dots$ of $\mathscr{R}'$, and a number i such that the label of v_j is (q, j) for all $j \geq i$. This implies $q \in Q_q^i$ for all $j \geq i$. Since r is accepting, we know that there exists $j > i$ such that $q \notin r(j)$ or $q^{\mathrm{d}} \in r(j)$. The first condition is an immediate contradiction. So assume $q^{\mathrm{d}} \in r(j)$ for some $j > i$. Since we have $q \in r(j-1)$, we also have $Q_q^j = \{q^{\mathrm{d}}\}$ by construction—a contradiction. Q.E.D.

Combining the previous lemma and Remark 4.5 yields an alternative proof of Corollary 4.6. Very weak alternating automata are interesting for another reason, too:

Theorem 4.9 (Rohde, [Roh97])**.** For every very weak alternating automaton $\mathscr{A}$ there exists an LTL formula φ such that $\mathscr{L}(\varphi) = \mathscr{L}(\mathscr{A})$.

This was also proved by Löding and Thomas [LT00] and a proof of it can be found in [DG].

4.4 LTL Satisfiability, Model Checking, and Realizability

We can now return to the problems we are interested in, satisfiability, validity, model checking, and realizability.

Theorem 4.10 (Clarke-Emerson-Sistla, [CES83])**.** LTL satisfiability is PSPACE-complete.

Proof. Given an LTL formula φ over a set P of propositional variables, we construct a Büchi automaton equivalent to φ and check this automaton for nonemptiness. Clearly, this procedure is correct. To determine its complexity, we use the following simple fact from complexity theory.

(†) Let $f\colon A^* \to B^*$ be a function computable in PSPACE and $L \subseteq B^*$ a problem solvable in nondeterministic logarithmic space. Then $f^{-1}(P) \in$ PSPACE.

When we apply (†) to the situation where f computes the above Büchi automaton equivalent to φ and L is the problem whether a Büchi automaton accepts some word, then we obtain that our problem is in PSPACE.

For the lower bound, we refer the reader to [CES83] or [Sch02]. Q.E.D.

For model checking, the situation is essentially the same as with S1S. When we are given a finite-state automaton $\mathscr{D}$ over the alphabet 2^P for some finite set of propositional variables and φ is an LTL formula over P, we write $\mathscr{D} \models \varphi$ if $u \models \varphi$ for all $u \in \mathscr{L}(\mathscr{D})$. LTL model checking is the problem, given $\mathscr{D}$ and φ, to determine whether $\mathscr{D} \models \varphi$, that is, whether $\mathscr{L}(\mathscr{D}) \subseteq \mathscr{L}(\varphi)$.

Theorem 4.11. (Sistla-Clarke-Lichtenstein-Pnueli, [SC85, LP85])

(i) LTL model checking is PSPACE-complete.

(ii) Given a formula φ with n subformulas and a finite-state automaton $\mathscr{D}$ of size m, whether $\mathscr{D} \models \varphi$ holds can be checked in time $2^{O(n)}m$.

Proof. The same approach as in Section 2.1 yields the desired upper bounds. Given a finite set of propositional variables P, a finite-state automaton $\mathscr{D}$ over 2^P, and an LTL formula over P, we first construct the product $\mathscr{A} \times \mathscr{D}$ where $\mathscr{A}$ is a Büchi automaton equivalent to $\neg\varphi$. We have $\mathscr{L}(\mathscr{A} \times \mathscr{D}) = \varnothing$ if and only if $\mathscr{D} \models \varphi$. So, to conclude, we apply an emptiness test.

The number of states of the product is at most $(k+1)2^n \cdot m$ where n is the size of φ, k is the number of XU-formulas in φ (after transformation to positive normal form), and m is the number of states of $\mathscr{D}$. Using the same complexity-theoretic argument as in the proof of Theorem 4.10, we obtain part 1.

Part 2 follows from the fact that an emptiness test for a Büchi automaton can be carried out in time linear in the size of the automaton.

For the lower bound, we refer the reader to [CES83]. Q.E.D.

Finally, we turn to realizability, which is defined as with S1S (see Section 2.4). An LTL realizability instance is an LTL formula over a set $P = \{p_0, \ldots, p_{m-1}, q_0, \ldots, q_{n-1}\}$ of propositional variables. Just as earlier in this section, we interpret such formulas in words over $[2]_{m+n}$, which means that a solution of such an instance is a function $f\colon [2]_m^+ \to [2]_n$ satisfying the requirement known from the S1S setting, that is, $u^\frown v \models \varphi$ holds for every $u \in [2]_m^\omega$ and $v \in [2]_n^\omega$ defined by $v(i) = f(u[0,i])$ for every i.

We can use the same technique as in Section 3 to obtain the following result:

Theorem 4.12 (Pnueli-Rosner, [PR89])**.** LTL realizability is complete for doubly exponential time. Moreover, for every positive instance a finite-state machine realizing a finite-state solution can be computed within the same time bound.

Proof. Consider the following algorithm for solving a given instance φ over $\{p_0, \ldots, p_{m-1}, q_0, \ldots, q_{n-1}\}$. First, consider the game $\mathscr{G}[\varphi]$ which is obtained using the construction from Figure 10 with the S1S formula replaced by the LTL formula. Second, compute a Büchi automaton $\mathscr{A}$ equivalent to φ according to Corollary 4.6. Third, turn $\mathscr{A}$ into a deterministic parity automaton $\mathscr{B}$ according to 2.14. Fourth, let $\mathscr{G} = \mathscr{G}[\varphi] \times \mathscr{B}$ be the game obtained from expanding $\mathscr{G}[\varphi]$ by $\mathscr{B}$. Fifth, solve the game $\mathscr{G}$ using Theorem 2.21. Player 0 wins $\mathscr{G}$ if and only if φ is a positive instance of realizability.

To prove the desired complexity bound let n be the number of subformulas of φ and observe the following. The size of $\mathscr{A}$ is at most $(n+1)2^n$. Therefore, the worst-case size of $\mathscr{B}$ is $2^{O(2^n n \log n)}$ and $\mathscr{B}$ has at most $3(n+1)2^n$ priorities. Theorem 2.21 now gives the desired upper bound.

The additional claim about the finite-state solution follows from Lemmas 2.16 and 2.17. For the lower bound, see [Ros92]. Q.E.D.

In the remainder of this section, we present an alternative approach to solving the realizability problem, which is interesting in its own right.

Let φ be an instance of the realizability problem as above. Formally, a solution of φ is a function $f\colon [2]_m^+ \to [2]_n$. Such a function is the same as a $[2]_m$-branching $[2]_n$-labeled tree (where the root label is ignored). In other words, the set of all solutions of a given instance of the realizability problem is a tree language. This observation transforms the realizability problem into the framework of tree languages and tree automata, and we can apply tree-automata techniques to solve it.

Let $t\colon [2]_m^* \to [2]_n$ be any $[2]_m$-branching $[2]_n$-labeled tree. The tree can be turned into a potential solution to the instance φ if the label of the root is forgotten. The resulting function is denoted by $t_{-\varepsilon}$. We set $\mathscr{L}_{\mathrm{sol}}(\varphi) = \{t\colon [2]_m^* \to [2]_n \colon t_{-\varepsilon} \text{ solves } \varphi\}$.

We next show that $\mathscr{L}_{\mathrm{sol}}(\varphi)$ is a tree language which can be recognized by a universal tree automaton. We need, however, a more general notion of universal tree automaton as in Section 3.3. Also, we need to massage the formula φ a little to arrive at a simple automata-theoretic construction.

A universal co-Büchi tree automaton with set of directions D is a tuple

$$(A, D, Q, q_I, \Delta, F)$$

where A, Q, q_I, and F are as usual, and where D is a finite set of directions and $\Delta \subseteq Q \times A \times D \times Q$ is a transition relation. Following the definition from Section 3.3, a word $r \in Q^\omega$ is said to be a run for branch $u \in D^\omega$ if $(r(i), t(u[0,i)), u(i), r(i+1)) \in \Delta$ for every i and $r(0) = q_I$. A tree is accepted if every $r \in Q^\omega$ which is a run for some branch satisfies the co-Büchi acceptance condition. The latter means that $r(i) \in F$ only for finitely many i.

The technical problem one faces when constructing an automaton for $\mathscr{L}_{\mathrm{sol}}(\varphi)$ is that a tree automaton has transitions of the form (q, a, d, q'), so, when applied to the above setting, in one transition the automaton consumes an output of the device we are looking for and the next input. For our construction it would be much better to have automata that in one transition consume an input and a corresponding output. Rather than modifying our standard automaton model, we resolve the issue on the logical side. For a given formula $\varphi = \varphi(p_0, \ldots, p_{m-1}, q_0, \ldots, q_{n-1})$ we consider the formula φ^{X} defined by

$$\varphi^{\mathsf{X}} = \varphi(p_0, \ldots, p_{m-1}, \mathsf{X}q_0, \ldots, \mathsf{X}q_{n-1}).$$

(Recall that X stands for the temporal operator "next".) This formula moves the output one position to the right, more precisely,

$$\mathscr{L}(\varphi) = \{d^0 {}^\frown a^1 d^1 {}^\frown a^2 \ldots : d^0 \frown a^0 d^1 {}^\frown a^1 \cdots \in \mathscr{L}(\varphi^{\mathsf{X}})\}. \tag{1.2}$$

A universal co-Büchi tree automaton for a given LTL formula φ as above is now easily constructed, as can be seen in Figure 22.

Lemma 4.13. Let $\varphi = \varphi(p_0, \ldots, p_{m-1}, q_0, \ldots, q_{n-1})$ be an instance of the LTL realizability problem. Then $\mathscr{L}(\mathscr{A}^{\mathrm{real}}[\varphi]) = \mathscr{L}_{\mathrm{sol}}(\varphi)$. Q.E.D.

Universal co-Büchi tree automata for D-branching trees as defined above are a special case of universal parity tree automata for D-branching trees, which can be turned into nondeterministic parity tree automata for D-branching trees in the same fashion as this was explained for automata on binary trees in Figure 16. The same is true for the emptiness test for parity tree automata on D-branching trees, which can be solved by constructing a parity game along the lines of the construction depicted in Figure 13 and solving this game.

Let $\varphi = \varphi(p_0, \ldots, p_{m-1}, q_0, \ldots, q_{n-1})$ be an instance of the LTL realizability problem and $\mathscr{A}$ a Büchi automaton such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\neg\varphi^{\mathsf{X}})$. The universal co-Büchi tree automaton for φ, denoted $\mathscr{A}^{real}[\varphi]$, is defined by

$$\mathscr{A}^{real}[\varphi] = ([2]_n, [2]_m, Q, q_I, \Delta', F)$$

where

$$\Delta' = \{(q, a, d, q') \colon (q, d^\frown a, q') \in \Delta\}.$$

FIGURE 22. From an LTL realizability instance to a universal tree automaton

4.5 Notes

The automata-theoretic decision procedure for LTL model checking described in this section has had a great practical impact, because it has been implemented in an industrial setting, see, for instance, [Hol97], and used to verify real-world computing systems (mostly hardware). Much research has gone into improving the algorithm in several respects, but also into extending its applicability, for instance, more expressive logics and larger classes of devices have been looked at, see, for instance, [BFG$^+$05, CRST06, FWW97, KPV02]. It is also noteworthy that LTL is the basis for industrial specification languages such as as ForSpec [AFF$^+$02] and PSL [EF06] and that the automata-theoretic approach underlies industrial implementations of specification languages [AKTZ06].

An important aspect of this section is the use of alternating automata, which were introduced into the theory of automata on infinite objects by Muller, Schupp, and Saoudi [MSS88]. The only gain from this presented in the current section is Theorem 4.9, but this is probably the least important aspect in this context. What is more important is that weak alternating automata are as powerful as nondeterministic Büchi automata, which was proved by Kupferman and Vardi [KV97, KV99]. This result motivated new research, which, for instance, brought about new complementation constructions [KV97, KV99, Tho99]. As we see in the subsequent section, alternation is even more important in the context of tree languages.

We refer to [Var07] for a collection of open algorithmic issues with regard to automata-theoretic LTL model checking.

5 Computation Tree Logic

Certain temporal properties of a system cannot be specified when runs of the system are considered separately, as we do this with LTL. For instance, when one wants to specify that no matter which state a system is in there is some way to get back to a default state, then this cannot be stated in LTL. The reason is that the property says something about how a run can evolve into different runs.

This observation motivates the introduction of specification logics that compensate for the lack of expressive power in this regard. The first logic of this type, called UB, was introduced by Ben-Ari, Manna, and Pnueli [BAMP81] in 1981. Another logic of this type is computation tree logic (CTL), designed by Clarke and Emerson [EC82], which is interpreted in the "computation tree" of a given transition system. This is the logic we study in this section, in particular, we study satisfiability and model checking for this logic.

Many of the proofs in this section are very similar to proofs in the previous section. In these cases, we only give sketches, but describe the differences in detail.

5.1 CTL and Monadic Second-Order Logic

CTL mixes path quantifiers and temporal operators in a way such that a logic arises for which model checking can be carried out in polynomial time. The syntax of CTL is as follows:

- tt and ff are CTL formulas,
- every propositional variable is a CTL formula,
- if φ is a CTL formula, then so is $\neg\varphi$,
- if φ and ψ are formulas, then so are $\varphi \vee \psi$, $\mathsf{E}(\varphi\,\mathsf{XU}\,\psi)$, and $\mathsf{A}(\varphi\,\mathsf{XU}\,\psi)$.

CTL formulas are interpreted in transition systems, which we introduce next. Such a system is a simple, state-based abstraction of a computing device. Formally, it is a tuple

$$\mathscr{S} = (P, S, \to, l)$$

where P is a finite set of propositional variables, S is a set of states, $\to \subseteq S \times S$ is a transition relation in infix notation, and $l\colon S \to 2^P$ is a labeling function assigning to each state which propositional variables are true in it. A computation of such a transition system starting in a state s is a word $u \in S^+ \cup S^\infty$ such that

(i) $u(0) = s$,

(ii) $u(i) \to u(i+1)$ for all i with $i+1 < |u|$, and

(iii) u is maximal in the sense that if u is finite, then $u(*)$ must not have any successor.

Given a CTL formula φ, a transition system $\mathscr{S}$ over the same set P of propositional variables, and a state s of $\mathscr{S}$, it is defined whether φ holds true in $\mathscr{S}$ at s, denoted $\mathscr{S}, s \models \varphi$:

- $\mathscr{S}, s \models \mathsf{tt}$ and $\mathscr{S}, s \not\models \mathsf{ff}$,
- $\mathscr{S}, s \models p$ if $p \in l(s)$,
- $\mathscr{S}, s \models \neg\varphi$ if $\mathscr{S}, s \not\models \varphi$,
- $\mathscr{S}, s \models \psi \vee \chi$ if $\mathscr{S}, s \models \psi$ or $\mathscr{S}, s \models \chi$, for ψ and χ CTL formulas,
- $\mathscr{S}, s \models \mathsf{E}(\psi\, \mathsf{XU}\, \chi)$ if there exists a computation u of $\mathscr{S}$ starting at s and $j > 0$ such that $\mathscr{S}, u(j) \models \chi$ and $\mathscr{S}, u(i) \models \psi$ for all i with $0 < i < j$.
- $\mathscr{S}, s \models \mathsf{A}(\psi\, \mathsf{XU}\, \chi)$ if for all computations u of $\mathscr{S}$ starting at s there exists $j > 0$ such that $\mathscr{S}, u(j) \models \chi$ and $\mathscr{S}, u(i) \models \psi$ for all i with $0 < i < j$.

Just as with LTL, other operators can be defined:

- "in all computations always" is defined by $\mathsf{AG}\varphi = \varphi \wedge \neg\mathsf{E}(\mathsf{tt}\, \mathsf{XU}\, \neg\varphi)$,
- "in some computation eventually" is defined by $\mathsf{EF}\varphi = \varphi \vee \mathsf{E}(\mathsf{tt}\, \mathsf{XU}\, \varphi)$.

An interesting property one can express in CTL is the one discussed above, namely that from every state reachable from a given state a distinguished state, indicated by the propositional variable p_d, can be reached:

$$\mathsf{AG}\, \mathsf{EF} p_d. \tag{1.3}$$

Another property that can be expressed is that every request, indicated by the propositional variable p_r, is eventually acknowledged, indicated by the propositional variable p_a:

$$\mathsf{AG}(p_r \to \mathsf{AX}\, \mathsf{AF} p_a). \tag{1.4}$$

It is interesting to compare the expressive power of CTL with that of LTL. To this end, it is reasonable to restrict the considerations to infinite computations only and to say that a CTL formula φ and an LTL formula ψ are equivalent if for every transition system $\mathscr{S}$ and every state $s \in S$

the following holds: $\mathscr{S}, s \models \varphi$ iff $l(u(0))l(u(1)) \ldots \models \psi$ for all infinite computations u of $\mathscr{S}$ starting in s.

The second property from above can be expressed easily in LTL, namely by the formula $\mathsf{G}(p_r \to \mathsf{XF}p_a)$, that is, this formula and (1.4) are equivalent. Clarke and Draghicescu showed that a CTL property is equivalent to some LTL formula if and only if it is equivalent to the LTL formula obtained by removing the path quantifiers [CD88]. But it is not true that every LTL formula which can be expressed in CTL is expressible by a CTL formula which uses universal path quantifiers only. This was shown by Bojanczyk [Boj07].

An LTL formula which is not expressible in CTL is

$$\mathsf{GF}p, \tag{1.5}$$

which was already pointed out by Lamport [Lam80].

In order to be able to recast satisfiability and model checking in a (tree) automata setting, it is crucial to observe that CTL formulas cannot distinguish between a transition system and the transition system obtained by "unraveling" it. Formally, the unraveling of the transition system $\mathscr{S}$ at state $s \in S$, denoted $\mathscr{T}_s(\mathscr{S})$, is the tree inductively defined by:

- s is the root of $\mathscr{T}_s(\mathscr{S})$,
- if $v \in S^+$ is an element of $V^{\mathscr{T}_s(\mathscr{S})}$ and $v(*) \to s'$, then $vs' \in V^{\mathscr{T}_s(\mathscr{S})}$ and $(v, vs') \in E^{\mathscr{T}_s(\mathscr{S})}$,
- $l^{\mathscr{T}_s(\mathscr{S})}(v) = l^{\mathscr{S}}(v(*))$ for every $v \in V^{\mathscr{T}_s(\mathscr{S})}$.

Henceforth, a tree with labels from 2^P, such as the unraveling of a transition system, is viewed as a transition system in the canonical way. When we interpret a CTL formula in a tree and do not indicate a vertex, then the formula is interpreted at the root of the tree.

The formal statement of the above observation can now be phrased as follows.

Lemma 5.1. For every CTL formula φ, transition system $\mathscr{S}$, and state $s \in S$,

$$\mathscr{S}, s \models \varphi \qquad \text{iff} \qquad \mathscr{T}_s(\mathscr{S}) \models \varphi.$$

Proof. This can be proved by a straightforward induction on the structure of φ, using a slightly more general claim:

$$\mathscr{S}, s' \models \varphi \qquad \text{iff} \qquad \mathscr{T}_s(\mathscr{S}), v \models \varphi$$

for every state $s' \in S$ and every vertex v of $\mathscr{T}_s(\mathscr{S})$ where $v(*) = s'$. Q.E.D.

The previous lemma says that we can restrict attention to trees, in particular, a CTL formula is satisfiable if and only if there is a tree which is a model of it. So when we translate CTL formulas into logics on trees which satisfiability is decidable for, then we also know that CTL satisfiability is decidable.

We present a simple translation of CTL into monadic second-order logic. There is, however, an issue to be dealt with: S2S formulas specify properties of binary trees, but CTL is interpreted in transition systems where each state can have more than just two successors. A simple solution is to use a variant of S2S which allows any number of successors but has only a single successor predicate, suc. Let us call the resulting logic SUS. As with LTL, we identify the elements of 2^P for $P = \{p_0, \ldots, p_{n-1}\}$ with the elements of $[2]_n$.

Proposition 5.2. Let $P = \{p_0, \ldots, p_{n-1}\}$ be an arbitrary finite set of propositional variables. For every CTL formula φ over P an SUS formula $\tilde{\varphi} = \tilde{\varphi}(X_0, \ldots, X_{n-1})$ can be constructed such that $\mathscr{T} \models \varphi$ if and only if $\mathscr{T} \models \tilde{\varphi}$ for all trees $\mathscr{T}$ over 2^P (or $[2]_n$).

Proof. What we actually prove is somewhat stronger, analogous to the proof for LTL. We construct a formula $\hat{\varphi} = \hat{\varphi}(X_0, \ldots, X_{n-1}, x)$ such that $\mathscr{T}, v \models \varphi$ if and only if $\mathscr{T}, v \models \hat{\varphi}$ for all trees $\mathscr{T}$ and $v \in V^{\mathscr{T}}$. We can then set $\tilde{\varphi} = \exists x(\varphi_{root}(x) \wedge \hat{\varphi})$ where $\varphi_{root}(x) = \forall y(\neg \mathrm{suc}(y, x))$ specifies that x is the root.

For the induction base, assume $\varphi = p_i$. We can set $\hat{\varphi}$ to $x \in X_i$. Similarly, for $\varphi = \neg p_i$ we can set $\hat{\varphi}$ to $\neg x \in X_i$.

In the inductive step, we consider only one of the interesting cases, namely where $\varphi = \mathsf{A}(\psi \,\mathsf{XU}\, \chi)$. We start with a formula $\varphi_{closed} = \varphi_{closed}(X)$ which is true if every element of X has a successor in X provided it has a successor at all:

$$\varphi_{closed} = \forall x(x \in X \wedge \exists y(\mathrm{suc}(x, y)) \rightarrow \exists y(\mathrm{suc}(x, y) \wedge y \in X)).$$

We next write a formula $\varphi_{path}(x, X)$ which is true if X is a maximum path starting in x:

$$\varphi_{path} = x \in X \wedge \varphi_{closed}(X) \wedge \forall Y(x \in Y \wedge \varphi_{closed}(Y) \wedge Y \subseteq X \rightarrow X = Y).$$

We can then set

$$\hat{\varphi} = \forall X(\varphi_{path}(x, X) \rightarrow \exists z(z \in X \wedge \neg z = x \wedge \hat{\chi}(z) \wedge \forall y(x < y < z \rightarrow \hat{\psi}(y))).$$

The other CTL operators can be dealt with in a similar fashion. Q.E.D.

The desired decidability result now follows from the following result on SUS.

Theorem 5.3 (Walukiewicz, [Wal96])**.** SUS satisfiability is decidable.

This result can be proved just as we proved the decidability of satisfiability for S2S, that is, using an analogue of Rabin's Theorem. This analogue will use a different kind of tree automaton model which takes into account that the branching degree of the trees considered is unbounded and that there is one predicate for all successors. More precisely, a transition in such an automaton is of the form $(q, a, Q^{\mathsf{E}}, Q^{\mathsf{A}})$ where $Q^{\mathsf{E}}, Q^{\mathsf{A}} \subseteq Q$. Such a transition is to be read as follows: If the automaton is in state q at a vertex labeled a, then for every $q' \in Q^{\mathsf{E}}$ there exists exactly one successor that gets assigned q' and all the successors that do not get assigned any state in this fashion get assigned a state from Q^{A}. In particular, if $Q^{\mathsf{E}} = Q^{\mathsf{A}} = \varnothing$, then the vertex must not have a successor. In [Wal96], Walukiewicz actually presents a theorem like Büchi's and Rabin's: He shows that there is a translation in both directions, from SUS formulas to such automata and back.

Corollary 5.4. CTL satisfiability and model checking are decidable.

That model checking is decidable follows from the simple observation that in SUS one can define the unraveling of every finite transition system.

We conclude this introduction to CTL with further remarks on SUS and its relationship to CTL. There is a logic related to SUS which was already studied by Rabin and which he denoted SωS. This is the logic interpreted in the countably branching tree ω^* where, for each i, there is a separate successor relation $\mathrm{suc}_i(\cdot, \cdot)$. Observe that—as noted in [JW96]—in this logic one cannot even express that all successors of the root belong to a certain set, which can easily be expressed in CTL and SUS.

Observe, too, that in SUS one can express that every vertex of a tree has at least two successors, namely by

$$\forall x(\exists y_0 \exists y_1(\mathrm{suc}(x, y_0) \wedge \mathrm{suc}(x, y_1) \wedge \neg y_0 = y_1).$$

This is, however, impossible in CTL. More precisely, CTL cannot distinguish between bisimilar transition systems whereas SUS can do this easily.

5.2 From CTL to Nondeterministic Tree Automata

We next show how to arrive at good complexity bounds for satisfiability and model checking by following a refined automata-theoretic approach. For satisfiability, we can use nondeterministic automata and vary the approach we used for handling LTL in Section 4, while for model checking, we have to use alternating tree automata.

As pointed out above, the nondeterministic tree automaton model we defined in Section 3 was suited for binary trees only, which is not enough in the context of CTL. Here, we need an automaton model that can handle trees with arbitrary branching degree. We could use the tree automaton model explained in Section 5.1, but there is another model which is more appropriate. Following Janin and Walukiewicz [JW95], we use a tree automaton model which takes into account that properties like the one mentioned at the end of Section 5.1 cannot be expressed in CTL.

A generalized Büchi tree automaton in this context is a tuple

$$\mathscr{A} = (A, Q, Q_I, \Delta, \mathscr{F})$$

where A, Q, Q_I, and $\mathscr{F}$ are as with generalized Büchi (word) automata and $\Delta \subseteq Q \times A \times 2^Q$ is a transition relation.

A transition of the form (q, a, Q') is to be read as follows: If the automaton is in state q at vertex v and reads the label a, then it sends each state from Q' to at least one of the successors of v and every successor of v is at least sent one of the states from Q'; the same successor can get sent several states.

Formally, a run of $\mathscr{A}$ on a tree $\mathscr{T}$ is a $(Q \times V^{\mathscr{T}})$-labeled tree $\mathscr{R}$ satisfying the following conditions.

(i) The root of $\mathscr{R}$ is labeled $(q, \mathrm{root}(\mathscr{T}))$ for some $q \in Q_I$.

(ii) For every vertex $w \in V^{\mathscr{R}}$, if (q, v) is the label of w, then there exists a transition $(q, l(v), Q') \in \Delta$ such that:

  (a) For every $v' \in \mathrm{succs}_{\mathscr{T}}(v)$ there exists $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ labeled (q', v') for some $q' \in Q'$, that is, every successor of v occurs in a label of a successor of w.

  (b) For every $q' \in Q'$ there exist $v' \in \mathrm{succs}_{\mathscr{T}}(v)$ and $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that w' is labeled (q', v). That is, every state from Q' occurs at least once among all successors of w.

Such a run is accepting if every branch is accepting with respect to the given generalized Büchi condition just as this was defined for generalized Büchi word automata.

Observe that in this model the unlabeled tree underlying a run may not be the same as the unlabeled tree underlying a given input tree. Copies of subtrees may occur repeatedly.

As an example, let $P = \{p\}$ and $A = 2^P$ and consider the tree language L which contains all trees over A that satisfy the property that every branch is either finite or labeled $\{p\}$ infinitely often. An appropriate Büchi automaton

has two states, q_I and $q_{\{p\}}$, where q_I is initial and q_p is final, and the transitions are $(q, a, \{q_a\})$ and $(q, a, \varnothing)$ for any state q and letter a.

The idea for translating a given CTL formula into a nondeterministic tree automaton follows the translation of LTL into nondeterministic word automata: In each vertex, the automaton guesses which subformulas of the given formula are true and verifies this. The only difference is that the path quantifiers E and A are taken into account, which is technically somewhat involved. The details are given in Figure 23, where the following notation and terminology is used. Given a set Ψ of CTL formulas over a finite set P of propositional variables and a letter $a \in 2^P$ we say that Ψ is consistent with a if

- $\mathsf{ff} \notin \Psi$,
- $p \in \Psi$ iff $p \in a$, for all $p \in P$, and
- for $\psi \in \Psi$, if $\psi = \psi_0 \vee \psi_1$, then $\psi_i \in \Psi$ for some $i < 2$, and if $\psi = \psi_0 \wedge \psi_1$, then $\{\psi_0, \psi_1\} \subseteq \Psi$.

Further, a set Ψ' is a witness for $\mathsf{E}(\psi \mathsf{XU} \chi)$ if $\chi \in \Psi'$ or $\{\psi, \mathsf{E}(\psi \mathsf{XU} \chi)\} \subseteq \Psi'$. Similarly, Ψ' is a witness for $\mathsf{E}(\psi \mathsf{XR} \chi)$ if $\{\psi, \chi\} \subseteq \Psi'$ or $\{\chi, \mathsf{E}(\psi \mathsf{XR} \chi)\} \subseteq \Psi'$. The analogue terminology is used for A-formulas. When Ψ is a set of CTL formulas, then Ψ_{E} denotes the formulas of the form $\mathsf{E}(\psi \mathsf{XU} \chi)$ and $\mathsf{E}(\psi \mathsf{XR} \chi)$, that is, the set of all E-formulas in Ψ, and, similarly, Ψ_{A} denotes the set of all A-formulas in Ψ.

The only interesting aspect of the construction is (iv) of the definition of a transition. It would be more natural to omit (iv), and, indeed, the construction would then also be correct, but the resulting automaton would be too large. On the other hand, (iv) is not a real restriction, because the semantics of CTL requires only one "witness" for every existential path formula. The size of Q' must be by one larger than the number of existential subformulas because some of the successors of a vertex may not witness any existential formula, but they must be assigned a state.

Before formally stating the correctness of the construction, we introduce a notion referring to the number of different states which can be assigned in a transition. We say that a nondeterministic tree automaton $\mathscr{A}$ is m-bounded if $|Q'| \leq m$ holds for every $(q, a, Q') \in \Delta$.

Lemma 5.5. Let φ be an arbitrary CTL formula with n subformulas, m E-subformulas, and k U-subformulas. Then $\mathscr{A}[\varphi]$ is an $(m+1)$-bounded generalized Büchi tree automaton with 2^n states, k acceptance sets, and such that $\mathscr{L}(\mathscr{A}[\varphi]) = \mathscr{L}(\varphi)$.

Proof sketch. The claim about the size of the automaton is trivial. The proof of its correctness can be carried out similar to the proof of Theo-

Let P be a finite set of propositional variables, φ a CTL formula over P in positive normal form, and Φ the set of subformulas of φ. The generalized Büchi tree automaton for φ with respect to P, denoted $\mathscr{A}[\varphi]$, is defined by

$$\mathscr{A}[\varphi] = (2^P, 2^{\mathrm{sub}(\varphi)}, Q_I, \Delta, \mathscr{F})$$

where $Q_I = \{\Psi \subseteq 2^{\mathrm{sub}(\varphi)} \colon \varphi \in \Psi\}$ and

$$\mathscr{F} = \{F_{\mathsf{Q}[\psi \mathsf{XU} \chi]} \colon \mathsf{Q}[\psi \,\mathsf{XU}\, \chi] \in \mathrm{sub}(\varphi) \text{ and } \mathsf{Q} \in \{\mathsf{E}, \mathsf{A}\}\}$$

with

$$F_{\mathsf{Q}[\psi \mathsf{XU} \chi]} = \{\Psi \subseteq \Phi \colon \chi \in \Psi \text{ or } \mathsf{Q}[\psi \,\mathsf{XU}\, \chi] \notin \Psi\},$$

and where Δ contains a transition (Ψ, a, Q') if the following conditions are satisfied:

(i) Ψ is consistent with a,

(ii) for every $\psi \in \Psi_{\mathsf{E}}$ there exists $\Psi' \in Q'$ which witnesses it,

(iii) every $\Psi' \in Q'$ witnesses every $\psi \in \Psi_{\mathsf{A}}$,

(iv) $|Q'| \leq |\mathrm{sub}(\varphi)_{\mathsf{E}}| + 1$.

FIGURE 23. From CTL to generalized Büchi tree automata

rem 4.4, that is, one proves $\mathscr{L}(\mathscr{A}[\varphi]) \subseteq \mathscr{L}(\varphi)$ by induction on the structure of φ and $\mathscr{L}(\varphi) \subseteq \mathscr{L}(\mathscr{A}[\varphi])$ by constructing an accepting run directly. Q.E.D.

It is very easy to see that the construction from Figure 18 can also be used in this context to convert a generalized Büchi tree automaton into a Büchi automaton. To be more precise, an m-bounded generalized Büchi tree automaton with n states and k acceptance sets can be converted into an equivalent m-bounded Büchi tree automaton with $(k+1)n$ states.

So in order to solve the satisfiability problem for CTL we only need to solve the emptiness problem for Büchi tree automata in this context. There is a simple way to perform an emptiness test for nondeterministic tree

Let $\mathscr{A}$ be a nondeterministic Büchi tree automaton. The emptiness game for $\mathscr{A}$, denoted $\mathscr{G}_\varnothing[\mathscr{A}]$, is defined by

$$\mathscr{G}_\varnothing[\mathscr{A}] = (Q, \Delta, q_I, M_0 \cup M_1, F)$$

where

$$M_0 = \{(q, Q') \colon \exists a((q, a, Q') \in \Delta)\}, \quad M_1 = \{(Q', q) \colon q \in Q'\}.$$

FIGURE 24. Emptiness game for nondeterministic Büchi tree automaton

automata, namely by using the same approach as for nondeterministic tree automata working on binary trees: The nonemptiness problem is phrased as a game. Given a nondeterministic Büchi tree automaton $\mathscr{A}$, we define a game which Player 0 wins if and only if some tree is accepted by $\mathscr{A}$. To this end, Player 0 tries to suggest suitable transitions while Player 1 tries to argue that Player 0's choices are not correct. The details of the construction are given in Figure 24.

Lemma 5.6. Let $\mathscr{A}$ be a nondeterministic Büchi tree automaton. Then the following are equivalent:
(A) $\mathscr{L}(\mathscr{A}) \neq \varnothing$.
(B) Player 0 wins $\mathscr{G}_\varnothing[\mathscr{A}]$.

Proof. The proof of the lemma can be carried out along the lines of the proof of Lemma 3.3. The only difference is due to the arbitrary branching degree, which can easily be taken care of. One only needs to observe that if there exists a tree which is accepted by $\mathscr{A}$, then there is a tree with branching degree at most $|Q|$ which is accepted. Q.E.D.

We have the following theorem:

Theorem 5.7 (Emerson-Halpern-Fischer-Ladner, [EH85, FL79])**.** CTL satisfiability is complete for deterministic exponential time.

Proof. The decision procedure is as follows. A given CTL formula φ is first converted into an equivalent generalized Büchi tree automaton $\mathscr{A}$ using the construction from Figure 23. Then $\mathscr{A}$ is converted into an equivalent Büchi tree automaton $\mathscr{B}$ using the natural adaptation of the construction presented in Figure 18 to trees. In the third step, $\mathscr{B}$ is converted into the Büchi

game $\mathscr{G}_{\varnothing}[\mathscr{B}]$, and, finally, the winner of this game is determined. (Recall that a Büchi condition is a parity condition with two different priorities.)

From Theorem 2.21 on the complexity of parity games it follows immediately that Büchi games (parity games with two different priorities) can be solved in polynomial time, which means we only need to show that the size of $\mathscr{G}_{\varnothing}[\mathscr{B}]$ is exponential in the size of the given formula φ and can be constructed in exponential time. The latter essentially amounts to showing that $\mathscr{B}$ is of exponential size.

Let n be the number of subformulas of φ. Then $\mathscr{A}$ is n-bounded with 2^n states and at most n acceptance sets. This means that the number of sets Q' occurring in the transitions of $\mathscr{A}$ is at most 2^{n^2}, so there are at most 2^{n^2+2n} transitions (recall that there are at most 2^n letters in the alphabet). Similarly, $\mathscr{B}$ is n-bounded, has at most $(n+1)2^n$ states, and $2^{O(n^2)}$ transitions.

The upper bound is given in [FL79]. Q.E.D.

5.3 From CTL to Alternating Tree Automata

One of the crucial results of Emerson and Clarke on CTL is that model checking of CTL can be carried out in polynomial time. The decision procedure they suggested in [CES86] is a simple labeling algorithms. For every subformula ψ of a given formula φ they determine in which states of a given transition system $\mathscr{S}$ the formula ψ holds and in which it does not hold. This is trivial for atomic formulas. It is straightforward for conjunction and disjunction, provided it is known which of the conjuncts and disjuncts, respectively, hold. For XR- and XU-formulas, it amounts to simple graph searches.

Emerson and Clarke's procedure cannot easily be seen as a technique which could also be derived following an automata-theoretic approach. Consider the nondeterministic tree automaton we constructed in Figure 23. Its size is exponential in the size of the given formula (and this cannot be avoided), so it is unclear how using this automaton one can arrive at a polynomial-time procedure.

The key for developing an automata-theoretic approach, which is due to Kupferman, Vardi, and Wolper [KVW00], is to use alternating tree automata similar to how we used alternating automata for LTL in Section 4 and to carefully analyze their structure.

An alternating Büchi tree automaton is a tuple

$$\mathscr{A} = (P, Q, q_I, \delta, F)$$

where P, Q, q_I, and F are as usual and δ is the transition function which assigns to each state a transition condition. The set of transition conditions over P and Q, denoted $\mathrm{TC}(P, Q)$, is the smallest set such that

(i) $\mathsf{tt}, \mathsf{ff} \in \mathrm{TC}(P, Q)$,

(ii) $p, \neg p \in \mathrm{TC}(P, Q)$ for every $p \in P$,

(iii) every positive boolean combination of states is in $\mathrm{TC}(P, Q)$,

(iv) $\Diamond\gamma, \Box\gamma \in \mathrm{TC}(P, Q)$ where γ is a positive boolean combination of states.

This definition is very similar to the definition for alternating automata on words. The main difference reflects that in a tree a "position" can have several successors: $\Diamond$ expresses that a copy of the automaton should be sent to one successor, while $\Box$ expresses that a copy of the automaton should be sent to all successors. So $\Diamond$ and $\Box$ are the two variants of $\bigcirc$.

There is another, minor difference: For tree automata, we allow positive boolean combinations of states as transitions conditions (without $\Diamond$ or $\Box$ occurring in it). We could have allowed this for word automata, too, but it would not have helped us. Here, it makes our constructions simpler, but the proofs will be slightly more involved.

Let $\mathscr{T}$ be a 2^P-labeled tree. A tree $\mathscr{R}$ with labels from $\mathrm{TC}(P, Q) \times V^{\mathscr{T}}$ is a run of $\mathscr{A}$ on $\mathscr{T}$ if $l^{\mathscr{R}}(\mathrm{root}(\mathscr{R})) = (q_I, \mathrm{root}(\mathscr{T}))$ and the following conditions are satisfied for every vertex $w \in V^{\mathscr{R}}$ with label (γ, v):

- $\gamma \neq \mathsf{ff}$,
- if $\gamma = p$, then $p \in l^{\mathscr{T}}(w)$, and if $\gamma = \neg p$, then $p \notin l^{\mathscr{T}}(w)$,
- if $\gamma = \Diamond\gamma'$, then there exists $v' \in \mathrm{succs}_{\mathscr{T}}(v)$ and $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that $l^{\mathscr{R}}(w') = (\gamma', v')$,
- if $\gamma = \Box\gamma'$, then for every $v' \in \mathrm{succs}_{\mathscr{T}}(v)$ there exists $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that $l^{\mathscr{R}}(w') = (\gamma', v')$,
- if $\gamma = \gamma_0 \vee \gamma_1$, then there exists $i < 2$ and $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that $l^{\mathscr{R}}(w') = (\gamma_i, v)$,
- if $\gamma = \gamma_0 \wedge \gamma_1$, then for every $i < 2$ there exists $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that $l^{\mathscr{R}}(w') = (\gamma_i, v)$.

Such a run is accepting if on every infinite branch there exist infinitely many vertices w labeled with an element of F in the first component.

The example language from above can be recognized by an alternating Büchi automaton which is slightly more complicated than the nondeterministic automaton, because of the restrictive syntax for transition conditions.

Let φ be a CTL formula in positive normal form over P and Q the set which contains for each $\psi \in \mathrm{sub}(\varphi)$ an element denoted $[\psi]$. The automaton $\mathscr{A}^{alt}[\varphi]$ is defined by

$$\mathscr{A}^{alt}[\varphi] = (P, Q, [\varphi], \delta, F)$$

where

$$\begin{aligned} \delta([\mathsf{tt}]) &= \mathsf{tt}, & \delta([\mathsf{ff}]) &= \mathsf{ff}, \\ \delta([p]) &= p, & \delta([\neg p]) &= \neg p, \\ \delta([\psi \vee \chi]) &= [\varphi] \vee [\psi], & \delta([\psi \wedge \chi]) &= [\varphi] \wedge [\psi], \end{aligned}$$

$$\begin{aligned} \delta([\mathsf{E}(\psi\,\mathsf{XU}\,\chi)]) &= \Diamond([\chi] \vee ([\psi] \wedge [\mathsf{E}(\psi\,\mathsf{XU}\,\chi)])), \\ \delta([\mathsf{E}(\psi\,\mathsf{XR}\,\chi)]) &= \Diamond([\chi] \wedge ([\chi] \vee [\mathsf{E}(\psi\,\mathsf{XR}\,\chi)])), \\ \delta([\mathsf{A}(\psi\,\mathsf{XU}\,\chi)]) &= \Box([\chi] \vee ([\psi] \wedge [\mathsf{A}(\psi\,\mathsf{XU}\,\chi)])), \\ \delta([\mathsf{A}(\psi\,\mathsf{XR}\,\chi)]) &= \Box([\chi] \wedge ([\chi] \vee [\mathsf{A}(\psi\,\mathsf{XR}\,\chi)])), \end{aligned}$$

and F contains all the elements $[\psi]$ where ψ is not an XU-formula.

FIGURE 25. From CTL to alternating tree automata

We use the same states as above and four further states, q, $q'_{\{p\}}$, $q_\perp$, and $q'_\perp$. The transition function is determined by

$$\begin{aligned} \delta(q_I) &= q_\perp \vee q, \\ \delta(q_{\{p\}}) &= q'_{\{p\}} \wedge (q_\perp \vee q), & \delta(q'_{\{p\}}) &= p, \\ \delta(q_\perp) &= \Box q'_\perp, & \delta(q'_\perp) &= \mathsf{ff}, \\ \delta(q) &= \Box(q_I \vee q_{\{p\}}). \end{aligned}$$

The state $q_\perp$ is used to check that the automaton is at a vertex without successor.

In analogy to the construction for LTL, we can now construct an alternating tree automaton for a given CTL formula. This construction is depicted in Figure 25.

Compared to the construction for LTL, there are the following minor differences. First, the definition of the transition function is no longer inductive, because we allow positive boolean combinations in the transition

function. Second, we have positive boolean combinations of states in the scope of $\Diamond$ and $\Box$. This was not necessary with LTL, but it is necessary here. For instance, if we instead had $\delta([\mathsf{E}(\psi \mathsf{X}\mathsf{U}\chi)]) = \Diamond[\chi] \vee (\Diamond[\psi] \wedge \Diamond[\mathsf{E}(\psi \mathsf{X}\mathsf{U}\chi)])$, then this would clearly result in a false automaton because of the second disjunct.

We can make a similar observation as with the alternating automata that we constructed for LTL formulas. The automata are very weak in the sense that when we turn the subformula ordering into a linear ordering $\leq$ on the states, then for each state q, the transition conditions $\delta(q)$ contains only states q' such that $q \geq q'$.

Lemma 5.8 (Kupferman-Vardi-Wolper, [KVW00])**.** Let φ be a CTL formula with n subformulas. The automaton $\mathscr{A}^{alt}[\varphi]$ is a very weak alternating tree automaton with n states and such that $\mathscr{L}(\mathscr{A}^{alt}[A]) = \mathscr{L}(\varphi)$.

Proof. The proof can follow the lines of the proof of Lemma 4.7. Since the automaton is very weak, a simple induction on the structure of the formula can be carried out, just as in the proof of Lemma 4.7. Branching makes the proof only technically more involved, no new ideas are necessary to carry it out. Q.E.D.

As pointed out above, it is not our goal to turn $\mathscr{A}^{alt}[\varphi]$ into a nondeterministic automaton (although this is possible), because such a translation cannot be useful for solving the model checking problem. What we rather do is to define a product of an alternating automaton with a transition system, resulting in a game, in such a way that the winner of the product of $\mathscr{A}^{alt}[\varphi]$ with some transition system $\mathscr{S}$ reflects whether φ holds true in a certain state s_I of $\mathscr{S}$.

The idea is that a position in this game is of the form (γ, s) where γ is a transition condition and s is a state of the transition system. The goal is to design the game in such a way that Player 0 wins the game starting from (q_I, s_I) if and only if there exists an accepting run of the automaton on the unraveling of the transition system starting at s_I. This means, for instance, that if γ is a disjunction, then we make the position (γ, s) a position for Player 0, because by moving to the right position he should show which of the disjuncts holds. If, on the other hand, $\gamma = \Box\gamma'$, then we make the position a position for Player 1, because she should be able to challenge Player 0 with any successor of s. The details are spelled out in Figure 26, where the following notation and terminology is used. Given an alternating automaton $\mathscr{A}$, we write $\mathrm{sub}(\mathscr{A})$ for the set of subformulas of the values of the transition function of $\mathscr{A}$. In addition, we write $\mathrm{sub}^+(\mathscr{A})$ for the set of all $\gamma \in \mathrm{sub}(\mathscr{A})$ where the maximum state occurring belongs to the set of final states.

Let $\mathscr{A}$ be an alternating Büchi tree automaton, $\mathscr{S}$ a transition system over the same set of propositional variables, and $s_I \in S$. The product of $\mathscr{A}$ and $\mathscr{S}$ at s_I, denoted $\mathscr{A} \times_{s_I} \mathscr{S}$, is the Büchi game defined by

$$\mathscr{A} \times_{s_I} \mathscr{S} = (P_0, P_1, (q_I, s_I), M, \mathrm{sub}^+(\mathscr{A}) \times S)$$

where

- P_0 is the set of pairs $(\gamma, s) \in \mathrm{sub}(\mathscr{A}) \times S$ where γ is
  - (i) a disjunction,
  - (ii) a $\Diamond$-formula,
  - (iii) p for $p \notin l(s)$,
  - (iv) $\neg p$ for $p \in l(s)$, or
  - (v) ff,

  and
- P_1 is the set of pairs $(\gamma, s) \in \mathrm{sub}(\mathscr{A}) \times S$ where γ is
  - (i) a conjunction,
  - (ii) a $\Box$-formula,
  - (iii) p for some $p \in l(s)$,
  - (iv) $\neg p$ for some $p \notin l(s)$, or
  - (v) tt.

Further, M contains for every $\gamma \in \mathrm{sub}(\mathscr{A})$ and every $s \in S$ moves according to the following rules:

- if $\gamma = q$ for some state q, then $((\gamma, s), (\delta(q), s)) \in M$,
- if $\gamma = \gamma_0 \vee \gamma_1$ or $\gamma = \gamma_0 \wedge \gamma_1$, then $((\gamma, s), (\gamma_i, s)) \in M$ for $i < 2$,
- if $\gamma = \Diamond\gamma'$ or $\gamma = \Box\gamma'$, then $((\gamma, s), (\gamma', s')) \in M$ for all $s' \in \mathrm{succ}_{\mathscr{S}}(s)$.

FIGURE 26. Product of a transition system and an alternating automaton

Assume $\mathscr{A}$ is a very weak alternating Büchi automaton. Then $\mathscr{A} \times_{s_I} \mathscr{S}$ will need to be very weak in general in the sense that the game graph can be extended to a linear ordering. Observe, however, that the following is true for every position (q, s): All states in the strongly connected component of (q, s) are of the form (γ, s') where q is the largest state occurring in γ. So, by definition of $\mathscr{A} \times_{s_I} \mathscr{S}$, all positions in a strongly connected component of $\mathscr{A} \times_{s_I} \mathscr{S}$ are either final or nonfinal. We turn this into a definition. We say that a Büchi game is weak if for every strongly connected component of the game graph it is true that either all its positions are final or none of them is.

Lemma 5.9. Let $\mathscr{A}$ be an alternating Büchi tree automaton, $\mathscr{S}$ a transition system over the same finite set of propositional variables, and $s_I \in S$. Then $\mathscr{T}_{s_I}(\mathscr{S}) \in \mathscr{L}(\mathscr{A})$ iff Player 0 wins $\mathscr{A} \times_{s_I} \mathscr{S}$. Moreover, if $\mathscr{A}$ is a very weak alternating automaton, then $\mathscr{A} \times_{s_I} \mathscr{S}$ is a weak game.

Proof. The additional claim is obvious. For the other claim, first assume $\mathscr{R}$ is an accepting run of $\mathscr{A}$ on $\mathscr{T}_{s_I}(\mathscr{S})$. We convert $\mathscr{R}$ into a winning strategy σ for Player 0 in $\mathscr{A} \times \mathscr{S}$. To this end, let w be a vertex of $\mathscr{R}$ with label (γ, v) such that (γ, v) is a position for Player 0. Since $\mathscr{R}$ is an accepting run, w has a successor, say w'. Assume $l^{\mathscr{R}}(w') = (\gamma', v')$. We set $\sigma(u) = (\gamma, v'(*))$ where u is defined as follows. First, let $n = |v|$. Assume $l^{\mathscr{R}}(u(i)) = (\gamma_i, v_i)$ for every $i < n$. We set $u = (\gamma_0, v_0(*))(\gamma_1, v_1(*)) \ldots (\gamma_{n-1}, v_{n-1}(*))$. It can be shown that this defines a strategy. Moreover, since $\mathscr{R}$ is accepting, σ is winning.

For the other direction, a winning strategy is turned into an accepting run in a similar manner. Q.E.D.

The proof shows that essentially there is no difference between a run and a strategy—one can think of a run as a strategy. From this point of view, an alternating automaton defines a family of games, for each tree a separate game, and the tree language recognized by the tree automaton is the set of all trees which Player 0 wins the game for.

The additional claim in the above lemma allows us to prove the desired complexity bound for the CTL model checking problem:

Theorem 5.10 (Clarke-Emerson-Sistla, [CES86])**.** The CTL model checking problem can be solved in time $O(mn)$ where m is the size of the transition system and n the number of subformulas of the CTL formula.

Proof. Consider the following algorithm, given a CTL formula φ, a transition system $\mathscr{S}$, and a state $s_I \in S$. First, construct the very weak alternating Büchi automaton $\mathscr{A}^{alt}[\varphi]$. Second, build the product $\mathscr{A}^{alt}[\varphi] \times_{s_I} \mathscr{S}$. Third, solve $\mathscr{A}^{alt}[\varphi] \times_{s_I} \mathscr{S}$. Then Player 0 is the winner if and only if $\mathscr{S}, s_I \models \varphi$.

The claim about the complexity follows from the fact that the size of $\mathscr{A}^{alt}[\varphi] \times_{s_I} \mathscr{S}$ is mn and from Theorem 2.21. Note that weak games are parity games with one priority in each strongly connected component. Q.E.D.

Obviously, given a CTL formula φ, a transition system $\mathscr{S}$, and a state s_I one can directly construct a game that reflects whether $\mathscr{S}, s_I \models \varphi$. This game would be called the model checking game for $\mathscr{S}$, s_I, and φ. The construction via the alternating automaton has the advantage that starting from this automaton one can solve both, model checking and satisfiability, the latter by using a translation from alternating Büchi tree automata into nondeterministic tree automata. We present such a translation in Section 6.

The translation from CTL into very weak alternating automata has another interesting feature. Just as the translation from LTL to weak alternating automata, it has a converse. More precisely, following the lines of the proof of Theorem 4.9, one can prove:

Theorem 5.11. Every very weak alternating tree automaton is equivalent to a CTL formula. Q.E.D.

5.4 Notes

The two specification logics that we have dealt with, LTL and CTL, can easily be combined into a single specification logic. This led Emerson and Halpern to introduce CTL* in 1986 [EH86].

An automata-theoretic proof of Corollary 5.7 was given first by Vardi and Wolper in 1986 [VW86b]. Kupferman, Vardi, and Wolper, when proposing an automata-theoretic approach to CTL model checking in [KVW00], also showed how other model checking problems can be solved following the automata-theoretic paradigm. One of their results is that CTL model checking can be solved in space polylogarithmic in the size of the transition system.

6 Modal μ-Calculus

The logics that have been discussed thus far—S1S, S2S, LTL, and CTL—could be termed declarative in the sense that they are used to *describe* properties of sequences, trees, or transition systems rather than to specify how it can be determined whether such properties hold. This is different for the logic we discuss in this section, the modal μ-calculus (MC), introduced by Kozen in 1983 [Koz83]. This calculus has a rich and deep mathematical and algorithmic theory, which has been developed over more than 20 years. Fundamental work on it has been carried out by Emerson, Streett, and Jutla [SE84, EJ91b], Walukiewicz [Wal00], Bradfield and Lenzi [Len96, Bra98], and others, and it has been treated extensively in books, for instance, by Arnold and Niwiński [AN01] and Stirling [Sti01]. In

this section, we study satisfiability (and model checking) for MC from an automata-theoretic perspective. Given that MC is much more complex than LTL or CTL, our exposition is less detailed, but gives a good impression of how the automata-theoretic paradigm works for MC.

6.1 MC and Monadic Second-Order Logic

MC is a formal language consisting of expressions which are evaluated in transition systems; every closed expression (without free variables) is evaluated to a set of states. The operations available for composing sets of states are boolean operations, local operations, and fixed point operations.

Formally, the set of MC expressions is the smallest set containing

- p and $\neg p$ for any propositional variable p,
- any fixed-point variable X,
- $\varphi \wedge \psi$ and $\varphi \vee \psi$ if φ and ψ are MC expressions,
- $\langle\rangle\varphi$ and $[]\varphi$ if φ is an MC expression, and
- $\mu X\varphi$ and $\nu X\varphi$ if X is a fixed-point variable and φ an MC expression.

The operators μ and ν are viewed as quantifiers in the sense that one says they bind the following variable. As usual, an expression without free occurrences of variables is called closed. The set of all variables occurring free in an MC expression φ is denoted by $\mathrm{free}(\varphi)$. An expression is called a fixed-point expression if it starts with μ or ν.

To define the semantics of MC expressions, let φ be an MC expression over some finite set P of propositional variables, $\mathscr{S}$ a transition system, and α a variable assignment which assigns to every fixed-point variable a set of states of $\mathscr{S}$. The value of φ with respect to $\mathscr{S}$ and α, denoted $||\varphi||^{\alpha}_{\mathscr{S}}$, is defined as follows. The fixed-point variables and the propositional variables are interpreted according to the variable assignment and the transition system:

$$||p||^{\alpha}_{\mathscr{S}} = \{s \in S^{\mathscr{S}} : p \in l^{\mathscr{S}}(s)\}, \qquad ||\neg p||^{\alpha}_{\mathscr{S}} = \{s \in S^{\mathscr{S}} : p \notin l^{\mathscr{S}}(s)\},$$

and

$$||X||^{\alpha}_{\mathscr{S}} = \alpha(X).$$

Conjunction and disjunction are translated into union and intersection:

$$||\varphi \wedge \psi||^{\alpha}_{\mathscr{S}} = ||\varphi||^{\alpha}_{\mathscr{S}} \cap ||\psi||^{\alpha}_{\mathscr{S}}, \qquad ||\varphi \vee \psi||^{\alpha}_{\mathscr{S}} = ||\varphi||^{\alpha}_{\mathscr{S}} \cup ||\psi||^{\alpha}_{\mathscr{S}}.$$

The two local operators, $\langle\,\rangle$ and $[\,]$, are translated into graph-theoretic operations:

$$||\langle\,\rangle\varphi||^{\alpha}_{\mathscr{S}} = \{s \in S \colon \mathrm{succs}_{\mathscr{S}}(s) \cap ||\varphi||^{\alpha}_{\mathscr{S}} \neq \varnothing\},$$
$$||[\,]\varphi||^{\alpha}_{\mathscr{S}} = \{s \in S \colon \mathrm{succs}_{\mathscr{S}}(s) \subseteq ||\varphi||^{\alpha}_{\mathscr{S}}\}.$$

The semantics of the fixed-point operators is based on the observation that for every expression φ, the function $S' \mapsto ||\varphi||^{\alpha[X \mapsto S']}_{\mathscr{S}}$ is a monotone function on 2^S with set inclusion as ordering, where $\alpha[X \mapsto S']$ denotes the variable assignment which coincides with α, except for the value of the variable X, which is S'. The Knaster–Tarski Theorem then guarantees that this function has a least and a greatest fixed point:

$$||\mu X\varphi||^{\alpha}_{\mathscr{S}} = \bigcap \left\{ S' \subseteq S \colon ||\varphi||^{\alpha[X \mapsto S']}_{\mathscr{S}} = S' \right\},$$
$$||\nu X\varphi||^{\alpha}_{\mathscr{S}} = \bigcup \left\{ S' \subseteq S \colon ||\varphi||^{\alpha[X \mapsto S']}_{\mathscr{S}} = S' \right\}.$$

In the first equation the last equality sign can be replaced by $\subseteq$, while in the second equation it can be replaced by $\supseteq$. The above equations are—contrary to what was said at the beginning of this section—declarative rather than operational, but this can easily be changed because of the Knaster–Tarski Theorem. For a given system $\mathscr{S}$, a variable assignment α, an MC expression φ, and a fixed-point variable X, consider the ordinal sequence $(S_\lambda)_\lambda$, called approximation sequence for $||\mu X\varphi||^{\alpha}_{\mathscr{S}}$, defined by

$$S_0 = \varnothing, \qquad S_{\lambda+1} = ||\varphi||^{\alpha[X \mapsto S_\lambda]}_{\mathscr{S}}, \qquad S_{\lambda'} = \bigcup_{\lambda < \lambda'} S_\lambda,$$

where λ' stands for a limit ordinal. Because of monotonicity, we have $S_0 \subseteq S_1 \subseteq \ldots$. The definition of the sequence implies that if $S_\lambda = S_{\lambda+1}$ for any λ, then $S_{\lambda'} = S_\lambda = ||\mu X\varphi||^{\alpha}_{\mathscr{S}}$ for all $\lambda' \geq \lambda$. Clearly, we have $\lambda \leq \mathrm{card}(S)$ for the smallest such λ, which, for finite transition systems, means there is a simple (recursive) way to evaluate $\mu X\varphi$. The same holds true for $\nu X\varphi$, where the approximation is from above, that is, $S_0 = S$ and the inclusion order is reversed.

For notational convenience, we also use $\mathscr{S}, \alpha, s \models \varphi$ to denote $s \in ||\varphi||^{\alpha}_{\mathscr{S}}$ for any state $s \in S$. When φ is a closed MC expression, then the variable assignment α is irrelevant for its interpretation, so we omit it and simply write $||\varphi||_{\mathscr{S}}$ or $\mathscr{S}, s \models \varphi$.

To give some examples for useful expressions, recall the CTL formula (1.3) from Section 5.1. We can express its subformula $\mathsf{EF}p_d$ by

$$\varphi_{\mathit{inner}} = \mu X(p_d \vee \langle\,\rangle X),$$

so that the full formula can be written as

$$\nu Y(\varphi_{inner} \wedge [\,]Y).$$

In a similar fashion, (1.4) can be expressed:

$$\nu Y((\neg p_r \vee \mu X(p_a \vee [\,]X)) \wedge [\,]Y).$$

It is more complicated to express the LTL formula (1.5); it needs a nested fixed-point expression with mutually dependent fixed-point variables. We first build an expression which denotes all states from which on all paths a state is reachable where p is true and which belongs to a set Y:

$$\varphi'_{inner} = \mu X((p \wedge Y) \vee [\,]X).$$

Observe that Y occurs free in φ'_{inner}. The desired expression can then be phrased as a greatest fixed point:

$$\nu Y \varphi'_{inner}.$$

It is no coincidence that we are able to express the two CTL formulas in MC:

Proposition 6.1. For every CTL formula φ there exists a closed MC expression $\tilde{\varphi}$ such that for every transition system $\mathscr{S}$ and $s \in S$,

$$\mathscr{S}, s \models \varphi \qquad \text{iff} \qquad \mathscr{S}, s \models \tilde{\varphi}.$$

Proof. The proof is a straightforward induction. We describe one case of the inductive step. Assume ψ and χ are CTL formulas and $\tilde{\psi}$ and $\tilde{\chi}$ are MC expressions such that the claim holds. We consider $\varphi = \mathsf{E}(\psi \,\mathsf{XU}\, \chi)$ and want to construct $\tilde{\varphi}$ as desired. We simply describe the semantics of φ by a fixed-point computation:

$$\tilde{\varphi} = \langle\,\rangle \mu X(\tilde{\chi} \vee (\tilde{\psi} \wedge \langle\,\rangle \tilde{X})).$$

The other cases can be dealt with in the same fashion. Q.E.D.

The next observation is that as far as satisfiability is concerned, we can restrict our considerations to trees, just with CTL (recall Lemma 5.1).

Lemma 6.2. For every MC expression φ, transition system $\mathscr{S}$, variable assignment α, and state $s \in S$,

$$\mathscr{S}, \alpha, s \models \varphi \qquad \text{iff} \qquad \mathscr{T}_s(\mathscr{S}), \alpha \models \varphi.$$

(Recall that when we view a tree as a transition system, then we interpret formulas in the root of the tree unless stated otherwise.)

Proof. This can be proved by a straightforward induction on the structure of φ, using the following inductive claim:

$$\{v \in V^{\mathscr{T}_s(\mathscr{S})} \colon \mathscr{S}, \alpha, v(*) \models \varphi\} = ||\varphi||^{\alpha}_{\mathscr{T}_s(\mathscr{S})}.$$

This simply says that with regard to MC, there is no difference between a state s' in a given transition system $\mathscr{S}$ and every vertex v with $v(*) = s'$ in the unraveling of $\mathscr{S}$. Q.E.D.

Just as with CTL, the lemma allows us to work henceforth in the tree framework. For a closed MC expression φ with propositional variables from a set $P = \{p_0, \ldots, p_{n-1}\}$, the tree language defined by φ, denoted $\mathscr{L}(\varphi)$, is the set of all trees $\mathscr{T}$ over 2^P such that $\mathscr{T} \models \varphi$.

The next observation is that every MC expression can be translated into a monadic second-order formula, similar to Proposition 5.2. Before we can state the result, we define an appropriate equivalence relation between SUS formulas and MC expressions. Recall that an SUS formula is true or not for a given tree, while an MC expression evaluates to a set of vertices.

Let $P = \{p_0, \ldots, p_{n-1}\}$ be a set of propositional variables and φ an MC expression over P with free fixed-point variables among $X_0, \ldots, X_{m-1}$. We view the variables $X_0, \ldots, X_{m-1}$ as further propositional variables and identify each X_i with a set variable V_i and each p_j with a set variable V_{m+j}. So we can interpret φ and every SUS formula $\psi = \psi(V_0, \ldots, V_{m+n-1})$ in trees over $[2]_{m+n}$. We say φ is equivalent with such a formula ψ if $\mathscr{L}(\varphi) = \mathscr{L}(\psi)$.

Proposition 6.3. For every MC expression φ, an equivalent SUS formula $\tilde{\varphi}$ can be constructed.

Proof. This can be proved by induction on the structure of φ, using a more general claim. For every MC expression φ as above, we construct an SUS formula $\hat{\varphi} = \hat{\varphi}(V_0, \ldots, V_{m+n-1}, z)$ such that for every tree $\mathscr{T}$ over $[2]_{m+n}$ and $v \in V^{\mathscr{T}}$, we have:

$$\mathscr{T}{\downarrow}v \models \varphi \qquad \text{iff} \qquad \mathscr{T}, v \models \hat{\varphi}(V_0, \ldots, V_{m+n-1}, z),$$

where $\mathscr{T}, v \models \hat{\varphi}(V_0, \ldots, V_{m+n-1}, z)$ is defined in the obvious way, see Section 4.1 for a similar definition in the context of LTL. We can then set $\tilde{\varphi} = \exists x(\forall y(\neg \mathrm{suc}(y, x) \wedge \hat{\varphi}(V_0, \ldots, V_{m+n-1}, x)))$.

The interesting cases in the inductive step are the fixed-point operators. So let $\varphi = \mu X_i \psi$ and assume $\hat{\psi}$ is already given. The formula $\hat{\varphi}$ simply says that z belongs to a fixed point and that every other fixed point is a superset of it:

$$\begin{aligned}\hat{\varphi} = \exists Z(z \in Z \wedge \forall z'(\hat{\psi}(\ldots, V_{i-1}, Z, V_{i+1}, \ldots, z') \leftrightarrow z' \in Z) \wedge \\ \forall Z'(\forall z'(\psi(\ldots, V_{i-1}, Z', V_{i+1}, \ldots, z') \leftrightarrow z' \in Z') \rightarrow Z \subseteq Z')).\end{aligned}$$

For the greatest fixed-point operator, the construction is analogous. Q.E.D.

As a consequence, we can state:

Corollary 6.4 (Kozen-Parikh, [KP84])**.** MC satisfiability is decidable.

But, just as with LTL and CTL, by a translation into monadic second-order logic we get only a nonelementary upper bound for the complexity.

6.2 From MC to Alternating Tree Automata

Our overall objective is to derive a good upper bound for the complexity of MC satisfiability. The key is a translation of MC expressions into nondeterministic tree automata via alternating parity tree automata. We start with the translation of MC expressions into alternating parity tree automata.

Alternating parity tree automata are defined exactly as nondeterministic Büchi tree automata are defined in Section 5.3 except that the Büchi acceptance condition is replaced by a parity condition π.

Just as with LTL and CTL, the translation into alternating automata reflects the semantics of the expressions in a direct fashion. The fixed-point operators lead to loops, which means that the resulting tree automata will no longer be very weak (not even weak). For least fixed points these loops may not be traversed infinitely often, while this is necessary for greatest fixed points. To control this, priorities are used: Even priorities are used for greatest fixed-points, odd priorities for least fixed points. Different priorities are used to take into account the nesting of fixed points, the general rule being that outer fixed points have smaller priorities, because they are more important.

For model checking, it will be important to make sure as few different priorities as possible are used. That is why a careful definition of alternation depth is needed. In the approach by Emerson and Lei [EL86], one counts the number of alternations of least and greatest fixed points on the paths of the parse tree of a given expression. Niwiński's approach [Niw86] yields a coarser hierarchy, which gives better upper bounds for model checking. It requires that relevant nested subexpressions are "mutually recursive".

Let $\leq$ denote the relation "is subexpression of", that is, $\psi \leq \varphi$ if $\psi \in \mathrm{sub}(\varphi)$. Let φ be an MC expression. An *alternating μ-chain* in φ of length l is a sequence

$$\varphi \geq \mu X_0 \psi_0 > \nu X_1 \psi_1 > \mu X_2 \psi_2 > \cdots > \mu/\nu X_{l-1} \psi_{l-1} \tag{1.6}$$

where, for every $i < l-1$, the variable X_i occurs free in every formula ψ with $\psi_i \geq \psi \geq \psi_{i+1}$. The maximum length of an alternating μ-chain in φ is denoted by $m^\mu(\varphi)$. Symmetrically, ν-chains and $m^\nu(\varphi)$ are defined. The *alternation depth* of a μ-calculus expression φ is the maximum of $m^\mu(\varphi)$ and $m^\nu(\varphi)$ and is denoted by $\mathrm{d}(\varphi)$.

We say an MC expression is in normal form if for every fixed-point variable X occurring the following holds:

- every occurrence of X in φ is free or
- all occurrences of X in φ are bound in the same subexpression $\mu X\psi$ or $\nu X\psi$, which is then denoted by φ_X.

Clearly, every MC expression is equivalent to an MC expression in normal form.

The full translation from MC into alternating parity tree automata can be found in Figure 27, where the following notation is used. When φ is an MC expression and $\mu X\psi \in \mathrm{sub}(\varphi)$, then

$$\mathrm{d}_\varphi(\mu X\psi) = \begin{cases} \mathrm{d}(\varphi) + 1 - 2\lceil \mathrm{d}(\mu X\psi)/2 \rceil, & \text{if } \mathrm{d}(\varphi) \bmod 2 = 0, \\ \mathrm{d}(\varphi) - 2\lfloor \mathrm{d}(\mu X\psi)/2 \rfloor, & \text{otherwise.} \end{cases}$$

Similarly, when $\nu X\psi \in \mathrm{sub}(\varphi)$, then

$$\mathrm{d}_\varphi(\nu X\psi) = \begin{cases} \mathrm{d}(\varphi) - 2\lfloor \mathrm{d}(\nu X\psi)/2 \rfloor, & \text{if } \mathrm{d}(\varphi) \bmod 2 = 0, \\ \mathrm{d}(\varphi) + 1 - 2\lceil \mathrm{d}(\nu X\psi)/2 \rceil, & \text{otherwise.} \end{cases}$$

This definition reverses alternation depth so it can be used for defining the priorities in the alternating parity automaton for an MC expression. Recall that we want to assign priorities such that the higher the alternation depth the lower the priority and, at the same time, even priorities go to ν-formulas and odd priorities to μ-formulas. This is exactly what the above definition achieves.

It is obvious that $\mathscr{A}[\varphi]$ will have $\mathrm{d}(\varphi) + 1$ different priorities in certain cases, but from a certain point of view, these cases are not harmful. To explain this, we introduce the notion of index of an alternating tree automaton. The transition graph of an alternating tree automaton $\mathscr{A}$ is the graph with vertex set Q and where (q, q') is an edge if q' occurs in $\delta(q)$. The index of $\mathscr{A}$ is the maximum number of different priorities in the strongly connected components of the transition graph of $\mathscr{A}$. Clearly, $\mathscr{A}[\varphi]$ has index $\mathrm{d}(\varphi)$.

Theorem 6.5 (Emerson-Jutla, [EJ91b])**.** Let φ be an MC expression in normal form with n subformulas. Then $\mathscr{A}[\varphi]$ is an alternating parity tree automaton with n states and index $\mathrm{d}(\varphi)$ such that $\mathscr{L}(\mathscr{A}[\varphi]) = \mathscr{L}(\varphi)$.

To be more precise, $\mathscr{A}[\varphi]$ may have $\mathrm{d}(\varphi) + 1$ different priorities, but in every strongly connected component of the transition graph of $\mathscr{A}[\varphi]$ there are at most $\mathrm{d}(\varphi)$ different priorities, see also Theorem 2.21.

Let φ be a closed MC expression in normal form and Q a set which contains for every $\psi \in \mathrm{sub}(\varphi)$ a state $[\psi]$. The alternating parity tree automaton for φ, denoted $\mathscr{A}[\varphi]$, is defined by

$$\mathscr{A}[\varphi] = (P, Q, \varphi, \delta, \pi)$$

where the transition function is given by

$$\begin{aligned}
\delta([p]) &= p, & \delta([\neg p]) &= \neg p,\\
\delta([\psi \vee \chi]) &= [\psi] \vee [\chi], & \delta([\psi \wedge \chi]) &= [\psi] \wedge [\chi],\\
\delta([\langle\rangle\psi]) &= \Diamond[\psi], & \delta([[]\psi]) &= \Box[\psi],\\
\delta([\mu X\psi]) &= [\psi], & \delta([\nu X\psi]) &= [\psi],\\
\delta([X]) &= [\varphi_X],
\end{aligned}$$

and where

$$\pi([\psi]) = \mathrm{d}_\varphi(\psi)$$

for every fixed-point expression $\psi \in \mathrm{sub}(\varphi)$.

FIGURE 27. From μ-calculus to alternating tree automata

Proof. The claims about the number of states and the index are obviously true. The proof of correctness is more involved than the corresponding proofs for LTL and CTL, because the automata which result from the translation are, in general, not weak.

The proof of the claim is by induction on the structure of φ. The base case is trivial and so are the cases in the inductive step except for the cases where fixed-point operators are involved. We consider the case where $\varphi = \mu X\psi$.

So assume $\varphi = \mu X\psi$ and $\mathscr{T} \models \varphi$. Let $f\colon 2^V \to 2^V$ be defined by $f(V') = ||\psi||_{\mathscr{T}}^{X \mapsto V'}$. Let $(V_\lambda)_\lambda$ be the sequence defined by $V_0 = \varnothing$, $V_{\lambda+1} = f(V_\lambda)$, and $V_{\lambda'} = \bigcup_{\lambda<\lambda'} V_\lambda$ for limit ordinals λ'. We know that f has a least fixed point, which is the value of φ in $\mathscr{T}$, and that there exists κ such that V_κ is the least fixed-point of f. We show by induction on λ that there exists an accepting run of $\mathscr{A}[\varphi]$ on $\mathscr{T}{\downarrow}v$ for every $v \in V_\lambda$. This is trivial when $\lambda = 0$ or when λ is a limit ordinal. When λ is a successor ordinal, say $\lambda = \lambda_0 + 1$, then $V_\lambda = f(V_{\lambda_0})$. Consider the automaton $\mathscr{A}[\psi]$ where X is

viewed as a propositional variable. By the outer induction hypothesis, there exists an accepting run $\mathscr{R}$ of $\mathscr{A}[\psi]$ on $\mathscr{T}[X \mapsto V_{\lambda_0}]\downarrow v$, where $\mathscr{T}[X \mapsto V_{\lambda_0}]$ is the obvious tree over $2^{P\cup\{X\}}$. We can turn $\mathscr{R}$ into a prefix $\mathscr{R}'$ of a run of $\mathscr{A}[\varphi]$ on $\mathscr{T}\downarrow v$ by adding a new root labeled $([\varphi], v)$ to it. Observe that some of the leaves w of $\mathscr{R}'$ may be labeled (X, v') with $v' \in V_{\lambda_0}$. For each such v' there exists, by the inner induction hypothesis, an accepting run $\mathscr{R}_{v'}$ of $\mathscr{A}[\varphi]$ on $\mathscr{T}\downarrow v$. Replacing w by $\mathscr{R}_{v'}$ for every such leaf w yields a run $\hat{\mathscr{R}}$ of $\mathscr{A}[\varphi]$ on $\mathscr{T}\downarrow v$. We claim this run is accepting. To see this, observe that each infinite branch of $\hat{\mathscr{R}}$ is an infinite branch of $\mathscr{R}'$ or has an infinite path of $\mathscr{R}_{v'}$ for some v' as a suffix. In the latter case, the branch is accepting for a trivial reason, in the former case, the branch is accepting because the priorities in $\mathscr{A}[\psi]$ differ from the priorities in $\mathscr{A}[\varphi]$ by a fixed even number. This completes the inductive proof. Since, by assumption, the root of $\mathscr{T}$ belongs to V_κ, we obtain the desired result.

For the other direction, assume $\mathscr{T}$ is accepted by $\mathscr{A}[\varphi]$, say by a run $\mathscr{R}$. Let W be the set of all $w \in V^{\mathscr{R}}$ such that φ is the first component of $l^{\mathscr{R}}(w)$. Observe that because of the definition of the priority function π there can only be a finite number of elements from W on each branch of $\mathscr{R}$. This is because the priority function π is defined in a way such that if $\psi \in \mathrm{sub}(\varphi)$ is a fixed-point formula with $[\psi]$ in the strongly connected component of $[\varphi]$ in the transition graph of $\mathscr{A}[\varphi]$, then $\pi([\varphi]) \leq \pi([\psi])$.

Consider the sequence $(V_\lambda)_\lambda$ of subsets of $V^{\mathscr{R}}$ defined as follows:

- $V_0 = \varnothing$,
- $w \in V_{\lambda+1}$ if all proper descendants of w in $\mathscr{R}$ belong to $V_\lambda \cup V^{\mathscr{R}} \setminus W$, and
- $V_{\lambda'} = \bigcup_{\lambda<\lambda'} V_\lambda$ for every limit ordinal λ'.

Using the induction hypothesis, one can prove by induction on λ that for every $w \in V_\lambda$ the second component of its label belongs to $||\varphi||_{\mathscr{T}}$.

Since there are only a finite number of elements from W on each branch of V_λ, one can also show that $\mathrm{root}(\mathscr{R}) \in W$, which proves the claim. Q.E.D.

Before we turn to the conversion of alternating into nondeterministic parity tree automata, we discuss model checking MC expressions briefly. Model checking an MC expression, that is, evaluating it in a finite transition system is "trivial" in the sense that one can simply evaluate the expression according to its semantics, using approximation for evaluating fixed-point operators as explained in Section 6.1. Using the fact that fixed-points of the same type can be evaluated in parallel one arrives at an algorithm which is linear in the product of the size of the expression and the size of the system, but exponential in the depth of the alternation between least and greatest fixed points.

An alternative approach to model checking MC expressions is to proceed as with CTL. Given a finite transition system $\mathscr{S}$, an initial state $s_I \in S$, and an expression φ, one first constructs the alternating automaton $\mathscr{A}[\varphi]$, then the product game $\mathscr{A}[\varphi] \times_{s_I} \mathscr{S}$ (with a parity condition rather than a Büchi condition), and finally solves this game. (Of course, on can also directly construct the game.) As a consequence of the previous theorem and Theorem 2.21, one obtains:

Theorem 6.6 (Seidl-Jurdziński, [Sei96, Jur00])**.** An MC expression of size l and alternation depth d can be evaluated in a finite transition system with m states and n transitions in time $O((lm + ln(lm)^{\lfloor d/2 \rfloor}))$. Q.E.D.

In fact, there is a close connection between MC model checking and solving parity games: The two problems are interreducible, which means all the remarks on the complexity of solving parity games at the end of Section 2.5 are equally valid for MC model checking.

The above theorem tells us something about AMC, the set of all MC expressions with alternation depth ≤ 1. These expressions can be evaluated in time linear in the product of the size of the transition system and the length of the formula, which was first proved by Cleaveland, Klein, and Steffen [CKS92] in general and by Kupferman, Vardi, and Wolper using automata-theoretic techniques [KVW00]. This yields a different proof of Theorem 5.10: The straightforward translation from CTL into the μ-calculus, see Proposition 6.1, yields alternation-free expressions of linear size. From a practical point, it is interesting to note that model checking tools indeed use the translation of CTL into AMC, see [McM93].

6.3 From Alternating to Nondeterministic Tree Automata

In view of Theorem 6.5, what we need to solve MC satisfiability is a translation of alternating tree automata into nondeterministic tree automata, because we already know how to decide emptiness for these automata. To be precise, we proved this only for Büchi acceptance conditions, see Figure 24, but the proof of this theorem extends to parity tree automata in a straightforward manner.

One way of achieving a translation from alternating into nondeterministic automata is to proceed in two steps, where the intermediate result is an alternating automaton with very restrictive transition conditions. We say a transition condition is in normal form if it is a disjunction of transition conditions of the form

$$\bigwedge_{q \in Q^{\mathsf{A}}} \Box q \wedge \bigwedge_{q \in Q^{\mathsf{E}}} \Diamond q.$$

The conversion of an ordinary alternating tree automaton into an alternating tree automaton with transition conditions in normal form is similar to

removing ε-transitions. We describe it here for the case where the transition conditions are simpler as in the general case, namely where each subformula $\Box\gamma$ or $\Diamond\gamma$ is such that γ is a state. Observe that all the transition conditions in the construction described in Figure 27 are of this form. At the same time, we change the format of the transition function slightly. We say an alternating automaton is in normal form if its transition function δ is of the form $Q \times 2^P \to \mathrm{TC}(P, Q)$ where $\delta(q, a)$ is a transition condition in normal form for $q \in Q$ and $a \in 2^P$. The notion of a run of an alternating automaton is appropriately adapted.

To convert alternating automata into normal form, we start with a crucial definition. Let $\mathscr{A}$ be an alternating parity tree automaton, $a \in 2^P$, and $q \in Q$. We say a tree $\mathscr{R}$ labeled with transition conditions is a transition tree for q and a if its root is labeled q and every vertex w with label γ satisfies the following conditions:

- if $\gamma = p$, then $p \in a$, and if $\gamma = \neg p$, then $p \notin a$,
- if $\gamma = q'$, then there exists $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that $l^{\mathscr{R}}(w') = \delta(q')$,
- if $\gamma = \Diamond q'$ or $\gamma = \Box q'$, then w has no successor,
- if $\gamma = \gamma_0 \vee \gamma_1$, then there exists $i < 2$ and $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that $l^{\mathscr{R}}(w') = \gamma_i$,
- if $\gamma = \gamma_0 \wedge \gamma_1$, then for every $i < 2$ there exists $w' \in \mathrm{succs}_{\mathscr{R}}(w)$ such that $l^{\mathscr{R}}(w') = \gamma_i$.

Further, every infinite branch of $\mathscr{R}$ is accepting with respect to π.

A transition tree as above can easily be turned into a transition condition in normal form over an extended set of states, namely $\bar{Q} = Q \times \pi(Q)$. The second component is used to remember the minimum priority seen on a path of a transition tree, as explained below. Let Q^{A} be the set of pairs (q', i) such that $\Box q'$ is a label of a leaf of $\mathscr{R}$, say w, and i is the minimum priority on the path from the root of $\mathscr{R}$ to w. Similarly, let Q^{E} be the set of pairs (q', i) such that $\Diamond q'$ is a label of a leaf of $\mathscr{R}$, say w, and i is the minimum priority on the path from the root of $\mathscr{R}$ to w. The transition condition for the transition tree $\mathscr{R}$, denoted $\gamma_{\mathscr{R}}$, is defined by

$$\gamma_{\mathscr{R}} = \bigwedge_{(q',i)\in Q^{\mathsf{A}}} \Box(q', i) \wedge \bigwedge_{(q',i)\in Q^{\mathsf{E}}} \Diamond(q', i).$$

The entire normalization construction is depicted in Figure 28.

Lemma 6.7. Let $\mathscr{A}$ be an alternating parity tree automaton with n states and k different priorities. Then $\mathscr{A}^{norm}$ is an alternating parity tree automaton in normal form with kn states and k different priorities. Q.E.D.

Let $\mathscr{A}$ be an alternating parity tree automaton. The normalization of $\mathscr{A}$ is the alternating parity tree automaton $\mathscr{A}^{norm}$ defined by

$$\mathscr{A}^{norm} = (P, Q \times \pi(Q), (q_I, j), \delta', \pi')$$

where

- j is any element of $\pi(Q)$,
- $\pi'((q,i)) = i$ for all $q \in Q$ and $i \in \pi(Q)$, and
- $\delta'((q,i),a) = \bigvee \gamma_{\mathscr{R}}$ for $q \in Q$, $i \in \pi(Q)$, and $a \in 2^P$, with $\mathscr{R}$ ranging over all transition trees for q and a.

FIGURE 28. Normalizing transition conditions of alternating tree automata

The second step in our construction is a conversion of an alternating automaton in normal form into a nondeterministic tree automaton, similar to the conversion of universal parity tree automata into nondeterministic tree automata explained in Section 3.3. Again, we heavily draw on the generic automaton introduced in that section. Recall that given a finite state set Q and a priority function π, the generic automaton is a deterministic automaton over $\mathscr{Q}$, the alphabet consisting of all binary relations over Q, which accepts a word $u \in \mathscr{Q}^\omega$ if all $v \in \langle u \rangle$ satisfy the parity condition π.

Given an alternating automaton in normal form, a set $Q' \subseteq Q$, and a letter $a \in 2^P$, a set $\mathscr{Q}' \subseteq \mathscr{Q}$ is a choice for Q' and a if for every $q \in Q'$ there exists a disjunct in $\delta(q)$ of the form

$$\bigwedge_{q' \in Q_q^{\mathsf{A}}} \Box q' \wedge \bigwedge_{q' \in Q_q^{\mathsf{E}}} \Diamond q'$$

such that the following conditions are satisfied:

(i) $R \in \mathscr{Q}'$ where $R = \{(q,q') \colon q \in Q' \wedge q' \in Q_q^{\mathsf{A}}\}$,

(ii) $R \subseteq R'$ for every $R' \in \mathscr{Q}'$ with R as defined in the previous condition,

(iii) for every $q \in Q'$ and every $q' \in Q_q^{\mathsf{E}}$ there exists $R' \in \mathscr{Q}'$ such that $(q,q') \in R'$, and

(iv) $|\mathscr{Q}'| \leq |Q| \times |Q| + 1$.

Let $\mathscr{A}$ be an alternating parity tree automaton in normal form and $\mathscr{B} = \mathscr{A}[Q^{\mathscr{A}}, \pi^{\mathscr{A}}]$ the generic automaton for $Q^{\mathscr{A}}$ and $\pi^{\mathscr{A}}$.

The nondeterministic automaton $\mathscr{A}^{nd}$ is defined by

$$\mathscr{A}^{nd} = (2^P, 2^{Q^{\mathscr{A}}} \times Q^{\mathscr{B}}, (\{q_I^{\mathscr{A}}\}, q_I^{\mathscr{B}}), \Delta, \pi)$$

where $\pi((Q', q)) = \pi^{\mathscr{B}}(q)$ and $((Q', q), a, \bar{\mathscr{Q}}) \in \Delta$ if there exists a choice $\mathscr{Q}$ for Q' and a such that

$$\bar{\mathscr{Q}} = \{(Q'R, \delta^{\mathscr{B}}(q, R))\colon R \in \mathscr{Q}\}.$$

FIGURE 29. From alternating to nondeterministic tree automata

For a set $Q' \subseteq Q$ and a relation $R \subseteq Q \times Q$, we write $Q'R$ for the set $\{q' \in Q\colon \exists q(q \in Q' \wedge (q, q') \in R\}$.

The details of the conversion from alternating parity tree automata in normal form into nondeterministic tree automata can be found in Figure 29. It is analogous to the construction depicted in Figure 16, which describes how a universal parity tree automaton over binary trees can be turned into a nondeterministic parity tree automaton. The situation for alternating automata is different in the sense that the transition conditions of the form $\Diamond q'$ have to be taken care of, too, but this is captured by (iii) in the above definition.

Lemma 6.8. Let $\mathscr{A}$ be an alternating parity automaton in normal form with n states and k different priorities. Then A^{nd} is an equivalent nondeterministic automaton with a number of states exponential in n and a number of priorities polynomial in n.

Proof. The claims about the number of states and number of priorities are obvious. The correctness proof can be carried out almost in the same fashion as the proof of Lemma 3.10, except for one issue. In order to see that it is admissible to merge all branches of a run on a certain branch of a given tree into an element of $\mathscr{Q}^\omega$, one has to use Theorem 2.20, the memoryless determinacy of parity games. Q.E.D.

As a consequence of Theorem 6.5 and Lemmas 6.7 and 6.8, we obtain:

Corollary 6.9. (Emerson-Streett-Jutla, [EJ91b]) Every MC expression can be translated into an equivalent nondeterministic parity tree automaton with an exponential number of states and a polynomial number of different priorities.

In view of Lemma 5.6 and Theorem 2.21, we can also conclude:

Corollary 6.10 (Emerson-Jutla, [EJ91a])**.** MC satisfiability is complete for exponential time.

For the lower bound, we refer to [EJ91a]. We finally note that a converse of Corollary 6.9 also holds:

Theorem 6.11 (Niwiński-Emerson-Jutla-Janin-Walukiewicz, [Niw88, EJ91b, JW95])**.** Let P be a finite set of propositional variables. For every alternating parity tree automaton and every nondeterministic tree automaton over 2^P, there exists an equivalent closed MC expression.

6.4 Notes

Satisfiability for MC is not only complexity-wise simpler than satisfiability for S2S. The proofs for showing decidability of satisfiability for S2S all make use of a determinization construction for automata on infinite words. The "safraless decision procedures" advocated by Kupferman and Vardi [KV05] avoid this, but they still use the fact that equivalent deterministic word automata of a bounded size exist.

The nondeterministic tree automaton models for SUS and MC are not only similar on the surface: A fundamental result by Janin and Walukiewicz [JW96] states that the bisimulation-invariant tree languages definable in SUS are exactly the tree languages definable in MC, where the notion of bisimulation exactly captures the phenomenon that MC expressions (just as CTL formulas) are resistant against duplicating subtrees.

MC has been extended in various ways with many different objectives. With regard to adding to its expressive power while retaining decidability, one of the most interesting results is by Grädel and Walukiewicz [GW99], which says that satisfiable is decidable for guarded fixed-point logic. This logic can be seen as an extension of the modal μ-calculus insofar as guarded logic is considered a natural extension of modal logic, and guarded fixed-point logic is an extension of guarded logic just as modal μ-calculus is an extension of model logic by fixed-point operators. For further extensions, see [LS02, VV04] and [KL]. Other important work with regard to algorithmic handling of MC was carried out by Walukiewicz in [Wal01], where he studies the evaluation of MC expressions on pushdown graphs.

References

[ADNM] André Arnold, Jacques Duparc, Damian Niwiński, and Filip Murlak. On the topological complexity of tree languages. This volume.

[AFF+02] Roy Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211. Springer, 2002.

[AKTZ06] Roy Armoni, D. Korchemny, A. Tiemeyer, and Moshe Y. Vardi Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proc. Workshop on Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer, 2006.

[AN01] André Arnold and Damian Niwiński. *Rudiments of μ-Calculus*. Elsevier, Amsterdam, The Netherlands, 2001.

[Arn83] André Arnold. Rational omega-languages are non-ambiguous. *Theor. Comput. Sci.*, 26:221–223, 1983.

[ATW06] Christoph Schulte Althoff, Wolfgang Thomas, and Nico Wallmeier. Observations on determinization of büchi automata. *Theor. Comput. Sci.*, 363(2):224–233, 2006.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The logic of nexttime. In *Proc. 8th ACM Symp. on Principles of Programming Languages (POPL)*, pages 164–176, 1981.

[BCL] Achim Blumensath, Thomas Colcombet, and Christof Löding. Logical theories and compatible operations. This volume.

[BDHK06] Dietmar Berwanger, Anuj Dawar, Paul Hunter, and Stephan Kreutzer. Dag-width and parity games. In Bruno Durand and Wolfgang Thomas, editors, *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23–25, 2006, Proceedings*, volume 3884 of *Lecture Notes in Computer Science*, pages 524–536, 2006.

[BFG+05] D. Bustan, A. Flaisher, Orna Grumberg, O. Kupferman, and Moshe Y. Vardi. Regular vacuity. In *Proc. 13th Conf. on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2005.

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problems*. Springer, 1997.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[Boj07] Mikolaj Bojanczyk. The common fragment of ctl and ltl needs existential modalities. Available as `http://www.mimuw.edu.pl/~bojan/papers/paradox.pdf`, 2007.

[Bra98] Julian C. Bradfield. The modal μ-calculus alternation hierarchy is strict. *Theor. Comput. Sci.*, 195(2):133–153, 1998.

[Büc77] J. Richard Büchi. Using determinancy of games to eliminate quantifiers. In *FCT*, pages 367–378, 1977.

[Bur74] Rod M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing 74*, pages 308–312, Stockholm, Sweden, August 1974. International Federation for Information Processing, North-Holland Pub. Co.

[Büc60] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[Büc62] J. Richard Büchi. On a decision method in restricted second-order arithmetic. In E. Nagel, P. Suppes, and A. Tarski, editors, *Logic, Methodology, and Philosophy of Science: Proc. of the 1960 International Congress*, pages 1–11, Stanford, Calif., 1962. Stanford University Press.

[Büc65] J. Richard Büchi. Decision methods in the theory of ordinals. *Bull. Am. Math. Soc*, 71:767–770, 1965.

[Cau] Didier Caucal. Deterministic graph grammars. This volume.

[CD88] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for*

Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30–June 3, 1988, Proceedings, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 1988.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: a practical approach. In ACM, editor, *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 117–126, Austin, Texas, 1983. ACM Press.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Chu60] Alonzo Church. Application of recursive arithmetics to the problem of circuit analysis. In Institute for Defense Analysis Communications Research Division, editor, *Summaries of Talks Presented at the Summer Institute for Symbolic Logic, Ithaca, Cornell University, July 1957*, pages 3–50, 1960.

[Chu63] Alonzo Church. Logic, arithmetics, and automata. In Institut Mittag-Leffler, editor, *Proc. Int. Congress of Mathematicians, 1962*, pages 23–35, 1963.

[CKS92] Rance Cleaveland, Marion Klein, and Bernhard Steffen. Faster model checking for the modal μ-calculus. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 – July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 410–422. Springer, 1992.

[CL07] Arnaud Carayol and Christof Löding. MSO on the infinite binary tree: Choice and order. In *CSL'07*, volume 4646 of *LNCS*, pages 161–176. Springer, 2007.

[CPP] Olivier Carton, Dominique Perrin, and Jean-Éric Pin. Automata and semigroups recognizing infinite words. This volume.

[CRST06] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From psl to nba: A modular symbolic encoding. In *Proc. 6th Int'l Conf. on Formal Methods in Computer-Aided design*, 2006.

[DG] Volker Diekert and Paul Gastin. First-order definable languages. This volume.

[DG05] Manfred Droste and Paul Gastin. Weighted automata and weighted logics. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 2005.

[Don70] John Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4(5):406–451, October 1970.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL.* Springer, 2006.

[EH85] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. System Sci.*, 30:1–24, 1985.

[EH86] E. Allen Emerson and Joseph Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *J. Assoc. Comput. Mach.*, 33(1):151–178, 1986.

[EJ91a] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science*, pages 328–337, San Juan, Puerto Rico, 1991. IEEE.

[EJ91b] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science*, pages 368–377, San Juan, Puerto Rico, October 1991. IEEE.

[EL86] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *1st IEEE Symposium on Symposium on Logic in Computer Science*, pages 267–278, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[Elg61] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–51, January 1961.

[EM93] Werner Ebinger and Anca Muscholl. Logical definability on infinite traces. In Andrzej Lingas, Rolf Karlsson, and Svante Carlsson, editors, *Automata, Languages and Programming: 20th International Colloquium*, volume 700 of *Lecture Notes in Computer Science*, pages 335–346, Lund, Sweden, 1993. EATCS, Springer.

[ES84a] E. Allen Emerson and A. Prasad Sistla. Deciding branching time logic. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, 1984, Washington, D.C., USA*, pages 14–24. ACM, 1984.

[ES84b] E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, 1984.

[FKV06] Ehud Friedgut, Orna Kupferman, and Moshe Y. Vardi. Büchi complementation made tighter. *Int. J. Found. Comput. Sci.*, 17(4):851–868, 2006.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. System Sci.*, 18:194–211, 1979.

[FWW97] Alain Finkel, B. Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown automata. In F. Moller, editor, *Proc. 2nd Int. Workshop on Verification of Infinite States Systems*, 1997.

[GH82] Yuri Gurevich and Leo Harrington. Trees, automata, and games. In *14th ACM Symposium on the Theory of Computing*, pages 60–65, San Francisco, 1982. ACM Press.

[GKSV03] S. Gurumurthy, O. Kupferman, F. Somenzi, and M.Y. Vardi. On complementing nondeterministic Büchi automata. In *Proc. 12th Conf. on Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2003.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[GPSS80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Conference Record of the 12th ACM Symposium on Principles of Programming Languages*, pages 163–173, Las Vegas, Nev., 1980.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[GR] Dora Giammarresi and Antonio Restivo. Matrix based complexity functions and recognizable picture languages. This volume.

[GRST96] Dora Giammarresi, Antonio Restivo, Sebastian Seibert, and Wolfgang Thomas. Monadic second-order logic over rectangular pictures and recognizability by tiling systems. *Information and Computation*, 125(1):32–45, February 1996.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[GW99] Erich Grädel and Igor Walukiewicz. Guarded fixed point logic. In *LICS*, pages 45–54, 1999.

[HJJ+95] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *TACAS: Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19–20, 1995, Proceedings*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110, 1995.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Engrg.*, 23(5):279–295, 1997.

[IK89] Neil Immerman and Dexter Kozen. Definability with bounded number of bound variables. *Information and Computation*, 83(2):121–139, November 1989.

[JPZ06] Marcin Jurdziński, Mike Paterson, and Uri Zwick. A deterministic subexponential algorithm for solving parity games. In

Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, pages 117–123. ACM/SIAM, 2006.

[Jur98] Marcin Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, November 1998.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.

[JW95] David Janin and Igor Walukiewicz. Automata for the modal mu-calculus and related results. In Jirí Wiedermann and Petr Hájek, editors, *Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, August 28 – September 1, 1995, Proceedings (MFCS)*, pages 552–562, 1995.

[JW96] David Janin and Igor Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 263–277, 1996.

[Kam68] Johan Anthony Willem Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, Calif., 1968.

[KL] Stephan Kreutzer and Martin Lange. Non-regular fixed-point logics and games. This volume.

[Kla91] Nils Klarlund. Progress measures for complementation of ω-automata with applications to temporal logic. In *32nd Annual Symposium on Foundations of Computer Science, 1–4 October 1991, San Juan, Puerto Rico*, pages 358–367, 1991.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KP84] Dexter Kozen and Rohit Parikh. A decision procedure for the propositional μ-calculus. In *Logics of Programs*, volume 164 of

Lecture Notes in Computer Science, pages 313–325. Springer, 1984.

[KPV02] O. Kupferman, N. Piterman, and Moshe Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc 14th Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2002.

[Krö77] Fred Kröger. LAR: A logic of algorithmic reasoning. *Acta Informatica*, 8(3), August 1977.

[KV97] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. In *ISTCS*, pages 147–158, 1997.

[KV99] Orna Kupferman and Moshe Y. Vardi. The weakness of self-complementation. In *STACS*, pages 455–466, 1999.

[KV01] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Logic*, 2(3):408–429, 2001.

[KV05] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings (FOCS)*, pages 531–542. IEEE Computer Society, 2005.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. Assoc. Comput. Mach.*, 47(2):312–360, 2000.

[Käh01] Detlef Kähler. Determinisierung von É-Automaten. Diploma thesis, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, 2001.

[Lam80] Leslie Lamport. "Sometimes" is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symp. on Principles of Programming Languages (POPL)*, pages 174–185, 1980.

[Len96] Giacomo Lenzi. A hierarchy theorem for the μ-calculus. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8-12 July 1996, Proceedings*, volume 1099 of *Lecture Notes in Computer Science*, pages 87–97. Springer, 1996.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages"*, pages 97–107, 1985.

[LS02] Martin Lange and Colin Stirling. Model checking fixed point logic with chop. In Mogens Nielsen and Uffe Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 250–263. Springer, 2002.

[LT00] Christof Löding and Wolfgang Thomas. Alternating automata and logics over infinite words. In *Proceedings of the IFIP International Conference on Theoretical Computer Science, IFIP TCS2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 521–535. Springer, 2000.

[Mar75] Donald A. Martin. Borel determinacy. *Ann. Math.*, 102:363–371, 1975.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[MH84] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. In *CAAP*, pages 195–210, 1984.

[Mic88] M. Michel. Complementation is more difficult with automata on infinite words, 1988.

[Mos91] Andrzej W. Mostowski. Games with forbidden positions. Preprint 78, Uniwersytet Gdańsk, Instytyt Matematyki, 1991.

[MS] Oliver Matz and Nicole Schweikardt. Expressive power of monadic logics on words, trees, pictures, and graphs. This volume.

[MS85] David E. Muller and Paul E. Schupp. Alternating automata on infinite objects, determinacy and rabin's theorem. In Maurice Nivat and Dominique Perrin, editors, *Automata on Infinite Words, Ecole de Printemps d'Informatique Théorique, Le MontDore, May 14–18, 1984*, volume 192 of *Lecture Notes in Computer Science*, pages 100–107. Springer, 1985.

[MS95] David E. Muller and Paul E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of rabin, mcnaughton and safra. *Theor. Comput. Sci.*, 141(1&2):69–107, 1995.

[MSS88] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5–8 July 1988, Edinburgh, Scotland, UK (LICS)*, pages 422–427. IEEE Computer Society, 1988.

[Mul63] David E. Muller. Infinite sequences and finite machines. In *Proceedings of the 4th Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, 1963.

[Niw86] Damian Niwiński. On fixed point clones. In Laurent Kott, editor, *Automata, Languages and Programming: 13th International Colloquium*, volume 226 of *Lecture Notes in Computer Science*, pages 464–473, Rennes, France, 1986. Springer-Verlag, Berlin.

[Niw88] Damian Niwinski. Fixed points vs. infinite generation. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5–8 July 1988, Edinburgh, Scotland, UK (LICS)*, pages 402–409. IEEE Computer Society, 1988.

[NW] Damian Niwiński and Igor Walukiewicz. Ambiguity problem for automata on infinite trees. Unpublished note.

[Per86] Dominique Perrin. Recent results on automata on infinite words. In Laurent Kott, editor, *13th Intern. Coll. on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 134–148, Rennes, France, 1986. Springer-Verlag, Berlin.

[Pit06] Nir Piterman. From nondeterministic büchi and streett automata to deterministic parity automata. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 255–264. IEEE Computer Society, 2006.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Rhode Island, Providence, 1977. IEEE.

[PP03] Dominique Perrin and Jean-Éric Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Pure and Applied Mathematics. Elsevier, 2003.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

[Rab69] Michael Ozer Rabin. Decidability of second-order theories and finite automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[Rab70] Michael O. Rabin. Weakly definable relations and special automata. In *Proc. Symp. Mathematical Logic and Foundation of Set Theory*, pages 1–23. North Holland, 1970.

[Rei01] Klaus Reinhardt. The complexity of translating logic to finite automata. In *Automata, Logics, and Infinite Games*, pages 231–238, 2001.

[Roh97] Gareth Scott Rohde. *Alternating automata and the temporal logic of ordinals*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[Ros92] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[Saf88] Shmuel Safra. On the complexity of ω-automata. In *29th Annual Symposium on Foundations of Computer Science*, pages 319–327, White Plains, New York, 1988. IEEE.

[SC82] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, 5–7 May 1982, San Francisco, California, USA (STOC)*, pages 159–168. ACM, 1982.

[SC85] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. Assoc. Comput. Mach.*, 32(3):733–749, 1985.

[Sch02] Philippe Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic*, pages 393–436. King's College Publications, 2002.

[SE84] Robert S. Streett and E. Allen Emerson. The propositional mu-calculus is elementary. In Jan Paredaens, editor, *Automata,*

Languages and Programming, 11th Colloquium, Antwerp, Belgium, July 16-20, 1984, Proceedings (ICALP), volume 172 of *Lecture Notes in Computer Science*, pages 465–472. Springer, 1984.

[Sei96] Helmut Seidl. Fast and simple nested fixpoints. *Information Processing Letters*, 59(6):303–308, 1996.

[SM73] Larry Joseph Stockmeyer and Albert R. Meyer. Word problems requiring exponential time. In *Fifth Annual ACM Symposium on Theory of Computation*, pages 1–9, Austin, Texas, 1973. ACM Press.

[Sti01] Colin Stirling. *Modal and temporal properties of processes*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[Sto74] Larry Joseph Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Dept. of Electrical Engineering, MIT, Boston, Mass., 1974.

[Str82] Robert S. Streett. Propositional dynamic logic of looping and converse. *Inform. Contr.*, 54:121–141, 1982.

[Tho79] Wolfgang Thomas. Star-free regular sets of ω-sequences. *Inform. and Control*, 42:148–156, 1979.

[Tho82] Wolfgang Thomas. Classifying regular events in symbolic logic. *Journal of Computer and System Sciences*, 25:360–376, 1982.

[Tho90a] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, pages 134–191. Elsevier, Amsterdam, 1990.

[Tho90b] Wolfgang Thomas. On logical definability of regular trace languages. In Volker Diekert, editor, *Proceedings of a Workshop of the ESPRIT Basic Research Action No. 3166: Algebraic and Syntactic Methods in Computer Science (ASMICS)*, pages 172–182, Kochel am See, BRD, 1990. Bericht TUM-I901902, Technische Universität München.

[Tho97] Wolfgang Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 389–455. Springer, Berlin, 1997.

[Tho99] Wolfgang Thomas. Complementation of büchi automata revised. In Juhani Karhumäki, Hermann A. Maurer, Gheorghe Paun, and Grzegorz Rozenberg, editors, *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 109–120. Springer, 1999.

[Tra62] Boris A. Trakhtenbrot. Finite automata and the logic of one-place predicates. *Siberian Math. J.*, 3:103–131, 1962. (English translation in: AMS Transl. 59 (1966) 23–55.).

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order arithmetic. *Mathematical System Theory*, 2(1):57–81, 1968.

[Var07] Moshe Y. Vardi. Automata-theoretic model checking revisited. In *Proc. 7th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2007.

[VV04] Mahesh Viswanathan and Ramesh Viswanathan. A higher order modal fixed point logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 512–528. Springer, 2004.

[VW86a] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In Dexter Kozen, editor, *First Annual IEEE Symposium on Logic in Computer Science*, pages 322–331, Cambridge, Mass., 16–18 June 1986. IEEE.

[VW86b] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2):182–221, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.

[Wal96] Igor Walukiewicz. Monadic second order logic on tree-like structures. In Claude Puech and Rüdiger Reischuk, editors, *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22–24, 1996, Proceedings*, volume 1046 of *Lecture Notes in Computer Science*, pages 401–413, 1996.

[Wal00] Igor Walukiewicz. Completeness of kozen's axiomatisation of the propositional μ-calculus. *Inf. Comput.*, 157(1-2):142–182, 2000.

[Wal01] Igor Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comput.*, 164(2):234–263, 2001.

[Yan06] Qiqi Yan. Lower bounds for complementation of *mega*-automata via the full automata technique. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10–14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 589–600, 2006.

[Zuc86] Lenore D. Zuck. *Past Temporal Logic*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, August 1986.

Chapter 1

Classical Logic I: First-Order Logic

Wilfrid Hodges

1.1. First-Order Languages

The word 'logic' in the title of this chapter is ambiguous.

In its first meaning, a *logic* is a collection of closely related artificial languages. There are certain languages called *first-order languages*, and together they form first-order logic. In the same spirit, there are several closely related languages called modal languages, and together they form modal logic. Likewise second-order logic, deontic logic and so forth.

In its second but older meaning, *logic* is the study of the rules of sound argument. First-order languages can be used as a framework for studying rules of argument; logic done this way is called *first-order logic*. The contents of many undergraduate logic courses are first-order logic in this second sense.

This chapter will be about first-order logic in the first sense: a certain collection of artificial languages. In Hodges (1983), I gave a description of first-order languages that covers the ground of this chapter in more detail. That other chapter was meant to serve as an introduction to first-order logic, and so I started from arguments in English, gradually introducing the various features of first-order logic. This may be the best way in for beginners, but I doubt if it is the best approach for people seriously interested in the philosophy of first-order logic; by going gradually, one blurs the hard lines and softens the contrasts. So, in this chapter, I take the opposite route and go straight to the first-order sentences. Later chapters have more to say about the links with plain English.

The chief pioneers in the creation of first-order languages were Boole, Frege and C. S. Peirce in the nineteenth century; but the languages became public knowledge only quite recently, with the textbook of Hilbert and Ackermann (1950), first published in 1928 but based on lectures of Hilbert in 1917–22. (So first-order logic has been around for about 70 years, but Aristotle's syllogisms for well over 2000 years. Will first-order logic survive so long?)

From their beginnings, first-order languages have been used for the study of deductive arguments, but not only for this – both Hilbert and Russell used first-order formulas as an aid to definition and conceptual analysis. Today, computer

science has still more uses for first-order languages, e.g., in knowledge representation and in specifying the behavior of systems.

You might expect at this point to start learning what various sentences in first-order languages *mean*. However, first-order sentences were never intended to mean anything; rather they were designed to express *conditions which things can satisfy or fail to satisfy*. They do this in two steps.

First, each first-order language has a number of symbols called *nonlogical constants*; older writers called them *primitives*. For brevity, I shall call them simply *constants*. To use a first-order sentence ϕ, something in the world – a person, a number, a colour, whatever – is attached (or in the usual jargon, *assigned*) to each of the constants of ϕ. There are some restrictions on what kind of thing can be assigned to what constant; more on that later. The notional glue that does the attaching is called an *interpretation* or a *structure* or a *valuation*. These three words have precise technical uses, but for the moment 'interpretation' is used as the least technical term.

Second, given a first-order sentence ϕ and an interpretation I of ϕ, the *semantics* of the first-order language determine either that I makes ϕ true, or that I makes ϕ false. If I makes ϕ true, this is expressed by saying that I *satisfies* ϕ, or that I is a *model* of ϕ, or that ϕ is *true in* I or *under* I. (The most natural English usage seems to be 'true in a structure' but 'true under an interpretation.' Nothing of any importance hangs on the difference between 'under' and 'in,' and I will not be entirely consistent with them.) The *truth-value* of a sentence under an interpretation is Truth if the interpretation makes it true, and Falsehood if the interpretation makes it false.

The main difference between one first-order language and any other lies in its set of constants; this set is called the *signature* of the language. (First-order languages can also differ in other points of notation, but this shall be ignored here.) If σ is a signature of some first-order language, then an interpretation is said to be *of signature* σ if it attaches objects to exactly those constants that are in σ. So an interpretation of signature σ contains exactly the kinds of assignment that are needed to make a sentence of signature σ true or false.

Examples of first-order languages must wait until some general notions are introduced in the next section, but as a foretaste, many first-order languages have a sentence that is a single symbol

$$\bot$$

pronounced 'absurdity' or 'bottom.' Nobody knows or cares what this symbol *means*, but the semantic rules decree that it is false. So, it has no models. It is not a nonlogical constant; its truth-value does not depend on any assignment.

1.2. Some Fundamental Notions

In the definitions below, it is assumed that some fixed signature σ has been chosen; the sentences are those of the first-order language of signature σ and the

interpretations are of signature σ. So each interpretation makes each sentence either true or false:

true? false?
interpretations $I \leftrightarrow$ sentences ϕ

This picture can be looked at from either end. Starting from an interpretation I, it can be used as a knife to cut the class of sentences into two groups: the sentences which it satisfies and the sentences which it does not satisfy. The sentences satisfied by I are together known as the (first-order) *theory of I*. More generally, any set of sentences is called a *theory*, and I is a *model* of a theory T if it is a model of every sentence in T. By a standard mathematical convention, every interpretation is a model of the empty theory, because the empty theory contains no sentence that is false in the interpretation.

Alternatively, the picture can be read from right to left, starting with a sentence ϕ. The sentence ϕ separates the class of interpretations into two collections: those which satisfy it and those which do not. Those which satisfy ϕ are together known as the *model class* of ϕ. In fact, a similar definition can be given for any theory T: the *model class* of T is the class of all interpretations that satisfy T. If a particular class **K** of interpretations is the model class of a theory T, then T is a *set of axioms* for **K**. This notion is important in mathematical applications of first-order logic, because many natural classes of structures – e.g., the class of groups – are the model classes of first-order axioms.

Two theories are said to be *logically equivalent*, or more briefly *equivalent*, if they have the same model class. As a special case, two sentences are said to be *equivalent* if they are true in exactly the same interpretations. A theory is said to be (semantically) *consistent* if it has at least one model; otherwise, it is (semantically) *inconsistent*. There are many semantically inconsistent theories, for example the one consisting of the single sentence '$\bot$'. The word 'semantically' is a warning of another kind of inconsistency, discussed at the end of section 1.8.

Suppose T is a theory and ψ is a sentence. Then T *entails* ψ if there is no interpretation that satisfies T but not ψ. Likewise, ψ is *valid* if every interpretation makes ψ true. One can think of validity as a special case of entailment: a sentence is valid if and only if it is entailed by the empty theory.

The symbol '$\vdash$' is pronounced 'turnstile.' A *sequent* is an expression

$$T \vdash \psi$$

where T on the left is a theory and ψ on the right is a sentence. The sentences in T are called the *premises* of the sequent and ψ is called its *conclusion*. The sequent is *valid* if T entails ψ, and *invalid* otherwise. If T is a finite set of sentences, the sequent '$T \vdash \psi$' can be written as a *finite sequent*

$$\phi_1, \ldots, \phi_n \vdash \psi$$

listing the contents of T on the left. The language under discussion (i.e. the first-order language of signature σ) is said to be *decidable* if there is an algorithm (i.e. a

mechanical method which always works) for telling whether any given finite sequent is valid.

A *proof calculus* C consists of

(i) a set of rules for producing patterns of symbols called *formal proofs* or *derivations*, and
(ii) a rule which determines, given a formal proof and a sequent, whether the formal proof is a *proof* of the sequent.

Here 'proof of' is just a set of words; but one of the purposes of proof calculi is that they should give 'proofs of' all and only the valid sequents. The following definitions make this more precise:

1 A sequent

$$T \vdash \psi$$

is *derivable in* C, or in symbols

$$T \vdash_C \psi$$

if some formal proof in the calculus C is a proof of the sequent.
2 A proof calculus C is *correct* (or *sound*) if no invalid sequent is derivable in C.
3 C is *complete* if every valid sequent is derivable in C.

So a correct and complete proof calculus is one for which the derivable sequents are exactly the valid ones. One of the best features of first-order logic, from almost anybody's point of view, is that it has several excellent correct and complete proof calculi. Some are mentioned in section 1.8.

1.3. Grammar and Semantics

As in any language, the sentences of first-order languages have a grammatical structure. The details vary from language to language, but one feature that all first-order languages share is that *the grammatical structure of any given sentence is uniquely determined*. There are no grammatically ambiguous sentences like Chomsky's

They are flying planes.

This property of first-order languages is called the *unique parsing property*.

To guarantee unique parsing, first-order formulas generally have a large number of brackets. There are conventions for leaving out some of these brackets without introducing any ambiguity in the parsing. For example, if the first and last symbols of a sentence are brackets, they can be omitted. Any elementary textbook gives further details.

For historical reasons, there is a hitch in the terminology. With a first-order language, the objects that a linguist would call 'sentences' are called *formulas* (or in some older writers *well-formed formulas* or wff), and the word 'sentence' is reserved for a particular kind of formula, as follows.

Every first-order language has an infinite collection of symbols called *variables*:

$$x_0, x_1, x_2, \ldots$$

To avoid writing subscripts all the time, it is often assumed that

$$x, y, z, u, v$$

and a few similar symbols are variables too. Variables are not in the signature. From a semantic point of view, variables can occur in two ways: when a variable at some point in a formula needs to have an object assigned to it to give the formula a truth-value, this occurrence of the variable is called *free*; when no such assignment is needed, it is called *bound*. A *sentence* is a formula with no free occurrences of variables. To avoid confusing variables with constants, an assignment of objects to the variables is called a *valuation*. So, in general, a first-order formula needs an interpretation I of its constants *and* a valuation v of its variables to have a truth-value. (It will always be clear whether 'v' means a variable or a valuation.)

The definitions of the previous section all make sense if 'sentence' is read as 'first-order sentence'; they also make sense if 'sentence' is read as 'first-order formula' and 'interpretation' as 'interpretation plus valuation'. Fortunately, the two readings do not clash; for example, a sequent of first-order sentences is valid or invalid, regardless of whether the first-order sentences are regarded as sentences or as formulas. That needs a proof – one that can be left to the mathematicians. Likewise, according to the mathematicians, a first-order language is decidable in terms of sentences if and only if it is decidable in terms of formulas. (Be warned though that 'first-order theory' normally means 'set of first-order sentences' in the narrow sense. To refer to a set of first-order formulas it is safest to say 'set of first-order formulas.')

The next few sections present the semantic rules in what is commonly known as the *Tarski style* (in view of Tarski (1983) and Tarski and Vaught (1957). In this style, to find out what interpretations-plus-valuations make a complex formula ϕ true, the question is reduced to the same question for certain formulas that are simpler than ϕ. The Tarski style is not the only way to present the semantics. A suggestive alternative is the Henkin–Hintikka description in terms of games; see Hintikka (1973, ch. V), or its computer implementation by Barwise and Etchemendy (1999). Although the Tarski-style semantics and the game semantics look very different, they always give the same answer to the question: 'What is the truth-value of the sentence ϕ under the interpretation I?'

Throughout this chapter, symbols such as 'ϕ', 'α' are used to range over the formulas or terms of a first-order language. They are *metavariables*; in other words, they are not in the first-order languages but are used for talking about these languages. On the other hand, so as not to saddle the reader with still more metavariables for the other expressions of a first-order language, for example, when making a

general statement about variables, a typical variable may be used as if it was a meta-variable. Thus 'Consider a variable x' is common practice. More generally, quotation marks are dropped when they are more of a nuisance than a help.

1.4. The First-Order Language with Empty Signature

The simplest first-order language is the one whose signature is empty. In this section, this language is referred to as L.

An interpretation I with empty signature does not interpret any constants, but – for reasons that will appear very soon – it does have an associated class of objects, called its *universe* or *domain*. (The name 'domain' is perhaps more usual; but it has other meanings in logic so, to avoid confusion, 'universe' is used instead.) Most logicians require that the universe shall have at least one object in it; but, apart from this, it can be any class of objects. The members of the domain are called the *elements* of the interpretation; some older writers call them the *individuals*. A *valuation in* I is a rule v for assigning to each variable x_i an element $v(x_i)$ in the universe of I.

For the grammatical constructions given here, an interpretation I and a valuation v in I are assumed. The truth-values of formulas will depend partly on I and partly on v. A formula is said to be *true in* I *under* v.

Some expressions of L are called *atomic formulas*. There are two kinds:

- Every expression of the form '$(x = y)$', where x and y are variables, is an atomic formula of L.
- '$\perp$' is an atomic formula of L.

It has already been noted that '$\perp$' is false in I. The truth-value of $(x_1 = x_3)$, to take a typical example of the other sort of atomic formula, is

Truth	if $v(x_1)$ is the same element as $v(x_3)$
Falsehood	if not

So, given I and v, truth-values are assigned to the atomic formulas.

Next, a class of expressions called the *formulas* of L is defined. The atomic formulas are formulas, but many formulas of L are not atomic. Take the five symbols

$$\sim \quad \& \quad \vee \quad \supset \quad \equiv$$

which are used to build up complex formulas from simpler ones. There is a grammatical rule for all of them.

If ϕ and ψ are formulas, then so are each of these expressions:

$$(\sim \phi) \qquad (\phi \,\&\, \psi) \qquad (\phi \vee \psi) \qquad (\phi \supset \psi) \qquad (\phi \equiv \psi)$$

This chart, called a *truth-table*, shows which of these formulas are true, depending on whether ϕ and ψ are true:

ϕ	ψ	$(\sim \phi)$	$(\phi \,\&\, \psi)$	$(\phi \vee \psi)$	$(\phi \supset \psi)$	$(\phi \equiv \psi)$
T	T	F	T	T	T	T
T	F		F	T	F	F
F	T	T	F	T	T	F
F	F		F	F	T	T

(Here T = Truth and F = Falsehood.) Because of this table, the five symbols '$\sim$', '&', '$\vee$', '$\supset$', and '$\equiv$' are known as the *truth-functional symbols.*

For example, the formula

$$(((x_2 = x_3) \,\&\, (x_5 = x_2)) \supset (x_5 = x_3))$$

is false in just one case, namely where $v(x_2)$ and $v(x_3)$ are the same element, and $v(x_5)$ and $v(x_2)$ are the same element, but $v(x_5)$ and $v(x_3)$ are not the same element. Since this case can never arise, the formula is true regardless of what I and v are.

There remain just two grammatical constructions. The grammatical rule for them both is:

> If ϕ is any formula and x is any variable, then the expressions
>
> $(\forall x)\phi \qquad (\exists x)\phi$
>
> are both formulas.

The expressions '$(\forall x)$' and '$(\exists x)$' are called respectively a *universal quantifier* and an *existential quantifier*, and read respectively as 'for all x' and 'there is x'. In the two formulas given by the rule, the occurrence of x inside the quantifier is said to *bind* itself and any occurrences of the *same* variable in the formula ϕ. These occurrences stay bound as still more complex formulas are built. In any formula, an occurrence of a variable which is not bound by a quantifier in the formula is said to be *free*. A formula with no free variables is called a *sentence*. (And this is what was meant earlier by 'sentences' of first-order languages. The syntactic definitions just given are equivalent to the semantic explanation in the previous section.) For example, this is not a sentence:

$$(\exists y)(\sim (x = y))$$

because it has a free occurrence of x (though both occurrences of y are bound by the existential quantifier). But this is a sentence:

$$(\forall x)(\exists y)(\sim (x = y))$$

because its universal quantifier binds the variable x.

The semantic rules for the quantifiers are one of the harder concepts of first-order logic. For more than two millennia, some of the best minds of Europe struggled to formulate semantic rules that capture the essence of the natural language expressions 'all' and 'there is.'

- '$(\forall x)\phi$' is true in I under v if: for every element a in the universe of I, if w is taken to be the valuation exactly like v except that $w(x)$ is a, then ϕ is true in I under w.
- '$(\exists x)\phi$' is true in I under v if: there is an element a in the universe of I, such that if w is taken to be the valuation exactly like v except that $w(x)$ is a, then ϕ is true in I under w.

For example, the formula

$$(\exists y)(\sim (x = y))$$

is true in I under v, iff (if and only if) there is an element a such that $v(x)$ is not the same element as a. So the sentence

$$(\forall x)(\exists y)(\sim (x = y))$$

is true in I under v iff for every element b there is an element a such that b is not the same element as a. In other words, it is true iff the universe of I contains at least two different elements.

Note that this last condition depends only on I and not at all on v. One can prove that the truth-value of a formula ϕ in I and v never depends on $v(x)$ for any variable x that does not occur free in ϕ. Since sentences have no free variables, their truth-value depends only on I and the valuation slips silently away.

These rules capture the essence of the expressions 'all' and 'there is' by stating precise conditions under which a sentence starting with one of these phrases counts as true. The same applies to the truth-functional symbols, which are meant, in some sense, to capture at least the mathematical use of the words 'not', 'and', 'or', 'if . . . then', and 'if and only if'.

1.5. Some Notation

The notation in this section applies to all first-order languages, not just the language with empty signature.

Writing a formula as $\phi(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are different variables means that the formula is ϕ and it has no occurrences of free variables except perhaps for $x_1, \ldots, x_n$. Then

$$I \vDash \phi[a_1, \ldots, a_n]$$

means that ϕ is true in the interpretation I and under some valuation v for which $v(x_1), \ldots, v(x_n)$ are $a_1, \ldots, a_n$ respectively (or under any such valuation v – it makes no difference). When ϕ is a sentence, the $a_1, \ldots, a_n$ are redundant and

$$I \models \phi$$

simply means that ϕ is true in I.

Here is another useful piece of notation:

$$\phi(y/x)$$

means the formula obtained by replacing each free occurrence of x in ϕ by an occurrence of y. Actually, this is not quite right, but the correction in the next paragraph is rather technical. What is intended is that the formula $\phi(y/x)$ 'says about y' the same thing that ϕ 'said about x.'

Suppose, for example, that ϕ is the formula

$$(\forall y)(x = y)$$

which expresses that x is identical with everything. Simply putting y in place of each free occurrence of x in ϕ, gives

$$(\forall y)(y = y)$$

This says that each thing is identical to itself; whereas the intention was to make the more interesting statement that y is identical with everything. The problem is that y is put into a place where it immediately becomes bound by the quantifier $(\forall y)$. So $\phi(y/x)$ must be defined more carefully, as follows. First, choose another variable, say z, that does not occur in ϕ, and adjust ϕ by replacing all bound occurrences of y in ϕ by bound occurrences of z. After this, substitute y for free occurrences of x. (So $\phi(y/x)$ in our example now works out as

$$(\forall z)(y = z)$$

which says the right thing.) This more careful method of substitution is called *substitution avoiding clash of variables.*

The language L of the previous section is a very arid first-order language. The conditions that it can express on an interpretation I are very few. It can be used to say that I has at least one element, at least two elements, at least seven elements, either exactly a hundred or at least three billion elements, and similar things; but nothing else. (Is there a single sentence of L which expresses the condition that I has infinitely many elements? No. This is a consequence of the compactness theorem in section 1.10.)

Nevertheless L already shows some very characteristic features of first-order languages. For example, to work out the truth-value of a sentence ϕ under an interpretation I, one must generally consider the truth-values of subformulas of ϕ

under various valuations. As explained in section 1.3, the notion of a valid sequent applies to formulas as well as sentences; but for formulas it means that every interpretation-plus-valuation making the formulas on the left true makes the formula on the right true too.

Here are two important examples of valid sequents of the language L. The sequent

$$\vdash (x = x)$$

is valid because $v(x)$ is always the same element as $v(x)$. The sequent

$$(x = y) \vdash (\phi \supset \phi(y/x))$$

is valid because if two given elements are the same element, then they satisfy all the same conditions.

1.6. Nonlogical Constants: Monadic First-Order Logic

Section 1.5 ignored the main organ by which first-order formulas reach out to the world: the signature, the family of nonlogical constants.

The various constants can be classified by the kinds of feature to which they have to be attached in the world. For example, some constants are called *class symbols* because their job is to stand for classes. (Their more technical name is 1-*ary relation symbols*.) Some other constants are called *individual constants* because their job is to stand for individuals, i.e. elements. This section concentrates on languages whose signature contains only constants of these two kinds. Languages of this type are said to be *monadic*. Let L be a monadic language.

Usually, individual constants are lower-case letters 'a', 'b', 'c' etc. from the first half of the alphabet, with or without subscripts. Usually class symbols are capital letters 'P', 'Q', 'R' etc. from the second half of the alphabet, with or without number subscripts.

Grammatically these constants provide some new kinds of atomic formula. It is helpful first to define the *terms* of L. There are two kinds:

- Every variable is a term.
- Every individual constant is a term.

The definition of *atomic formula* needs revising:

- Every expression of the form '$(\alpha = \beta)$', where α and β are terms, is an atomic formula of L.
- If P is any class symbol and α any term, then '$P(\alpha)$' is an atomic formula.
- '$\perp$' is an atomic formula of L.

Apart from these new clauses, the grammatical rules remain the same as in section 1.4.

What should count as an interpretation for a monadic language? Every interpretation I needs a universe, just as before. But now it also needs to give the truth-value of $P(x)$ under a valuation that ties x to an element $v(x)$, which might be any element of the universe. In other words, the interpretation needs to give the class, written P^I, of all those elements a such that $P(x)$ is true under any valuation v with $v(x)$ equal to a. (Intuitively P^I is the class of all elements that satisfy $P(x)$ in I.) Here P^I might be any subclass of the universe.

In the branch of logic called model theory, the previous paragraph turns into a definition. A *structure* is an interpretation I of the following form:

- I has a universe, which is a set (generally taken to be non-empty).
- For each class symbol P in the signature, I picks out a corresponding class P^I, called the *interpretation of P under I*, all of whose members are in the universe.
- For each individual constant a in the signature, I picks out a corresponding element a^I in the universe, and this element is called the *interpretation of a under I*.

Writing σ for the signature in question, this interpretation I is called a *σ-structure*. So a σ-structure contains exactly the information needed to give a truth-value to a sentence of signature σ. Note that the interpretations a^I are needed to deal with sentences such as $P(a)$. (The requirement that the universe should be a set rather than a class is no accident: a set is a mathematically well-behaved class. The precise difference is studied in texts of set theory. [See chapter 3.])

However, this model-theoretic definition is not as old as first-order logic. In the early days, logicians would give an interpretation for a by writing down a name or a description of a thing or person. They would give an interpretation for P by writing down a sentence of English or their own native language with x in place of one or more pronouns. Or sometimes they would write a sentence with 'He' or 'It' left off the front; or more drastically, a sentence with 'He is a' left off. For example an interpretation might contain the items:

P : x is kind to people who are kind to x.

Q : is mortal

R : taxpayer

The third style here is the least flexible, but anyway it is not needed; it can easily be converted to the second style by writing 'is a taxpayer.' The second style in turn is less flexible than the first, and again is not needed. Q and R could be written 'x is mortal', 'x is a taxpayer'. A sentence with variables in place of some pronouns is sometimes called a *predicate*.

Can every interpretation by predicates be converted into a structure? Yes, provided that each of the predicates has a certain property: *the question whether an element of the universe satisfies the predicate always has a definite answer* (*Yes or No*) *which depends only on the element and not on how it is described*. Predicates with this property are said to be *extensional*. The following predicates seem not to be extensional

– though this is an area where people have presented strong arguments for some quite surprising conclusions:

x is necessarily equal to 7.

I recognized x.

The prevailing view is that to handle predicates like these, a logic with a subtler semantics than first-order logic is needed. Modal logic takes on board the first example, epistemic logic the second. [See chapters 7 and 9.]

The predicate

x is bald.

also fails the test, not because it is possible to be bald under one name and bushy-haired under another, but because there are borderline cases – people who aren't definitely bald or definitely not bald. So this predicate does not succeed in defining a class of people. Truth to tell, most natural language predicates are at least slightly vague; even logicians have to live with the roughnesses of the real world.

Given an interpretation I that uses predicates, a first-order sentence ϕ can often be translated into an English sentence which is guaranteed to be true if and only if ϕ is true in I. The translation will generally need to mention the universe of the interpretation, unless a predicate is used to describe that too. Here are some examples, using the interpretation a couple of paragraphs above, together with the universe described by 'x is a person':

$(\forall x)(R(x) \to Q(x))$
Every person who is a taxpayer is mortal.

$(\exists x)P(x)$
At least one person is kind to people who are kind to him or her.

$(\exists x)(R(x) \,\&\, (\forall y)(R(y) \equiv (y = x)))$
Exactly one person is a taxpayer.

The reader may well agree with the following comment: If these first-order sentences are being used to express the English sentences in question, then it is artificial to ask for a universe at all. In ordinary speech, no one asks people to state their universes.

This comment needs answers on several levels. First, mathematical objects – such as groups, rings, boolean algebras and the like – consist of a set of objects with certain features picked out by nonlogical constants. So it was natural for the mathematical creators of first-order logic to think of this set of objects as a universe.

Second, there is a mathematical result that takes some of the sting out of the requirement that a universe has to be chosen. An occurrence of a universal quantifier is *restricted* if it occurs as follows:

$$(\forall x)(P(x) \supset \cdots)$$

i.e. followed immediately by a left bracket, a class symbol with the same variable, and then '$\supset$'. Likewise an occurrence of an existential quantifier is *restricted* if it looks like this:

$$(\exists x)(P(x) \,\&\, \cdots)$$

The mathematical result states:

Theorem 1.1 Let ϕ be a sentence of the first-order language L of signature σ, and suppose that all occurrences of quantifiers in ϕ are restricted. Let I be a σ-structure and J be another σ-structure which comes from I by removing some elements which are not inside P^I for any class symbol P. Then ϕ has the same truth-value in I and in J.

First-order sentences that serve as straightforward translations of English sentences usually have all their quantifiers restricted, as in the first and third examples above. (The second example can be rewritten harmlessly as

$$(\exists x)(P(x) \,\&\, P(x))$$

and then its quantifier is restricted too.) So the choice of universe may be largely *ad hoc*, but it is also largely irrelevant. (This theorem remains true for first-order languages that are not monadic.)

Third, if the class symbols are interpreted by predicates rather than by classes, the choice of universe certainly can make a difference to truth-values, even for sentences covered by the theorem just stated. Suppose, for example, that an interpretation is being used, with

P : x is a person.

Q : x will be dead before the year 2200.

With such an interpretation, the sentence

$$(\forall x)(P(x) \rightarrow Q(x))$$

expresses that every person will be dead before the year 2200. This is probably true of people alive now, but probably false if 'person' includes people yet to be born. So different universes give different truth-values. Why does this not contradict Theorem 1.1? Because the predicate 'x is a person' picks out different classes according as future people are excluded or included, so that the corresponding σ-structures differ in their assignments to P and Q, not just in their universes.

If a universe can contain future people, can it contain possible people, or fictional people, or even impossible people (like the man I met who wasn't there, in the

children's jingle)? Or to be more metaphysical, can a universe contain as separate elements myself-now and myself-ten-years-ago? First-order logic is very robust about questions like these: it doesn't give a damn. If you think that there are fictional people and that they have or fail to have this property or that, and can meaningfully be said to be the same individuals or not the same individuals as one another, then fine, put them in your universes. Likewise, if you think there are time-slices of people. If you don't, then leave them out.

All these remarks about universes apply equally well to the more general first-order languages of section 1.7. Here is a theorem that does not.

Theorem 1.2 If L is a monadic first-order language with a finite signature, then L is decidable.

See, for example, Boolos and Jeffrey (1974, ch. 25, 'Monadic versus dyadic logic') for a proof of this theorem.

1.7. Some More Nonlogical Constants

Most logicians before about 1850, if they had been set to work designing a first-order language, would probably have been happy to stick with the kinds of constant already introduced here. Apart from some subtleties and confusions about empty classes, the traditional syllogistic forms correspond to the four sentence-types

$(\forall x)(P(x) \supset Q(x))$ $\quad$ $(\forall x)(P(x) \supset \sim Q(x))$

$(\exists x)(P(x) \,\&\, Q(x))$ $\quad$ $(\exists x)(P(x) \,\&\, \sim Q(x))$

The main pressure for more elaborate forms came from mathematics, where geometers wanted symbols to represent predicates such as:

x is a point lying on the line y.

x is between y and z.

Even these two examples show that there is no point in restricting ourselves in advance to some fixed number of variables. So, class symbols are generalized to *n-ary relation symbols*, where the *arity*, n, is the number of distinct variables needed in a predicate that interprets the relation symbol.

Like class symbols, relation symbols are usually 'P', 'Q', etc., i.e. capital letters from near the end of the alphabet. An *ordered n-tuple* is a list

$\langle a_1, \ldots, a_n \rangle$

where a_i is the ith item in the list; the same object may appear as more than one item in the list. The interpretation R^I of a relation symbol R of arity n in an

interpretation I is a set of ordered n-tuples of elements in the universe of I. If R^I is specified by giving a particular predicate for R, then which variables of the predicate belong with which places in the lists must also be specified. An example shows how:

$R(x, y, w)$: w is between x and y.

Class symbols are included as the relation symbols of arity 1, by taking a list

$\langle a \rangle$

of length 1 to be the same thing as its unique item a.

There can also be relation symbols of arity 0 if it is decided that there is exactly one list $\langle\ \rangle$ of length 0. So the interpretation p^I of a 0-ary relation symbol p is either the empty set (call it Falsehood) or else the set whose one element is $\langle\ \rangle$ (call this set Truth). All this makes good sense set-theoretically. What matters here, however, is the outcome: relation symbols of arity 0 are called *propositional symbols*, and they are always interpreted as Truth or as Falsehood. A sentence which contains neither '=', quantifiers nor any nonlogical constants except propositional symbols is called a *propositional sentence*. Propositional logic is about propositional sentences.

The language can be extended in another way by introducing nonlogical symbols called *n-ary function symbols*, where n is a positive integer. The interpretation F^I of such a symbol F is a function which assigns an element of I to each ordered n-tuple of elements of I. (Again, there is a way of regarding individual constants as 0-ary function symbols, but the details can be skipped here.)

The new symbols require some more adjustments to the grammar. The clause for *terms* becomes:

- Every variable is a term.
- Every individual constant is a term.
- If F is a function symbol of arity n, and $\alpha_1, \ldots, \alpha_n$ are terms, then '$F(\alpha_1, \ldots, \alpha_n)$' is a term.

The definition of *atomic formula* becomes:

- Every expression of the form '$(\alpha = \beta)$', where α and β are terms, is an atomic formula of L.
- Every propositional symbol is an atomic formula.
- If R is any relation symbol of positive arity n and $\alpha_1, \ldots, \alpha_n$ are terms, then '$R(\alpha_1, \ldots, \alpha_n)$' is an atomic formula.
- '$\perp$' is an atomic formula.

The semantic rules are the obvious adjustments of those in the previous section.

Some notation from section 1.5 can be usefully extended. If ϕ is a formula and α is a term,

$\phi(\alpha/x)$

represents the formula obtained from ϕ by replacing all free occurrences of x by α. As in section 1.5, to avoid clash of variables, the bound variables in ϕ may need to be changed first, so that they do not bind any variables in α.

1.8. Proof Calculi

First-order logic has a range of proof calculi. With a very few exceptions, all these proof calculi apply to all first-order languages. So, for the rest of this section assume that L is a particular first-order language of signature σ.

The first proof calculi to be discovered were the *Hilbert-style* calculi, where one reaches a conclusion by applying deduction rules to axioms. An example is described later in this section. These calculi tend to be very *ad hoc* in their axioms, and maddeningly wayward if one is looking for proofs in them. However, they have their supporters, e.g., modal logicians who need a first-order base to which further axioms can be added.

In 1934, Gentzen (1969) invented two other styles of calculus. One was the *natural deduction calculus* (independently proposed by Jaśkowski slightly earlier). An intuitionistic natural deduction calculus is given in chapter 11, which, as noted there, can be extended to make a calculus for classical first-order logic by the addition of a rule for double-negation elimination. Gentzen's second invention was the *sequent calculus*, which could be regarded as a Hilbert-style calculus for deriving finite sequents instead of formulas. With this subtle adjustment, nearly all of the arbitrariness of Hilbert-style systems falls away, and it is even possible to convert each sequent calculus proof into a sequent calculus proof in a very simple form called a *cut-free proof*. The popular *tableau* or *truth-tree* proofs are really cut-free sequent proofs turned upside down. A proof of a sequent in any of the four kinds of calculi – Hilbert-style, natural deduction, sequent calculus, tableaux – can be mechanically converted to a proof of the same sequent in any of the other calculi; see Sundholm (1983) for a survey.

The *resolution calculus* also deserves a mention. This calculus works very fast on computers, but its proofs are almost impossible for a normal human being to make any sense of, and it requires the sentences to be converted to a normal form (not quite the one in section 1.10 below) before the calculation starts; see, for example, Gallier (1986).

To sketch a Hilbert-style calculus, called $\mathcal{H}$, first define the class of *axioms* of $\mathcal{H}$. This is the set of all formulas of the language L which have any of the following forms:

H1 $\phi \supset (\psi \supset \phi)$

H2 $(\phi \supset \psi) \supset ((\phi \supset (\psi \supset \chi)) \supset (\psi \supset \chi))$

H3 $(\sim \phi \supset \psi) \supset ((\sim \phi \supset \sim \psi) \supset \phi)$

H4 $((\phi \supset \bot) \supset \bot) \supset \phi$

H5 $\phi \supset (\psi \supset (\phi \,\&\, \psi))$

H6 $(\phi \,\&\, \psi) \supset \phi$, $(\phi \,\&\, \psi) \supset \psi$

H7 $\phi \supset (\phi \vee \psi)$, $\psi \supset (\phi \vee \psi)$

H8 $(\phi \supset \chi) \supset ((\psi \supset \chi) \supset ((\phi \vee \psi) \supset \chi))$

H9 $(\phi \supset \psi) \supset ((\psi \supset \phi) \supset (\phi \equiv \psi))$

H10 $(\phi \equiv \psi) \supset (\phi \supset \psi)$, $(\phi \equiv \psi) \supset (\psi \supset \phi)$

H11 $\phi(\alpha/x) \supset \exists x\phi$ (α any term)

H12 $\forall x\phi \supset \phi(\alpha/x)$ (α any term)

H13 $x = x$

H14 $x = y \supset (\phi \supset \phi(y/x))$

A *derivation* (or *formal proof*) in $\mathcal{H}$ is defined to be a finite sequence

$$(\langle \phi_1, m_1 \rangle, \ldots, \langle \phi_n, m_n \rangle)$$

such that $n \geq 1$, and for each i $(1 \leq i \leq n)$ one of the five following conditions holds. (Clauses (c)–(e) are known as the *derivation rules* of $\mathcal{H}$.)

(a) $m_i = 1$ and ϕ_i is an axiom.
(b) $m_i = 2$ and ϕ_i is any formula of L.
(c) $m_i = 3$ and there are j and k in $\{1, \ldots, i-1\}$ such that ϕ_k is $\phi_j \to \phi_i$.
(d) $m_i = 4$ and there is j $(1 \leq j < i)$ such that ϕ_j has the form $\psi \to \chi$, x is a variable not occurring free in ψ, and ϕ_i is $\psi \to \forall x\chi$.
(e) $m_i = 5$ and there is j $(1 \leq j < i)$ such that ϕ_j has the form $\psi \to \chi$, x is a variable not occurring free in χ, and ϕ_i is $\exists x\psi \to \chi$.

The *premises* of this derivation are the formulas ϕ_i for which $m_i = 2$. Its *conclusion* is ϕ_n. We say that ψ is *derivable in* $\mathcal{H}$ from a set T of formulas, in symbols

$$T \vdash_{\mathcal{H}} \psi$$

if there exists a derivation whose conclusion is ψ and all of whose premises are in T.

Proofs are usually written vertically rather than horizontally. For example here is a proof of $(\phi \supset \phi)$, where ϕ is any formula:

(1) $(\phi \supset (\phi \supset \phi)) \supset ((\phi \supset ((\phi \supset \phi) \supset \phi)) \supset (\phi \supset \phi))$ [Axiom H2]

(2) $\phi \supset (\phi \supset \phi)$ [Axiom H1]

(3) $(\phi \supset ((\phi \supset \phi) \supset \phi)) \supset (\phi \supset \phi)$ [Rule (c) from (1), (2)]

(4) $\phi \supset ((\phi \supset \phi) \supset \phi)$ [Axiom H1]

(5) $\phi \supset \phi$ [Rule (c) from (3), (4)]

To save the labor of writing this argument every time a result of the form $\phi \supset \phi$ is needed, this result can be quoted as a lemma in further proofs. Thus $\sim\bot$ can be proved as follows:

(1) $\sim\bot\supset\sim\bot$ [Lemma]

(2) $(\sim\bot\supset\sim\bot)\supset((\sim\bot\supset\bot)\supset(\sim\bot\supset\sim\bot))$ [Axiom H1]

(3) $(\sim\bot\supset\bot)\supset(\sim\bot\supset\sim\bot)$ [Rule (c) from (1), (2)]

(4) $((\sim\bot\supset\bot)\supset(\sim\bot\supset\sim\bot))\supset(((\sim\bot\supset\bot)\supset((\sim\bot\supset\sim\bot)\supset\bot))\supset((\sim\bot\supset\bot)\supset\bot))$ [Axiom H2]

(5) $((\sim\bot\supset\bot)\supset((\sim\bot\supset\sim\bot)\supset\bot))\supset((\sim\bot\supset\bot)\supset\bot)$ [Rule (c) from (3), (4)]

(6) $(\sim\bot\supset\bot)\supset((\sim\bot\supset\sim\bot)\supset\bot)$ [Axiom H3]

(7) $(\sim\bot\supset\bot)\supset\bot$ [Rule (c) from (5), (6)]

(8) $((\sim\bot\supset\bot)\supset\bot)\supset\sim\bot$ [Axiom H4]

(9) $\sim\bot$ [Rule (c) from (7), (8)]

Then this result can be quoted in turn as a lemma in a proof of $(\phi\supset\bot)\supset\sim\phi$, and so on.

A theory T is *$\mathcal{H}$-inconsistent* if there is some formula ϕ such that $(\phi\,\&\sim\phi)$ is derivable from T in $\mathcal{H}$. If the language L contains $\bot$, then it can be shown that this is equivalent to saying that $\bot$ is derivable from T in $\mathcal{H}$. T is *$\mathcal{H}$-consistent* if it is not $\mathcal{H}$-inconsistent. $\mathcal{H}$-inconsistency is one example of *syntactic inconsistency*; other proof calculi give other examples.

1.9. Correctness and Completeness

Theorem 1.3 (*Correctness Theorem for $\mathcal{H}$*) Suppose $\phi_1, \ldots, \phi_n$ and ψ are sentences. If ϕ is derivable in $\mathcal{H}$ from $\phi_1, \ldots, \phi_n$, then the sequent

$$\phi_1, \ldots, \phi_n \vdash \psi$$

is valid.

Proof sketch This is proved by induction on the length of the shortest derivation of ψ from $\phi_1, \ldots, \phi_n$. Unfortunately, the formulas in the derivation need not be sentences. So for the induction hypothesis something a little more general needs to be proved:

Suppose $\phi_1, \ldots, \phi_n$ are sentences and ψ is a formula whose free variables are all among $x_1, \ldots, x_m$. If ψ is derivable in $\mathcal{H}$ from $\phi_1, \ldots, \phi_n$, then the sequent

$\phi_1, \ldots, \phi_n \vdash \forall x_1 \cdots \forall x_m \psi$

is valid.

The argument splits into cases according to the last derivation rule used in the proof. Suppose, for example, that this was the rule numbered (5) above, and ψ is the formula $\exists y \theta \supset \chi$ where y is not free in χ. Then, from the induction hypothesis, the sequent

$\phi_1, \ldots, \phi_n \vdash \forall x_1 \cdots \forall x_n \forall y(\theta \supset \chi)$

is valid. Using the fact that y is not free in χ, it can be checked that the sequent

$\forall x_1 \cdots \forall x_n \forall y(\theta \supset \chi) \vdash \forall x_1 \cdots \forall x_n(\exists y \theta \supset \chi)$

is valid. By this and the induction hypothesis, the sequent

$\phi_1, \ldots, \phi_n \vdash \forall x_1 \cdots \forall x_n(\exists y \theta \supset \chi)$

is valid as required. QED

Now, the completeness question:

Theorem 1.4 (*Completeness Theorem for* $\mathcal{H}$) Suppose that T is a theory and ψ is a sentence such that the sequent

$T \vdash \psi$

is valid. Then ψ is derivable from T in $\mathcal{H}$.

In fact one proves the special case of the Completeness Theorem where ψ is $\perp$; in other words

If T is a theory with no models, then $T \vdash_{\mathcal{H}} \perp$.

This is as good as proving the whole theorem, since the sequent

$T \cup \{\sim \psi\} \vdash \perp$

is equivalent to '$T \vdash \psi$' both semantically and in terms of derivability in $\mathcal{H}$.

Here, the Completeness Theorem is proved by showing that if T is any $\mathcal{H}$-consistent theory then T has a model. A technical lemma about $\mathcal{H}$ is needed along the way:

Lemma 1.5 Suppose c is a constant which occurs nowhere in the formula ϕ, the theory T or the sentence ψ. If

$$T \vdash_{\mathcal{H}} \phi(c/x) \supset \psi$$

then

$$T \vdash_{\mathcal{H}} \exists x\phi \supset \psi$$

Proof sketch of the Completeness Theorem This is known as a *Henkin-style* proof because of three features: the constants added as witnesses, the construction of a maximal consistent theory, and the way that a model is built using sets of terms as elements. The proof uses a small amount of set theory, chiefly infinite cardinals and ordinals. [See chapter 3.]

Assume a $\mathcal{H}$-consistent theory T in the language L. Let κ be the number of formulas of L; κ is always infinite. Expand the language L to a first-order language L^+ by adding to the signature a set of κ new individual constants; these new constants are called *witnesses*. List the sentences of L^+ as $(\phi_i : i < \kappa)$. Now define for each $i < \kappa$ a theory T_i, so that

$$T = T_0 \subseteq T_1 \subseteq \cdots$$

and each T_i is $\mathcal{H}$-consistent. To start the process, put $T_0 = T$. When i is a limit ordinal, take T_i to be the union of the T_j with $j < i$; this theory is $\mathcal{H}$-consistent since any inconsistency would have a proof using finitely many sentences, all of which would lie in some T_j with $j < i$.

The important choice is where i is a successor ordinal, say $i = j + 1$. If $T_j \cup \{\phi_j\}$ is not $\mathcal{H}$-consistent, take T_{j+1} to be T_j. Otherwise, put $T_j' = T_j \cup \{\phi_j\}$. Then if ϕ_j is of the form $\exists x\psi$, choose a witness c that appears nowhere in any sentence of T_j', and put $T_{j+1} = T_j' \cup \{\psi(c/x)\}$; otherwise put $T_{j+1} = T_j'$. By Lemma 1.5, T_{j+1} is $\mathcal{H}$-consistent in all these cases.

Write T^+ for the union of all the theories T_i. It has the property that if ϕ_j is any sentence of L^+ for which $T^+ \cup \{\phi_j\}$ is $\mathcal{H}$-consistent, then $T_j \cup \{\phi_j\}$ was already $\mathcal{H}$-consistent and so ϕ_j is in T^+ by construction. (As noted, T^+ is *maximal consistent*.) Moreover if T^+ contains a sentence ϕ_j of the form $\exists x\psi$, then by construction it also contains $\psi(c/x)$ for some witness c.

Two witnesses c and d are *equivalent* if the sentence '$c = d$' is in T^+. Now if '$c = d$' and '$d = e$' are both in T^+, then (appealing to the axioms and rules of $\mathcal{H}$) the theory $T^+ \cup \{c = e\}$ is $\mathcal{H}$-consistent, and so '$c = e$' is also in T^+. This and similar arguments show that 'equivalence' is an equivalence relation on the set of witnesses. Now build a structure A^+ whose universe is the set of equivalence classes $c^\sim$ of witnesses c. For example, if P is a 2-ary relation symbol in the signature, then take P^{A+} to be the set of all ordered pairs $\langle c^\sim, d^\sim \rangle$ such that the sentence '$P(c, d)$' is in T^+. There are a number of details to be checked, but the outcome is that A^+ is a model of T^+. Now, stripping the witnesses out of the signature gives a structure A whose signature is that of L, and A is a model of all the sentences of L that are in T^+. In particular, A is a model of T, as required. (Note that A has at most κ elements, since there were only κ witnesses.) QED

1.10. Metatheory of First-Order Logic

The *metatheory* of a logic consists of those things that one can say *about* the logic, rather than in it. All the numbered theorems of this chapter are examples. The metatheory of first-order logic is vast. Here are a few high points, beginning with some consequences of the Completeness Theorem for $\mathcal{H}$.

Theorem 1.6 (*Compactness Theorem*) Suppose T is a first-order theory, ψ is a first-order sentence and T entails ψ. Then there is a finite subset U of T such that U entails ψ.

Proof If T entails ψ then the sequent

$$T \vdash \psi$$

is valid, and so by the completeness of the proof calculus $\mathcal{H}$, the sequent has a formal proof. Let U be the set of sentences in T which are used in this proof. Since the proof is a finite object, U is a finite set. But the proof is also a proof of the sequent

$$U \vdash \psi$$

So by the correctness of $\mathcal{H}$, U entails ψ. QED

Corollary 1.7 Suppose T is a first-order theory and every finite subset of T has a model. Then T has a model.

Proof Working backwards, it is enough to prove that if T has no model then some finite subset of T has no model. If T has no model then T entails $\perp$, since $\perp$ has no models. So by the Compactness Theorem, some finite subset U of T entails $\perp$. But this implies that U has no model. QED

The next result is the weakest of a family of theorems known as the *Downward Löwenheim–Skolem Theorem*.

Theorem 1.8 Suppose L is a first-order language with at most countably many formulas, and let T be a consistent theory in L. Then T has a model with at most countably many elements.

Proof Assuming T is semantically consistent, it is $\mathcal{H}$-consistent by the correctness of $\mathcal{H}$. So the sketch proof of the Completeness Theorem in section 1.9 constructs a model A of T. By the last sentence of section 1.9, A has at most countably many elements. QED

There is also an *Upward Löwenheim–Skolem Theorem*, which says that every first-order theory with infinite models has arbitrarily large models.

A *basic conjunction* is a formula of the form

$(\phi_1 \,\&\, \cdots \,\&\, \phi_m)$

where each ϕ_i is either an atomic formula or an atomic formula preceded by $\sim$. (Note that $m = 1$ is allowed, so that a single atomic formula, with or without $\sim$, counts as a basic conjunction.) A formula is in *disjunctive normal form* if it has the form

$(\psi_1 \vee \cdots \vee \psi_n)$

where each ψ_j is a basic conjunction. (Again, $n = 1$ is allowed, so that a basic conjunction counts as being in disjunctive normal form.)

A first-order formula is said to be *prenex* if it consists of a string of quantifiers followed by a formula with no quantifiers in it. (The string of quantifiers may be empty, so that a formula with no quantifiers counts as being prenex.)

A formula is in *normal form* if it is prenex and the part after the quantifiers is in disjunctive normal form.

Theorem 1.9 (*Normal Form Theorem*) Every first-order formula ϕ is equivalent to a first-order formula ψ of the same signature as ϕ, which has the same free variables as ϕ and is in normal form.

The next theorem, Lyndon's Interpolation Theorem, deserves to be better known. Among other things, it is the first-order form of some laws which were widely known to logicians of earlier centuries as the Laws of Distribution (Hodges, 1998). It is stated here for sentences in normal form; by Theorem 1.9, this implies a theorem about all first-order sentences.

Suppose ϕ is a first-order sentence in normal form. An occurrence of a relation symbol in ϕ is called *positive* if it has no '$\sim$' immediately in front of it, and *negative* if it has.

Theorem 1.10 (*Lyndon's Interpolation Theorem*) Suppose ϕ and ψ are first-order sentences in normal form, and ϕ entails ψ. Then there is a first-order sentence θ in normal form, such that

- ϕ entails θ and θ entails ψ
- every relation symbol which has a positive occurrence in θ has positive occurrences in both ϕ and ψ, and
- every relation symbol which has a negative occurrence in θ has negative occurrences in both ϕ and ψ.

Lyndon's theorem can be proved either by analyzing proofs of the sequent '$\phi \vdash \psi$', or by a set-theoretic argument using models of ϕ and ψ. Both arguments are too complicated to give here.

An important corollary of Lyndon's Interpolation Theorem is Craig's Interpolation Theorem, which was proved a few years before Lyndon's.

Corollary 1.11 (*Craig's Interpolation Theorem*) Suppose ϕ and ψ are first-order sentences, and ϕ entails ψ. Then there is a first-order sentence θ such that

- ϕ entails θ and θ entails ψ
- every relation symbol that occurs in θ occurs both in ϕ and in ψ.

Craig's Interpolation Theorem in turn implies Beth's Definability Theorem, which was proved earlier still. But all these theorems are from the 1950s, perhaps the last great age of elementary metatheory.

Corollary 1.12 (*Beth's Definability Theorem*) Suppose ϕ is a first-order sentence in which a relation symbol R of arity n occurs, and suppose also that there are not two models I and J of ϕ which are identical except that R^I is different from R^J. Then ϕ entails some first-order sentence of the form

$$(\forall x_1) \cdots (\forall x_n)\ (\psi \equiv R(x_1, \ldots, x_n))$$

where ψ is a formula in which R never occurs.

Finally, note a metatheorem of a different kind, to contrast with Theorem 1.2 above: a form of *Church's Theorem on the Undecidability of First-Order Logic*.

Theorem 1.13 Suppose L is a first-order language whose signature contains at least one n-ary relation symbol with $n > 1$. Then L is not decidable.

A reference for all the metatheorems in this section except Theorems 1.9 and 1.13 is Hodges (1997). Theorem 1.9 is proved in both Kleene (1952, pp. 134f, 167) and Ebbinghaus et al. (1984, p. 126), together with a wealth of other mathematical information about first-order languages. Boolos and Jeffrey (1974) contains a proof of the undecidability of first-order logic (though to reach Theorem 1.13 above from its results, some coding devices are needed).

Suggested further reading

There are many places where the subjects of this chapter can be pursued to a deeper level. Of those mentioned already in this chapter, Boolos and Jeffrey (1974) is a clear introductory text aimed at philosophers, while Hodges (1983) is a survey with an eye on philosophical issues. Ebbinghaus et al. (1984) is highly recommended for those prepared to face some nontrivial mathematics. Of older books, Church (1956) is still valuable for its philosophical and historical remarks, and Tarski (1983) is outstanding for its clear treatment of fundamental questions.

References

Barwise, J. and Etchemendy, J. 1999: *Tarski's World 5.1*, available with *Language, Proof and Logic*, (Seven Bridges Press, New York and London).

Boolos, G. S. and Jeffrey, R. C. 1974: *Computability and Logic*, (Cambridge University Press, Cambridge).

Church, A. 1956: *Introduction to Mathematical Logic*, (Princeton University Press, Princeton, NJ).

Ebbinghaus, H.-D., Flum, J. and Thomas, W. 1984: *Mathematical Logic*, (Springer, New York).

Gallier, J. H. 1986: *Logic for Computer Science: Foundations of Automated Theorem Proving*, (Harper and Row, New York).

Gentzen, G. 1969: "Untersuchungen über das Logische Schliessen," translated in *The Collected Papers of Gerhard Gentzen*, M. Szabo, ed., (North-Holland, Amsterdam), 68–131.

Hilbert, D. and Ackermann, W. 1950: *Grundzuge der Theoretische Logik* (1928); English translation *Principles of Mathematical Logic*, R. E. Luce, ed., (Chelsea Publishing Company, New York).

Hintikka, J. 1973: *Logic, Language-Games and Information*, (Oxford University Press, Oxford).

Hodges, W. 1983: "Elementary Predicate Logic," in *Handbook of Philosophical Logic – Vol. I: Elements of Classical Logic*, D. M. Gabbay and F. Guenthner, eds., (D. Reidel, Dordrecht), 1–131.

Hodges, W. 1997: *A Shorter Model Theory*, (Cambridge University Press, Cambridge).

Hodges, W. 1998: "The Laws of Distribution for Syllogism," *Notre Dame Journal of Formal Logic*, 39, 221–30.

Kleene, S. C. 1952: *Introduction to Metamathematics*, (North-Holland, Amsterdam).

Sundholm, G. 1983: "Systems of Deduction," in *Handbook of Philosophical Logic – Vol. I: Elements of Classical Logic*, D. M. Gabbay and F. Guenthner, eds., (D. Reidel, Dordrecht), 133–88.

Tarski, A. 1983: "The Concept of Truth in Formalized Languages," in *Logic, Semantics, Metamathematics, Papers from 1923 to 1938*, J. H. Woodger, tr., and J. Corcoran, ed., (Hackett Publishing Co., Indianapolis), 152–278.

Tarski, A. and Vaught, R. 1957: "Arithmetical Extensions of Relational Systems," *Compositio Math.*, 13, 81–102.

Chapter 8

Resolution In First-Order Logic

8.1 Introduction

In this chapter, the resolution method presented in Chapter 4 for propositional logic is extended to first-order logic without equality. The point of departure is the Skolem-Herbrand-Gödel theorem (theorem 7.6.1). Recall that this theorem says that a sentence A is unsatisfiable iff some compound instance C of the Skolem form B of A is unsatisfiable. This suggests the following procedure for checking unsatisfiability:

Enumerate the compound instances of B systematically one by one, testing each time a new compound instance C is generated, whether C is unsatisfiable.

If we are considering a first-order language without equality, there are algorithms for testing whether a quantifier-free formula is valid (for example, the *search* procedure) and, if B is unsatisfiable, this will be eventually discovered. Indeed, the *search* procedure halts for every compound instance, and for some compound instance C, $\neg C$ will be found valid.

If the logic contains equality, the situation is more complex. This is because the *search* procedure does not necessarily halt for quantifier-free formulae that are not valid. Hence, it is possible that the procedure for checking unsatisfiability will run forever even if B is unsatisfiable, because the *search* procedure can run forever for some compound instance that is not unsatisfiable. We can fix the problem as follows:

Interleave the generation of compound instances with the process of checking whether a compound instance is unsatisfiable, proceeding by rounds. A round consists in running the *search* procedure a fixed number of steps for each compound instance being tested, and then generating a new compound instance. The process is repeated with the new set of compound instances. In this fashion, at the end of each round, we have made progress in checking the unsatisfiability of all the activated compound instances, but we have also made progress in the number of compound instances being considered.

Needless to say, such a method is horribly inefficient. Actually, it is possible to design an algorithm for testing the unsatisfiability of a quantifier-free formula with equality by extending the congruence closure method of Oppen and Nelson (Nelson and Oppen, 1980). This extension is presented in Chapter 10.

In the case of a language without equality, any algorithm for deciding the unsatisfiability of a quantifier-free formula can be used. However, the choice of such an algorithm is constrained by the need for efficiency. Several methods have been proposed. The *search* procedure can be used, but this is probably the least efficient choice. If the compound instances C are in CNF, the resolution method of Chapter 4 is a possible candidate. Another method called the method of *matings* has also been proposed by Andrews (Andrews, 1981).

In this chapter, we are going to explore the method using resolution. Such a method is called *ground resolution*, because it is applied to quantifier-free clauses with no variables.

From the point of view of efficiency, there is an undesirable feature, which is the need for systematically generating compound instances. Unfortunately, there is no hope that the process of finding a refutation can be purely mechanical. Indeed, by Church's theorem (mentioned in the remark after the proof of theorem 5.5.1), there is no algorithm for deciding the unsatisfiability (validity) of a formula.

There is a way of avoiding the systematic generation of compound instances due to J. A. Robinson (Robinson, 1965). The idea is not to generate compound instances at all, but instead to generalize the resolution method so that it applies directly to the clauses in B, as opposed to the (ground) clauses in the compound instance C. The completeness of this method was shown by Robinson. The method is to show that every ground refutation can be lifted to a refutation operating on the original clauses, as opposed to the closed (or ground) substitution instances. In order to perform this lifting operation the process of *unification* must be introduced. We shall define these concepts in the following sections.

It is also possible to extend the resolution method to first-order languages with equality using the *paramodulation* method due to Robinson and Wos (Robinson and Wos, 1969, Loveland, 1978), but the completeness proof is

rather delicate. Hence, we will restrict our attention to first-order languages without equality, and refer the interested reader to Loveland, 1978, for an exposition of paramodulation.

As in Chapter 4, the resolution method for first-order logic (without equality) is applied to special conjunctions of formulae called clauses. Hence, it is necessary to convert a sentence A into a sentence A' in clause form, such that A is unsatisfiable iff A' is unsatisfiable. The conversion process is defined below.

8.2 Formulae in Clause Form

First, we define the notion of a formula in clause form.

Definition 8.2.1 As in the propositional case, a *literal* is either an atomic formula B, or the negation $\neg B$ of an atomic formula. Given a literal L, its *conjugate* $\overline{L}$ is defined such that, if $L = B$ then $\overline{L} = \neg B$, else if $L = \neg B$ then $\overline{L} = B$. A sentence A is in *clause form* iff it is a conjunction of (prenex) sentences of the form $\forall x_1 ... \forall x_m C$, where C is a disjunction of literals, and the sets of bound variables $\{x_1, ..., x_m\}$ are disjoint for any two distinct clauses. Each sentence $\forall x_1 ... \forall x_m C$ is called a *clause*. If a clause in A has no quantifiers and does not contain any variables, we say that it is a *ground clause*.

For simplicity of notation, the universal quantifiers are usually omitted in writing clauses.

Lemma 8.2.1 For every (rectified) sentence A, a sentence B' in clause form such that A is valid iff B' is unsatisfiable can be constructed.

Proof: Given a sentence A, first $B = \neg A$ is converted to B_1 in NNF using lemma 6.4.1. Then B_1 is converted to B_2 in Skolem normal form using the method of definition 7.6.2. Next, by lemma 7.2.1, B_2 is converted to B_3 in prenex form. Next, the matrix of B_3 is converted to conjunctive normal form using theorem 3.4.2, yielding B_4. In this step, theorem 3.4.2 is applicable because the matrix is quantifier free. Finally, the quantifiers are distributed over each conjunct using the valid formula $\forall x(A \wedge B) \equiv \forall x A \wedge \forall x B$, and renamed apart using lemma 5.3.4.

Let the resulting sentence be called B'. The resulting formula B' is a conjunction of clauses.

By lemma 6.4.1, B is unsatisfiable iff B_1 is. By lemma 7.6.3, B_1 is unsatisfiable iff B_2 is. By lemma 7.2.1, B_2 is unsatisfiable iff B_3 is. By theorem 3.4.2 and lemma 5.3.7, B_3 is unsatisfiable iff B_4 is. Finally, by lemma 5.3.4 and lemma 5.3.7, B_4 is unsatisfiable iff B' is. Hence, B is unsatisfiable iff B' is. Since A is valid iff $B = \neg A$ is unsatisfiable, then A is valid iff B' is unsatisfiable. □

EXAMPLE 8.2.1

Let

$$A = \neg\exists y\forall z(P(z,y) \equiv \neg\exists x(P(z,x) \wedge P(x,z))).$$

First, we negate A and eliminate $\equiv$. We obtain the sentence

$$\exists y\forall z[(\neg P(z,y) \vee \neg\exists x(P(z,x) \wedge P(x,z))) \wedge (\exists x(P(z,x) \wedge P(x,z)) \vee P(z,y))].$$

Next, we put in this formula in NNF:

$$\exists y\forall z[(\neg P(z,y) \vee \forall x(\neg P(z,x) \vee \neg P(x,z))) \wedge (\exists x(P(z,x) \wedge P(x,z)) \vee P(z,y))].$$

Next, we eliminate existential quantifiers, by the introduction of Skolem symbols:

$$\forall z[(\neg P(z,a) \vee \forall x(\neg P(z,x) \vee \neg P(x,z))) \wedge ((P(z,f(z)) \wedge P(f(z),z)) \vee P(z,a))].$$

We now put in prenex form:

$$\forall z\forall x[(\neg P(z,a) \vee (\neg P(z,x) \vee \neg P(x,z))) \wedge ((P(z,f(z)) \wedge P(f(z),z)) \vee P(z,a))].$$

We put in CNF by distributing $\wedge$ over $\vee$:

$$\forall z\forall x[(\neg P(z,a) \vee \neg P(z,x) \vee \neg P(x,z)) \wedge (P(z,f(z)) \vee P(z,a)) \wedge (P(f(z),z)) \vee P(z,a))].$$

Omitting universal quantifiers, we have the following three clauses:

$$C_1 = (\neg P(z_1,a) \vee \neg P(z_1,x) \vee \neg P(x,z_1)),$$
$$C_2 = (P(z_2,f(z_2)) \vee P(z_2,a)) \text{ and}$$
$$C_3 = (P(f(z_3),z_3) \vee P(z_3,a)).$$

We will now show that we can prove that $B = \neg A$ is unsatisfiable, by instantiating C_1, C_2, C_3 to ground clauses and use the resolution method of Chapter 4.

8.3 Ground Resolution

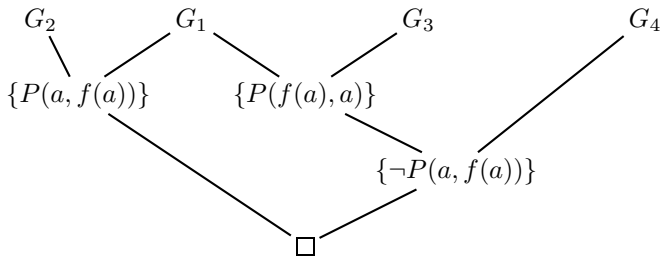
The *ground resolution method* is the resolution method applied to sets of ground clauses.

EXAMPLE 8.3.1

Consider the following ground clauses obtained by substitution from C_1, C_2 and C_3:

$G_1 = (\neg P(a,a))$ (from C_1, substituting a for x and z_1)
$G_2 = (P(a,f(a)) \vee P(a,a))$ (from C_2, substituting a for z_2)
$G_3 = (P(f(a),a)) \vee P(a,a))$ (from C_3, substituting a for z_3).
$G_4 = (\neg P(f(a),a) \vee \neg P(a,f(a)))$ (from C_1, substituting $f(a)$ for z_1 and a for x).

The following is a refutation by (ground) resolution of the set of ground clauses G_1, G_2, G_3, G_4.



We have the following useful result.

Lemma 8.3.1 (Completeness of ground resolution) The ground resolution method is complete for ground clauses.

Proof: Observe that the systems G' and $GCNF'$ are complete for quantifier-free formulae of a first-order language without equality. Hence, by theorem 4.3.1, the resolution method is also complete for sets of ground clauses. □

However, note that this is not the case for quantifier-free formulae with equality, due to the need for equality axioms and for inessential cuts, in order to retain completeness.

Since we have shown that a conjunction of ground instances of the clauses C_1, C_2, C_3 of example 8.2.1 is unsatisfiable, by the Skolem-Herbrand-Gödel theorem, the sentence A of example 8.2.1 is valid.

Summarizing the above, we have a method for finding whether a sentence B is unsatisfiable known as *ground resolution*. This method consists in converting the sentence B into a set of clauses B', instantiating these clauses to ground clauses, and applying the ground resolution method.

By the completeness of resolution for propositional logic (theorem 4.3.1), and the Skolem-Herbrand-Gödel theorem (actually the corollary to theorem 7.6.1 suffices, since the clauses are in CNF, and so in NNF), this method is complete.

However, we were lucky to find so easily the ground clauses G_1, G_2, G_3 and G_4. In general, all one can do is enumerate ground instances one by one, testing for the unsatisfiabiliy of the current set of ground clauses each time. This can be a very costly process, both in terms of time and space.

8.4 Unification and the Unification Algorithm

The fundamental concept that allows the lifting of the ground resolution method to the first-order case is that of a most general unifier.

8.4.1 Unifiers and Most General Unifiers

We have already mentioned that Robinson has generalized ground resolution to arbitrary clauses, so that the systematic generation of ground clauses is unnecessary.

The new ingredient in this new form of resolution is that in forming the resolvent, one is allowed to apply substitutions to the parent clauses.

For example, to obtain $\{P(a, f(a))\}$ from

$$C_1 = (\neg P(z_1, a) \vee \neg P(z_1, x) \vee \neg P(x, z_1)) \quad \text{and}$$
$$C_2 = (P(z_2, f(z_2)) \vee P(z_2, a)),$$

first we substitute a for z_1, a for x, and a for z_2, obtaining

$$G_1 = (\neg P(a, a)) \quad \text{and} \quad G_2 = (P(a, f(a)) \vee P(a, a)),$$

and then we resolve on the literal $P(a, a)$.

Note that the two sets of literals $\{P(z_1, a), P(z_1, x), P(x, z_1)\}$ and $\{P(z_2, a)\}$ obtained by dropping the negation sign in C_1 have been "unified" by the substitution $(a/x, a/z_1, a/z_2)$.

In general, given two clauses B and C whose variables are disjoint, given a substitution σ having as support the union of the sets of variables in B and C, if $\sigma(B)$ and $\sigma(C)$ contain a literal Q and its conjugate, there must be a subset $\{B_1, ..., B_m\}$ of the sets of literals of B, and a subset $\{\overline{C_1}, ..., \overline{C_n}\}$ of the set of literals in C such that

$$\sigma(B_1) = ... = \sigma(B_m) = \sigma(C_1) = ... = \sigma(C_n).$$

We say that σ is a *unifier* for the set of literals $\{B_1, ..., B_m, C_1, ..., C_n\}$. Robinson showed that there is an algorithm called the *unification algorithm*, for deciding whether a set of literals is unifiable, and if so, the algorithm yields what is called a *most general unifier* (Robinson, 1965). We will now explain these concepts in detail.

Definition 8.4.1 Given a substitution σ, let $D(\sigma) = \{x \mid \sigma(x) \neq x\}$ denote the *support* of σ, and let $I(\sigma) = \bigcup_{x \in D(\sigma)} FV(\sigma(x))$. Given two substitutions σ and θ, their *composition* denoted $\sigma \circ \theta$ is the substitution $\sigma \circ \widehat{\theta}$ (recall that $\widehat{\theta}$ is the unique homomorphic extension of θ). It is easily shown that the substitution $\sigma \circ \theta$ is the restriction of $\widehat{\sigma} \circ \widehat{\theta}$ to $\mathbf{V}$. If σ has support $\{x_1, ..., x_m\}$ and $\sigma(x_i) = s_i$ for $i = 1, ..., m$, we also denote the substitution σ by $(s_1/x_1, ..., s_m/x_m)$.

The notions of a unifier and a most general unifier are defined for arbitrary trees over a ranked alphabet (see Subsection 2.2.6). Since terms and atomic formulae have an obvious representation as trees (rigorously, since they are freely generated, we could define a bijection recursively), it is perfectly suitable to deal with trees, and in fact, this is intuitively more appealing due to the graphical nature of trees.

Definition 8.4.2 Given a ranked alphabet Σ, given any set $S = \{t_1, ..., t_n\}$ of finite Σ-trees, we say that a substitution σ is a *unifier of* S iff

$$\sigma(t_1) = ... = \sigma(t_n).$$

We say that a substitution σ is a *most general unifier* of S iff it is a unifier of S, the support of σ is a subset of the set of variables occurring in the set S, and for any other unifier σ' of S, there is a substitution θ such that

$$\sigma' = \sigma \circ \theta.$$

The tree $t = \sigma(t_1) = ... = \sigma(t_n)$ is called a *most common instance* of t_1,...,t_n.

EXAMPLE 8.4.1

(i) Let $t_1 = f(x, g(y))$ and $t_2 = f(g(u), g(z))$. The substitution $(g(u)/x, y/z)$ is a most general unifier yielding the most common instance $f(g(u), g(y))$.

(ii) However, $t_1 = f(x, g(y))$ and $t_2 = f(g(u), h(z))$ are not unifiable since this requires $g = h$.

(iii) A slightly more devious case of non unifiability is the following:

Let $t_1 = f(x, g(x), x)$ and $t_2 = f(g(u), g(g(z)), z)$. To unify these two trees, we must have $x = g(u) = z$. But we also need $g(x) = g(g(z))$, that is, $x = g(z)$. This implies $z = g(z)$, which is impossible for finite trees.

This last example suggest that unifying trees is similar to solving systems of equations by variable elimination, and there is indeed such an analogy. This analogy is explicated in Gorn, 1984. First, we show that we can reduce the problem of unifying any set of trees to the problem of unifying two trees.

Lemma 8.4.1 Let t_1,...,t_m be any m trees, and let $\#$ be a symbol of rank m not occurring in any of these trees. A substitution σ is a unifier for the set $\{t_1, ..., t_m\}$ iff σ is a unifier for the set $\{\#(t_1, ..., t_m), \#(t_1, ..., t_1)\}$.

Proof: Since a substitution σ is a homomorphism (see definition 7.5.3),

$$\sigma(\#(t_1, ..., t_m)) = \#(\sigma(t_1), ..., \sigma(t_m)) \quad \text{and}$$
$$\sigma(\#(t_1, ..., t_1)) = \#(\sigma(t_1), ..., \sigma(t_1)).$$

Hence,

$$\sigma(\#(t_1, ..., t_m)) = \sigma(\#(t_1, ..., t_1)) \quad \text{iff}$$
$$\#(\sigma(t_1), ..., \sigma(t_m)) = \#(\sigma(t_1), ..., \sigma(t_1)) \quad \text{iff}$$
$$\sigma(t_1) = \sigma(t_1),\ \sigma(t_2) = \sigma(t_1), ...,\ \sigma(t_m) = \sigma(t_1) \quad \text{iff}$$
$$\sigma(t_1) = ... = \sigma(t_m). \quad \square$$

Before showing that if a set of trees is unifiable then it has a most general unifier, we note that most general unifiers are essentially unique when they exist. Lemma 8.4.2 holds even if the support of mgu's is not a subset of $FV(S)$.

Lemma 8.4.2 If a set of trees S is unifiable and σ and θ are any two most general unifiers for S, then there exists a substitution ρ such that $\theta = \sigma \circ \rho$, ρ is a bijection between $I(\sigma) \cup (D(\theta) - D(\sigma))$ and $I(\theta) \cup (D(\sigma) - D(\theta))$, and $D(\rho) = I(\sigma) \cup (D(\theta) - D(\sigma))$ and $\rho(x)$ is a variable for every $x \in D(\rho)$.

Proof: First, note that a bijective substitution must be a bijective renaming of variables. Let $f|_A$ denote the restriction of a function f to A. If ρ is bijective, there is a substitution ρ' such that $(\rho \circ \rho')|_{D(\rho)} = Id$ and $(\rho' \circ \rho)|_{D(\rho')} = Id$. But then, if $\rho(x)$ is not a variable for some x in the support of ρ, $\rho(x)$ is a constant or a tree t of depth ≥ 1. Since $(\rho \circ \rho')|_{D(\rho)} = Id$, we have $\rho'(t) = x$. Since a substitution is a homomorphism, if t is a constant c, $\rho'(c) = c \neq x$, and otherwise $\rho'(t)$ has depth at least 1, and so $\rho'(t) \neq x$. Hence, $\rho(x)$ must be a variable for every x (and similarly for ρ'). A reasoning similar to the above also shows that for any two substitutions σ and ρ, if $\sigma = \sigma \circ \rho$, then ρ is the identity on $I(\sigma)$. But then, if both σ and θ are most general unifiers, there exist σ' and θ' such that $\theta = \sigma \circ \theta'$ and $\sigma = \theta \circ \sigma'$. Thus, $D(\sigma') = I(\theta) \cup (D(\sigma) - D(\theta))$, $D(\theta') = I(\sigma) \cup (D(\theta) - D(\sigma))$, $\theta = \theta \circ (\sigma' \circ \theta')$, and $\sigma = \sigma \circ (\theta' \circ \sigma')$. We claim that $(\sigma' \circ \theta')|_{D(\sigma')} = Id$, and $(\theta' \circ \sigma')|_{D(\theta')} = Id$. We prove that $(\sigma' \circ \theta')|_{D(\sigma')} = Id$, the other case being similar. For $x \in I(\theta)$, $\sigma' \circ \theta'(x) = x$ follows from above. For $x \in D(\sigma) - D(\theta)$, then $x = \theta(x) = \theta'(\sigma(x))$, and so, $\sigma(x) = y$, and $\theta'(y) = x$, for some variable y. Also, $\sigma(x) = y = \sigma'(\theta(x)) = \sigma'(x)$. Hence, $\sigma' \circ \theta'(x) = x$. Since $D(\theta')$ and $D(\sigma')$ are finite, θ' is a bijection between $D(\theta')$ and $D(\sigma')$. Letting $\rho = \theta'$, the lemma holds. $\square$

We shall now present a version of Robinson's unification algorithm.

8.4.2 The Unification Algorithm

In view of lemma 8.4.1, we restrict our attention to pairs of trees. The main idea of the unification algorithm is to find how two trees "disagree," and try

to force them to agree by substituting trees for variables, if possible. There are two types of disagreements:

(1) Fatal disagreements, which are of two kinds:

  (i) For some tree address u both in $dom(t_1)$ and $dom(t_2)$, the labels $t_1(u)$ and $t_2(u)$ are not variables and $t_1(u) \neq t_2(u)$. This is illustrated by case (ii) in example 8.4.1;

  (ii) For some tree address u in both $dom(t_1)$ and $dom(t_2)$, $t_1(u)$ is a variable say x, and the subtree t_2/u rooted at u in t_2 is not a variable and x occurs in t_2/u (or the symmetric case in which $t_2(u)$ is a variable and t_1/u isn't). This is illustrated in case (iii) of example 8.4.1.

(2) Repairable disagreements: For some tree address u both in $dom(t_1)$ and $dom(t_2)$, $t_1(u)$ is a variable and the subtree t_2/u rooted at u in t_2 does not contain the variable $t_1(u)$.

In case (1), unification is impossible (although if we allowed infinite trees, disagreements of type (1)(ii) could be fixed; see Gorn, 1984). In case (2), we force "local agreement" by substituting the subtree t_2/u for all occurrences of the variable x in both t_1 and t_2.

It is rather clear that we need a systematic method for finding disagreements in trees. Depending on the representation chosen for trees, the method will vary. In most presentations of unification, it is usually assumed that trees are represented as parenthesized expressions, and that the two strings are scanned from left to right until a disagreement is found. However, an actual method for doing so is usually not given explicitly. We believe that in order to give a clearer description of the unification algorithm, it is better to be more explicit about the method for finding disagreements, and that it is also better not to be tied to any string representation of trees. Hence, we will give a recursive algorithm inspired from J. A. Robinson's original algorithm, in which trees are defined in terms of tree domains (as in Section 2.2), and the disagreements are discovered by performing two parallel top-down traversals of the trees t_1 and t_2.

The type of traversal that we shall be using is a recursive traversal in which the root is visited first, and then, from left to right, the subtrees of the root are recursively visited (this kind of traversal is called a *preorder traversal*, see Knuth, 1968, Vol. 1). We define some useful functions on trees. (The reader is advised to review the definitions concerning trees given in Section 2.2.)

Definition 8.4.3 For any tree t, for any tree address $u \in dom(t)$:

$leaf(u) =$ *true* iff u is a leaf;

$variable(t(u)) =$ *true* iff $t(u)$ is a variable;

$left(u) =$ *if* $leaf(u)$ *then nil else* $u1$;

$right(ui) =$ *if* $u(i+1) \in dom(t)$ *then* $u(i+1)$ *else nil*.

We also assume that we have a function $dosubstitution(t, \sigma)$, where t is a tree and σ is a substitution.

Definition 8.4.4 (A unification algorithm) The formal parameters of the algorithm *unification* are the two input trees t_1 and t_2, an output flag indicating whether the two trees are unifiable or not ($unifiable$), and a most general unifier ($unifier$) (if it exists).

The main program *unification* calls the recursive procedure *unify*, which performs the unification recursively and needs procedure *test-and-substitute* to repair disagreements found, as in case (2) discussed above. The variables $tree1$ and $tree2$ denote trees (of type *tree*), and the variables *node*, *newnode* are tree addresses (of type $treereference$). The variable $unifier$ is used to build a most general unifier (if any), and the variable *newpair* is used to form a new substitution component (of the form (t/x), where t is a tree and x is a variable). The function *compose* is simply function composition, where $compose(unifier, newpair)$ is the result of composing $unifier$ and *newpair*, in this order. The variables $tree1$, $tree2$, and *node* are global variables to the procedure *unification*. Whenever a new disageement is resolved in *test-and-substitute*, we also apply the substitution *newpair* to $tree1$ and $tree2$ to remove the disagreement. This step is not really necessary, since at any time, $dosubstitution(t_1, unifier) = tree1$ and $dosubstitution(t_2, unifier) = tree2$, but it simplifies the algorithm.

Procedure to Unify Two Trees t_1 and t_2

```
procedure unification(t1, t2 : tree; var unifiable : boolean;
                                     var unifier : substitution);
  var node : treereference; tree1, tree2 : tree;

  procedure test-and-substitute(var node : treereference;
    var tree1, tree2 : tree;
    var unifier : substitution; var unifiable : boolean);
  var newpair : substitution;

  {This procedure tests whether the variable tree1(node) belongs
  to the subtree of tree2 rooted at node. If it does, the
  unification fails. Otherwise, a new substitution newpair
  consisting of the subtree tree2/node and the variable
  tree1(node) is formed, the current unifier is composed with
  newpair, and the new pair is added to the unifier.
  To simplify the algorithm, we also apply newpair to tree1
  and tree2 to remove the disagreement}

  begin
  {test whether the variable tree1(node) belongs to the
   subtree tree2/node, known in the literature as "occur check"}
```

```
        if tree1(node) ∈ tree2/node then
          unifiable := false
        else

          {create a new substitution pair consisting of the
          subtree tree2/node at address node, and the
          variable tree1(node) at node in tree1}

          newpair := ((tree2/node)/tree1(node));

          {compose the current partial unifier with
          the new pair newpair}

          unifier := compose(unifier, newpair);

          {updates the two trees so that they now agree on
          the subtrees at node}

          tree1 := dosubstitution(tree1, newpair);
          tree2 := dosubstitution(tree2, newpair)
        endif
      end test-and-substitute;

procedure unify(var node : treereference;
    var unifiable : boolean; var unifier : substitution);
var newnode : treereference;

{Procedure unify recursively unifies the subtree
of tree1 at node and the subtree of tree2 at node}

  begin
    if tree1(node) <> tree2(node) then
      {the labels of tree1(node) and tree2(node) disagree}
      if variable(tree1(node)) or variable(tree2(node)) then
        {one of the two labels is a variable}
        if variable(tree1(node)) then
          test-and-substitute(node, tree1, tree2, unifier, unifiable)
        else
          test-and-substitute(node, tree2, tree1, unifier, unifiable)
        endif
      else
        {the labels of tree1(node) and tree2(node)
        disagree and are not variables}
        unifiable := false
      endif
    endif;
```

```
    {At this point, if unifiable = true, the labels at node
    agree. We recursively unify the immediate subtrees of node
    in tree1 and tree2 from left to right, if node is not a leaf}

    if (left(node) <> nil) and unifiable then
      newnode := left(node);
      while (newnode <> nil) and unifiable do
        unify(newnode, unifiable, unifier);
        if unifiable then
          newnode := right(newnode)
        endif
      endwhile
    endif
  end unify;
```

Body of Procedure Unification

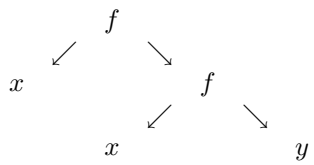
```
  begin
    tree1 := t1;
    tree2 := t2;
    unifiable := true;
    unifier := nil;        {empty unification}
    node := e;             {start from the root}
    unify(node, unifiable, unifier)
  end unification
```

Note that if successful, the algorithm could also return the tree *tree*1 (or *tree*2), which is a most common form of t_1 and t_2. As presented, the algorithm performs a single parallel traversal, but we also have the cost of the *occur check* in *test-and-substitute*, and the cost of the substitutions. Let us illustrate how the algorithm works.

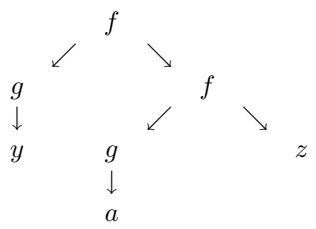
EXAMPLE 8.4.2

Let $t_1 = f(x, f(x, y))$ and $t_2 = f(g(y), f(g(a), z))$, which are represented as trees as follows:

Tree t_1

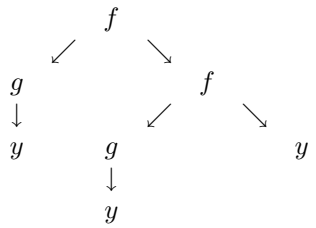


Tree t_2

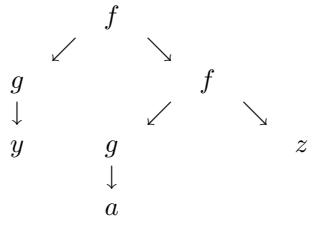


Initially, $tree1 = t_1$, $tree2 = t_2$ and $node = e$. The first disagreement is found for $node = 1$. We form $newpair = (g(y)/x)$, and $unifier = newpair$. After applying $newpair$ to $tree1$ and $tree2$, we have:

Tree $tree1$

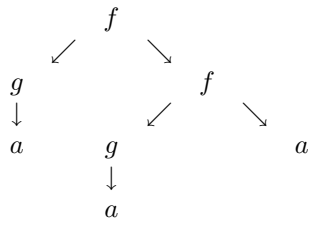


Tree $tree2$

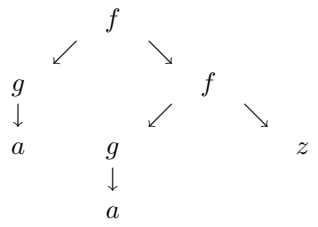


The next disagreement is found for $node = 211$. We find that $newpair = (a/y)$, and compose $unifier = (g(y)/x)$ with $newpair$, obtaining $(g(a)/x, a/y)$. After applying $newpair$ to $tree1$ and $tree2$, we have:

Tree $tree1$



Tree $tree2$



The last disagreement occurs for $node = 22$. We form $newpair = (a/z)$, and compose $unifier$ with $newpair$, obtaining

$$unifier = (g(a)/x, a/y, a/z).$$

The algorithm stops successfully with the most general unifier $(g(a)/x, a/y, a/z)$, and the trees are unified to the last value of $tree1$.

In order to prove the correctness of the unification algorithm, the following lemma will be needed.

Lemma 8.4.3 Let # be any constant. Given any two trees $f(s_1, ..., s_n)$ and $f(t_1, ..., t_n)$ the following properties hold:

(a) For any i, $1 \leq i \leq n$, if σ is a most general unifier for the trees

$$\begin{array}{rcll} f(s_1, ..., s_{i-1}, \#, ..., \#) & \text{and} & f(t_1, ..., t_{i-1}, \#, ..., \#), & \text{then} \\ f(s_1, ..., s_i, \#, ..., \#) & \text{and} & f(t_1, ..., t_i, \#, ..., \#) & \text{are unifiable iff} \\ \sigma(f(s_1, ..., s_i, \#, ..., \#)) & \text{and} & \sigma(f(t_1, ..., t_i, \#, ..., \#)) & \text{are unifiable.} \end{array}$$

(b) For any i, $1 \leq i \leq n$, if σ is a most general unifier for the trees $f(s_1, ..., s_{i-1}, \#, ..., \#)$ and $f(t_1, ..., t_{i-1}, \#, ..., \#)$, and θ is a most general unifier for the trees $\sigma(s_i)$ and $\sigma(t_i)$, then $\sigma \circ \theta$ is a most general unifier for the trees $f(s_1, ..., s_i, \#, ..., \#)$ and $f(t_1, ..., t_i, \#, ..., \#)$.

Proof: (a) The case $i = 1$ is trivial. Clearly, if σ is a most general unifier for the trees $f(s_1, ..., s_{i-1}, \#, ..., \#)$ and $f(t_1, ..., t_{i-1}, \#, ..., \#)$ and if the trees $\sigma(f(s_1, ..., s_i, \#, ..., \#))$ and $\sigma(f(t_1, ..., t_i, \#, ..., \#))$ are unifiable, then $f(s_1, ..., s_i, \#, ..., \#)$ and $f(t_1, ..., t_i, \#, ..., \#)$ are unifiable.

We now prove the other direction. Let θ be a unifier for

$$f(s_1, ..., s_i, \#, ..., \#) \quad \text{and} \quad f(t_1, ..., t_i, \#, ..., \#).$$

Then,

$$\theta(s_1) = \theta(t_1), ..., \theta(s_i) = \theta(t_i).$$

Hence, θ is a unifier for

$$f(s_1, ..., s_{i-1}, \#, ..., \#) \quad \text{and} \quad f(t_1, ..., t_{i-1}, \#, ..., \#).$$

Since σ is a most general unifier, there is some θ' such that $\theta = \sigma \circ \theta'$. Then,

$$\begin{aligned}\theta'(\sigma(f(s_1, ..., s_i, \#, ..., \#))) &= \theta(f(s_1, ..., s_i, \#, ..., \#)) \\ &= \theta(f(t_1, ..., t_i, \#, ..., \#)) = \theta'(\sigma(f(t_1, ..., t_i, \#, ..., \#))),\end{aligned}$$

which shows that θ' unifies

$$\sigma(f(s_1, ..., s_i, \#, ..., \#)) \quad \text{and} \quad \sigma(f(t_1, ..., t_i, \#, ..., \#)).$$

(b) Again, the case $i = 1$ is trivial. Otherwise, clearly,

$$\sigma(s_1) = \sigma(t_1), ..., \sigma(s_{i-1}) = \sigma(t_{i-1}) \text{ and } \theta(\sigma(s_i)) = \theta(\sigma(t_i))$$

implies that $\sigma \circ \theta$ is a unifier of

$$f(s_1, ..., s_i, \#, ..., \#) \quad \text{and} \quad f(t_1, ..., t_i, \#, ..., \#).$$

If λ unifies $f(s_1, ..., s_i, \#, ..., \#)$ and $f(t_1, ..., t_i, \#, ..., \#)$, then

$$\lambda(s_1) = \lambda(t_1), ..., \lambda(s_i) = \lambda(t_i).$$

Hence, λ unifies

$$f(s_1, ..., s_{i-1}, \#, ..., \#) \quad \text{and} \quad f(t_1, ..., t_{i-1}, \#, ..., \#).$$

Since σ is a most general unifier of these two trees, there is some σ' such that $\lambda = \sigma \circ \sigma'$. But then, since $\lambda(s_i) = \lambda(t_i)$, we have $\sigma'(\sigma(s_i)) = \sigma'(\sigma(t_i))$, and since θ is a most general unifier of $\sigma(s_i)$ and $\sigma(t_i)$, there is some θ' such that $\sigma' = \theta \circ \theta'$. Hence,

$$\lambda = \sigma \circ (\theta \circ \theta') = (\sigma \circ \theta) \circ \theta',$$

which proves that $\sigma \circ \theta$ is a most general unifier of $f(s_1, ..., s_i, \#, ..., \#)$ and $f(t_1, ..., t_i, \#, ..., \#)$. □

We will now prove the correctness of the unification algorithm.

Theorem 8.4.1 (Correctness of the unification algorithm) (i) Given any two finite trees t_1 and t_2, the unification algorithm always halts. It halts with output $unifiable =$ **true** iff t_1 and t_2 are unifiable.

(ii) If t_1 and t_2 are unifiable, then they have a most general unifier and the output of procedure $unify$ is a most general unifier.

Proof: Clearly, the procedure *test-and-substitute* always terminates, and we only have to prove the termination of the $unify$ procedure. The difficulty

in proving termination is that the trees $tree1$ and $tree2$ may grow. However, this can only happen if *test-and-substitute* is called, and in that case, since unifiable is not **false** iff the variable $x = tree1(node)$ does not belong to $t = tree2/node$, after the substitution of t for all occurrences of x in both $tree1$ and $tree2$, the variable x has been completely eliminated from both $tree1$ and $tree2$. This suggests to try a proof by induction over the well-founded lexicographic ordering $<<$ defined such that, for all pairs (m, t) and (m', t'), where m, m' are natural numbers and t, t' are finite trees,

$$(m, t) << (m', t') \quad \text{iff either } m < m',$$
$$\text{or } m = m' \text{ and } t \text{ is a proper subtree of } t'.$$

We shall actually prove the input-output correctness assertion stated below for the procedure $unify$.

Let s_0 and t_0 be two given finite trees, σ a substitution such that none of the variables in the support of σ is in $\sigma(s_0)$ or $\sigma(t_0)$, u any tree address in both $dom(\sigma(s_0))$ and $dom(\sigma(t_0))$, and let $s = \sigma(s_0)/u$ and $t = \sigma(t_0)/u$. Let $tree1_0$, $tree2_0$, $node_0$, $unifiable_0$ and $unifier_0$ be the input values of the variables $tree1$, $tree2$, $unifiable$, and $unifier$, and $tree1'$, $tree2'$, $node'$ $unifiable'$ and $unifier'$ be their output value (if any). Also, let m_0 be the sum of the number of variables in $\sigma(s_0)$ and $\sigma(t_0)$, and m' the sum of the number of variables in $tree1'$ and $tree2'$.

Correctness assertion:

$$\text{If } tree1_0 = \sigma(s_0), \quad tree2_0 = \sigma(t_0), \quad node_0 = u,$$
$$unifiable_0 = \textbf{true} \quad \text{and} \quad unifier_0 = \sigma, \text{ then}$$

the following holds:

(1) The procedure $unify$ always terminates;

(2) $unifiable' = \textbf{true}$ iff s and t are unifiable and, if $unifiable' = \textbf{true}$, then $unifier' = \sigma \circ \theta$, where θ is a most general unifier of s and t, $tree1' = unifier'(s_0)$, $tree2' = unifier'(t_0)$, and no variable in the support of $unifier'$ occurs in $tree1'$ or $tree2'$.

(3) If $tree1' \neq \sigma(s_0)$ or $tree2' \neq \sigma(t_0)$ then $m' < m_0$, else $m' = m_0$.

Proof of assertion: We proceed by complete induction on (m, s), where m is the sum of the number of variables in $tree1$ and $tree2$ and s is the subtree $tree1/node$.

(i) Assume that s is a constant and t is not a variable, the case in which t is a constant being similar. Then u is a leaf node in $\sigma(s_0)$. If $t \neq s$, the comparison of $tree1(node)$ and $tree2(node)$ fails, and $unifiable$ is set to **false**. The procedure terminates with failure. If $s = t$, since u is a leaf node in $\sigma(s_0)$ and $\sigma(t_0)$, the procedure terminates with success, $tree1' = \sigma(s_0)$,

$tree2' = \sigma(t_0)$, and $unifier' = \sigma$. Hence the assertion holds with the identity substitution for θ.

(ii) Assume that s is a variable say x, the case in which t is a variable being similar. Then u is a leaf node in $\sigma(s_0)$. If $t = s$, this case reduces to case (i). Otherwise, $t \neq x$ and the occur check is performed in *test-and-substitute*. If x occurs in t, then $unifiable$ is set to **false**, and the procedure terminates. In this case, it is clear that x and t are not unifiable, and the assertion holds. Otherwise, the substitution $\theta = (t/x)$ is created, $unifier' = \sigma \circ \theta$, and $tree1' = \theta(\sigma(s_0)) = unifier'(s_0)$, $tree2' = \theta(\sigma(t_0)) = unifier'(t_0)$. Clearly, θ is a most general unifier of x and t, and since x does not occur in t, since no variable in the support of σ occurs in $\sigma(s_0)$ or $\sigma(t_0)$, no variable in the support of $unifier'$ occurs in $tree1' = \theta(\sigma(s_0))$ or $tree2' = \theta(\sigma(s_0))$. Since the variable x does not occur in $tree1'$ and $tree2'$, (3) also holds. Hence, the assertion holds.

(iii) Both s and t have $depth \geq 1$. Assume that $s = f(s_1, ..., s_m)$ and $t = f'(t_1, ..., t_n)$. If $f \neq f'$, the test $tree1(node) = tree2(node)$ fails, and $unify$ halts with failure. Clearly, s and t are not unifiable, and the claim holds. Otherwise, $s = f(s_1, ..., s_n)$ and $t = f(t_1, ..., t_n)$.

We shall prove the following claim by induction:

Claim: (1) For every i, $1 \leq i \leq n + 1$, the first $i - 1$ recursive calls in the while loop in $unify$ halt with success iff $f(s_1, ..., s_{i-1}, \#, ..., \#)$ and $f(t_1, ..., t_{i-1}, \#, ..., \#)$ are unifiable, and otherwise one of the calls halts with failure;

(2) If the first $i - 1$ recursive calls halt with success, the input values at the end of the $(i - 1)$-th iteration are:

$$node_i = ui, \quad unifiable_i = \textbf{true}, \quad unifier_i = \sigma \circ \theta_{i-1},$$

where θ_{i-1} is a most general unifier for the trees $f(s_1, ..., s_{i-1}, \#, ..., \#)$ and $f(t_1, ..., t_{i-1}, \#, ..., \#)$, (with $\theta_0 = Id$, the identity substitution),

$$tree1_i = unifier_i(s_0), \quad tree2_i = unifier_i(t_0),$$

and no variable in the support of $unifier_i$ occurs in $tree1_i$ or $tree2_i$.

(3) If $tree1_i \neq tree1_0$ or $tree2_i \neq tree2_0$, if m_i is the sum of the number of variables in $tree1_i$ and $tree2_i$, then $m_i < m_0$.

Proof of claim: For $i = 1$, the claim holds because before entering the while loop for the first time,

$$tree1_1 = s_0, \quad tree2_1 = t_0, \quad node_1 = u1,$$
$$unifier_1 = \sigma, \quad unifiable_1 = \textbf{true}.$$

Now, for the induction step. We only need to consider the case where the first $i - 1$ recursive calls were successful. If we have $tree1_i = tree1_0$ and

$tree2_i = tree2_0$, then we can apply the induction hypothesis for the assertion to the address ui, since $tree1_0/ui$ is a proper subtree of $tree1_0/u$. Otherwise, $m_i < m_0$, and we can also also apply the induction hypothesis for the assertion to address ui. Note that

$$\begin{aligned} &tree1_i/u = \theta_{i-1}(f(s_1, ..., s_i, ..., s_n)) \quad \text{and} \\ &tree2_i/u = \theta_{i-1}(f(t_1, ..., t_i, ..., s_n)), \quad \text{since} \\ &unifier_i = \sigma \circ \theta_{i-1}. \end{aligned}$$

By lemma 8.4.3(a), since θ_{i-1} is a most general unifier for the trees

$$\begin{aligned} f(s_1, ..., s_{i-1}, \#, ..., \#) \quad &\text{and} \quad f(t_1, ..., t_{i-1}, \#, ..., \#), \quad \text{then} \\ f(s_1, ..., s_i, \#, ..., \#) \quad &\text{and} \quad f(t_1, ..., t_i, \#, ..., \#) \quad \text{are unifiable, iff} \\ \theta_{i-1}(f(s_1, ..., s_i, \#, ..., \#)) \quad &\text{and} \quad \theta_{i-1}f((t_1, ..., t_i, \#, ..., \#)) \quad \text{are unifiable.} \end{aligned}$$

Hence, $unify$ halts with success for this call for address ui, iff

$$f(s_1, ..., s_i, \#, ..., \#) \quad \text{and} \quad f(t_1, ..., t_i, \#, ..., \#) \quad \text{are unifiable.}$$

Otherwise, $unify$ halts with failure. This proves part (1) of the claim.

By part (2) of the assertion, the output value of the variable $unifier$ is of the form $unifier_i \circ \lambda_i$, where λ_i is a most general unifier for $\theta_{i-1}(s_i)$ and $\theta_{i-1}(t_i)$ (the subtrees at ui), and since θ_{i-1} is a most general unifier for

$$f(s_1, ..., s_{i-1}, \#, ..., \#) \quad \text{and} \quad f(t_1, ..., t_{i-1}, \#, ..., \#),$$

λ_i is a most general unifier for

$$\theta_{i-1}(f(s_1, ..., s_i, \#, ..., \#)) \quad \text{and} \quad \theta_{i-1}f((t_1, ..., t_i, \#, ..., \#)).$$

By lemma 8.4.3(b), $\theta_{i-1} \circ \lambda_i$ is a most general unifier for

$$f(s_1, ..., s_i, \#, ..., \#) \quad \text{and} \quad f(t_1, ..., t_i, \#, ..., \#).$$

Letting

$$\theta_i = \theta_{i-1} \circ \lambda_i,$$

it is easily seen that part (2) of the claim is satisfied. By part (3) of the assertion, part (3) of the claim also holds.

This concludes the proof of the claim. □

For $i = n + 1$, we see that all the recursive calls in the while loop halt successfully iff s and t are unifiable, and if s and t are unifiable, when the loop is exited, we have

$$unifier_{n+1} = \sigma \circ \theta_n,$$

where θ_n is a most general unifier of s and t,

$$tree1_{n+1} = unifier_{n+1}(s_0), \quad tree2_{n+1} = unifier_{n+1}(t_0),$$

and part (3) of the assertion also holds. This concludes the proof of the assertion. □

But now, we can apply the assertion to the input trees t_1 and t_2, with $u = e$, and σ the identity substitution. The correctness assertion says that $unify$ always halts, and if it halts with success, the output variable $unifier$ is a most general unifier for t_1 and t_2. This concludes the correctness proof. □

The subject of unification is the object of current research because fast unification is crucial for the efficiency of programming logic systems such as PROLOG. Some fast unification algorithms have been published such as Paterson and Wegman, 1978; Martelli and Montanari, 1982; and Huet, 1976. For a survey on unification, see the article by Siekmann in Shostak, 1984a. Huet, 1976, also contains a thorough study of unification, including higher-order unification.

PROBLEMS

8.4.1. Convert the following formulae to clause form:

$$\forall y(\exists x(P(y,x) \vee \neg Q(y,x)) \wedge \exists x(\neg P(x,y) \vee Q(x,y)))$$
$$\forall x(\exists y P(x,y) \wedge \neg Q(y,x)) \vee (\forall y \exists z(R(x,y,z) \wedge \neg Q(y,z)))$$
$$\neg(\forall x \exists y P(x,y) \supset (\forall y \exists z \neg Q(x,z) \wedge \forall y \neg \forall z R(y,z)))$$
$$\forall x \exists y \forall z(\exists w(Q(x,w) \vee R(x,y)) \equiv \neg \exists w \neg \exists u(Q(x,w) \wedge \neg R(x,u)))$$

8.4.2. Apply the unification algorithm to the following clauses:

$$\{P(x,y), P(y,f(z))\}$$
$$\{P(a,y,f(y)), P(z,z,u)\}$$
$$\{P(x,g(x)), P(y,y)\}$$
$$\{P(x,g(x),y), P(z,u,g(u))\}$$
$$\{P(g(x),y), P(y,y), P(u,f(w))\}$$

8.4.3. Let S and T be two finite sets of terms such that the set of variables occurring in S is disjoint from the set of variables occurring in T. Prove that if $S \cup T$ is unifiable, σ_S is a most general unifier of S, σ_T is a most general unifier of T, and $\sigma_{S,T}$ is a most general unifier of $\sigma_S(S) \cup \sigma_T(T)$, then

$$\sigma_S \circ \sigma_T \circ \sigma_{S,T}$$

is a most general unifier of $S \cup T$.

8.4.4. Show that the most general unifier of the following two trees contains a tree with 2^{n-1} occurrences of the variable x_1:

$$f(g(x_1, x_1), g(x_2, x_2), ..., g(x_{n-1}, x_{n-1})) \quad \text{and}$$
$$f(x_2, x_3, ..., x_n)$$

∗ **8.4.5.** Define the relation $\leq$ on terms as follows: Given any two terms t_1, t_2,

$$t_1 \leq t_2 \quad \text{iff} \quad \text{there is a substitution } \sigma \text{ such that } t_2 = \sigma(t_1).$$

Define the relation $\cong$ such that

$$t_1 \cong t_2 \quad \text{iff} \quad t_1 \leq t_2 \text{ and } t_2 \leq t_1.$$

(a) Prove that $\leq$ is reflexive and transitive and that $\cong$ is an equivalence relation.

(b) Prove that $t_1 \cong t_2$ iff there is a bijective renaming of variables ρ such that $t_1 = \rho(t_2)$. Show that the relation $\leq$ induces a partial ordering on the set of equivalence classes of terms modulo the equivalence relation $\cong$.

(c) Prove that two terms have a least upper bound iff they have a most general unifier (use a separating substitution, see Section 8.5).

(d) Prove that any two terms always have a greatest lower bound.

Remark: The structure of the set of equivalence classes of terms modulo $\cong$ under the partial ordering $\leq$ has been studied extensively in Huet, 1976. Huet has shown that this set is well founded, that every subset has a greatest lower bound, and that every bounded subset has a least upper bound.

8.5 The Resolution Method for First-Order Logic

Recall that we are considering first-order languages without equality. Also, recall that even though we usually omit quantifiers, clauses are universally quantified sentences. We extend the definition of a resolvent given in definition 4.3.2 to arbitrary clauses using the notion of a most general unifier.

8.5.1 Definition of the Method

First, we define the concept of a separating pair of substitutions.

Definition 8.5.1 Given two clauses A and A', a *separating pair of substitutions* is a pair of substitutions ρ and ρ' such that:

ρ has support $FV(A)$, ρ' has support $FV(A')$, for every variable x in A, $\rho(x)$ is a variable, for every variable y in A', $\rho'(y)$ is a variable, ρ and ρ' are bijections, and the range of ρ and the range of ρ' are disjoint.

Given a set S of literals, we say that S is *positive* if all literals in S are atomic formulae, and we say that S is *negative* if all literals in S are negations of atomic formulae. If a set S is positive or negative, we say that the literals in S are of the *same sign*. Given a set of literals $S = \{A_1, ..., A_m\}$, the *conjugate* of S is defined as the set

$$\overline{S} = \{\overline{A_1}, ..., \overline{A_m}\}$$

of conjugates of literals in S. If S is a positive set of literals we let $|S| = S$, and if S is a negative set of literals, we let $|S| = \overline{S}$.

Definition 8.5.2 Given two clauses A and B, a clause C is a *resolvent* of A and B iff the following holds:

(i) There is a subset $A' = \{A_1, ..., A_m\} \subseteq A$ of literals all of the same sign, a subset $B' = \{B_1, ..., B_n\} \subseteq B$ of literals all of the opposite sign of the set A', and a separating pair of substitutions (ρ, ρ') such that the set

$$|\rho(A') \cup \rho'(\overline{B'})|$$

is unifiable;

(ii) For some most general unifier σ of the set

$$|\rho(A') \cup \rho'(\overline{B'})|,$$

we have

$$C = \sigma(\rho(A - A') \cup \rho'(B - B')).$$

EXAMPLE 8.5.1

Let

$$A = \{\neg P(z, a), \neg P(z, x), \neg P(x, z)\} \quad \text{and}$$
$$B = \{P(z, f(z)), P(z, a)\}.$$

Let

$$A' = \{\neg P(z, a), \neg P(z, x)\} \quad \text{and} \quad B' = \{P(z, a)\},$$
$$\rho = (z_1/z), \quad \rho' = (z_2/z).$$

Then,

$$|\rho(A') \cup \rho'(\overline{B'})| = \{P(z_1, a), P(z_1, x), P(z_2, a)\}$$

is unifiable,

$$\sigma = (z_1/z_2, a/x)$$

is a most general unifier, and

$$C = \{\neg P(a, z_1), P(z_1, f(z_1))\}$$

is a resolvent of A and B.

If we take $A' = A$, $B' = \{P(z, a)\}$,

$$|\rho(A') \cup \rho'(\overline{B'})| = \{P(z_1, a), P(z_1, x), P(x, z_1), P(z_2, a)\}$$

is also unifiable,

$$\sigma = (a/z_1, a/z_2, a/x)$$

is the most general unifier, and

$$C = \{P(a, f(a))\}$$

is a resolvent.

Hence, two clauses may have several resolvents.

The generalization of definition 4.3.3 of a resolution DAG to the first-order case is now obvious.

Definition 8.5.3 Given a set $S = \{C_1, ..., C_n\}$ of first-order clauses, a *resolution DAG* for S is any finite set

$$G = \{(t_1, R_1), ..., (t_m, R_m)\}$$

of distinct DAGs labeled in the following way:

(1) The leaf nodes of each underlying tree t_i are labeled with clauses in S.

(2) For every DAG (t_i, R_i), every nonleaf node u in t_i is labeled with some triple $(C, (\rho, \rho'), \sigma)$, where C is a clause, (ρ, ρ') is a separating pair of substitutions, σ is a substitution and the following holds:

For every nonleaf node u in t_i, u has exactly two successors $u1$ and $u2$, and if $u1$ is labeled with a clause C_1 and u_2 is labeled with a clause C_2 (not necessarily distinct from C_1), then u is labeled with the triple $(C, (\rho, \rho'), \sigma)$, where (ρ, ρ') is a separating pair of substitutions for C_1 and C_2 and C is the resolvent of C_1 and C_2 obtained with the most general unifier σ.

A resolution DAG is a *resolution refutation* iff it consists of a single DAG (t, R) whose root is labeled with the empty clause. The nodes of a DAG that are not leaves are also called *resolution steps*.

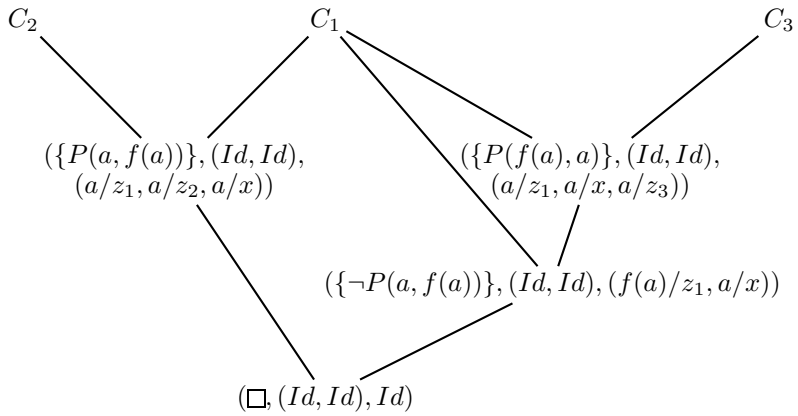
We will often use a simplified form of the above definition by dropping (ρ, ρ') and σ from the interior nodes, and consider that nodes are labeled with clauses. This has the effect that it is not always obvious how a resolvent is obtained.

EXAMPLE 8.5.2

Consider the following clauses:

$$C_1 = \{\neg P(z_1, a), \neg P(z_1, x), \neg P(x, z_1)\},$$
$$C_2 = \{P(z_2, f(z_2)), P(z_2, a)\} \quad \text{and}$$
$$C_3 = \{P(f(z_3), z_3), P(z_3, a)\}.$$

The following is a resolution refutation:



8.5.2 Soundness of the Resolution Method

In order to prove the soundness of the resolution method, we prove the following lemma, analogous to lemma 4.3.1.

Lemma 8.5.1 Given two clauses A and B, let $C = \sigma(\rho(A - A') \cup \rho'(B - B'))$ be any resolvent of A and B, for some subset $A' \subseteq A$ of literals of A, subset $B' \subseteq B$ of literals of B, separating pair of substitutions (ρ, ρ'), with $\rho = (z_1/x_1, ..., z_m/x_m)$, $\rho' = (z_{m+1}/y_1, ..., z_{m+n}/y_n)$ and most general unifier $\sigma = (t_1/u_1, ..., t_k/u_k)$, where $\{u_1, ..., u_k\}$ is a subset of $\{z_1, ..., z_{m+n}\}$. Also, let $\{v_1, ..., v_p\} = FV(C)$. Then,

$$\models (\forall x_1 ... \forall x_m A \wedge \forall y_1 ... \forall y_n B) \supset \forall v_1 ... \forall v_p C.$$

Proof: We show that we can constuct a G-proof for

$$(\forall x_1 ... \forall x_m A \wedge \forall y_1 ... \forall y_n B) \rightarrow \forall v_1 ... \forall v_p C.$$

Note that $\{z_1, ..., z_{m+n}\} - \{u_1, ..., u_k\}$ is a subset of $\{v_1, ..., v_p\}$. First, we perform $p\ \forall : right$ steps using p entirely new variables w_1,...,w_p. Let

$$\sigma' = \sigma \circ (w_1/v_1, ..., w_p/v_p) = (t'_1/z_1, ..., t'_{m+n}/z_{m+n}),$$

be the substitution obtained by composing σ and the substitution replacing each occurrence of the variable v_i by the variable w_i. Then, note that the support of σ' is disjoint from the set $\{w_1, ..., w_p\}$, which means that for every tree t,

$$\sigma'(t) = t[t'_1/z_1]...[t'_{m+n}/z_{m+n}]$$

(the order being irrelevant). At this point, we have the sequent

$$(\forall x_1 ... \forall x_m A \wedge \forall y_1 ... \forall y_n B) \rightarrow \sigma'(\rho(A - A')), \sigma'(\rho'(B - B')).$$

Then apply the $\wedge : left$ rule, obtaining

$$\forall x_1 ... \forall x_m A, \forall y_1 ... \forall y_n B \rightarrow \sigma'(\rho(A - A')), \sigma'(\rho'(B - B')).$$

At this point, we apply $m+n\ \forall : left$ rules as follows: If $\rho(x_i)$ is some variable u_j, do the substitution t'_j/x_i, else $\rho(x_i)$ is some variable v_j not in $\{u_1, .., u_k\}$, do the substitution w_j/v_j.

If $\rho'(y_i)$ is some variable u_j, do the substitution t'_j/y_i, else $\rho'(y_j)$ is some variable v_j not in $\{u_1, ..., u_k\}$, do the substitution w_j/v_j.

It is easy to verify that at the end of these steps, we have the sequent

$$(\sigma'(\rho(A - A')), Q), (\sigma'(\rho'(B - B')), \overline{Q}) \rightarrow \sigma'(\rho(A - A')), \sigma'(\rho'(B - B'))$$

where $Q = \sigma'(\rho(A'))$ and $\overline{Q} = \sigma'(\rho'(B'))$ are conjugate literals, because σ is a most general unifier of the set $|\rho(A') \cup \rho'(\overline{B'})|$.

Hence, we have a quantifier-free sequent of the form

$$(A_1 \vee Q), (A_2 \vee \neg Q) \rightarrow A_1, A_2,$$

and we conclude that this sequent is valid using the proof of lemma 4.3.1. □

As a consequence, we obtain the soundness of the resolution method.

Lemma 8.5.2 (Soundness of resolution without equality) If a set of clauses has a resolution refutation DAG, then S is unsatisfiable.

Proof: The proof is identical to the proof of lemma 4.3.2, but using lemma 8.5.1, as opposed to lemma 4.3.1. □

8.5.3 Completeness of the Resolution Method

In order to prove the completeness of the resolution method for first-order languages without equality, we shall prove the following lifting lemma.

Lemma 8.5.3 (Lifting lemma) Let A and B be two clauses, σ_1 and σ_2 two substitutions such that $\sigma_1(A)$ and $\sigma_2(B)$ are ground, and assume that D is a resolvent of the ground clauses $\sigma_1(A)$ and $\sigma_2(B)$. Then, there is a resolvent C of A and B and a substitution θ such that $D = \theta(C)$.

Proof: First, let (ρ, ρ') be a separating pair of substitutions for A and B. Since ρ and ρ' are bijections they have inverses ρ^{-1} and $\rho^{'-1}$. Let σ be the substitution formed by the union of $\rho^{-1} \circ \sigma_1$ and $\rho^{'-1} \circ \sigma_2$, which is well defined, since the supports of ρ^{-1} and $\rho^{'-1}$ are disjoint. It is clear that

$$\sigma(\rho(A)) = \sigma_1(A) \quad \text{and} \quad \sigma(\rho'(B)) = \sigma_2(B).$$

Hence, we can work with $\rho(A)$ and $\rho'(B)$, whose sets of variables are disjoint. If D is a resolvent of the clauses $\sigma_1(A)$ and $\sigma_2(B)$, there is a ground literal Q such that $\sigma(\rho(A))$ contains Q and $\sigma(\rho'(B))$ contains its conjugate. Assume that Q is positive, the case in which Q is negative being similar. Then, there must exist subsets $A' = \{A_1, ..., A_m\}$ of A and $B' = \{\neg B_1, ..., \neg B_n\}$ of B, such that

$$\sigma(\rho(A_1)) = ... = \sigma(\rho(A_m)) = \sigma(\rho'(B_1)) = ..., \sigma(\rho'(B_n)) = Q,$$

and σ is a unifier of $\rho(A') \cup \rho'(\overline{B'})$. By theorem 8.4.1, there is a most general unifier λ and a substitution θ such that

$$\sigma = \lambda \circ \theta.$$

Let C be the resolvent

$$C = \lambda(\rho(A - A') \cup \rho'(B - B')).$$

Clearly,

$$\begin{aligned} D &= (\sigma(\rho(A)) - \{Q\}) \cup (\sigma(\rho'(B)) - \{\neg Q\}) \\ &= (\sigma(\rho(A - A')) \cup \sigma(\rho'(B - B'))) \\ &= \theta(\lambda(\rho(A - A') \cup \rho'(B - B'))) = \theta(C). \end{aligned}$$

□

Using the above lemma, we can now prove the following lemma which shows that resolution DAGs of ground instances of clauses can be lifted to resolution DAGs using the original clauses.

Lemma 8.5.4 (Lifting lemma for resolution refutations) Let S be a finite set of clauses, and S_g be a set of ground instances of S, so that every clause in

S_g is of the form $\sigma_i(C_i)$ for some clause C_i in S and some ground substitution σ_i.

For any resolution DAG H_g for S_g, there is a resolution DAG H for S, such that the DAG H_g is a homomorphic image of the DAG H in the following sense:

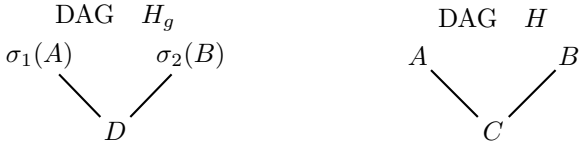
There is a function $F : H \to H_g$ from the set of nodes of H to the set of nodes of H_g, such that, for every node u in H, if $u1$ and $u2$ are the immediate descendants of u, then $F(u1)$ and $F(u2)$ are the immediate descendants of $F(u)$, and if the clause C (not necessarily in S_g) is the label of u, then $F(u)$ is labeled by the clause $\theta(C)$, where θ is some ground substitution.

Proof: We prove the lemma by induction on the underlying tree of H_g.

(i) If H_g has a single resolution step, we have clauses $\sigma_1(A)$, $\sigma_2(B)$ and their resolvent D. By lemma 8.5.3, there exists a resolvent C of A and B and a substitution θ such that $\theta(C) = D$. Note that it is possible that A and B are distinct, but $\sigma_1(A)$ and $\sigma_2(B)$ are not. In the first case, we have the following DAGs:

The homomorphism F is such that $F(e) = e$, $F(1) = 1$ and $F(2) = 1$.

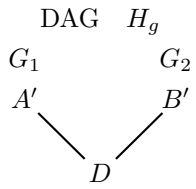
In the second case, $\sigma_1(A) \neq \sigma_2(B)$, but we could have $A = B$. Whether or not $A = B$, we create the following DAG H with three distinct nodes, so that the homomorphism is well defined:



The homomorphism F is the identity on nodes.

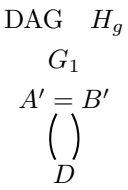
(ii) If H_g has more than one resolution step, it is of the form either

(ii)(a)



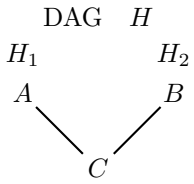
where A' and B' are distinct, or of the form

(ii)(b)



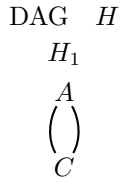
if $A' = B'$.

(a) In the first case, by the induction hypothesis, there are DAGs H_1 and H_2 and homomorphisms $F_1 : H_1 \to G_1$ and $F_2 : H_2 \to G_2$, where H_1 is rooted with some formula A and H_2 is rooted with some formula B, and for some ground substitutions θ_1 and θ_2, we have, $A' = \theta_1(A)$ and $B' = \theta_2(B)$. By lemma 8.5.3, there is a resolvent C of A and B and a substitution θ such that $\theta(C) = D$. We can construct H as the DAG obtained by making C as the root, and even if $A = B$, by creating two distinct nodes 1 and 2, with 1 labeled A and 2 labeled B:



The homomorphism $F : H \to H_g$ is defined such that $F(e) = e$, $F(1) = 1$, $F(2) = 2$, and it behaves like F_1 on H_1 and like F_2 on H_2. The root clause C is mapped to $\theta(C) = D$.

(b) In the second case, by the induction hypothesis, there is a DAG H_1 rooted with some formula A and a homomorphism $F_1 : H_1 \to G_1$, and for some ground substitution θ_1, we have $A' = \theta_1(A)$. By lemma 8.5.3, there is a resolvent C of A with itself, and a substitution θ such that $\theta(C) = D$. It is clear that we can form H so that C is a root node with two edges connected to A, and F is the homomorphism such that $F(e) = e$, $F(1) = 1$, and F behaves like F_1 on H_1.



The clause C is mapped onto $D = \theta(C)$. This concludes the proof. □

EXAMPLE 8.5.3

The following shows a lifting of the ground resolution of example 8.3.1 for the clauses:

$$C_1 = \{\neg P(z_1, a), \neg P(z_1, x), \neg P(x, z_1)\}$$
$$C_2 = \{P(z_2, f(z_2)), P(z_2, a)\}$$
$$C_3 = \{P(f(z_3), z_3), P(z_3, a)\}.$$

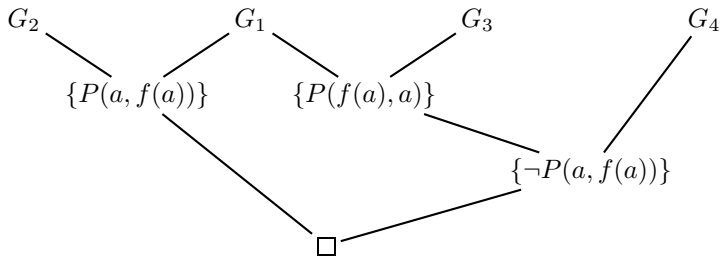
Recall that the ground instances are

$$G_1 = \{\neg P(a, a)\}$$
$$G_2 = \{P(a, f(a)), P(a, a)\}$$
$$G_3 = \{P(f(a), a), P(a, a)\}$$
$$G_4 = \{\neg P(f(a), a), \neg P(a, f(a))\},$$

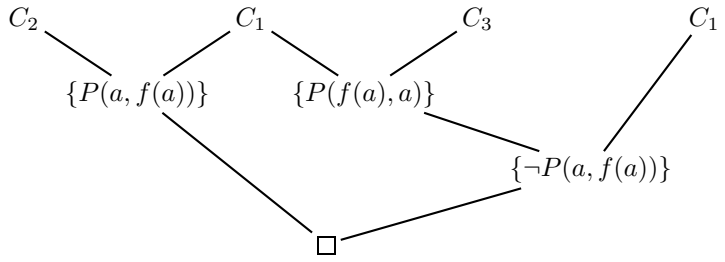
and the substitutions are

$$\sigma_1 = (a/z_2)$$
$$\sigma_2 = (a/z_1, a/x)$$
$$\sigma_3 = (a/z_3)$$
$$\sigma_4 = (f(a)/z_1, a/x).$$

Ground resolution-refutation H_g
for the set of ground clauses G_1, G_2, G_3, G_4



Lifting H of the above resolution refutation
for the clauses C_1, C_2, C_3



The homomorphism is the identity on the nodes, and the substitutions are, (a/z_2) for node 11 labeled C_2, $(a/z_1, a/x)$ for node 12 labeled C_1,

(a/z_3) for node 212 labeled C_3, and $(f(a)/z_1, a/x)$ for node 22 labeled C_1.

Note that this DAG is not as concise as the DAG of example 8.5.1. This is because is has been designed so that there is a homomorphism from H to H_g.

As a consequence of the lifting theorem, we obtain the completeness of resolution.

Theorem 8.5.1 (Completeness of resolution, without equality) If a finite set S of clauses is unsatisfiable, then there is a resolution refutation for S.

Proof: By the Skolem-Herbrand-Gödel theorem (theorem 7.6.1, or its corollary), S is unsatisfiable iff a conjunction S_g of ground substitution instances of clauses in S is unsatisfiable. By the completeness of ground resolution (lemma 8.3.1), there is a ground resolution refutation H_g for S_g. By lemma 8.5.4, this resolution refutation can be lifted to a resolution refutation H for S. This concludes the proof. □

Actually, we can also prove the following type of Herbrand theorem for the resolution method, using the constructive nature of lemma 7.6.2.

Theorem 8.5.2 (A Herbrand-like theorem for resolution) Consider a first-order language without equality. Given any prenex sentence A whose matrix is in CNF, if $A \rightarrow$ is LK-provable, then a resolution refutation of the clause form of A can be obtained constructively.

Proof: By lemma 7.6.2, a compound instance C of the Skolem form B of A can be obtained constructively. Observe that the Skolem form B of A is in fact a clause form of A, since A is in CNF. But C is in fact a conjunction of ground instances of the clauses in the clause form of A. Since $\neg C$ is provable, the *search* procedure will give a proof that can be converted to a $GCNF'$-proof. Since theorem 4.3.1 is constructive, we obtain a ground resolution refutation H_g. By the lifting lemma 8.5.4, a resolution refutation H can be constructively obtained for S_g. Hence, we have shown that a resolution refutation for the clause form of A can be constructively obtained from an LK-proof of $A \rightarrow$. □

It is likely that theorem 8.5.2 has a converse, but we do not have a proof of such a result. A simpler result is to prove the converse of lemma 8.5.4, the lifting theorem. This would provide another proof of the soundness of resolution. It is indeed possible to show that given any resolution refutation H of a set S of clauses, a resolution refutation H_g for a certain set S_g of ground instances of S can be constructed. However, the homomorphism property does not hold directly, and one has to exercise care in the construction. The interested reader should consult the problems.

It should be noted that a Herbrand-like theorem for the resolution method and a certain Hilbert system has been proved by Joyner in his Ph.D

thesis (Joyner, 1974). However, these considerations are somewhat beyond the scope of this text, and we will not pursue this matter any further.

PROBLEMS

8.5.1. Give separating pairs of substitutions for the following clauses:

$$\{P(x, y, f(z))\},\ \{P(y, z, f(z))\}$$
$$\{P(x, y), P(y, z)\},\ \{Q(y, z), P(z, f(y))\}$$
$$\{P(x, g(x))\},\ \{P(x, g(x))\}$$

8.5.2. Find all resolvents of the following pairs of clauses:

$$\{P(x, y), P(y, z)\},\ \{\neg P(u, f(u))\}$$
$$\{P(x, x), \neg R(x, f(x))\},\ \{R(x, y), Q(y, z)\}$$
$$\{P(x, y), \neg P(x, x), Q(x, f(x), z)\},\ \{\neg Q(f(x), x, z), P(x, z)\}$$
$$\{P(x, f(x), z), P(u, w, w)\},\ \{\neg P(x, y, z), \neg P(z, z, z)\}$$

8.5.3. Establish the unsatisfiability of each of the following formulae using the resolution method.

$$(\forall x \exists y P(x, y) \wedge \exists x \forall y \neg P(x, y))$$

$$(\forall x \exists y \exists z (L(x, y) \wedge L(y, z) \wedge Q(y) \wedge R(z) \wedge (P(z) \equiv R(x))) \wedge$$
$$\forall x \forall y \forall z ((L(x, y) \wedge L(y, z)) \supset L(x, z)) \wedge \exists x \forall y \neg (P(y) \wedge L(x, y)))$$

8.5.4. Consider the following formulae asserting that a binary relation is symmetric, transitive, and total:

$$S_1 : \forall x \forall y (P(x, y) \supset P(y, x))$$
$$S_2 : \forall x \forall y \forall z ((P(x, y) \wedge P(y, z)) \supset P(x, z))$$
$$S_3 : \forall x \exists y P(x, y)$$

Prove by resolution that

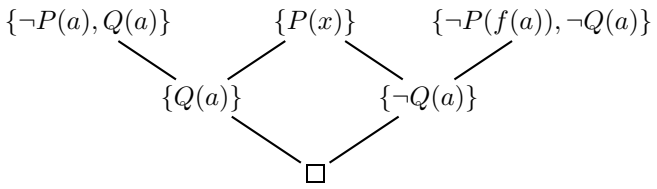
$$S_1 \wedge S_2 \wedge S_3 \supset \forall x P(x, x).$$

In other words, if P is symmetric, transitive and total, then P is reflexive.

8.5.5. Complete the details of the proof of lemma 8.5.1.

∗ **8.5.6.** (a) Prove that given a resolution refutation H of a set S of clauses, a resolution refutation H_g for a certain set S_g of ground instances of S can be constructed.

Apply the above construction to the following refutation:



(b) Using (a), give another proof of the soundness of the resolution method.

∗ **8.5.7.** As in the propositional case, another way of presenting the resolution method is as follows. Given a (finite) set S of clauses, let

$$R(S) = S \cup \{C \mid C \text{ is a resolvent of two clauses in } S\}.$$

Also, let

$$\begin{aligned} R^0(S) &= S, \\ R^{n+1}(S) &= R(R^n(S)), (n \geq 0), \text{ and let} \\ R^*(S) &= \bigcup_{n \geq 0} R^n(S). \end{aligned}$$

(a) Prove that S is unsatisfiable if and only if $R^*(S)$ is unsatisfiable.

(b) Prove that if S is finite, there is some $n \geq 0$ such that

$$R^*(S) = R^n(S).$$

(c) Prove that there is a resolution refutation for S if and only if the empty clause $\square$ is in $R^*(S)$.

(d) Prove that S is unsatisfiable if and only if $\square$ belongs to $R^*(S)$.

8.5.8. Prove that the resolution method is still complete if the resolution rule is restricted to clauses that are not tautologies (that is, clauses not containing both A and $\neg A$ for some atomic formula A).

∗ **8.5.9.** We say that a clause C_1 *subsumes* a clause C_2 if there is a substitution σ such that $\sigma(C_1)$ is a subset of C_2. In the version of the resolution method described in problem 8.5.7, let

$$\begin{aligned} R_1(S) &= R(S) - \{C \mid C \text{ is subsumed by some clause in } R(S)\}. \\ \text{Let } R_1^0 &= S, \\ R_1^{n+1}(S) &= R_1(R_1^n(S)) \text{ and} \\ R_1^*(S) &= \bigcup_{n \geq 0} R_1^n(S). \end{aligned}$$

Prove that S is unsatisfiable if and only if $\square$ belongs to $R_1^*(S)$.

8.5.10. The resolution method described in problem 8.5.7 can be modified by introducing the concept of *factoring*. Given a clause C, if C' is any subset of C and C' is unifiable, the clause $\sigma(C)$ where σ is a most general unifier of C' is a *factor* of C. The *factoring rule* is the rule that allows any factor of a clause to be added to $R(S)$. Consider the simplification of the resolution rule in which a resolvent of two clauses A and B is obtained by resolving sets A' and B' consisting of a single literal. This restricted version of the resolution rule is sometimes called *binary resolution*.

(a) Show that binary resolution together with the factoring rule is complete.

(b) Show that the factoring rule can be restricted to sets C' consisting of a pair of literals.

(c) Show that binary resolution alone is not complete.

8.5.11. Prove that the resolution method is also complete for infinite sets of clauses.

8.5.12. Write a computer program implementing the resolution method.

8.6 A Glimpse at Paramodulation

As we have noted earlier, equality causes complications in automatic theorem proving. Several methods for handling equality with the resolution method have been proposed, including the *paramodulation method* (Robinson and Wos, 1969), and the *E-resolution method* (Morris, 1969; Anderson, 1970). Due to the lack of space, we will only define the *paramodulation rule*, but we will not give a full treatment of this method.

In order to define the paramodulation rule, it is convenient to assume that the *factoring rule* is added to the resolution method. Given a clause A, if A' is any subset of A and A' is unifiable, the clause $\sigma(A)$ where σ is a most general unifier of A' is a *factor* of A. Using the factoring rule, it is easy to see that the resolution rule can be simplified, so that a resolvent of two clauses A and B is obtained by resolving sets A' and B' consisting of a single literal. This restricted version of the resolution rule is sometimes called *binary resolution* (this is a poor choice of terminology since both this restricted rule and the general resolution rule take two clauses as arguments, but yet, it is used in the literature!). It can be shown that binary resolution alone is not complete, but it is easy to show that it is complete together with the factoring rule (see problem 8.5.10).

The *paramodulation rule* is a rule that treats an equation $s \doteq t$ as a (two way) *rewrite rule*, and allows the replacement of a subterm r unifiable with

s (or t) in an atomic formula Q, by the other side of the equation, modulo substitution by a most general unifier.

More precisely, let

$$A = ((s \doteq t) \vee C)$$

be a clause containing the equation $s \doteq t$, and

$$B = (Q \vee D)$$

be another clause containing some literal Q (of the form $Pt_1...t_n$ or $\neg Pt_1...t_n$, for some predicate symbol P of rank n, possibly the equality symbol $\doteq$, in which case $n = 2$), and assume that for some tree address u in Q, the subterm $r = Q/u$ is unifiable with s (or that r is unifiable with t). If σ is a most general unifier of s and r, then the clause

$$\sigma(C \vee Q[u \leftarrow t] \vee D)$$

(or $\sigma(C \vee Q[u \leftarrow s] \vee D)$, if r and t are unifiable) is a *paramodulant* of A and B. (Recall from Subsection 2.2.5, that $Q[u \leftarrow t]$ (or $Q[u \leftarrow s]$) is the result of replacing the subtree at address u in Q by t (or s)).

EXAMPLE 8.6.1

Let

$$A = \{f(x, h(y)) \doteq g(x, y), P(x)\}, \quad B = \{Q(h(f(h(x), h(a))))\}.$$

Then

$$\{Q(h(g(h(z), h(a)))), P(h(z))\}$$

is a paramodulant of A and B, in which the replacement is performed in B at address 11.

EXAMPLE 8.6.2

Let

$$A = \{f(g(x), x) \doteq h(a)\}, \quad B = \{f(x, y) \doteq h(y)\}.$$

Then,

$$\{h(z) \doteq h(a)\}$$

is a paramodulant of A and B, in which the replacement is performed in A at address e.

It can be shown that the resolution method using the (binary) resolution rule, the factoring rule, and the paramodulation rule, is complete for any finite set S of clauses, provided that the reflexity axiom and the functional reflexivity axioms are added to S. The *reflexivity axiom* is the clause

$$\{x \doteq x\},$$

and the *functional reflexivity axioms* are the clauses

$$\{f(x_1, ..., x_n) \doteq f(x_1, ..., x_n)\},$$

for each function symbol f occurring in S, of any rank $n > 0$.

The proof that this method is complete is more involved than the proof for the case of a first-language without equality, partly because the lifting lemma does not extend directly. It can also be shown that paramodulation is complete without the functional reflexivity axioms, but this is much harder. For details, the reader is referred to Loveland, 1978.

Notes and Suggestions for Further Reading

The resolution method has been studied extensively, and there are many refinements of this method. Some of the refinements are still complete for all clauses (linear resolution, model elimination), others are more efficient but only complete for special kinds of clauses (unit or input resolution). For a detailed exposition of these methods, the reader is referred to Loveland, 1978; Robinson, 1979, and to the collection of original papers compiled in Siekmann and Wrightson, 1983. One should also consult Boyer and Moore, 1979, for advanced techniques in automatic theorem proving, induction in particular. For a more introductory approach, the reader may consult Bundy, 1983, and Kowalski, 1979.

The resolution method has also been extended to higher-order logic by Andrews. The interested reader should consult Andrews, 1971; Pietrzykowski, 1973; and Huet, 1973.

Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)

ROBERT NIEUWENHUIS AND ALBERT OLIVERAS

Technical University of Catalonia, Barcelona, Spain

AND

CESARE TINELLI

The University of Iowa, Iowa City, Iowa

Abstract. We first introduce *Abstract DPLL*, a rule-based formulation of the Davis–Putnam–Logemann–Loveland (DPLL) procedure for propositional satisfiability. This abstract framework allows one to cleanly express practical DPLL algorithms and to formally reason about them in a simple way. Its properties, such as soundness, completeness or termination, immediately carry over to the modern DPLL implementations with features such as backjumping or clause learning.

We then extend the framework to Satisfiability Modulo background Theories (SMT) and use it to model several variants of the so-called *lazy approach* for SMT. In particular, we use it to introduce a few variants of a new, efficient and modular approach for SMT based on a general DPLL(X) engine, whose parameter X can be instantiated with a specialized solver $Solver_T$ for a given theory T, thus producing a DPLL(T) system. We describe the high-level design of DPLL(X) and its cooperation with $Solver_T$, discuss the role of *theory propagation*, and describe different DPLL(T) strategies for some theories arising in industrial applications.

Our extensive experimental evidence, summarized in this article, shows that DPLL(T) systems can significantly outperform the other state-of-the-art tools, frequently even in orders of magnitude, and have better scaling properties.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational logic*; *verification*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Deduction (e.g., natural, rule-based)*

General Terms: Theory, Verification

Additional Key Words and Phrases: SAT solvers, Satisfiability Modulo Theories

---

This work was partially supported by Spanish Ministry of Education and Science through the Logic Tools project TIN2004-03382 (all authors), the FPU grant AP2002-3533 (Oliveras) and National Science Foundation (NSF) grant 0237422 (Tinelli and Oliveras).

Authors' addresses: R. Nieuwenhuis and A. Oliveras, Technical University of Catalonia, Campus Nord—Edif. Omega, C. Jordi Girona, 1–3, 08034 Barcelona, Spain; C. Tinelli, University of Iowa, Department of Computer Science, 14 MacLean Hall, Iowa City, IA, 52242.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 0004-5411/06/1100-0937 $5.00

1. *Introduction*

The problem of deciding the satisfiability of propositional formulas (SAT) does not only lie at the heart of the most important open problem in complexity theory (P vs. NP), it is also at the basis of many practical applications in such areas as Electronic Design Automation, Verification, Artificial Intelligence, and Operations Research. Thanks to recent advances in SAT-solving technology, propositional solvers are becoming the tool of choice for attacking more and more practical problems.

Most state-of-the-art SAT solvers [Moskewicz et al. 2001; Goldberg and Novikov 2002; Eén and Sörensson 2003; Ryan 2004] today are based on different variations of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis and Putnam 1960; Davis et al. 1962]. Starting essentially with the work on the GRASP, SATO and Relsat systems [Marques-Silva and Sakallah 1999; Zhang 1997; Bayardo and Schrag 1997], the spectacular improvements in the performance of DPLL-based SAT solvers achieved in the last years are due to (i) better implementation techniques, such as the *two-watched literal* approach for unit propagation, and (ii) several conceptual enhancements on the original DPLL procedure, aimed at reducing the amount of explored search space, such as *backjumping* (a form of *non-chronological backtracking*), *conflict-driven lemma learning*, and *restarts*. These advances make it now possible to decide the satisfiability of industrial SAT problems with tens of thousands of variables and millions of clauses.

Because of their success, both the DPLL procedure and its enhancements have been adapted to handle satisfiability problems in more expressive logics than propositional logic. In particular, they have been used to build efficient algorithms for the *Satisfiability Modulo Theories (SMT)* problem: deciding the satisfiability of ground first-order formulas with respect to background theories such as the theory of equality, of the integer or real numbers, of arrays, and so on [Armando et al. 2000, 2004; Filliâtre et al. 2001; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Ganzinger et al. 2004; Bozzano et al. 2005]. SMT problems arise in many industrial applications, especially in formal verification (see Section 3 for examples). They may contain thousands of clauses like $p \vee \neg q \vee a = f(b - c) \vee g(g(b)) \neq c \vee a - c \leq 7$, with purely propositional atoms as well as atoms over (combined) theories, such as the theory of the integers, or of Equality with Uninterpreted Functions (EUF).

Altogether, many variants and extensions of the DPLL procedure exist today. They are typically described in the literature informally and with the aid of pseudocode fragments. Therefore, it has become difficult for the newcomer to understand the precise nature of all these procedures, and for the expert to formally reason about their properties.

The first main contribution of this article is to address these shortcomings by providing *Abstract DPLL*, a uniform, declarative framework for describing DPLL-based solvers, both for propositional satisfiability and for satisfiability modulo theories. The framework allows one to describe the essence of various prominent approaches and techniques in terms of simple transition rules and rule application strategies. By abstracting away heuristics and implementation issues, it facilitates the understanding of DPLL at a conceptual level as well as its correctness and termination. For DPLL-based SMT approaches, it moreover provides a clean formulation and a basis for comparison of the different approaches.

The second main contribution of this article is a new modular architecture for building SMT solvers in practice, called DPLL(T), and a careful study of *theory propagation*, a refinement of SMT methods that can have a crucial impact on their performance.

The architecture is based on a general DPLL(X) engine, whose parameter X can be instantiated with a specialized solver $Solver_T$ for a given theory T, thus producing a system DPLL(T). Such systems can be implemented extremely efficiently and have good scaling properties: our Barcelogic implementation of DPLL(T) won four divisions at the 2005 SMT Competition [Barrett et al. 2005] (for the other three existing divisions it had no $Solver_T$ yet). The insights provided by our Abstract DPLL framework were an important factor in the success of our DPLL(T) architecture and its Barcelogic implementation. For instance, the abstract framework helped us in understanding the interactions between the DPLL(X) engine and the solvers, especially concerning the different forms of theory propagation, as well as in defining a good interface between both.

Section 2 of this article presents the propositional version of Abstract DPLL. It models DPLL procedures by means of simple *transition systems*. While abstract and declarative in nature, these transition systems can explicitly model the salient conceptual features of state-of-the-art DPLL-based SAT solvers, thus bridging the gap between logic-based calculi for DPLL and actual implementations. Within the Abstract DPLL formalism, we discuss in a clean and uniform way properties such as soundness, completeness, and termination. These properties immediately carry over to modern DPLL implementations with features such as backjumping and learning.

For backjumping systems, for instance, we achieve this by modeling backjumping by a general rule that encompasses several backtracking strategies—including basic chronological backtracking—and explaining how different systems implement the rule. Similarly, we model learning by general rules that show how devices such as conflict graphs are just one possibility for computing new lemmas. We also provide a general and simple termination argument for DPLL procedures that does not depend on an exhaustive enumeration of truth assignments; instead, it relies on a notion of search progress neatly expressing that search advances with the deduction of new unit clauses—the higher up in the search tree the better—which is the very essence of backjumping.

In Section 3, we go beyond propositional satisfiability, and extend the framework to *Abstract DPLL Modulo Theories*. As in the purely propositional case, this again allows us to express—and formally reason about—a number of current DPLL-based techniques for SMT, such as the various variants of the so-called *lazy approach* [Armando et al. 2000, 2004; Filliâtre et al. 2001; Audemard et al. 2002; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Ball et al. 2004].

In Section 4, based on the Abstract DPLL Modulo Theories framework, we introduce our DPLL(T) approach for building SMT systems. We first describe two variants of DPLL(T), depending on whether theory propagation is done exhaustively or not. Once the DPLL(X) engine has been implemented, this approach becomes extremely flexible: a DPLL(T) system for a theory T is obtained by simply plugging in the corresponding theory solver $Solver_T$, which must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements. We discuss the design of DPLL(X) and describe how DPLL(X) and $Solver_T$ cooperate. We also show that practical T-solvers can be designed to include theory propagation in an efficient way. A nontrivial issue

is how to deal with conflict analysis and clause learning adequately in the context of theory propagation. Different options and possible problems for doing this are analyzed and discussed in detail in Section 5.

In Section 6, we discuss some experiments with our Barcelogic implementation of DPLL(T). The results show that it can significantly outperform the best state-of-the-art tools and, in addition, scales up very well.

This article consolidates and improves upon preliminary ideas and results presented at the JELIA [Tinelli 2002], LPAR [Nieuwenhuis and Oliveras 2003; Nieuwenhuis et al. 2005], and CAV [Ganzinger et al. 2004; Nieuwenhuis and Oliveras 2005a] conferences.

2. *Abstract DPLL in the Propositional Case*

We start this section with some formal preliminaries on propositional logic and on transition systems. Then we introduce several variants of Abstract DPLL and prove their correctness properties, showing at the same time how the different features of actual DPLL implementations are modeled by these variants.

2.1. Formulas, Assignments, and Satisfaction.

Let P be a fixed finite set of propositional symbols. If $p \in P$, then p is an *atom* and p and $\neg p$ are *literals* of P. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$. A *unit clause* is a clause consisting of a single literal. A (CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \cdots \wedge C_n$. When it leads to no ambiguities, we will sometimes also write such a formula in set notation $\{C_1, \ldots, C_n\}$, or simply replace the $\wedge$ connectives by commas.

A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. A literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. A literal is *defined* in M if it is either true or false in M. The assignment M is *total* over P if no literal of P is undefined in M. A clause C is true in M if at least one of its literals is in M. It is false in M if all its literals are false in M, and it is undefined in M otherwise. A formula F is true in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In that case, M is a *model* of F. If F has no models then it is *unsatisfiable*. If F and F' are formulas, we write $F \models F'$ if F' is true in all models of F. Then, we say that F' *is entailed by* F, or is a *logical consequence* of F. If $F \models F'$ and $F' \models F$, we say that F and F' are *logically equivalent*.

In what follows, (possibly subscripted or primed) lowercase l *always* denote literals. Similarly, C and D always denote clauses, F and G denote formulas, and M and N denote assignments. If C is a clause $l_1 \vee \cdots \vee l_n$, we sometimes write $\neg C$ to denote the formula $\neg l_1 \wedge \cdots \wedge \neg l_n$.

2.2. States and Transition Systems in Abstract DPLL.

DPLL can be fully described by simply considering that a *state* of the procedure is either the distinguished state *FailState* or a pair of the form $M \parallel F$, where F is a CNF formula, that is, a finite set of clauses, and M is, essentially, a (partial) assignment.

More precisely, M is a *sequence* of literals, never containing both a literal and its negation, where each literal has an *annotation*, a bit that marks it as a *decision* literal (see below) or not. Frequently, we will consider M just as a partial assignment, or as a set or conjunction of literals (and hence as a formula), ignoring both the annotations and the order between its elements.

The concatenation of two such sequences will be denoted by simple juxtaposition. When we want to emphasize that a literal l is annotated as a decision literal we will write it as l^{d}. We will denote the empty sequence of literals (or the empty assignment) by $\emptyset$. We say that a clause C is *conflicting* in a state $M \parallel F, C$ if $M \models \neg C$.

We will model each DPLL procedure by means of a set of states together with a binary relation $\Longrightarrow$ over these states, called the *transition relation*. As usual, we use infix notation, writing $S \Longrightarrow S'$ instead of $(S, S') \in \Longrightarrow$. If $S \Longrightarrow S'$ we say that there is a *transition* from S to S'. We denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$. We call any sequence of transitions of the form $S_0 \Longrightarrow S_1, \quad S_1 \Longrightarrow S_2, \cdots$ a *derivation*, and denote it by $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \cdots$. We call any subsequence of a derivation a *subderivation*.

In what follows, transition relations will be defined by means of conditional *transition rules*. For a given state S, a transition rule precisely defines whether there is a transition from S by this rule and, if so, to which state S'. Such a transition is called an *application step* of the rule.

A *transition system* is a set of transition rules defined over some given set of states. Given a transition system R, the transition relation defined by R will be denoted by $\Longrightarrow_R$. If there is no transition from S by $\Longrightarrow_R$, we will say that S is *final* with respect to R (examples of a transition system and a final state with respect to it can be found in Definition 2.1 and Example 2.2).

2.3. THE CLASSICAL DPLL PROCEDURE. A very simple DPLL system, faithful to the classical DPLL algorithm, consists of the following five transition rules. We give this system here mainly for explanatory and historical reasons. The informally stated results for it are easily obtained by adapting the more general ones given in Section 2.5.

Definition 2.1. The *Classical DPLL system* is the transition system $C\,l$ consisting of the following five transition rules. In this system, the literals added to M by all rules except Decide are annotated as non-decision literals.

UnitPropagate:

$$M \parallel F,\ C \vee l \quad \Longrightarrow \quad M\ l \parallel F,\ C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M. \end{cases}$$

PureLiteral:

$$M \parallel F \quad \Longrightarrow \quad M\ l \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M. \end{cases}$$

Decide:

$$M \parallel F \quad \Longrightarrow \quad M\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M. \end{cases}$$

Fail:

$$M \parallel F,\ C \quad \Longrightarrow \quad \mathit{FailState} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals.} \end{cases}$$

Backtrack:

$$M\ l^{\mathsf{d}}\ N \parallel F,\ C \quad \Longrightarrow \quad M\ \neg l \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C \\ N \text{ contains no decision literals.} \end{cases}$$

One can use the transition system Cl for deciding the satisfiability of an input formula F simply by generating an arbitrary derivation $\emptyset \parallel F \Longrightarrow_{\mathrm{Cl}} \cdots \Longrightarrow_{\mathrm{Cl}} S_n$, where S_n is a final state with respect to Cl. The applicability of each of the five rules is easy to check and, as we will see, their application always leads to finite derivations. Moreover, for every derivation like the above ending in a final state S_n, (i) F is unsatisfiable if, and only if, S_n is *FailState*, and (ii) if S_n is of the form $M \parallel F$ then M is a model of F. Note that in this Classical DPLL system the second component of a state remains unchanged, a property that does not hold for the other transition systems we introduce later.

We now briefly comment on what the different rules do. In the following, if M is a sequence of the form $M_0\, l_1\, M_1 \cdots l_k\, M_k$, where the l_i are all the decision literals in M, we say that the state $M \parallel F$ is *at decision level* k, and that all the literals of each $l_i\ M_i$ *belong to decision level* i.

—UnitPropagate: To satisfy a CNF formula, all its clauses have to be true. Hence, if a clause of F contains a literal l whose truth value is not defined by the current assignment M while all the remaining literals of the clause are false, then M must be extended to make l true.

—PureLiteral: If a literal l is *pure* in F, that is, it occurs in F while its negation does not, then F is satisfiable only if it has a model that makes l true. Thus, if M does not define l it can be extended to make l true.

—Decide: This rule represents a case split. An undefined literal l is chosen from F, and added to M. The literal is annotated as a *decision literal*, to denote that if $M\ l$ cannot be extended to a model of F then the alternative extension $M\ \neg l$ must still be considered. This is done by means of the Backtrack rule.

—Fail: This rule detects a *conflicting clause* C and produces the *FailState* state whenever M contains no decision literals.

—Backtrack: If a conflicting clause C is detected and Fail does not apply, then the rule backtracks one *decision level*, by replacing the most recent decision literal l^{d} by $\neg l$ and removing any subsequent literals in the current assignment. Note that $\neg l$ is annotated as a nondecision literal, since the other possibility l has already been explored.

Example 2.2. The following is a derivation in the Classical DPLL system, with each transition annotated by the rule that makes it possible. To improve readability we denote atoms by natural numbers, and negation by overlining.

$$
\begin{array}{rcl}
\emptyset & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}}\ \overline{2} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}}\ \overline{2}\ 3 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
1^{\mathsf{d}}\ \overline{2}\ 3\ 4 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1}\ 4 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1}\ 4\ \overline{3}^{\mathsf{d}} & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4 \\
\overline{1}\ 4\ \overline{3}^{\mathsf{d}}\ 2 & \parallel & \overline{1}\vee\overline{2},\ \ 2\vee3,\ \ \overline{1}\vee\overline{3}\vee4,\ \ 2\vee\overline{3}\vee\overline{4},\ \ 1\vee4
\end{array}
\qquad
\begin{array}{l}
\Longrightarrow_{\mathrm{Cl}} \text{(Decide)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(Backtrack)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(Decide)} \\
\Longrightarrow_{\mathrm{Cl}} \text{(UnitPropagate)} \\
\phantom{\Longrightarrow_{\mathrm{Cl}}}
\end{array}
$$

The last state of this derivation is final. The (total) assignment in it is a model of the formula.

The Davis–Putnam procedure [Davis and Putnam 1960] was originally presented as a two-phase proof-procedure for first-order logic. The unsatisfiability of a formula was to be proved by first generating a suitable set of ground instances which then, in the second phase, were shown to be propositionally unsatisfiable.

Subsequent improvements, such as the Davis-Logemann-Loveland procedure of Davis et al. [1962], mostly focused on the propositional phase. What most authors now call the *DPLL Procedure* is a satisfiability procedure for propositional logic based on this propositional phase. Originally, this procedure amounted to the depth-first search algorithm with backtracking modeled by our Classical DPLL system.

2.4. MODERN DPLL PROCEDURES. The major modern DPLL-based SAT solvers do not implement the Classical DPLL system. For example, due to efficiency reasons the pure literal rule is normally only used as a preprocessing step—hence, we will not consider this rule in the following. Moreover, *backjumping*, a more general and more powerful backtracking mechanism, is now commonly used in place of chronological backtracking.

The usefulness of a more sophisticated backtracking mechanism for DPLL solvers is perhaps best illustrated with another example of derivation in the Classical DPLL system.

Example 2.3.

$$
\begin{aligned}
\emptyset \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Decide)}\\
1^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2 \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Decide)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4 \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Decide)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &\Longrightarrow_{\mathrm{B}} \text{(Backtrack)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5} \;\|\; \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2}
\end{aligned}
$$

Before the Backtrack step, the clause $6 \vee \overline{5} \vee \overline{2}$ is conflicting: it is false in the assignment $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6}$. This is a consequence of the unit propagation 2 of the decision 1^{d}, together with the decision 5^{d} and its unit propagation $\overline{6}$.

Therefore, one can infer that the decision 1^{d} is incompatible with the decision 5^{d}, that is, that the given clause set entails $\overline{1}\vee\overline{5}$. Similarly, it also entails $\overline{2}\vee\overline{5}$.

Such entailed clauses are called *backjump clauses* if their presence would have allowed a unit propagation at an earlier decision level. This is precisely what *backjumping* does: given a backjump clause, it goes back to that level and adds the unit propagated literal. For example, using $\overline{2}\vee\overline{5}$ as a backjump clause, the last Backtrack step could be replaced by a backjump to a state with first component $1^{\mathsf{d}}\ 2\ \overline{5}$.

We model all this in the next system with the Backjump rule, of which Backtrack is a particular case. In this rule, the clause $C' \vee l'$ is the backjump clause, where l'

is the literal that can be unit propagated ($\overline{5}$ in our example). Below we show that the rule is effective: a backjump clause can always be found.

Definition 2.4. The *Basic DPLL system* is the four-rule transition system B consisting of the rules UnitPropagate, Decide, Fail from Classical DPLL, and the following Backjump rule:

Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F,\ C \implies M\ l' \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N. \end{cases}$$

We call clause C in Backjump the *conflicting* clause and clause $C' \vee l'$ the *backjump* clause.

Chronological backtracking, modeled by Backtrack, always undoes the *last* decision l, going back to the previous level and adding $\neg l$ to it. *Conflict-driven* backjumping, as modeled by Backjump, is generally able to backtrack further than chronological backtracking by analyzing the reasons that produced the conflicting clause. Backjump can frequently undo *several* decisions at once, going back to a lower decision level than the previous level and adding some new literal to that lower level. It jumps over levels that are irrelevant to the conflict. In the previous example, it jumps over the decision 3^{d} and its consequence 4, which are totally unrelated with the reasons for the falsity of the conflicting clause $6 \vee \overline{5} \vee \overline{2}$. Moreover, intuitively, the search state $1^{\mathsf{d}}\ 2\ \overline{5}$ reached after Backjump is more *advanced* than the state $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5}$ reached after Backtrack. This notion of "being more advanced" is formalized in Theorem 2.10 below.

We show in the proof of Lemma 2.8 below that the literals of the backjump clause can always be chosen among the negations of the decision literals—although better choices usually exist. When the negations of *all* the decision literals are included in the backjump clause, the Backjump rule simulates the Backtrack rule of Classical DPLL. We remark that, in fact, Lemma 2.8 shows that, whenever a state $M \parallel F$ contains a conflicting clause, either Fail applies, if there are no decision literals in M, or otherwise Backjump applies.

Most modern DPLL implementations make additional use of backjump clauses: they add them to the clause set as *learned* clauses, also called *lemmas*, implementing what is usually called *conflict-driven learning*.

In Example 2.3, learning the clause $\overline{2} \vee \overline{5}$ will allow the application of UnitPropagate to any state whose assignment contains either 2 or 5. Hence, it will prevent any conflict caused by having both 2 and 5 in M. Reaching such *similar* conflicts frequently happens in industrial problems having some regular structure, and learning such lemmas has been shown to be very effective in improving performance.

Since a lemma is aimed at preventing future similar conflicts, when these conflicts are not very likely to be found again the lemma can be removed. In practice, a lemma is removed when its *relevance* (see, e.g., Bayardo and Schrag [1997]) or its *activity* level drops below a certain threshold; the activity can be, for example, the

number of times it becomes a unit or a conflicting clause [Goldberg and Novikov 2002].

To model lemma learning and removal we consider the following extension of the Basic DPLL system.

Definition 2.5. The *DPLL system with learning*, denoted by L, consists of the four transition rules of the Basic DPLL system and the two additional rules:

Learn:

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models C. \end{cases}$$

Forget:

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \qquad \textbf{if} \begin{cases} F \models C. \end{cases}$$

In any application step of Learn, the clause C is said to be *learned* if it did not already belong to F. Similarly, it is said to be *forgotten* by Forget.

Observe that the Learn rule allows one to add to the current formula F an arbitrary clause C entailed by F, as long as all the atoms of C occur in F or M. This models not only conflict-driven lemma learning but also any other techniques that produce consequences of F, such as limited forms of resolution (see the following example).

Similarly, the Forget rule can be used in principle to remove from F *any* clause that is entailed by the rest of F, not just those previously added to the clause set by Learn. The applicability of the two rules in their full scope, however, is limited in practice by the relative cost of determining such entailments in general.

The six rules of the DPLL system with learning model the high-level conceptual structure of DPLL implementations. These rules will allow us to formally reason about properties such as correctness or termination.

Example 2.6. We now show how the Backjump rule can be guided by means of a *conflict graph* for finding backjump clauses. In this example we assume a strategy that is followed in most SAT solvers: (i) Decide is applied only if no other Basic DPLL rule is applicable (Theorem 5.2 of Section 5 shows that this is not needed, but here we require it for simplicity) and (ii) after each application of Backjump, the backjump clause is learned.

Consider a state of the form $M \parallel F$ where, among other clauses, F contains:

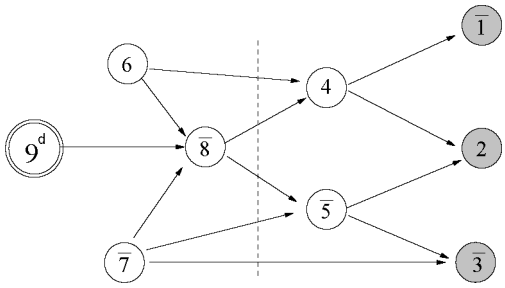
$$\overline{9} \vee \overline{6} \vee 7 \vee \overline{8} \quad 8 \vee 7 \vee \overline{5} \quad \overline{6} \vee 8 \vee 4 \quad \overline{4} \vee \overline{1} \quad \overline{4} \vee 5 \vee 2 \quad 5 \vee 7 \vee \overline{3} \quad 1 \vee \overline{2} \vee 3$$

and M is of the form: $\ldots 6 \ldots \overline{7} \ldots 9^{\mathsf{d}}\ \overline{8}\ \overline{5}\ 4\ \overline{1}\ 2\ \overline{3}$.

It is easy to see that this state can be reached after the last decision 9^{d} by six applications of UnitPropagate. For example, $\overline{8}$ is implied by 9, 6, and $\overline{7}$ because of the clause $\overline{9} \vee \overline{6} \vee 7 \vee \overline{8}$. A typical DPLL implementation will save the sequence of propagated literals and remember for each one of them the clause that caused its propagation. Now, in the state $M \parallel F$ above the clause $1 \vee \overline{2} \vee 3$ is conflicting, since M contains $\overline{1}$, 2 and $\overline{3}$. Using the saved information, the DPLL implementation can trace back the reasons for this conflicting clause. For example, the saved data will show that $\overline{3}$ was implied by $\overline{5}$ and $\overline{7}$, due to the clause $5 \vee 7 \vee \overline{3}$. The literal $\overline{5}$ was in turn implied by $\overline{8}$ and $\overline{7}$, and so on.

This way, *working backwards from the conflicting clause* and in the opposite order in which each literal was propagated, it is possible to build the following

conflict graph, where the nodes corresponding to the conflicting clause are shown in gray:



This figure shows the graph obtained when the decision literal of the current decision level (here, 9^{d}) is reached in this backwards process—which is why this node and the nodes belonging to earlier decision levels (in this example, literals 6 and $\overline{7}$) have no incoming arrows.

To find a backjump clause, it suffices to cut the graph into two parts. The first part must contain (at least) all the literals with no incoming arrows. The second part must contain (at least) all the literals with no outgoing arrows, that is, the negated literals of the conflicting clause (in our example, $\overline{1}$, 2 and $\overline{3}$). It is not hard to see that in such a cut no model of F can satisfy all the literals whose outgoing edges are cut.

For instance, consider the cut indicated by the dotted line in the graph, where the literals with cut outgoing edges are $\overline{8}$, $\overline{7}$, and 6. From these three literals, by unit propagation using five clauses of F, one can infer the negated literals of the conflicting clause. Hence, one can infer from F that $\overline{8}$, $\overline{7}$, and 6 cannot be simultaneously true, that is, one can infer the clause $8 \vee 7 \vee \overline{6}$. In this case, this is a possible backjump clause, that is, the clause $C' \vee l'$ in the definition of the Backjump rule, with the literal 8 playing the role of l'. The clause allows one to backjump to the decision level of $\overline{7}$ and add 8 to it. After that, the clause $8 \vee 7 \vee \overline{6}$ has to be learned to explain in future conflicts the presence of 8 as a propagation from 6 and $\overline{7}$.

The kind of cuts we have described produce backjump clauses provided that exactly one of the literals with cut outgoing edges belongs to the current decision level. The negation of this literal will act as the literal l' in the backjump rule. In the SAT literature, the literal is called a *Unique Implication Point (UIP)* of the conflict graph. Formally, UIPs are defined as follows. Let D be the set of all the literals of a conflicting clause C that have become false at the current decision level (this set is always nonempty, since Decide is applied only if Fail or Backjump do not apply). A UIP in the conflict graph of C is any literal that belongs to all paths in the graph from the current decision literal to the negation of a literal in D. Note that a conflict graph always contains at least one UIP, the decision literal itself, but in general it can contain more (in our example 9^{d} and $\overline{8}$ are both UIPs).

In practice, it is not actually necessary to build the conflict graph to produce a backjump clause; it suffices to work backwards from the conflicting clause, maintaining only a *frontier* list of literals yet to be expanded, until the *first* UIP (first in the reverse propagation ordering) has been reached [Marques-Silva and Sakallah 1999; Zhang et al. 2001].

The construction of the backjump clause can also be seen as a derivation in the resolution calculus, constructed according to the following *backwards conflict resolution* process. In our example, the clause $8 \vee 7 \vee \overline{6}$ is obtained by successive resolution steps on the conflicting clause, resolving away the literals $3, \overline{2}, 1, \overline{4}$ and 5, in the reverse order their negations were propagated, with the respective clauses that caused the propagations:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{5 \vee 7 \vee \overline{\mathbf{3}} \qquad 1 \vee \overline{2} \vee \mathbf{3}}{\overline{4} \vee 5 \vee \mathbf{2}} \qquad 5 \vee 7 \vee 1 \vee \overline{\mathbf{2}}}{\overline{4} \vee \overline{\mathbf{1}}} \qquad \overline{4} \vee 5 \vee 7 \vee \mathbf{1}}{\overline{6} \vee 8 \vee \mathbf{4}} \qquad 5 \vee 7 \vee \overline{\mathbf{4}}}{8 \vee 7 \vee \overline{\mathbf{5}}} \qquad \overline{6} \vee 8 \vee 7 \vee \mathbf{5}}{8 \vee 7 \vee \overline{6}}$$

The process stops once it generates a clause with only one literal of the current decision level, which is precisely the first UIP (in our example, the literal 8 in the clause $8 \vee 7 \vee \overline{6}$). Some SAT solvers, such as Siege, also learn some of the intermediate clauses in such resolution derivations [Ryan 2004].

2.5. CORRECTNESS OF DPLL WITH LEARNING. In this section, we show how the DPLL system with learning can be used as a decision procedure for the satisfiability of CNF formulas.

Deciding the satisfiability of an input formula F will be done by generating an arbitrary derivation of the form $\emptyset \parallel F \Longrightarrow_{\mathrm{L}} \cdots \Longrightarrow_{\mathrm{L}} S_n$ such that S_n is *final with respect to the Basic DPLL system*. Note that final states with respect to the DPLL system with Learning do not always exist, since the same clause could be learned and forgotten infinitely many times.

For all rules their applicability is easy to check and, as we will show in Theorem 2.11, if infinite subderivations with only Learn and Forget steps are avoided, one always reaches a state that is final with respect to the Basic DPLL system. This state S is moreover easily recognizable as final, because it is either *FailState* or of form $M \parallel F'$ where F' has no conflicting clauses and all of its literals are defined in M. Furthermore, similarly to the Classical DPLL system and as proved below in Theorem 2.12; in the first case, F is unsatisfiable, in the second case, it is satisfied by M.

We emphasize that these formal results apply to *any* procedure modeled by the DPLL system with learning, and can moreover be extended to DPLL Modulo Theories. This generalizes the less formal correctness proof for the concrete pseudo code of the Chaff algorithm given in Zhang and Malik [2003], which has the same underlying proof idea.

The starting point for our results is the next lemma, which lists a few properties that are invariant for all the states derived in the DPLL system with learning from initial states of the form $\emptyset \parallel F$.

LEMMA 2.7. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} M \parallel G$, *then the following hold.*

(1) *All the atoms in M and all the atoms in G are atoms of F.*
(2) *M contains no literal more than once and is indeed an assignment, that is, it contains no pair of literals of the form p and $\neg p$.*
(3) *G is logically equivalent to F.*

(4) *If M is of the form $M_0\ l_1\ M_1 \cdots l_n\ M_n$, where $l_1, \ldots, l_n$ are all the decision literals of M, then $F, l_1, \ldots, l_i \models M_i$ for all i in $0 \ldots n$.*

PROOF. Since all four properties trivially hold in the initial state $\emptyset \parallel F$, we only need to prove that all six rules preserve them. Consider a step $M' \parallel F' \Longrightarrow_{\mathrm{L}} M'' \parallel F''$ and assume all properties hold in $M' \parallel F'$. Property 1 holds in $M'' \parallel F''$ because the only atoms that may be added to M'' or F'' are the ones in F' or M', all of which belong to F. The side conditions of the rules clearly preserve Property 2. As for Property 3, only Learn and Forget may break the invariant. But learning (or forgetting) a clause C that is a logical consequence clearly preserves equivalence between F' and F''.

For the fourth property, consider that M' is of the form $M'_0\ l_1\ M'_1 \cdots l_n\ M'_n$, and $l_1, \ldots, l_n$ are all the decision literals of M'. If the step is an application of Decide, there is nothing to prove. For Learn or Forget, it easily follows since M' is M'' and F'' is logically equivalent to F'. The remaining rules are:

UnitPropagate: Since M'' will be of the form $M'l$ (we use l and C as in the definition of the rule), we only have to prove that $F, l_1, \ldots, l_n \models l$, which holds since (i) $F, l_1, \ldots, l_n \models M'$, (ii) $M' \models \neg C$, (iii) $C \vee l$ is a clause of F' and (iv) F and F' are equivalent.

Backjump: Assume that, in the Backjump rule, l^{d} is l_{j+1}, the $j+1$-th decision literal. Then (using l' and C' as in the definition of the rule), M'' is of the form $M'_0\ l_1\ M'_1 \cdots l_j\ M'_j\ l'$. We only need to show that $F, l_1, \ldots, l_j \models l'$. This holds as for the UnitPropagate case, since we have (i) $F, l_1, \ldots, l_j \models M'_0\ l_1\ M'_1 \cdots l_j\ M'_j$, (ii) $M'_0\ l_1\ M'_1 \cdots l_j\ M'_j \models \neg C'$, (iii) $F' \models C' \vee l'$ and (iv) F and F' are equivalent. □

The most interesting property of this lemma is probably Property 4. It shows that every nondecision literal added to an assignment M is a logical consequence of the previous decision literals of M and the initial formula F. In other words, we have that $F, l_1, \ldots, l_n \models M$. Hence, the only arbitrary additions to M are the ones made by Decide.

Another important property concerns the applicability of Backjump. Given a state with a conflicting clause, it may not be clear a priori whether Backjump is applicable or not, mainly due to the need to find an appropriate backjump clause. Below we show that, if there is a conflicting clause, it is always the case that either Backjump or Fail applies. Moreover, whenever the first precondition of Backjump holds ($M\ l^{\mathsf{d}}\ N \models \neg C$), a backjump clause $C' \vee l'$ always exists and can be easily computed.

LEMMA 2.8. *Assume that $\emptyset \parallel F \Longrightarrow^*_{\mathrm{L}} M \parallel F'$ and that $M \models \neg C$ for some clause C in F'. Then either Fail or Backjump applies to $M \parallel F'$.*

PROOF. If there is no decision literal in M, it is immediate that Fail applies. Otherwise, M is of the form $M_0\ l_1\ M_1 \cdots l_n\ M_n$ for some $n > 0$, where $l_1, \ldots, l_n$ are all the decision literals of M. Since $M \models \neg C$ we have, due to Lemma 2.7-4, that $F, l_1, \ldots, l_n \models \neg C$. If we now consider any i in $1 \cdots n$ such that $F, l_1, \ldots, l_i \models \neg C$, and any j in $0 \cdots i-1$ such that $F, l_1, \ldots, l_j, l_i \models \neg C$, we can show that then backjumping to decision level j is possible.

Let C' be the clause $\neg l_1 \vee \cdots \vee \neg l_j$, and note that M is also of the form $M'\ l_{j+1}\ N$. Then Backjump is applicable to $M \parallel F'$, yielding the state $M'\ \neg l_i \parallel F'$. That is because the clause $C' \vee \neg l_i$ satisfies all the side conditions of the Backjump rule:

(i) $F' \models C' \vee \neg l_i$ because $F, l_1, \ldots, l_j, l_i \models \neg C$, which implies, given that C is in F' and F' is equivalent to F (by Lemma 2.7-3), that $F, l_1, \ldots, l_j, l_i$ is unsatisfiable or, equivalently, that $F \models \neg l_1 \vee \cdots \vee \neg l_j \vee \neg l_i$; furthermore, $M' \models \neg C'$ by construction of C';

(ii) $\neg l_i$ is undefined in M' (by Lemma 2.7-2);

(iii) l_i occurs in M. □

It is interesting to observe that, the smaller one can choose the value j in the previous proof, the higher one can backjump. Note also that, if we construct the backjump clause as in the proof and take i to be n and j to be $n-1$ then the Backjump rule models standard backtracking.

We stress that backjump clauses need not be built as in the proof above, out of the decision literals of the current assignment. It follows from the termination and correctness results given in this section that in practice one is free to apply the backjump rule with *any* backjump clause. In fact, backjump clauses may be built to contain no decision literals at all, as is for instance possible in backjumping SAT solvers relying on the first UIP learning scheme illustrated in Example 2.6.

Given the previous lemma, it is easy to prove that final states with respect to Basic DPLL will be either *FailState* or $M \parallel F'$, where M is a model of the original formula F. More formally:

LEMMA 2.9. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} S$, *and S is final with respect to Basic DPLL, then S is either FailState, or it is of the form* $M \parallel F'$, *where*

(1) *all literals of* F' *are defined in* M,

(2) *there is no clause* C *in* F' *such that* $M \models \neg C$, *and*

(3) M *is a model of* F.

PROOF. Assume S is not *FailState*. If (1) does not hold, then S cannot be final, since Decide would be applicable. Similarly, for (2): by Lemma 2.8, either Fail or Backjump would apply. Together (1) and (2) imply that all clauses of F' are defined and true in M, and since by Lemma 2.7(3), F and F' are logically equivalent this implies that M is a model of F. □

We now prove termination of the Basic DPLL system.

THEOREM 2.10. *There are no infinite derivations of the form* $\emptyset \parallel F \Longrightarrow_{\mathrm{B}} S_1 \Longrightarrow_{\mathrm{B}} \cdots$.

PROOF. It suffices to define a well-founded strict partial ordering $\succ$ on states, and show that each step $M \parallel F \Longrightarrow_{\mathrm{B}} M' \parallel F$ is decreasing with respect to this ordering, that is, $M \parallel F \succ M' \parallel F$. Note that such an ordering must be entirely based on the first component of the states, because in this system without Learn and Forget the second component of states remains constant.

Let M be of the form $M_0\ l_1\ M_1 \cdots l_p\ M_p$, where $l_1, \ldots, l_p$ are all the decision literals of M. Similarly, let M' be $M'_0\ l'_1\ M'_1 \cdots l'_{p'}\ M'_{p'}$.

Let n be the number of distinct atoms (propositional variables) in F. By Lemma 2.7(1,2), we have that p, p' and the length of M and M' are always smaller than or equal to n.

For each assignment N, define $m(N)$ to be $n - \mathit{length}(N)$, that is, $m(N)$ is the number of literals "missing" in N for N to be total. Now define: $M \parallel F' \succ M' \parallel F''$ if

(i) there is some i with $0 \leq i \leq p$, p' such that

$$m(M_0) = m(M'_0), \ldots, m(M_{i-1}) = m(M'_{i-1}), \quad m(M_i) > m(M'_i) \text{ or}$$

(ii) $m(M_0) = m(M'_0), \ldots, m(M_p) = m(M'_p)$ and $m(M) > m(M')$.

Note that, in case (ii), we have $p' > p$, and all decision levels up to p coincide in number of literals. Comparing the number of missing literals in sequences is clearly a strict ordering (i.e., it is an irreflexive and transitive relation) and it is also well-founded, and hence this also holds for its lexicographic extension on tuples of sequences of bounded length. It is easy to see that all Basic DPLL rules are decreasing with respect to $\succ$ if *FailState* is added as an additional minimal element. The rules UnitPropagate and Backjump decrease by case (i) of the definition and Decide decreases by case (ii). □

It is nice to see in this proof that, in contrast to the classical, depth-first DPLL procedure, progress in backjumping DPLL procedures is not measured by the number of decision literals that have been *tried* with both truth values, but by the number of defined literals that are added to earlier decision levels. The Backjump rule makes progress in this sense by increasing by one the number of defined literals in the decision level it backjumps to. The lower this decision level is (i.e., the higher up in the depth-first search tree), the more progress is made with respect to $\succ$.

As an immediate consequence of this theorem, we obtain the termination of the DPLL system with learning if infinite subderivations with only Learn and Forget steps are avoided. The reason is that the other steps (the Basic DPLL ones) decrease the first components of the states with respect to the well-founded ordering, while the Learn and Forget steps do not modify that component.

THEOREM 2.11. *Every derivation* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}} S_1 \Longrightarrow_{\mathrm{L}} \cdots$ *by the DPLL system with Learning is finite if it contains no infinite subderivations consisting of only* Learn *and* Forget *steps.*

Note that this condition is very weak and easily enforced. Learn is typically only applied together with Backjump in order to learn the corresponding backjump clause. The theorem entails that such a strategy eventually reaches a state where only Learn and/or Forget apply, that is, a state that is final with respect to the Basic DPLL system. As already mentioned, by Lemma 2.9, this state is moreover easily recognizable because it is *FailState* or else it has the form $M \parallel G$ with all literals of G defined in M and no conflicting clause.

Actually, we could have alternatively defined a state $M \parallel G$ to be final if M is a *partial* assignment satisfying all clauses of G, hence allowing some literals of G to remain undefined. Then the correctness argument would have been exactly the same but without the use of Lemma 2.9—which now is needed mostly to show that the current definition of a final state $M \parallel G$ is a sufficient condition for M to be a model of G. However, in typical DPLL implementations, checking each time whether a partial assignment is a model of the current formula G is more expensive, because of the necessary additional bookkeeping, than just extending a partial model of G to a total one, which can be done with no search. But note that things may be different in the SMT case (see a brief discussion at the end of Section 3), or when the goal is to enumerate *all* models (perhaps in some compact representation) of the initial formula F.

We are now ready to prove that DPLL with learning provides a decision procedure for the satisfiability of CNF formulas.

THEOREM 2.12. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} S$ *where* S *is final with respect to Basic DPLL, then*

(1) S *is FailState if, and only if,* F *is unsatisfiable.*
(2) *If* S *is of the form* $M \parallel F'$ *then* M *is a model of* F.

PROOF. For Property 1, if S is *FailState* it is because there is some state $M \parallel F'$ such that $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} M \parallel F' \Longrightarrow_{\mathrm{L}} \mathit{FailState}$. By the definition of the Fail rule, there is no decision literal in M and there is a clause C in F' such that $M \models \neg C$. Since F and F' are equivalent by Lemma 2.7(3), we have that $F \models C$. However, if $M \models \neg C$, by Lemma 2.7(4), then also $F \models \neg C$, which implies that F is unsatisfiable. For the right-to-left implication, if S is not *FailState* it has to be of the form $M \parallel F'$. But then, by Lemma 2.9(3), M is a model of F and hence F is satisfiable.

For Property 2, if S is $M \parallel F'$, then, again by Lemma 2.9(3), M is a model of F. □

Note that the previous theorem does not guarantee confluence in the sense of rewrite systems, say. With unsatisfiable formulas, the only possible final (with respect to Basic DPLL) state for a sequence is *FailState*. If, on the other hand, the formula is satisfiable, different states that are final with respect to Basic DPLL may be reachable. However, all of them will be of the form $M \parallel F'$, with M a model of the original formula.

Although Theorem 2.12 was given for the relation $\Longrightarrow_{\mathrm{L}}$, it also holds for $\Longrightarrow_{\mathrm{B}}$, since the existence of Learn or Forget is not required in the proof.

THEOREM 2.13. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{B}}^{*} S$ *where* S *is final with respect to Basic DPLL, then*

(1) S *is FailState if, and only if,* F *is unsatisfiable.*
(2) *If* S *is of the form* $M \parallel F'$*, then* M *is a model of* F.

2.6. ABOUT PRACTICAL IMPLEMENTATIONS AND RESTARTS. State-of-the art SAT-solvers [Moskewicz et al. 2001; Goldberg and Novikov 2002; Eén and Sörensson 2003; Ryan 2004] essentially apply Abstract DPLL with Learning using efficient implementation techniques for UnitPropagate (such as the two-watched literal scheme for unit propagation [Moskewicz et al. 2001]), and good heuristics for selecting the decision literal when applying the Decide rule. As said, conflict analysis procedures for applying Backjump and the possibility of applying learning by other forms of resolution have also been well studied.

In addition, modern DPLL implementations *restart* the DPLL procedure whenever the search is not making enough progress according to some measure. The rationale behind this idea is that upon each restart, the additional knowledge of the search space compiled into the newly learned lemmas will lead the heuristics for Decide to behave differently, and possibly cause the procedure to explore the search space in a more compact way. The combination of learning and restarts has been shown to be powerful not only in practice, but also in theory. Essentially, any Basic DPLL derivation to *FailState* is equivalent to a *tree-like* refutation by resolution. But for some classes of problems tree-like proofs are always exponentially larger

than the smallest *general*, that is, DAG-like, resolution ones [Bonet et al. 2000]. The good news is that DPLL with learning and restarts becomes again equivalent to general resolution with respect to such notions of proof complexity [Beame et al. 2003].

In our formalism, restarts can be simply modeled by the following rule:

Definition 2.14. The Restart rule is:

$$M \parallel F \implies \emptyset \parallel F.$$

Adding the Restart rule to DPLL with Learning, it is obvious that all results of this section hold as long as one can ensure that a final state with respect to Basic DPLL is eventually reached. This is usually done in practice by periodically increasing the minimal number of Basic DPLL steps between each pair of restart steps. This is formalized below.

Definition 2.15. Consider a derivation by the DPLL system with learning extended with the Restart rule. We say that Restart *has increasing periodicity* in the derivation if, for each subderivation $S_i \implies \cdots \implies S_j \implies \cdots \implies S_k$ where the steps producing S_i, S_j, and S_k are the only Restart steps, the number of Basic DPLL steps in $S_i \implies \cdots \implies S_j$ is strictly smaller than in $S_j \implies \cdots \implies S_k$.

THEOREM 2.16. *Any derivation* $\emptyset \parallel F \implies S_1 \implies \cdots$ *by the transition system* L *extended with the* Restart *rule is finite if it contains no infinite subderivations consisting of only* Learn *and* Forget *steps, and* Restart *has increasing periodicity in it.*

PROOF. By contradiction, assume Der is an infinite derivation fulfilling the requirements. Let $\succ$ be the well-founded ordering on (the first components of) states defined in the proof of Theorem 2.10. In a subderivation of Der without Restart steps, at each step either this first component decreases with respect to $\succ$ (by the Basic DPLL steps) or it remains equal (by the Learn and Forget steps). Therefore, since there is no infinite subderivation consisting of only Learn and Forget steps, there must be infinitely many Restart steps in Der. Also, if between two states there is at least one Basic DPLL step and no Restart step, these states do not have the same first component. Therefore, if n denotes the (fixed, finite) number of different first components of states that exist for the given finite set of propositional symbols, there cannot be any subderivations with more than n Basic DPLL steps between two Restart steps. This contradicts the fact that there are infinitely many Restart steps if Restart has increasing periodicity in Der. □

In conclusion, in this section, we have formally described a large family of practical implementations of DPLL with learning and restarts, and proved that they provide a decision procedure for propositional satisfiability.

3. *Abstract DPLL Modulo Theories*

For many applications, encoding the problems into propositional logic is not the right choice. Frequently, a better alternative is to express the problems in a richer non-propositional logic, considering satisfiability with respect to a background theory T.

For example, some properties of timed automata are naturally expressed in *Difference Logic*, where formulas contain atoms of the form $a - b \leq k$, which are interpreted with respect to a background theory T of the integers, rationals or reals [Alur 1999]. Similarly, for the verification of pipelined microprocessors it is convenient to consider a logic of *Equality with Uninterpreted Functions (EUF)*, where the background theory T specifies a congruence [Burch and Dill 1994]. To mention just one further example, the conditions arising from program verification usually involve arrays, lists and other data structures, so it becomes very natural to consider satisfiability problems *modulo* the combined theory T of these data structures. In such applications, typical formulas consist of large sets of clauses such as:

$$p \quad \vee \quad \neg q \quad \vee \quad a = f(b - c) \quad \vee \quad read(s,\ f(b - c)) = d \quad \vee \quad a - g(c) \leq 7$$

containing purely propositional atoms as well as atoms over the combined theory. This is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory T: given a formula F, determine whether F is T-satisfiable, that is, whether there exists a model of T that is also a model of F.

In this section, we show that many of the existing techniques for handling SMT, of which SAT is a particular case if we consider T to be the empty theory, can be described and discussed within the Abstract DPLL framework.

3.1. Formal Preliminaries on Satisfiability Modulo Theories. Throughout this section, we consider the same definitions and notation given in Section 2 for the propositional case, except that here the set P over which formulas are built is a fixed finite set of *ground* (i.e., variable-free) first-order atoms, instead of propositional symbols.

In addition to these propositional notions, here we also consider some notions of first-order logic (see e.g., Hodges [1993]). A *theory* T is a set of closed first-order formulas. A formula F is *T-satisfiable* or *T-consistent* if $F \wedge T$ is satisfiable in the first-order sense. Otherwise, it is called *T-unsatisfiable* or *T-inconsistent*.

As in the previous section, a partial assignment M will sometimes also be seen as a conjunction of literals and hence as a formula. If M is a T-consistent partial assignment and F is a formula such that $M \models F$, that is, M is a (propositional) model of F, then we say that *M is a T-model of F*. If F and G are formulas, then *F entails G in T*, written $F \models_T G$, if $F \wedge \neg G$ is T-inconsistent. If $F \models_T G$ and $G \models_T F$, we say that F and G are *T-equivalent*. A *theory lemma* is a clause C such that $\emptyset \models_T C$.

The SMT problem for a theory T is the problem of determining, given a formula F, whether F is T-satisfiable, or, equivalently, whether F has a T-model.

As usual in SMT, given a background theory T, we will only consider the SMT problem for *ground* (and hence quantifier-free) CNF formulas F. Such formulas may contain *free* constants, that is, constant symbols not in the signature of T, which, as far as satisfiability is concerned, can be equivalently seen as existential variables. Other than free constants, all other predicate and function symbols in the formulas will instead come from the signature of T. From now on, when we say formula we will mean a formula satisfying these restrictions.

We will consider here only theories T such that the T-satisfiability of conjunctions of such ground literals is decidable. We will call any decision procedure for this problem a *T-solver*.

3.2. AN INFORMAL PRESENTATION OF SMT PROCEDURES. The current techniques for deciding the satisfiability of a ground formula F with respect to a background theory T can be broadly divided into two main categories: *eager* and *lazy*.

3.2.1. *Eager SMT Techniques.* In eager techniques, the input formula is translated using a satisfiability-preserving transformation into a propositional CNF formula which is then checked by a SAT solver for satisfiability (see, e.g., Bryant et al. [2001], Bryant and Velev [2002], and Strichman [2002]).

One of the strengths of this eager approach is that it can always use the best available SAT solver off the shelf. When the new generation of efficient SAT solvers such as Chaff [Moskewicz et al. 2001] became available, impressive results using the eager SMT approach were achieved by Bryant's group at CMU with the solver *UCLID* [Lahiri and Seshia 2004] for the verification of pipelined processors.

However, eager techniques are not very flexible: to make them efficient, sophisticated ad-hoc translations are required for each theory. For example, for EUF and for Difference Logic there exist the *per-constraint* encoding [Bryant and Velev 2002; Strichman et al. 2002], the *small domain* encoding (or *range-allocation* techniques), [Pnueli et al. 1999; Bryant et al. 2002; Talupur et al. 2004; Meir and Strichman 2005], and several hybrid approaches [Seshia et al. 2003]. The eager encoding approach can also handle integer linear arithmetic and the theory of arrays (see Seshia [2005]).

In spite of the effort spent in devising efficient translations, on many practical problems the translation process or the SAT solver run out of time or memory (see de Moura and Ruess [2004]). The current alternative techniques explained below are in many cases several orders of magnitude faster.

The correctness of the eager approach for SMT relies on the correctness of both the SAT solver and the translation, which is specific for each theory. It is out of the scope of this article to discuss the correctness of these ad-hoc translations. Assuming them to be correct, the correctness of the eager techniques follows from the results of Section 2.

3.2.2. *Lazy SMT Techniques.* As an alternative to the eager approach, one can use a specialized T-solver for deciding the satisfiability of conjunctions of theory literals. Then, a decision procedure for SMT is easily obtained by converting the given formula into disjunctive normal form (DNF) and using the T-solver to check whether any of the DNF conjuncts is satisfiable. However, the exponential blowup usually caused by the conversion into DNF makes this approach too inefficient.

A lot of research has then looked into ways to combine the strengths of specialized T-solvers with the strengths of state-of-the-art SAT solvers in dealing with the Boolean structure of formulas. The most widely used approach in the last few years is usually referred to as the *lazy* approach [Armando et al. 2000; Filliâtre et al. 2001; Audemard et al. 2002; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Armando et al. 2004; Ball et al. 2004]. In this approach, each atom occurring in a formula F to be checked for satisfiability is initially considered simply as a propositional symbol, *forgetting* about the theory T. Then the formula is given to a SAT solver. If the SAT solver determines it to be (propositionally) unsatisfiable, then F is T-unsatisfiable as well. If the SAT solver returns instead a propositional model M of F, then this assignment (seen as a conjunction of literals) is checked by a T-solver. If M is found T-consistent then it is a T-model of F. Otherwise, the T-solver builds a ground clause that is a logical consequence of T, that is, a theory

lemma, precluding that assignment. This lemma is added to F and the SAT solver is started again. This process is repeated until the SAT solver finds a T-model or returns unsatisfiable.

Example 3.1. Assume we are deciding with a lazy procedure the T-satisfiability of a large EUF formula, where T is the theory of equality, and assume that the model M found by the SAT solver contains, among many others, the four literals:

$$b=c, \quad f(b)=c, \quad a\neq g(b), \quad g(f(c))=a.$$

Then the T-solver detects that M is not a T-model, since

$$b=c \ \wedge \ f(b)=c \ \wedge \ g(f(c))=a \quad \models_T \quad a=g(b).$$

Therefore, the lazy procedure has to be restarted after the corresponding theory lemma has been added to the clause set. In principle, one can take as theory lemma simply the negation of M, that is, the disjunction of the negations of all the literals in M. However, this is usually not a good idea as the generated clause may end up containing thousands of literals. Lazy procedures are much more efficient if the T-solver is able instead to generate a small *explanation* of the T-inconsistency of M. In this example, the explanation could be simply the clause $b\neq c \ \vee \ f(b)\neq c \ \vee \ g(f(c))\neq a \ \vee \ a=g(b)$.

The main advantage of the lazy approach is its flexibility, since it can easily combine any SAT solver with any T-solver. More importantly, if the SAT solver used by the lazy SMT procedure is based on DPLL, then several refinements exist that make the SMT procedure much more efficient. Here we outline the most significant ones.

Incremental T-solver. The T-consistency of the assignment can be checked incrementally, while the assignment is being built by the DPLL procedure, without delaying the check until a propositional model has been found. This can save a large amount of useless work. It can be done fully eagerly, detecting T-inconsistencies as soon as they are generated, or, if that is too expensive, at regular intervals, for example, once every k literals added to the assignment. The idea was already mentioned in Audemard et al. [2002] under the name of *early pruning* and in Barrett [2003] under the name of *eager notification*. Currently, most SMT implementations work with incremental T-solvers. The incremental use of T-solvers poses different requirements on their implementation: to make the incremental approach effective in practice, the solver should (on average, say) be faster in processing one additional input literal l than in re-processing from scratch all previous inputs and l together. For many theories this can indeed be done; see, for example, Section 4.3, where we describe an incremental solver for Difference Logic.

On-line SAT Solver. When a T-inconsistency is detected by the incremental T-solver, one can ask the DPLL procedure simply to backtrack to some point where the assignment was still T-consistent, instead of restarting the search from scratch. For instance, if, in Abstract DPLL terms, the current state is of the form $M \parallel F$ and M has been detected to be T-inconsistent, then there is some subset $\{l_1 \cdots l_n\}$ of M such that $\neg l_1 \vee \cdots \vee \neg l_n$ is a theory lemma. This lemma can be added to the clause set, and, since it is conflicting, that is, it is false in M, Backjump or Fail can be applied. As we will formally prove below, after the backjump step this lemma is no longer needed for completeness and could be safely forgotten: the procedure

will search through all propositional models, finding a T-consistent one whenever it exists. Nevertheless, keeping theory lemmas can still be very useful for efficiency reasons, because it may cause an important amount of pruning later in the search. Theory lemmas are especially effective if they are small, as observed in, for example, de Moura and Rueß [2002] and Barrett [2003]. On-line SAT solvers (in combination with incremental T-solvers) are now common in SMT implementations, and state-of-the-art SAT solvers like zChaff or MiniSAT provide this functionality.

Theory Propagation. In the approach presented so far, the T-solver provides information only *after* a T-inconsistent partial assignment has been generated. In this sense, the T-solver is used only to *validate* the search *a posteriori*, not to *guide* it a priori. To overcome this limitation, the T-solver can also be used in a given DPLL state $M \parallel F$ to detect literals l occurring in F such that $M \models_T l$, allowing the DPLL procedure to move to the state $M\ l \parallel F$. We call this process *theory propagation*.

The idea of theory propagation was first mentioned in Armando et al. [2000] under the name of *Forward Checking Simplification*, and since then it has been applied, in limited form, in very few other systems (see Section 5). In contrast, theory propagation plays a major role in the DPLL(T) approach, introduced in Section 4 of this article. There we show that, somewhat against expectations, practical T-solvers can be designed to include this feature in an efficient way. A highly non-trivial issue is how to perform conflict analysis appropriately in the context of theory propagation. Different options and possible problems for doing this are analyzed and solved in detail in Section 5, something that, to our knowledge, had not been done before. In Section 6, we show that theory propagation, if handled well, has a crucial impact on the performance of SMT systems.

Exhaustive Theory Propagation. For some theories, it even pays off to perform *all* possible Theory Propagations before applying the Decide rule. This idea of exhaustive theory propagation is also introduced in the DPLL(T) approach presented here.

Lazy techniques that learn theory lemmas and do not perform any theory propagation in effect dump a large number of ground consequences of the theory into the clause set, duplicating theory information into the SAT solver. This duplication is instead completely unnecessary in a system with exhaustive theory propagation—and is greatly reduced with non-exhaustive theory propagation.

The reason is that any literal generated by unit propagation over a theory lemma can also be generated by theory propagation.[1]

For some logics, such as Difference Logic, for instance, exhaustive theory propagation usually yields speedups of several orders of magnitude, as we show in Section 6.

Using an incremental T-solver in combination with an on-line SAT solver is known to be crucial for efficiency. Possibly with the only exception of Verifun [Flanagan et al. 2003], an experimental system no longer under development, most, if not all, state-of-the-art SMT systems use incremental solvers. On the other hand, only a few SMT systems so far use theory propagation, as we will discuss in Section 5.

[1] But see the discussion about strategies with lazier theory propagation at the end of Section 5.1.

3.3. ABSTRACT DPLL MODULO THEORIES. In this section, we formalize the different enhancements of the lazy approach to Satisfiability Modulo Theories. We do this by adapting the Abstract DPLL framework for the propositional case presented in the previous section. One significant difference is that here we deal with ground first-order literals instead of propositional ones. Except for that, the rules Decide, Fail, UnitPropagate, and Restart remain unchanged: they will still regard all literals as syntactical items as in the propositional case. Only Learn, Forget and Backjump are slightly modified to work modulo theories: in these rules, entailment between formulas now becomes entailment in T. In addition, atoms of T-learned clauses can now also belong to M, and not only to F; this is required for Property 3.9 below, needed to recover from T-inconsistent states. Note that the theory version of Backjump below uses both the propositional notion of satisfiability ($\models$) and the first-order notion of entailment modulo theory ($\models_T$).

Definition 3.2. The rules T-Learn, T-Forget and T-Backjump are:

T-Learn:

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{cases}$$

T-Forget:

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \textbf{if} \begin{cases} F \models_T C \end{cases}$$

T-Backjump:

$$M\ l^{\mathsf{d}}\ N \parallel F, C \quad \Longrightarrow \quad M\ l' \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models_T C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N. \end{cases}$$

3.3.1. *Modeling the Naive Lazy Approach.* Using these rules, it is easy to model the basic lazy approach (without any of the refinements of incremental T-solvers, on-line SAT solvers or theory propagation). Each time a state $M \parallel F$ is reached that is final with respect to Decide, Fail, UnitPropagate, and T-Backjump, that is, final in a similar sense as in the previous section, M can be T-consistent or not. If it is, then M is indeed a T-model of F, as we will prove below. If M is not T-consistent, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. By one T-Learn step, the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$ can be learned and then Restart can be applied. As we will prove below, if these learned theory lemmas are never removed by the T-Forget rule, this strategy is terminating under similar requirements as those in the previous section, namely, the absence of infinite subderivations consisting of only Learn and Forget steps and the increasing periodicity of Restart steps. Then, the strategy is also sound and complete as stated in the previous section: the initial formula is T-unsatisfiable if, and only if, *FailState* is reached; moreover, if *FailState* is not reached then a T-model has been found.

3.3.2. *Modeling the Lazy Approach with an Incremental T-Solver.* Assume a state $M \parallel F$ has been reached where M is T-inconsistent. Note that in practice this

is detected by the incremental T-solver, and that this state need not be final now. Then, as in the naive lazy approach, there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. This theory lemma is then learned, producing the state $M \parallel F,\ \neg l_1 \vee \cdots \vee \neg l_n$. As in the previous case, Restart can then be applied and the same results hold.

3.3.3. *Modeling the Lazy Approach with an Incremental T-Solver and an on-Line SAT Solver.* As in the previous case, if a subset $\{l_1, \ldots, l_n\}$ of M is detected such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$, the theory lemma is learned, reaching the state $M \parallel F,\ \neg l_1 \vee \cdots \vee \neg l_n$. But now, since in addition we consider an online SAT solver, instead of completely restarting, the procedure *repairs* the T-inconsistency of the partial assignment by exploiting the fact that the recently learned theory lemma is a conflicting clause. As we show later, and similarly to what happened in the propositional case, if there is no decision literal in M then Fail applies, otherwise T-Backjump applies. Our results below prove that, even if the theory lemma is always forgotten immediately after backjumping, this approach is terminating, sound, and complete under similar conditions as the ones of the previous section.

3.3.4. *Modeling the Previous Refinements and Theory Propagation.* This requires the following additional rule:

Definition 3.3. The TheoryPropagate rule is:

$$M \parallel F \implies M\ l \parallel F \quad \textbf{if} \quad \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M. \end{cases}$$

The purpose of this rule is to prune the search by assigning a truth value to literals that are (propositionally) undefined by the current assignment M but T-entailed by it, rather than letting the Decide rule guess a value for them. As said, this sort of propagation can lead to dramatic improvements in performance. Below we prove that the correctness results mentioned for the previous three lazy approaches also hold in combination with arbitrary applications of this rule.

3.3.5. *Modeling the Previous Refinements and Exhaustive Theory Propagation.* Exhaustive theory propagation is modeled simply by assuming that TheoryPropagate is applied with a higher priority than Decide. The correctness of this approach follows immediately from the correctness of the previous one which had arbitrary applications of TheoryPropagate.

3.4. CORRECTNESS OF ABSTRACT DPLL MODULO THEORIES. Up to now, we have seen several different application strategies of (subsets) of the given rules, which lead to different SMT procedures. In this subsection we give a simple and uniform proof showing that all the approaches described in the previous subsection are indeed decision procedures for the SMT problem. The proofs are structured in the same way as the ones given in Section 2.5 for the propositional case, and hence here we focus on the variations and extensions that are needed.

Definition 3.4. The *Basic DPLL Modulo Theories system* consists of the rules Decide, Fail, UnitPropagate, TheoryPropagate, and T-Backjump.

Definition 3.5. The *Full DPLL Modulo Theories system*, denoted by FT, consists of the rules of Basic DPLL Modulo Theories and the rules T-Learn, T-Forget, and Restart.

As before, a decision procedure will be obtained by generating a derivation using the given rules with a particular strategy. However, here the aim of a derivation is to compute a state S to which the main theorem of this section, Theorem 3.10, can be applied, that is, a state S such that: (i) S is final with respect to the rules of Basic DPLL Modulo Theories and (ii) if S is of the form $M \parallel F$ then M is T-consistent.

Property 3.9 below provides a very general class of strategies in which such a state S is always reached, without violating the requirements of termination of Theorem 3.7 (also given below). Such a state S can be recognized in a similar way as in the propositional case: it is either *FailState* or it is of the form $M \parallel F$ where all the literals of F are defined in M, there are no conflicting clauses, and M is T-consistent.

The following lemma states invariants similar to the ones of Lemma 2.7 of the previous section.

LEMMA 3.6. *If* $\emptyset \parallel F \Longrightarrow_{\text{FT}}^{*} M \parallel G$, *then the following hold:*

(1) *All the atoms in* M *and all the atoms in* G *are atoms of* F.
(2) M *contains no literal more than once and is indeed an assignment, that is, it contains no pair of literals of the form* p *and* $\neg p$.
(3) G *is* T*-equivalent to* F.
(4) *If* M *is of the form* $M_0\ l_1\ M_1\ \cdots\ l_n\ M_n$, *where* $l_1, \ldots, l_n$ *are all the decision literals of* M, *then* $F, l_1, \ldots, l_i \models_T M_i$ *for all* i *in* $0 \cdots n$.

PROOF. As for Lemma 2.7, all rules preserve the properties. The new rule TheoryPropagate preserves them like UnitPropagate; the other rules as for their propositional versions. □

THEOREM 3.7 (TERMINATION). *Let* Der *be a derivation of the form:*
$\emptyset \parallel F = S_0 \Longrightarrow_{\text{FT}} S_1 \Longrightarrow_{\text{FT}} \cdots$
Then Der *is finite if the following two conditions hold:*

(1) Der *has no infinite subderivations consisting of only* T*-Learn and* T*-Forget steps.*
(2) *For every subderivation of* Der *of the form:*
$S_{i-1} \Longrightarrow_{\text{FT}} S_i \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_j \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_k$
where the only three Restart *steps are the ones producing* S_i, S_j, *and* S_k, *either:*
—*there are more Basic DPLL Modulo Theories steps in* $S_j \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_k$ *than in* $S_i \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_j$, *or*
—*a clause is learned*[2] *in* $S_j \Longrightarrow_{\text{FT}} \cdots \Longrightarrow_{\text{FT}} S_k$ *that is not forgotten in* Der.

PROOF. The proof is a slight extension of the one of Theorem 2.16. The only new aspect is that some Restart steps are applied with non-increasing periodicity. But since for each one of them a new clause has been learned that is never forgotten in Der, there can only be finitely many of them. From this, a contradiction follows as in Theorem 2.16. □

[2] See Definition 2.5.

LEMMA 3.8. *If* $\emptyset \parallel F \Longrightarrow^{*}_{\mathrm{FT}} M \parallel F'$ *and there is some conflicting clause in* $M \parallel F'$*, that is,* $M \models \neg C$ *for some clause* C *in* F'*, then either* Fail *or* T*-*Backjump *applies to* $M \parallel F'$*.*

PROOF. As in Lemma 2.8. □

PROPERTY 3.9. *If* $\emptyset \parallel F \Longrightarrow^{*}_{\mathrm{FT}} M \parallel F'$ *and* M *is* T*-inconsistent, then either there is a conflicting clause in* $M \parallel F'$*, or else* T*-*Learn *applies to* $M \parallel F'$*, generating a conflicting clause.*

PROOF. If M is T-inconsistent, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. Hence, the conflicting clause $\neg l_1 \vee \cdots \vee \neg l_n$ is either in $M \parallel F'$, or else it can be learned by one T-Learn step. □

Lemma 3.8 and Property 3.9 show that a rule of Basic DPLL modulo theories is always applicable to a state of the form $M \parallel F$, or to its successor after a single T-Learn step, whenever a literal of F is undefined in M, or F contains a conflicting clause, or M is T-inconsistent. Together with Theorem 3.7 (Termination), this shows how to compute a state to which the following main theorem is applicable.

THEOREM 3.10. *Let* Der *be a derivation* $\emptyset \parallel F \Longrightarrow^{*}_{\mathrm{FT}} S$*, where (i)* S *is final with respect to Basic DPLL Modulo Theories, and (ii) if* S *is of the form* $M \parallel F'$ *then* M *is* T*-consistent. Then*

(1) *S is FailState if, and only if, F is T-unsatisfiable.*

(2) *If S is of the form $M \parallel F'$, then M is a T-model of F.*

PROOF. The first result follows from Lemmas 3.6(3), 3.6(4), as in Theorem 2.12. The second part is proved as in Lemma 2.9 of the previous section, but using Lemma 3.8 and Lemma 3.6(3), instead of Lemma 2.8 and Lemma 2.7(3). □

The previous theorem shows that a large family of practical approaches provide a decision procedure for satisfiability modulo theories. Note that the results of this section are independent from the theory T under consideration, the only (obviously necessary) requirement being the decidability of the T-consistency of conjunctions of ground literals.

We conclude this section by observing that, as in the propositional case, our definition of final state for Abstract DPLL Modulo Theories forces the assignment M in a state of the form $M \parallel G$ to be total. We remarked in the previous section that the alternative definition of final state where M can be partial as long as it satisfies G is inefficient in practice in the SAT case. With theories, however, this is not always true. Depending on the theory T and the available T-solver, it may be considerably more expensive to insist on extending a satisfying partial assignment to a total one than to check periodically whether the current assignment has become a model of the current formula. The reason is that by Theorem 3.10 one can stop the search with a final state $M \parallel G$ only if M is also T-consistent, and T-consistency checks can have a high cost, especially when the T-satisfiability of conjunction of literals is NP-hard. We have maintained the same definition of final state for both Abstract DPLL and Abstract DPLL Modulo Theories mainly for simplicity, to make the lifting of the former to the latter clearer. We stress though that as in

the previous section essentially the same correctness proof applies if one uses the alternative definition of final state in this section.

4. *The DPLL(T) Approach*

We have now seen an abstract framework that allows one to model a large number of complete and terminating strategies for SMT. In this section, we describe the DPLL(T) approach for Satisfiability Modulo Theories, a general modular architecture on top of which actual implementations of such SMT strategies can be built. This architecture is based on a general DPLL engine, called DPLL(X), that is not dependent on any particular theory T. Instead, it is parameterized by a solver for a theory T of interest. A DPLL(T) system for T is produced by instantiating the parameter X with a module $Solver_T$ that can handle conjunctions of literals in T, that is, a T-solver.

The basic idea is similar to the $CLP(X)$ scheme for constraint logic programming [Jaffar and Maher 1994]: provide a clean and modular, but at the same time efficient, integration of specialized theory solvers within a general-purpose engine, in our case one based on DPLL.

The DPLL(T) architecture presented here combines the advantages of the eager and lazy approaches to SMT. On the one hand, the architecture allows for very efficient implementations, as witnessed by our system, Barcelogic, which implements DPLL(T) for a number of theories and compares very favorably with other SMT systems—see Section 6. On the other hand, DPLL(T) has the flexibility of the lazy approaches: more general logics can be dealt with by simply plugging in other solvers into the general DPLL(X) engine, provided that these solvers conform to a minimal interface.

4.1. Overall Architecture of DPLL(T).

At each state $M \parallel F$ of a derivation, the DPLL(X) engine knows M and F, but it treats all literals and clauses as purely propositional ones. As a consequence, all the needed theory-based inferences are exclusively localized in the theory solver $Solver_T$, which knows M but not the current F.

For the purposes of this article, it is not necessary to precisely define the interface between DPLL(X) and $Solver_T$. It suffices to know that $Solver_T$ provides operations that can be called by DPLL(X) to:

—Notify $Solver_T$ that a certain literal has been set to true.

—Ask $Solver_T$ to check whether the current partial assignment M, as a conjunction of literals, is T-inconsistent. This request can be made by DPLL(X) with different degrees of *strength*: for theories where deciding T-inconsistency is in general expensive, it might be more convenient to use cheaper, albeit incomplete, T-inconsistency checks for most of the derivation, and resort to a more expensive but complete check only when necessary.[3]

It is required that when $Solver_T$ detects a T-inconsistency it is also able to identify a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. This theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$, which we will call the *(theory) explanation* of the T-inconsistency, is then communicated by $Solver_T$ to the engine.

[3] Note that, according to the correctness results of Abstract DPLL modulo theories, a *decision* of T-inconsistency is only needed when a final state with respect to the Basic DPLL rules is reached.

—Ask $Solver_T$ to identify currently undefined input literals that are T-consequences of M. Again, this request can be made by DPLL(X) with different degrees of strength. $Solver_T$ answers with a (possibly empty) list of literals of the input formula that are newly detected T-consequences of M. Note that for this operation $Solver_T$ needs to know the set of input literals.

—Ask $Solver_T$ to provide a justification for the T-entailment of some theory propagated literal l. This is needed for the following reasons. In a concrete implementation of the DPLL(X) engine, backjumping is typically guided by a conflict graph, as explained in Example 2.6. But there is a difference with respect to the purely propositional case: a literal l at a node in the graph can now also be due to an application of theory propagation. Hence, building the graph requires that $Solver_T$ be able to recover and return as a justification of l a (preferably small, non-redundant) subset $\{l_1, \ldots, l_n\}$ of literals of the assignment M that T-entailed l when l was T-propagated. Computing that subset amounts to generating the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n \vee l$. We will call this lemma the *(theory) explanation* of l. (See Example 5.1, and also Section 4.3 and Section 5 for more details and refinements.)

—Ask $Solver_T$ to undo the last n notifications that a literal has been set to true.

In the rest of this section, we describe two concrete SMT strategies for the Abstract DPLL modulo theories framework, and show how they can be implemented using the DPLL(T) architecture.

The first one, described in Section 4.2, performs exhaustive theory propagation in a very eager way: in a state $M \parallel F$, TheoryPropagate is immediately applied whenever some input literal l is T-entailed by M. Therefore, $Solver_T$ is required to detect *all* such entailments immediately after a literal is set to true. In contrast, the second DPLL(T) system, described in Section 4.4, allows $Solver_T$ to sometimes fail to detect some entailed literals.

Each system is accompanied by a concrete motivating example of a theory of practical relevance, namely Difference Logic and EUF Logic, respectively. For Difference Logic, an efficient design for $Solver_T$ is described in Section 4.3.

Further refinements of theory propagation and conflict-driven clause learning are discussed in more detail in Section 5.

4.2. DPLL(T) WITH EXHAUSTIVE THEORY PROPAGATION AND DIFFERENCE LOGIC. Here, we deal with a particular application strategy of the rules of Abstract DPLL Modulo Theories modeling exhaustive theory propagation. We show how it can be implemented using the DPLL(T) architecture, and explain the roles of the DPLL(X) engine and the theory solver $Solver_T$ in it.

$Solver_T$ processes the input formula, stores the list of all literals occurring in it, and hands it over to DPLL(X), which treats it as a purely propositional CNF. After that, the various Abstract DPLL rules are applied by DPLL(X) as described below:

TheoryPropagate. Immediately after $Solver_T$ is notified that a literal l has been added to M, (e.g., as a consequence of UnitPropagate or Decide), $Solver_T$ is also requested to provide *all* input literals that are T-consequences of $M\, l$ but not of M alone. Then, for each one of them, TheoryPropagate is immediately applied by DPLL(X). Note that, this way, M never becomes T-inconsistent, a property that can be exploited by $Solver_T$ to increase its efficiency (see the next subsection for

the case of Difference Logic), and by the DPLL(X) engine since it will never need to ask for the T-consistency of M.

UnitPropagate. If TheoryPropagate is not applicable, DPLL(X) tries to apply UnitPropagate next, possibly triggering more rounds of theory propagation, and stops if it discovers a conflicting clause. (In a concrete implementation all this can be implemented with the commonly used two-watched-literals scheme.)

Backjump and Fail. If DPLL(X) detects a conflicting clause, it immediately applies T-Backjump or Fail, depending respectively on whether the current assignment contains a decision literal or not. (In a concrete implementation, an appropriate backjump clause can be computed as explained in the next section.) At each backjump, DPLL(X) tells $Solver_T$ how many literals have been removed from the assignment.

***T*-Learn.** Immediately after each T-Backjump application, the T-Learn rule is applied to learn the backjump clause. This is possible because this clause is always a T-consequence of the formula F in the current state $M \parallel F$. Note that, as explained in Section 3.2 for the case of exhaustive theory propagation, theory lemmas (clauses C such that $\emptyset \models_T C$) are never learned, since they are useless in this context.

Restart. For correctness with respect to the Abstract DPLL modulo theories framework, one must guarantee that Restart has increasing periodicity. Typically, this is achieved by only applying Restart when certain system parameters reach some prescribed limits, such as the number of conflicts or the number of new units derived, and increasing this restart limit periodically.

***T*-Forget.** For correctness with respect to Abstract DPLL modulo theories, it suffices to apply this rule only to previously T-learned clauses. This is what is usually done, removing part of these clauses according to their activity (e.g., the number of times involved in recent conflicts).

Decide. In this strategy, DPLL(X) applies Decide only if none of the other Basic DPLL rules apply. The choice of the decision literal is well known to have a strong impact on the search behavior. Numerous heuristics for this purpose exist.

4.3. DESIGN OF $Solver_T$ FOR DIFFERENCE LOGIC. To provide an example in this article of $Solver_T$ for a given T, here we briefly outline the design of a theory solver for Difference Logic. Despite its simplicity, Difference Logic has been used to express important practical problems, such as verification of timed systems, scheduling problems or the existence of paths in digital circuits with bounded delays.

In Difference Logic, the background theory T can be the theory of the integers, the rationals or the reals, depending on the application. Input formulas are restricted to Boolean combinations of atoms of the form $a \leq b + k$, where a and b are free constants and k is a (possibly negative) integer, rational or real constant. Over the integers, atoms of the form $a < b + k$ can be equivalently written as $a \leq b + (k - 1)$; for instance, $a < b + 7$ becomes $a \leq b + 6$. A similar transformation exists for the rationals and reals, by decreasing k by a small enough amount ε. For a given input formula, the ε to be applied to its literals can be computed in linear time [Schrijver 1987; Armando et al. 2004]. Similarly, negations and equalities can also be removed, and one can assume that all literals are of the form $a \leq b + k$. Their conjunction can be seen as a weighted graph G with an edge $a \xrightarrow{k} b$ for each literal $a \leq b + k$. Independently of whether T is the theory

of the integers, the rationals or the reals, such a conjunction is T-satisfiable if, and only if, there is no cycle in G with negative accumulated weight. Therefore, once all literals are of the form $a \leq b + k$, the specific theory does not matter any more.

4.3.1. *Initial Setup.* As said, for the initial setup of DPLL(T), $Solver_T$ reads the input CNF, stores the list of all literals occurring in it, and hands it over to DPLL(X) as a purely propositional CNF. For efficiency reasons, it is important that, in this CNF, the relation between literals and their negations is made explicit. For example, over the integers, if $a \leq b + 2$ and $b \leq a - 3$ occur in the input, then, since one is equivalent to the negation of the other, they should be abstracted by a propositional variable and its negation. This can be detected by using a canonical form during this setup process. For instance, one can impose an ordering on the free constants and require that the smallest one, say a in the example above, be always on the left-hand side of the $\leq$ symbol. So here we would have that $b \leq a - 3$ is handled as the negation of $a \leq b + 2$.

For reasons we will see below, $Solver_T$ also builds a data structure containing, for each constant symbol, the number of input literals it occurs in, and the list of all these literals.

4.3.2. *DPLL(X) Sets the Truth Value of a Literal.* Then, $Solver_T$ adds the corresponding edge to the graph. Here we will write $a_0 \xrightarrow{k\,*} a_n$ if there is a path in the graph of the form $a_0 \xrightarrow{k_1} a_1 \xrightarrow{k_2} \cdots \xrightarrow{k_{n-1}} a_{n-1} \xrightarrow{k_n} a_n$ with $n \geq 0$ and where $k = k_1 + \cdots + k_n$ is called the length of this path.

Note that one can assume that DPLL(X) does not communicate to $Solver_T$ any redundant edges (i.e., edges already entailed by G), since such consequences would already have been communicated by $Solver_T$ to DPLL(X). Similarly, DPLL(X) will not communicate to $Solver_T$ any edges that are inconsistent with the graph. Therefore, there will be no cycles of negative length. Here, $Solver_T$ must return to DPLL(X) all input literals that are new consequences of the graph once the new edge has been added. Essentially, for detecting the new consequences of a new edge $a \xrightarrow{k} b$, $Solver_T$ needs to check all paths $a_i \xrightarrow{k_i\,*} a \xrightarrow{k} b \xrightarrow{k'_j\,*} b_j$ and see whether there is any input literal that follows from $a_i \leq b_j + (k_i + k + k'_j)$, that is, an input literal of the form $a_i \leq b_j + k'$, with $k' \geq k_i + k + k'_j$. For checking all such paths from a_i to b_j that pass through the new edge from a to b, the graph is kept in double adjacency list representation. Then a standard single-source-shortest-path algorithm starting from a can be used for computing all a_i with their corresponding minimal k_i (and similarly for the b_j). Its cost, for M literals containing N different constant symbols, is $O(N \cdot M)$.

While doing this, the visited nodes are marked and inserted into two lists, one for the a_i's and one for the b_j's. At the same time, two counters are kept measuring the total number of input literals containing the a_i's and, respectively, the b_j's.

Then, if, without loss of generality, the a_i's are the ones that occur in less input literals, we check, for each input literal l containing some a_i, whether the other constant in l is some of the found b_j, and whether l is entailed or not (this can be checked in constant time since previously all b_j have been marked). The asymptotic worst-case cost of this part is $O(L)$, where L is the number of different input literals. In our experience, this is much faster than the $O(N^2)$ check of the Cartesian product of a_i's and b_j's.

4.3.3. *Implementation of Explain and Backtrack.* Whenever the mth edge is added to the graph, the edge is annotated with its *insertion number* m. When a literal l of the form $a \leq b + k$ is returned as a T-consequence of the mth edge, this m is recorded together with l. If later on the explanation for l is required, a path in the graph from a to b of length at most k is searched, using a depth-first search as before, but without traversing any edges with insertion number greater than m. This not only improves efficiency, but it is also needed for not returning "too new" explanations, which may create cycles in the implication graph (see Section 5). Each time DPLL(X) backjumps, it communicates to $Solver_T$ how many edges it has to remove, for example, up to some insertion number m. According to our experiments, the best way (with negligible cost) for dealing with this in $Solver_T$ is the naive one, that is, using a trail stack of edges with their insertion numbers and all their associated T-consequences.

4.4. DPLL(T) WITH NONEXHAUSTIVE THEORY PROPAGATION AND EUF LOGIC. For some logics, such as the logic of Equality with Uninterpreted Functions (EUF), exhaustive theory propagation is not the best strategy. In EUF, atoms consist of ground equations between terms, and the theory T consists of the axioms of reflexivity, symmetry, and transitivity of '=', as well as the *monotonicity* axioms, saying, for all f, that $f(x_1 \cdots x_n) = f(y_1 \cdots y_n)$ whenever $x_i = y_i$ for all i in $1 \cdots n$ (see also Example 3.1).

Our experiments with EUF revealed that a nonexhaustive strategy behaves better in practice than one with exhaustive theory propagation. More precisely, we found that detecting exhaustively all *negative* equality consequences is very expensive, whereas all positive equalities can propagated efficiently by means of a congruence closure algorithm [Downey et al. 1980]. It is beyond the scope of this article to describe the design of a theory solver for EUF. We refer the reader to Nieuwenhuis and Oliveras [2003] for a description and discussion of a modern incremental, backtrackable congruence closure algorithm for this purpose. We point out that efficiently retrieving explanations (for constructing the conflict graph and generating theory lemmas) inside an incremental congruence closure algorithm is nontrivial. Increasingly better techniques have been developed in de Moura et al. [2004] Stump and Tan [2005], and Nieuwenhuis and Oliveras [2005b].

We describe below an application strategy of Abstract DPLL Modulo Theories for DPLL(T) with nonexhaustive theory propagation. The emphasis will be on those aspects that differ from the exhaustive case, and on how and when T-inconsistent partial assignments M are detected and repaired.

TheoryPropagate. In this strategy, when $Solver_T$ is asked for new T-consequences, it may return only an incomplete list. Therefore, DPLL(X) can no longer maintain the invariant that the partial assignment is always T-consistent as in the exhaustive case of Section 4.2. For this reason, it is no longer necessary to ask for T-consequences as eagerly as in the exhaustive case. Instead, for efficiency reasons it is better to ask $Solver_T$ for new T-consequences only if no Basic DPLL rule other than Decide applies and the current assignment is T-consistent. For each returned T-consequence, TheoryPropagate is immediately applied by DPLL(X).

UnitPropagate. DPLL(X) applies this rule while possible unless it detects a conflicting clause.

Backjump and Fail. DPLL(X) may apply T-Backjump or Fail due to two possible situations. The first one is when it detects a conflicting clause, as usual. The second one is due to a T-inconsistency of the current partial assignment M.

$Solver_T$ is asked to check the T-consistency of M each time no Basic DPLL rule other than Decide applies—and before being asked for theory consequences. When M is T-inconsistent $Solver_T$ identifies a subset $\{l_1, \ldots, l_n\}$ of it such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$, and returns the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$ as an explanation of the inconsistency. *DPLL*(X) then handles the lemma as a conflicting clause, applying T-Backjump or Fail to it.

***T*-Learn.** Immediately after each T-Backjump application, the T-Learn rule is applied for learning the backjump clause.

Now, in backjumps due to T-inconsistencies, the backjump clause may sometimes be the theory lemma denoting the T-inconsistency itself (if it has only one literal of the current decision level). Therefore, in this case, sometimes theory lemmas will be learned. Another possibility is to always learn the theory lemma coming from a T-inconsistency, even if it is not the backjump clause. This may be useful, because it prevents the same T-inconsistency from occurring again.

Restart. This rule is applied as in the exhaustive strategy of Section 4.2.

***T*-Forget.** It is also applied as in the exhaustive case, but in this case among the (less active) lemmas that are removed there are also theory lemmas. This is again less simple than in the exhaustive case, because different forgetting policies could be applied to the two kinds of lemmas. Note that, in any case, none of the lemmas needs to be kept for completeness.

Decide. This rule is applied as in the exhaustive strategy of Section 4.2.

We conclude this section by summarizing the key differences between the two strategies for exhaustive and nonexhaustive theory propagation, described in Sections 4.2 and 4.4: in the former, which we applied to Difference Logic, the partial model never becomes T-inconsistent, since *all* input literals that are T-consequences are immediately set to true. In contrast, the DPLL(T) system described in Section 4.4, and applied to EUF logic, allows, for efficiency reasons, $Solver_T$ to fail sometimes to detect some entailed literals, and hence it must be able to recover from T-inconsistent partial assignments.

5. *Theory Propagation Strategies and Conflict Analysis*

The idea of theory propagation was first mentioned in Armando et al. [2000] under the name of *Forward Checking Simplification*, in the context of temporal reasoning. The authors suggest that a literal l can be propagated if $\neg l$ is inconsistent with the current state, but they also imply that this is expensive "since it requires a number of T-consistency checks roughly equal to the number of literals in [the whole formula] φ". A similar notion called *Enhanced Early Pruning* is mentioned in Audemard et al. [2002] in the context of the MathSAT system, but nothing is said about when and how it is applied, and how it relates to conflict analysis. Also, the new system Yices (see Section 6) appears to apply some form of theory propagation. Except for these systems and ours, and a forthcoming version of CVC Lite based on the work described here, we are not aware of any other systems that apply theory propagation,

nor of any other descriptions of theory propagation in the literature outside our own previous work on the subject.

We remark that the techniques proposed in, for example, Armando et al. [2004] and Flanagan et al. [2003], where the input formula is statically augmented with theory lemmas encoding transitivity constraints, may have effects similar to theory propagation. However, eagerly learning all such constraints is usually too expensive and typically only a subset of them is ever used at run-time.

In Ganzinger et al. [2004], we showed that, somewhat against expectations, practical T-solvers can be designed to do theory propagation efficiently. To the best of our knowledge, before that, the methods for detecting a theory consequence l were essentially based on sending $\neg l$ to the theory solver, and checking whether a T-inconsistency was derived.

Some essential and nontrivial issues about theory propagation have remained largely unstudied until now:

—when to compute the explanations for the theory propagated literals;

—how to handle conflict analysis adequately in the context of theory propagation;

—how eagerly to perform theory propagation.

In this section, we analyze these issues in detail. We point out that thinking in terms of Abstract DPLL Modulo Theories was crucial in giving us a sufficient understanding for doing this analysis in the first place, especially by helping us clearly separate correctness concerns from efficiency ones. We start with a running example illustrating some of the questions above.

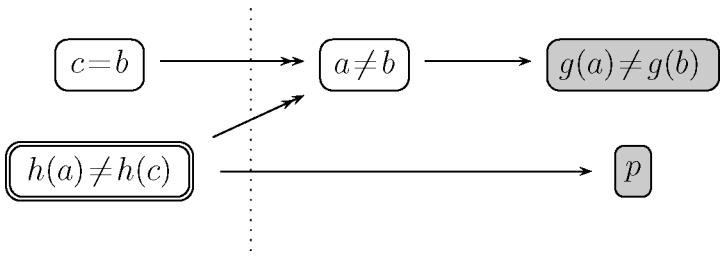
Example 5.1. Consider EUF logic and a clause set F containing, among others:

$$(1)\ a=b \ \vee\ g(a)\neq g(b)$$
$$(2)\ h(a)=h(c) \ \vee\ p$$
$$(3)\ g(a)=g(b) \ \vee\ \neg p.$$

Now consider a state of the form $M,\ c=b,\ f(a)\neq f(b) \parallel F,$ and the following sequence of derivation steps:

Step:	**New literal:**	**Reason:**
Decide	$h(a)\neq h(c)$	
TheoryPropagate	$a\neq b$	since $h(a)\neq h(c)\wedge c=b \models_T a\neq b$
UnitPropagate	$g(a)\neq g(b)$	because of $a\neq b$ and Clause 1
UnitPropagate	p	because of $h(a)\neq h(c)$ and Clause 2.

In the resulting state, Clause 3 is conflicting. When seen as a conflict graph, as done in Example 2.6 for the propositional case, the situation looks as follows:



In this graph, the double arrows $\twoheadrightarrow$ indicate theory propagations, whereas the single arrows denote unit propagations. The backjump clause $h(a) = h(c) \ \vee \ c \neq b$ can be produced by considering the indicated cut in the graph, as in Example 2.6. This clause can also be obtained by the backwards resolution process on the conflicting clause illustrated in Example 2.6, specifically, by resolving in reverse chronological order with the clauses that caused propagations, until a clause with exactly one literal from the current decision level is derived.

The only difference here with respect to the propositional case is that now we can have theory propagated literals as well. For each one of these literals, resolution is done with the theory lemma explaining its propagation (here, the leftmost premise of the last resolution step):

$$\dfrac{\dfrac{h(a)=h(c)\vee c\neq b\vee \mathbf{a}\neq\mathbf{b} \qquad \dfrac{a=b\vee \mathbf{g(a)}\neq\mathbf{g(b)} \qquad \dfrac{h(a)=h(c)\vee \mathbf{p} \qquad g(a)=g(b)\vee\neg\mathbf{p}}{\mathbf{g(a)}=\mathbf{g(b)}\vee h(a)=h(c)}}{h(a)=h(c)\vee \mathbf{a}=\mathbf{b}}}{h(a)=h(c) \ \vee \ c\neq b}$$

The resulting clause can be used as a backjump clause, in the same way as in Example 2.6 for the propositional case.

In what follows, we argue that in general it is not a good idea to compute these theory lemmas (or *explanations*) immediately, during theory propagation. Instead, it is usually better to compute each of them only as needed in resolution steps during conflict analysis. We also explain what problems may occur in delaying the computation of explanations until they are really needed, and give detailed results showing when and how a backjump clause can be found. □

5.1. When to Compute Explanations for the Theory Propagated Literals. Each time a theory propagation step of the form $M \parallel F \implies M\, l \parallel F$ takes place, this is because $l_1 \wedge \cdots \wedge l_n \models_T l$ for some subset $\{l_1, \ldots, l_n\}$ of M. Now, a very simple way of managing theory propagated literals for the purposes of conflict analysis is to use T-Learn immediately after each such a theory propagation step, adding the corresponding theory lemma $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ to the current formula. After that, the theory propagated literal l can be simply seen as a unit propagated literal by the newly learned clause. Hence, when a conflicting clause is detected, the backjump clause can be computed exactly as in the propositional case, as explained in Example 2.6.

Unfortunately, this approach, used for instance in the latest version of the MathSAT system [Bozzano et al. 2005], has some important drawbacks. We have done extensive experiments, running our DPLL(T) implementations on all the formulas available in the SMT-LIB benchmark library [Ranise and Tinelli 2003; Tinelli and Ranise 2005] for the logics EUF, RDL, IDL and UFIDL (see the next section). In these experiments, we have counted (i) the number of theory propagation steps and (ii) the number of times theory propagated literals are involved in a conflict, in other words, the number of resolution steps with explanations.

It turns out that, on average, theory propagations are around 250 times more frequent than resolution steps with explanations. For almost all examples, the ratio lies between 20 and 1500. Hence, immediately computing an explanation each time a theory propagation takes place, as done in MathSAT, is bound to be highly inefficient: on average just one of these lemmas out of every 250 is ever going

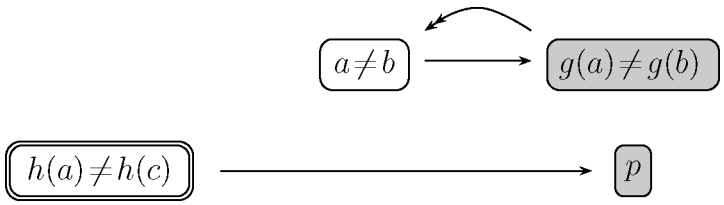
to be used (possibly, even less than that, as each theory propagated literal may occur in more than one conflict). The cost of generating explanations is twofold: it is the cost incurred by $Solver_T$ in computing the clause and that incurred by DPLL(X) in inserting the clause in the clause database and maintaining it under propagation.

There is however a potential advantage in the MathSAT approach, for strategies where TheoryPropagate is applied only if no other rule except Decide is applicable (this is what we did for EUF in Section 4.4). Assume as before that TheoryPropagate applies to a state $M \parallel F$ because $l_1 \wedge \cdots \wedge l_n \models_T l$ for for some subset $\{l_1, \ldots, l_n\}$ of M. Also assume that, due to a previous TheoryPropagate step, the explanation $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ is still present in F, although, due to backtracking, l has become again undefined in M. Then the effect of theory propagating l can now be achieved more efficiently by a unit propagation step with the clause $\neg l_1 \vee \cdots \vee \neg l_n \vee l$. If this leads to a conflict at the current decision level before TheoryPropagate is tried, then a gain in efficiency may be obtained. If, on the other hand, no conflict occurs before applying TheoryPropagate, then it is likely that repeated work is done by $Solver_T$, rediscovering the fact that l is a T-consequence of M.

For some theory solvers, it may be possible that, when computing a T-consequence, there is only a low additional cost in computing its explanation as well at the same time. But even then one usually would not want to pay the time and memory cost of adding the lemma as a new clause—since in many cases this is going to be wasted work and space. One could simply store the lemma as a passive clause, that is, not active in the DPLL procedure, or store some information on how to compute it later.

5.2. HANDLING CONFLICT ANALYSIS IN THE CONTEXT OF THEORY PROPAGATION. In the previous subsection we have argued that it is preferable to generate explanations only at the moment they are needed for conflict analysis. Here we analyze the possible problems that arise in doing so, and discuss when and how it is still possible to compute a backjump clause. In a state of the form $M_1\, l\, M_2\, l'\, M_3 \parallel F$, we say that l is *older* than l', and that l' is *newer* than l.

Too new explanations:
Let us first revisit Example 5.1. After the four steps, where Clause 3 is conflicting, if $Solver_T$ is asked to compute the explanation of $a \neq b$, it can also return $g(a) \neq g(b)$, instead of the "real" explanation $h(a) = h(c) \vee c \neq b \vee a \neq b$. Indeed $a \neq b$ is a T-consequence of $g(a) \neq g(b)$ as well. But $g(a) \neq g(b)$ is a *too new explanation*: it did not even belong to the partial assignment at the time $a \neq b$ was propagated, and was in fact deduced by UnitPropagate from $a \neq b$ itself and Clause 1. Too new explanations are problematic because they can cause cycles in the conflict graph:

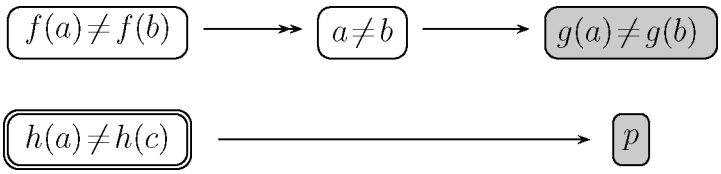


For the conflict graph above, the backwards resolution process computing the backjump clause does in fact loop:

$$\dfrac{g(a)=g(b)\vee \mathbf{a}\neq\mathbf{b} \qquad \dfrac{a=b\vee \mathbf{g(a)}\neq\mathbf{g(b)} \qquad \dfrac{h(a)=h(c)\vee \mathbf{p} \quad g(a)=g(b)\vee\neg\mathbf{p}}{\mathbf{g(a)}=\mathbf{g(b)}\vee h(a)=h(c)}}{h(a)=h(c)\vee \mathbf{a}=\mathbf{b}}}{g(a)=g(b)\ \vee\ h(a)=h(c)}$$

Therefore, to make sure that a backjump clause can be found, $Solver_T$ should never return too new explanations. A sufficient condition is to require that all literals in the explanation of a literal l be older than l. In other words, if the current state is of the form $M\ l\ N \parallel F$, then all literals in the explanation of l should occur in M.

Too old explanations:
In our example, when $Solver_T$ was asked to compute the explanation of $a \neq b$, it could also have returned $f(a) \neq f(b)$. This literal was already available before $a \neq b$ was obtained, but, as mentioned in Section 4.4, $Solver_T$ might have failed to detect $a \neq b$ as a negative consequence of it. It is interesting to observe that, with $f(a) = f(b) \vee a \neq b$ as the explanation of $a \neq b$, the resulting conflict graph has no unique implication point (UIP). In fact, there is not even a path from the current decision literal $h(a) \neq h(c)$ to the conflicting literal (of the current decision level) $g(a) \neq g(b)$:



However, looking at what happens with the backwards resolution procedure, one can see that it still produces a backjump clause, that is, a clause with exactly one literal from the current decision level:

$$\dfrac{f(a)=f(b)\vee \mathbf{a}\neq\mathbf{b} \qquad \dfrac{a=b\vee \mathbf{g(a)}\neq\mathbf{g(b)} \qquad \dfrac{h(a)=h(c)\vee \mathbf{p} \quad g(a)=g(b)\vee\neg\mathbf{p}}{\mathbf{g(a)}=\mathbf{g(b)}\vee h(a)=h(c)}}{h(a)=h(c)\vee \mathbf{a}=\mathbf{b}}}{h(a)=h(c)\ \vee\ f(a)=f(b)}$$

The following theorem states that in the backwards resolution process too old explanations are never a problem. It follows that just disallowing too new explanations suffices to guarantee that a backjump clause is always found.

THEOREM 5.2. *Assume that for any state of the form $M \parallel F$ and for any l in M due to* TheoryPropagate*, the explanation of l produced by $Solver_T$ contains no literals newer than l in M. Then, if some clause C is conflicting in a state S, either* Fail *applies to S, or else the backwards resolution process applied to C reaches a backjump clause.*

PROOF. Let d be the largest of the decision levels of the literals in C, and let D be the (nonempty) set of all literals of C that have become false at decision level d. If D is a singleton, C itself is a backjump clause. Otherwise, we can apply the

backwards conflict resolution process, resolving away literals of decision level d, until we reach a backjump clause having exactly one literal of level d. This process always terminates because each resolution step replaces a literal of decision level d by a finite number (zero in the case of a too old explanation) of strictly older literals of level d. The process is also guaranteed to produce a clause with just one literal of decision level d because, except for the decision literal itself, every literal of decision level d is resolvable. □

Note that the previous theorem is rather general by making no assumptions on the strategy followed in applying the DPLL rules. Also note that the theorem holds in the purely propositional case a well, where the theory T is empty and the theorem's assumption is vacuously true as TheoryPropagate never applies. Its generality entails that, for instance, one can apply Decide even in the presence of a conflicting clause, or if UnitPropagate also applies. In contrast, in Zhang and Malik [2003], the correctness proof of the Chaff algorithm assumes the fixed standard strategy in which unit propagation is done exhaustively before making any new decisions, which is considered an "important invariant". Theorem 5.2 instead shows that it is unproblematic for conflict analysis if a literal l is unit propagated at a decision level d when in fact it could have been propagated already at an earlier level. The reason is simply that in the backwards resolution step resolving on l replaces it by zero literals of level d, in perfect analogy to what happens to theory propagated literals with a too old explanation.

5.3. THE DEGREE OF EAGERNESS BY WHICH THEORY PROPAGATION SHOULD BE PERFORMED. So far, we have seen two possible strategies for theory propagation.

The first one, which we defined for Difference Logic, requires that $Solver_T$ returns *all* theory consequences (Section 4.2). In that strategy, TheoryPropagate is invoked each time a new literal is added to the current partial assignment. This is done to ensure that the partial assignment never becomes T-inconsistent.

The second strategy, defined for EUF logic, assumes that $Solver_T$ may return only some subset of the theory consequences, and applies TheoryPropagate only if no rule other than TheoryPropagate or Decide is applicable (Section 4.4).

However, there may also be expensive theories where one does not want to do full theory propagation (or check T-consistency) before every Decide step, but instead invoke it in some cheaper, incomplete way. The complete check is only required at the leaves of the search tree, that is, each time a propositional model has been found, in order to decide its T-consistency (this coincides with what is done in the naive lazy approach). The MathSAT approach [Bozzano et al. 2005] is based on a similar hierarchical view, where cheaper checks are performed more eagerly than expensive ones.

6. *Experiments with an Implementation of DPLL(T)*

We have experimented the DPLL(T) architecture with various implementations collaboratively developed in Barcelona and Iowa. We describe here our most advanced implementation, Barcelogic, developed mostly in Barcelona. The system follows the strategies presented in Section 4, and its solvers are as described in Section 4.3 and in Nieuwenhuis and Oliveras [2003, 2005b]. Its DPLL(X) engine implements state-of-the-art techniques such as the two-watched literal scheme for unit propagation, the first-UIP learning scheme, and VSIDS-like decision heuristics

[Moskewicz et al. 2001; Zhang et al. 2001]. The T-Forget rule is currently applied by DPLL(X) after each restart, removing a portion of the learned clauses according to their activity level [Goldberg and Novikov 2002], defined as the number of times they were involved in a conflict since the last restart.

6.1. THE 2005 SMT COMPETITION. The effectiveness of Barcelogic was shown at the 2005 SMT Competition [Barrett et al. 2005]. The competition used problems from the SMT-LIB library [Tinelli and Ranise 2005], a fairly large collection of benchmarks (around 1300) coming from such diverse areas as software and hardware verification, bounded model checking, finite model finding, and scheduling. These benchmarks were in the standard format of SMT-LIB [Ranise and Tinelli 2003], and were classified into 7 competition divisions according to their background theory and some additional syntactic restrictions. For each division, around 50 benchmarks were randomly chosen and given to each entrant system with a time limit of 10 minutes per benchmark.

Barcelogic entered and won all four divisions for which it had a theory solver: EUF, IDL and RDL (respectively, integer and real Difference Logic), and UFIDL (combining EUF and IDL). At least 10 systems participated in each of these divisions. Among the competitors were well-known SMT solvers such as SVC [Barrett et al. 1996], CVC [Barrett et al. 2002], CVC-Lite [Barrett and Berezin 2004], MathSAT [Bozzano et al. 2005], and two very recent successors of ICS [Filliâtre et al. 2001]: Yices (by Leonardo de Moura) and Simplics (by Dutertre and de Moura). Apart from EUF and Difference Logic, these systems also support other theories such as arrays (except MathSAT), and linear arithmetic (SVC only over the reals).

It is well-known that in practical problems over richer (combined) theories usually a large percentage of the work still goes into EUF and Difference Logic. For example, in [Bozzano et al. 2005] it is mentioned that in many calls a general solver is not needed: "very often, the unsatisfiability can be established in less expressive, but much easier, sub-theories". Similarly, Seshia and Bryant [2004], which deals with quantifier-free Presburger arithmetic, states that it has been found by them and others that literals are "mainly" difference logic.

The competition was run on 2.6 GHz, 512 MB, Pentium 4 Linux machines, with a 512 KB cache. For each division, the results of the best three systems are shown in the following table, where **Total time** is the total time in seconds spent by each system, with a timeout of 600 seconds, and **Time solved** is the time spent on the solved problems only:

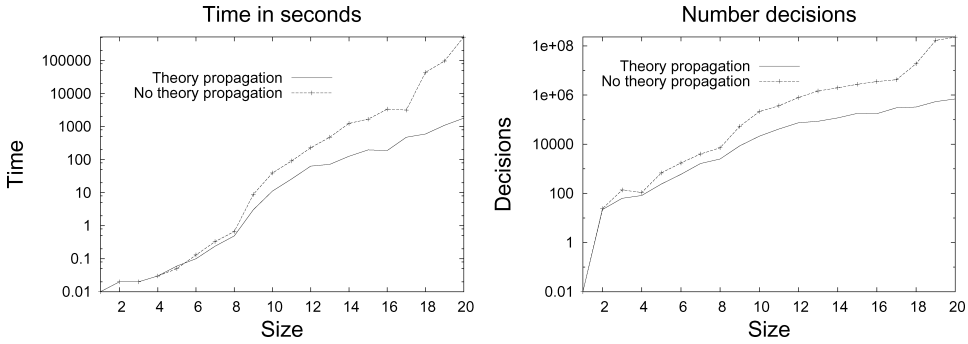
	Top-3 systems	# problems solved	Total time	Time solved
EUF (50 problems):	Barcelogic	39	8358	1758
	Yices	37	9601	1801
	MathSAT	33	12386	2186
RDL (50 pbms.):	Barcelogic	41	6341	940
	Yices	37	9668	1868
	MathSAT	37	10408	2608
IDL (51 pbms.):	Barcelogic	47	3531	1131
	Yices	47	4283	1883
	MathSAT	46	4295	1295
UFIDL (49 pbms.):	Barcelogic	45	2705	305
	Yices	36	9789	1989
	MathSAT	22	17255	1055

Not only did Barcelogic solve more problems than each of the other systems, it also did so in considerably less time, even—and in spite of the fact that it solved more problems—if only the time spent on the solved problems is counted.

6.2. EXPERIMENTS ON THE IMPACT OF THEORY PROPAGATION. In our experience, the overhead produced by theory propagation is almost always compensated by a significant reduction of the search space. In Ganzinger et al. [2004] we presented extensive experimental results showing its effectiveness in our DPLL(T) approach for EUF logic. In Nieuwenhuis and Oliveras [2005a] we discussed a large number of experiments for Difference Logic, with additional emphasis on the good scaling properties of the approach. The new SMT solver Yices now also heavily relies on theory propagation.

Of course, theory propagation may not pay off in certain specific problems where the theory plays an insignificant role, that is, where reasoning is done almost entirely at the Boolean level. Such situations can be detected on the fly by computing the percentage of conflicts caused by theory propagations. If this number is very low, theory propagation can be switched off automatically, or applied more lazily, to speed up the computation. (This is done in a forthcoming release of our system.)

In the following two figures, Barcelogic with and without theory propagation is compared, on the same type of machine as in the previous subsection, in terms of run time (in seconds) and number of decisions (applications of Decide) on a typical real-world Difference Logic suite (fisher6-mutex, see Tinelli and Ranise [2005]), consisting of 20 problems of increasing size.



The figures show the typical behavior on the larger problems where the theory plays a significant role: both the run time and the number of decisions are orders of magnitude smaller in the version with theory propagation (note that times and decisions are plotted on a logarithmic scale). In both cases, the DPLL(X) engine used was exactly the same, although in the exhaustive theory case some parts of the code were never executed (e.g., theory lemma learning).

6.3. EXPERIMENTS COMPARING BARCELOGIC WITH THE EAGER APPROACH. For completeness, we finally compare Barcelogic with UCLID, the best-known tool implementing the eager translation approach to SMT [Lahiri and Seshia 2004]. We show below run time results (in seconds) for three typical series of benchmarks for UFIDL coming from different methods for pipelined processor verification given in [Manolios and Srinivasan 2005a, 2005b] (more precisely, for the BIDW case (i) flushing, (ii) commitment good MA and (iii) commitment GFP). The benchmarks

were run on the same type of machine as in the previous two subsections, but this time with a one hour timeout. We used Siege [Ryan 2004] as the final SAT solver for UCLID, since it performed better than any other available SAT solver on these problems.

	UCLID	BLogic	UCLID	BLogic	UCLID	BLogic
6-stage:	258	1	3596	5	19	1
7-stage:	835	3	>3600	8	58	1
8-stage:	3160	15	>3600	18	226	1
9-stage:	>3600	23	>3600	18	664	1
10-stage:	>3600	54	>3600	29	>3600	2

We emphasize that these results are typical for the pipelined processor verification problems coming from this source, a finding that has also independently been reproduced by P. Manolios (private communication). We refer the reader to the results given in Ganzinger et al. [2004], showing that our approach also dominates UCLID in the pure EUF case, as well as for EUF with *integer offsets* (interpreted successor and predecessor symbols).

7. *Conclusions*

We have shown that the Abstract DPLL formalism introduced here can be very useful for understanding and formally reasoning about a large variety of DPLL-based procedures for SAT and SMT.

In particular, we have used it here to describe several variants of a new, efficient, and modular approach for SMT, called DPLL(T). Given a DPLL(X) engine, a DPLL(T) system for a theory T is obtained by simply plugging in the corresponding theory solver $Solver_T$, which must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements.

We are currently working on several—in our opinion very promising—ways to improve and extend both the abstract framework and the DPLL(T) architecture.

The abstract framework can be extended to deal more effectively with theories where the satisfiability of conjunctions of literals is already NP-hard by lifting, from the theory solver to the DPLL(X) engine, some or all of the case analysis done by the theory solver. Along those lines, the framework can also be nicely extended to a Nelson-Oppen style combination framework for handling formulas over several theories. The resulting $DPLL(T_1, \ldots, T_n)$ architecture can deal modularly and efficiently with the combined theories.

Preliminary experiments reveal that other applications of the DPLL(T) framework can produce competitive decision procedures as well for completely different (at least on the surface) kinds of problems. For example, optimization aspects of problems such as pseudo-Boolean constraints can be nicely expressed and efficiently solved in this framework by recasting them as particular SMT problems.

ACKNOWLEDGMENTS. We would like to thank Roberto Sebastiani for a number of insightful discussions and comments on the lazy SMT approach and DPLL(T). We are also grateful to the anonymous referees for their helpful suggestions on improving this article.

REFERENCES

ALUR, R. 1999. Timed automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)* (Trento, Italy), N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York, 8–22.

ARMANDO, A., CASTELLINI, C., AND GIUNCHIGLIA, E. 2000. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning* (Durham, UK), S. Biundo and M. Fox, Eds. Lecture Notes in Computer Science, vol. 1809. Springer-Verlag, New York, 97–108.

ARMANDO, A., CASTELLINI, C., GIUNCHIGLIA, E., AND MARATEA, M. 2004. A SAT-based decision procedure for the Boolean combination of difference constraints. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*. Lecture Notes in Computer Science. Springer-Verlag, New York.

AUDEMARD, G., BERTOLI, P., CIMATTI, A., KORNILOWICZ, A., AND SEBASTIANI, R. 2002. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *Proceedings of the CADE-18*. Lecture Notes in Computer Science, vol. 2392. Springer-Verlag, New York, 195–210.

BALL, T., COOK, B., LAHIRI, S. K., AND ZHANG, L. 2004. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York. 457–461.

BARRETT, C., DE MOURA, L., AND STUMP, A. 2005. SMT-COMP: Satisfiability modulo theories competition. In *Proceedings of the 17th International Conference on Computer Aided Verification*, K. Etessami and S. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 20–23. (See www.csl.sri.com/users/demoura/smt-comp.)

BARRETT, C., DILL, D., AND STUMP, A. 2002. Checking satisfiability of first-order formulas by incremental translation into SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, New York.

BARRETT, C., DILL, D. L., AND LEVITT, J. 1996. Validity checking for combinations of theories with equality. In *Proceedings of the 1st International Conference on Formal Methods in Computer Aided Design*. Lecture Notes in Computer Science, vol. 1166. Springer-Verlag, New York, 187–201.

BARRETT, C. W. 2003. Checking validity of quantifier-free formulas in combinations of first-order theories. Ph.D. dissertation. Stanford University, Stanford, CA.

BARRETT, C. W., AND BEREZIN, S. 2004. CVC lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 515–518.

BAYARDO, R. J. J., AND SCHRAG, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)* (Providence, RI), 203–208.

BEAME, P., KAUTZ, H., AND SABHARWAL, A. 2003. On the power of clause learning. In *Proceedings of IJCAI-03, 18th International Joint Conference on Artificial Intelligence* (Acapulco, MX).

BONET, M. L., ESTEBAN, J. L., GALESI, N., AND JOHANNSEN, J. 2000. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput. 30*, 5, 1462–1484.

BOZZANO, M., BRUTTOMESSO, R., CIMATTI, A., JUNTTILA, T. V. ROSSUM, P., SCHULZ, S., AND SEBASTIANI, R. 2005. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference (TACAS)*. Lecture Notes in Computer Science, vol. 3440. Springer-Verlag, New York, 317–333.

BRYANT, R., GERMAN, S., AND VELEV, M. 2001. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Computational Logic 2*, 1, 93–134.

BRYANT, R., LAHIRI, S., AND SESHIA, S. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, New York.

BRYANT, R. E., AND VELEV, M. N. 2002. Boolean satisfiability with transitivity constraints. *ACM Trans. Computational Logic 3*, 4, 604–627.

BURCH, J. R., AND DILL, D. L. 1994. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 818. Springer-Verlag, New York. 68–80.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Commun. ACM 5*, 7, 394–397.

DAVIS, M., AND PUTNAM, H. 1960. A computing procedure for quantification theory. *J ACM 7*, 201–215.

DE MOURA, L., AND RUEß, H. 2002. Lemmas on demand for satisfiability solvers. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*. 244–251.

DE MOURA, L., AND RUESS, H. 2004. An experimental evaluation of ground decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 162–174.

DE MOURA, L., RUEß, H., AND SHANKAR, N. 2004. Justifying equality. In *Proceedings of the 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning* (Cork, Ireland).

DOWNEY, P. J., SETHI, R., AND TARJAN, R. E. 1980. Variations on the common subexpressions problem. *J. ACM 27*, 4, 758–771.

EÉN, N., AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 502–518.

FILLIÂTRE, J.-C., OWRE, S., RUEß, H., AND SHANKAR, N. 2001. ICS: Integrated canonization and solving (tool presentation). In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'2001)*. G. Berry, H. Comon, and A. Finkel, Eds. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, New York, 246–249.

FLANAGAN, C., JOSHI, R., OU, X., AND SAXE, J. B. 2003. Theorem proving using lazy proof explanation. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2725. Springer-Verlag, New York.

GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2004. DPLL(T): Fast Decision Procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 175–188.

GOLDBERG, E., AND NOVIKOV, Y. 2002. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Symposium on Design, Automation, and Test in Europe (DATE '02)*. 142–149.

HODGES, W. 1993. *Model Theory*. Enclyclopedia of mathematics and its applications, vol. 42. Cambridge University Press, Cambridge, MA.

JAFFAR, J., AND MAHER, M. 1994. Constraint Logic Programming: A Survey. *J. Logic Prog. 19/20*, 503–581.

LAHIRI, S. K., AND SESHIA, S. A. 2004. The UCLID Decision Procedure. In *Computer Aided Verification, 16th International Conference (CAV)*. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 475–478.

MANOLIOS, P., AND SRINIVASAN, S. K. 2005a. A computationally efficient method based on commitment refinement maps for verifying pipelined machines. In *Proceedings of the ACM IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. ACM, New York.

MANOLIOS, P., AND SRINIVASAN, S. K. 2005b. Refinement maps for efficient verification of processor models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE)*. IEEE Computer Society, 1304–1309.

MARQUES-SILVA, J., AND SAKALLAH, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput. 48*, 5 (May), 506–521.

MEIR, O., AND STRICHMAN, O. 2005. Yet another decision procedure for equality logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)* (Edimburgh, Scotland). K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 307–320.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

NIEUWENHUIS, R., AND OLIVERAS, A. 2003. Congruence Closure with Integer Offsets. In *Proceedings of the 10h International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, M. Vardi and A. Voronkov, Eds. Lecture Notes in Artificial Intellegence, vol. 2850. Springer-Verlag, New York, 2850. 78–90.

NIEUWENHUIS, R., AND OLIVERAS, A. 2005a. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)* (Edimburgh, Scotland). K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 321–334.

NIEUWENHUIS, R., AND OLIVERAS, A. 2005b. Proof-producing congruence closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA'05)* (Nara, Japan). J. Giesl, Ed. Lecture Notes in Computer Science, vol. 3467. Springer-Verlag, New York, 453–468.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2005. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of the 11th International Conference Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. F. Baader and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 3452. Springer-Verlag, New York, 36–50.

PNUELI, A., RODEH, Y., SHTRICHMAN, O., AND SIEGEL, M. 1999. Deciding equality formulas by small domains instantiations. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York. 455–469.

RANISE, S., AND TINELLI, C. 2003. The SMT-LIB format: An initial proposal. In *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*. Miami.

RYAN, L. 2004. Efficient algorithms for clause-learning SAT solvers. M.S. dissertation, School of Computing Science, Simon Fraser University.

SCHRIJVER, A. 1987. *Theory of Linear and Integer Programming*. Wiley, New York.

SESHIA, S., LAHIRI, S., AND BRYANT, R. 2003. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proceedings of the 40th Design Automation Conference (DAC)*. 425–430.

SESHIA, S. A. 2005. Adaptive eager Boolean encoding for arithmetic reasoning in verification. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA.

SESHIA, S. A., AND BRYANT, R. E. 2004. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Computer Society Press, Los Alamitos, CA, 100–109.

STRICHMAN, O. 2002. On solving presburger and linear arithmetic with SAT. In *Proceedings of the Formal Methods in Computer-Aided Design, 4th International Conference (FMCAD 2002)* (Portland, OR). M. Aagaard and J. W. O'Leary, Eds. Lecture Notes in Computer Science, vol. 2517. Springer-Verlag, New York, 160–170.

STRICHMAN, O., SESHIA, S. A., AND BRYANT, R. E. 2002. Deciding separation formulas with SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol 2404. Springer-Verlag, New York. 209–222.

STUMP, A., AND TAN, L.-Y. 2005. The algebra of equality proofs. In *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05* (Nara, Japan). J. Giesl, Ed. Lecture Notes in Computer Science, vol. 3467. Springer-Verlag, New York, 469–483.

TALUPUR, M., SINHA, N., STRICHMAN, O., AND PNUELI, A. 2004. Range allocation for separation logic. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*. (Boston, MA July 13–17). Lecture Notes in Computer Science, Springer-Verlag, New York, 148–161.

TINELLI, C. 2002. A DPLL-based calculus for ground satisfiability modulo theories. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence*. Lecture Notes in Artificial Intelligence, vol. 2424. Springer-Verlag, New York, 308–319.

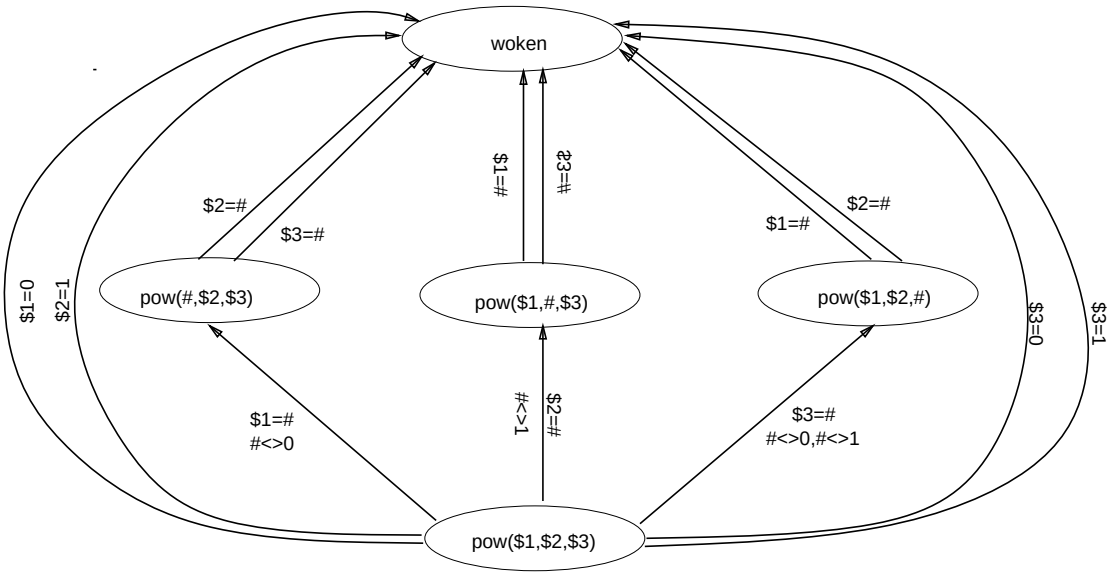
TINELLI, C., AND RANISE, S. 2005. *SMT-LIB: The Satisfiability Modulo Theories Library*. http://goedel.cs.uiowa.edu/smtlib/.

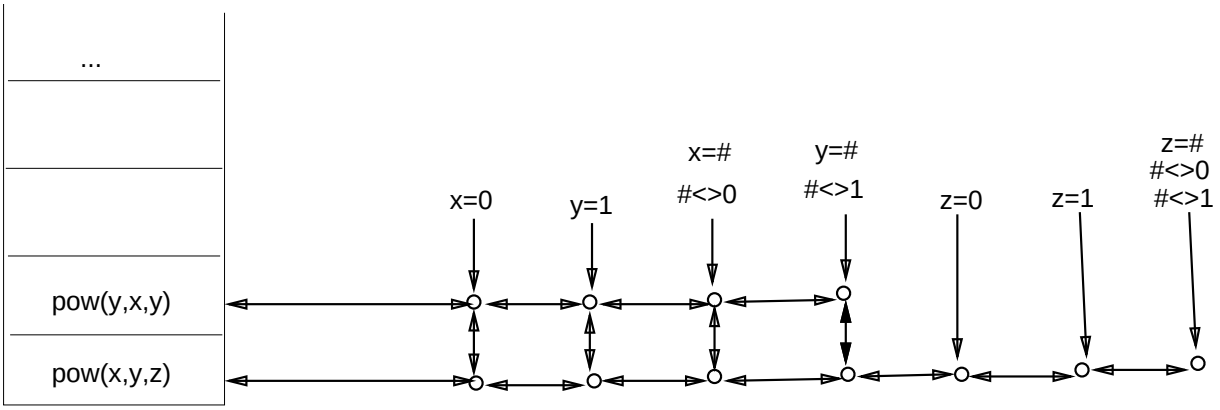
ZHANG, H. 1997. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*. Springer-Verlag, New York, 272–275.

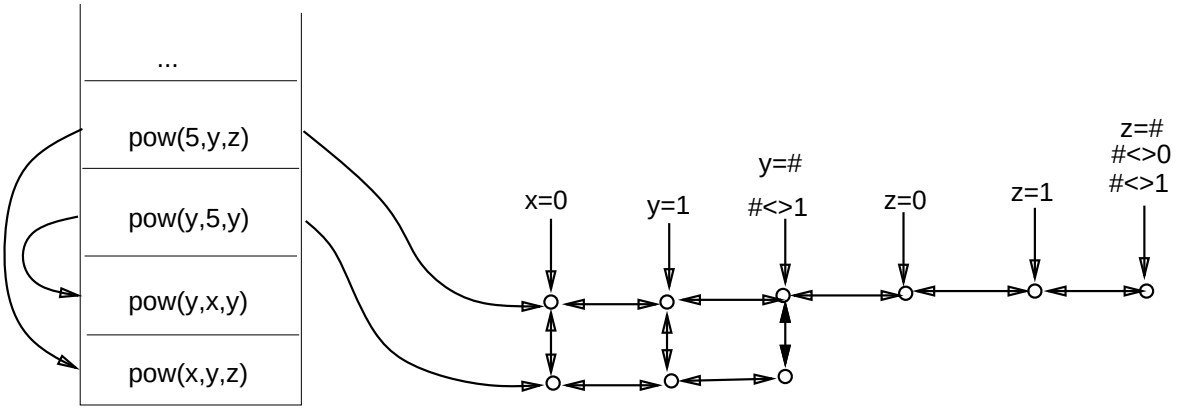
ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. W., AND MALIK, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*. 279–285.

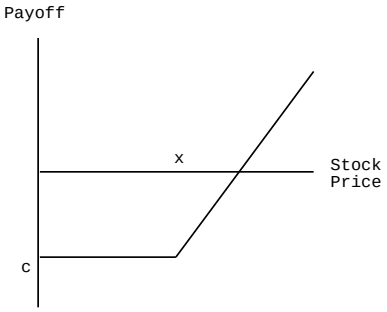
ZHANG, L., AND MALIK, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the 2003 Design, Automation and Test in Europe Conference (DATE 2003)*. IEEE Computer Society Press, Los Alamitos, CA, 10880–10885.

RECEIVED DECEMBER 2005; REVISED FEBRUARY 2006 AND JULY 2006; ACCEPTED NOVEMBER 2006

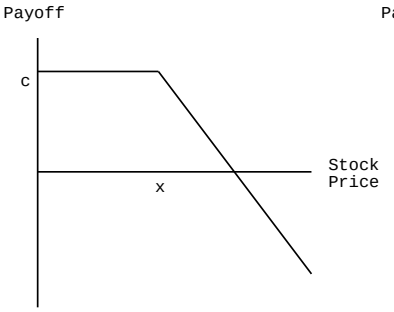

woken
pow(#,$2,$3)
pow($1,#,$3)
pow($1,$2,#)
pow($1,$2,$3)
$1=0
$2=1
$2=#
$3=#
$1=#
#<>0
$1=#
$2=#
#<>1
$2=#
$1=#
$3=0
$3=#
#<>0,#<>1
$3=1


...
pow(y,x,y)
pow(x,y,z)
x=0
y=1
x=#
#<>0
y=#
#<>1
z=0
z=1
z=#
#<>0
#<>1

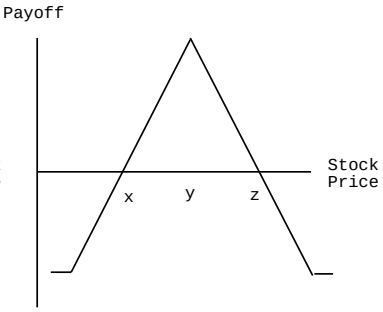

...
pow(5,y,z)
pow(y,5,y)
pow(y,x,y)
pow(x,y,z)
x=0
y=1
y=#
#<>1
z=0
z=1
z=#
#<>0
#<>1



Buy a Call



Sell a Call



Butterfly

A tableau-based decision procedure for LTL

Angelo Montanari

Department of Mathematics and Computer Science
University of Udine, Udine Italy

Outline

Outline

Basic coordinates - 1

Explicit vs. implicit methods for (modal and) temporal logics

Basic coordinates - 1

Explicit vs. implicit methods for (modal and) temporal logics

In **implicit methods** the accessibility relation is built-in into the structure of the tableau

This is the case with tableau methods for linear and branching time point temporal logics

Basic coordinates - 1

Explicit vs. implicit methods for (modal and) temporal logics

In **implicit methods** the accessibility relation is built-in into the structure of the tableau

This is the case with tableau methods for linear and branching time point temporal logics

Explicit methods keep track of the accessibility relation by means of some sort of external device

This is the case with tableau methods for interval temporal logics where structured labels are associated with nodes to constrain the corresponding formula, or set of formulae, to hold only at the domain element(s) identified by the label

Basic coordinates - 2

Declarative vs. incremental methods

Basic coordinates - 2

Declarative vs. incremental methods

Declarative methods first generate all possible sets of subformulae of a given formula and then they eliminate some (possibly all) of them

Declarative methods are generally easier to understand

Basic coordinates - 2

Declarative vs. incremental methods

Declarative methods first generate all possible sets of subformulae of a given formula and then they eliminate some (possibly all) of them

Declarative methods are generally easier to understand

Incremental methods generate only ‘meaningful’ sets of subformulae

Incremental methods are generally more efficient

Tableau systems for LTL and fragments/variants - 1

An exponential time declarative method to check LTL formulae has been developed by Wolper

P. Wolper, The tableau method for temporal logic: An overview, Logique et Analyse 28 (1985) 119–136

and later extended by Lichtenstein and Pnueli to Past LTL (PLTL)

O. Lichtenstein, A. Pnueli, Propositional temporal logic: Decidability and completeness, Logic Journal of the IGPL 8(1) (2000) 55–85

Tableau systems for LTL and fragments/variants - 2

A PSPACE incremental method for PLTL has been proposed by Kesten et al.

Y. Kesten, Z. Manna, H. McGuire, A. Pnueli, A decision algorithm for full propositional temporal logic, in: Proc. of the 5th International Conference on Computer Aided Verification, 1993, pp. 97–109

A labeled tableau system for the LTL-fragment LTL[F] has been proposed by Schmitt and Goubault-Larrecq

P. Schmitt, J. Goubault-Larrecq, A tableau system for linear-time temporal logic, in: E. Brinksma (Ed.), Proc. of the 3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 1217 of LNCS, Springer, 1997, pp. 130–144

Tableau systems for LTL and fragments/variants - 3

A tableau method for PLTL over bounded models has been developed by Cerrito and Cialdea-Mayer

S. Cerrito, M. Cialdea-Mayer, Bounded model search in linear temporal logic and its application to planning, in: Proc. of the International Conference TABLEAUX 1998, Vol. 1397 of LNAI, Springer, 1998, pp. 124–140

Later Cerrito et al. generalized the method to first-order PLTL

S. Cerrito, M. Cialdea-Mayer, S. Praud, First-order linear temporal logic over finite time structures, in: H. Ganzinger, D. McAllester, A. Voronkov (Eds.), Proc. of the 6th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Vol. 1705 of LNAI, Springer, 1999, pp. 62–76.

About complexity

The satisfiability problem for LTL / PLTL is PSPACE-complete

A. Sistla, E. Clarke, The complexity of propositional linear time temporal logics, Journal of the ACM 32 (3) (1985) 733–749

while that LTL[F] and for PLTL over bounded models of polynomial length is NP-complete

S. Cerrito, M. Cialdea-Mayer, Bounded model search in linear temporal logic and its application to planning, in: Proc. of the International Conference TABLEAUX 1998, Vol. 1397 of LNAI, Springer, 1998, pp. 124–140

A. Sistla, E. Clarke, The complexity of propositional linear time temporal logics, Journal of the ACM 32 (3) (1985) 733–749

Tableau systems for CTL

An implicit tableau method to check the satisfiability of CTL formulae, that generalizes Wolper's method for LTL, has been proposed by Emerson and Halpern

E. Emerson, J. Halpern, Decision procedures and expressiveness in the temporal logic of branching time, Journal of Computer and System Sciences 30 (1) (1985) 1–24

The satisfiability problem for CTL is known to be EXPTIME-complete. There exists an optimal incremental version of Emerson and Halpern's decision procedure

Outline

A tableau-based decision procedure for LTL

In the following, we describe in detail a tableau-based decision procedure for LTL

For the sake of clarity, among the various existing tableau systems for LTL, we selected Manna and Pnueli's implicit declarative one

Z. Manna, A. Pnueli, Temporal Verification of Reactive Systems: Safety, Springer, 1995

Expansion rules and closure

Expansion rules

- $Gp \approx p \wedge XGp$
- $Fp \approx p \vee XFp$
- $pUq \approx q \vee (p \wedge X(pUq))$

Expansion rules and closure

Expansion rules

- $Gp \approx p \wedge XGp$
- $Fp \approx p \vee XFp$
- $pUq \approx q \vee (p \wedge X(pUq))$

Closure Φ_φ of a formula φ

Φ_φ is the smallest set of formulae satisfying:

- $\varphi \in \Phi_\varphi$
- for every $p \in \Phi_\varphi$ and subformula q of p, $q \in \Phi_\varphi$
- for every $p \in \Phi_\varphi$, $\neg p \in \Phi_\varphi$ ($\neg\neg p \equiv p$)
- for every $\psi \in \{Gp, Fp, pUq\}$, if $\psi \in \Phi_\varphi$, then $X\psi \in \Phi_\varphi$

Example of closure

$\varphi : Gp \wedge F\neg p$

The closure is $\Phi_\varphi = \Phi^+_\varphi \cup \Phi^-_\varphi$, where

$$\Phi^+_\varphi = \{\varphi, Gp, F\neg p, XGp, XF\neg p, p\}$$

and

$$\Phi^-_\varphi = \{\neg\varphi, \neg Gp, \neg F\neg p, \neg XGp, \neg XF\neg p, \neg p\}$$

Example of closure

$\varphi : Gp \wedge F\neg p$

The closure is $\Phi_\varphi = \Phi_\varphi^+ \cup \Phi_\varphi^-$, where

$$\Phi_\varphi^+ = \{\varphi, Gp, F\neg p, XGp, XF\neg p, p\}$$

and

$$\Phi_\varphi^- = \{\neg\varphi, \neg Gp, \neg F\neg p, \neg XGp, \neg XF\neg p, \neg p\}$$

We have that $|\Phi_\varphi| \leq 4 \cdot |\varphi|$

$$Gp \rightarrow \{Gp, XGp, \neg Gp, \neg XGp\}$$

Classification of formulae

α and β tables

$\underline{\alpha}$	$\underline{k(\alpha)}$
$p \wedge q$	p, q
Gp	p, XGp

We have that an α-formula holds at position j iff all of $k(\alpha)$-formulae hold at j

Classification of formulae

α and β tables

$\underline{\alpha}$	$\underline{k(\alpha)}$
$p \wedge q$	p, q
Gp	p, XGp

We have that an α-formula holds at position j iff all of $k(\alpha)$-formulae hold at j

$\underline{\beta}$	$\underline{k_1(\beta)}$	$\underline{k_2(\beta)}$
$p \vee q$	p	q
Fp	p	XFp
pUq	q	$p, X(pUq)$

We have that a β-formula holds at position j iff either the $k_1(\beta)$-formula holds at j or all $k_2(\beta)$-formulae hold at j (or both)

Atoms

Atom over φ (φ-atom)

A φ-atom is a subset $A \subseteq \Phi_\varphi$ satisfying:

- R_{sat}: the conjunction of all local formulae in A is satisfiable
- $R_\neg$: for every $p \in \Phi_\varphi$, $p \in A$ iff $\neg p \ / \in A$ (i.e., for every $p \in \Phi_\varphi$, a φ-atom must contain either p or $\neg p$)
- R_α: for every α-formula $\alpha \in \Phi_\varphi$, $\alpha \in A$ iff $k(\alpha) \subseteq A$ (e.g., $Gp \in A$ iff both $p \in A$ and $XGp \in A$)
- R_β: for every β-formula $\beta \in \Phi_\varphi$, $\beta \in A$ iff either $k_1(\beta) \in A$ or $k_2(beta) \subseteq A$ (or both)

Atoms

Atom over φ (φ-atom)

A φ-atom is a subset $A \subseteq \Phi_\varphi$ satisfying:

- R_{sat}: the conjunction of all local formulae in A is satisfiable
- $R_\neg$: for every $p \in \Phi_\varphi$, $p \in A$ iff $\neg p \ / \in A$ (i.e., for every $p \in \Phi_\varphi$, a φ-atom must contain either p or $\neg p$)
- R_α: for every α-formula $\alpha \in \Phi_\varphi$, $\alpha \in A$ iff $k(\alpha) \subseteq A$ (e.g., $Gp \in A$ iff both $p \in A$ and $XGp \in A$)
- R_β: for every β-formula $\beta \in \Phi_\varphi$, $\beta \in A$ iff either $k_1(\beta) \in A$ or $k_2(beta) \subseteq A$ (or both)

Example ($\varphi : Gp \wedge F\neg p$)

$A_1 = \{\varphi, Gp, F\neg p, XGp, XF\neg p, p\}$ is an atom
$A_2 == \{\varphi, Gp, F\neg p, XGp, \neg XF\neg p, \neg p\}$ is not (R_α is violated)

Intended meaning of atoms

Atoms are used to represent maximal mutually satisfiable sets of formulae

Intended meaning of atoms

Atoms are used to represent maximal mutually satisfiable sets of formulae

Definition

A set of formulae $S \subseteq \Phi_\varphi$ is **mutually satisfiable** if there exist a model σ and a position $j \geq 0$ such that every formula $p \in S$ holds at position j

Intended meaning of atoms

Atoms are used to represent maximal mutually satisfiable sets of formulae

Definition

A set of formulae $S \subseteq \Phi_\varphi$ is **mutually satisfiable** if there exist a model σ and a position $j \geq 0$ such that every formula $p \in S$ holds at position j

Proposition

For any set of mutually satisfiable formulae $S \subseteq \Phi_\varphi$ there exists a φ-atom A such that $S \subseteq A$

Intended meaning of atoms

Atoms are used to represent maximal mutually satisfiable sets of formulae

Definition

A set of formulae $S \subseteq \Phi_\varphi$ is **mutually satisfiable** if there exist a model σ and a position $j \geq 0$ such that every formula $p \in S$ holds at position j

Proposition

For any set of mutually satisfiable formulae $S \subseteq \Phi_\varphi$ there exists a φ-atom A such that $S \subseteq A$

The opposite does not hold: it may happen that $S \subseteq \Phi_\varphi$ and there exists a φ-atom A such that $S \subseteq A$, but S is not mutually satisfiable (e.g., $Xp \wedge X\neg p$)

Basic (or elementary) formulae

Definition

Basic formulae are propositions or formulae of the form Xp

Basic (or elementary) formulae

Definition

Basic formulae are propositions or formulae of the form Xp

Property of basic formulae

The presence or absence of basic formulae in an atom A determine the presence or absence of all other closure formulae in A

Basic (or elementary) formulae

Definition

Basic formulae are propositions or formulae of the form Xp

Property of basic formulae

The presence or absence of basic formulae in an atom A determine the presence or absence of all other closure formulae in A

Example ($\varphi : Gp \wedge F\neg p$)

Suppose that $XGp \in A$ and $XF\neg p \in A$, while $p \not\in A$.
From $p \not\in A$, it follows that $\neg p \in A$
From $p \not\in A$ and $XGp \in A$, it follows that $\neg Gp \in A$
From $\neg p \in A$ and $XF\neg p \in A$, it follows that $F\neg p \in A$
From $Gp \not\in A$ and $F\neg p \in A$, it follows that $\neg\varphi \in A$

Tableau

Given a formula φ, construct a direct graph T_φ such that

Nodes and edges of T_φ

The nodes of T_φ are the atoms of φ and there exists an edge from an atom A to an atom B if for every $Xp \in \Phi_\varphi$, $Xp \in A$ iff $p \in B$

Tableau

Given a formula φ, construct a direct graph T_φ such that

Nodes and edges of T_φ

The nodes of T_φ are the atoms of φ and there exists an edge from an atom A to an atom B if for every $Xp \in \Phi_\varphi$, $Xp \in A$ iff $p \in B$

Tableau

T_φ is the tableau of φ

Tableau

Given a formula φ, construct a direct graph T_φ such that

Nodes and edges of T_φ

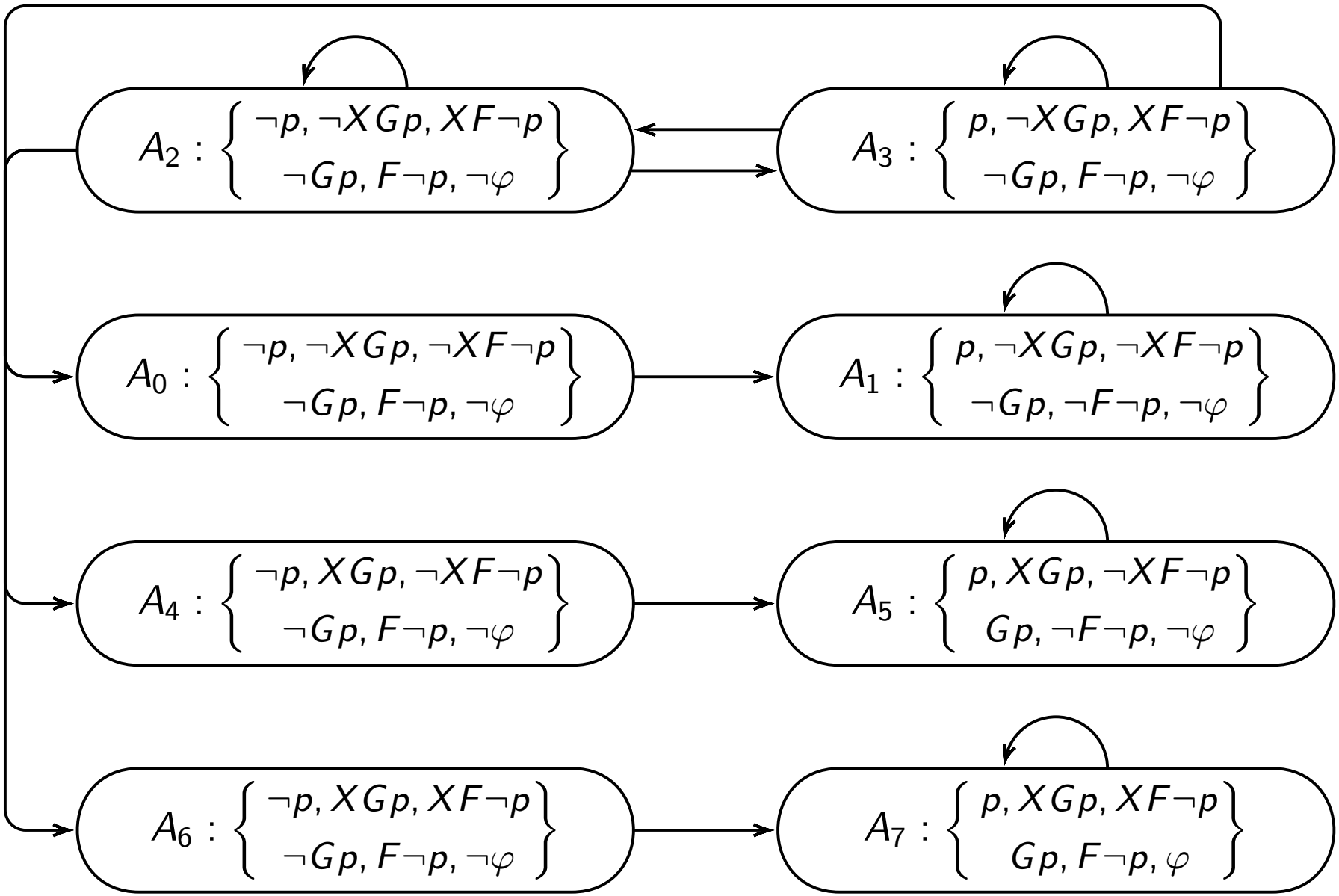
The nodes of T_φ are the atoms of φ and there exists an edge from an atom A to an atom B if for every $Xp \in \Phi_\varphi$, $Xp \in A$ iff $p \in B$

Tableau

T_φ is the tableau of φ

Example ($\varphi : Gp \wedge F\neg p$)

The tableau T_φ of $\varphi = Gp \wedge F\neg p$ is depicted in the next slide


$A_2 : \{\neg p, \neg XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_3 : \{p, \neg XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_0 : \{\neg p, \neg XGp, \neg XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_1 : \{p, \neg XGp, \neg XF\neg p, \neg Gp, \neg F\neg p, \neg\varphi\}$
$A_4 : \{\neg p, XGp, \neg XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_5 : \{p, XGp, \neg XF\neg p, Gp, \neg F\neg p, \neg\varphi\}$
$A_6 : \{\neg p, XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_7 : \{p, XGp, XF\neg p, Gp, F\neg p, \varphi\}$

Models and tableau paths - 1

Definition (induced path)

Given a model σ of φ, the infinite path $\pi_\sigma : A_0, A_1, \ldots$ in T_φ is induced by σ if for every position $j \geq 0$ and every $p \in \Phi_\varphi$, $(\sigma, j) \Vdash p$ iff $p \in A_j$ (in particular, $\varphi \in A_0$)

Models and tableau paths - 1

Definition (induced path)

Given a model σ of φ, the infinite path $\pi_\sigma : A_0, A_1, \ldots$ in T_φ is induced by σ if for every position $j \geq 0$ and every $p \in \Phi_\varphi$, $(\sigma, j) \Vdash p$ iff $p \in A_j$ (in particular, $\varphi \in A_0$)

Proposition

Given a formula φ and a tableau T_φ for it, for every model $\sigma : s_0, s_1, \ldots$ of φ there exists an infinite path $\pi_\sigma : A_0, A_1, \ldots$ in T_φ such that π_σ is induced by σ.

Models and tableau paths - 2

Sketch of the proof

Let $\sigma : s_0, s_1, \ldots$ be a model. For every $j \geq 0$, let A_j be the subset of Φ_ϕ that contains all formulas $p \in \Phi_\phi$ such that $(\sigma, j) \models p$. For every $j \geq 0$, we have that (i) A_j satisfies all the requirements of an atom and (ii) the pair (A_j, A_{j+1}) satisfies the condition on edges. Hence, $\pi_\sigma : A_0, A_1, \ldots$ is an infinite path in T_φ induced by σ.

Models and tableau paths - 2

Sketch of the proof

Let $\sigma : s_0, s_1, \ldots$ be a model. For every $j \geq 0$, let A_j be the subset of Φ_ϕ that contains all formulas $p \in \Phi_\phi$ such that $(\sigma, j) \models p$. For every $j \geq 0$, we have that (i) A_j satisfies all the requirements of an atom and (ii) the pair (A_j, A_{j+1}) satisfies the condition on edges. Hence, $\pi_\sigma : A_0, A_1, \ldots$ is an infinite path in T_φ induced by σ.

An immediate consequence

Since σ is a model of ϕ, we have that $(\sigma, 0) \models \phi$ and thus $\phi \in A_0$

Models and tableau paths - 2

Sketch of the proof

Let $\sigma : s_0, s_1, \ldots$ be a model. For every $j \geq 0$, let A_j be the subset of Φ_ϕ that contains all formulas $p \in \Phi_\phi$ such that $(\sigma, j) \models p$. For every $j \geq 0$, we have that (i) A_j satisfies all the requirements of an atom and (ii) the pair (A_j, A_{j+1}) satisfies the condition on edges. Hence, $\pi_\sigma : A_0, A_1, \ldots$ is an infinite path in T_φ induced by σ.

An immediate consequence

Since σ is a model of ϕ, we have that $(\sigma, 0) \models \phi$ and thus $\phi \in A_0$

The opposite does not hold: not every infinite path in T_φ is induced by some model σ

A (counter)example

The infinite path A_7^ω, where $A_7 = \{p, XGp, XF\neg p, Gp, F\neg p, \varphi\}$, is not induced by any model:

every formula $q \in A_7$ should hold at all positions j, but there exists no model σ such that $F\neg p$ holds at position 0 and p holds at all positions $j \geq 0$.

A (counter)example

The infinite path A_7^ω, where $A_7 = \{p, XGp, XF\neg p, Gp, F\neg p, \varphi\}$, is not induced by any model:

every formula $q \in A_7$ should hold at all positions j, but there exists no model σ such that $F\neg p$ holds at position 0 and p holds at all positions $j \geq 0$.

For what kind of paths does the opposite hold?

Promises and promising formulae

Promise

A formula $\psi \in \Phi_\varphi$ is said **to promise** a formula r if ψ has one of the following forms:

$$Fr \qquad pUr \qquad \neg G \neg r$$

Promises and promising formulae

Promise

A formula $\psi \in \Phi_\varphi$ is said **to promise** a formula r if ψ has one of the following forms:

$$Fr \qquad pUr \qquad \neg G \neg r$$

Property 1

If $(\sigma, j) \Vdash \psi$, then $(\sigma, k) \Vdash r$, for some $k \geq j$

Promises and promising formulae

Promise

A formula $\psi \in \Phi_\varphi$ is said **to promise** a formula r if ψ has one of the following forms:

$$Fr \qquad pUr \qquad \neg G \neg r$$

Property 1

If $(\sigma, j) \Vdash \psi$, then $(\sigma, k) \Vdash r$, for some $k \geq j$

Property 2

The model σ contains infinitely many positions $j \geq 0$ such that

$$(\sigma, j) \Vdash \neg\psi \quad \text{or} \quad (\sigma, j) \Vdash r$$

Fulfilling atoms and paths

Fulfilling atom

An **atom** A **fulfills** a formula ψ, that promises r, if $\neg\psi \in A$ or $r \in A$

Fulfilling atoms and paths

Fulfilling atom

An **atom** A **fulfills** a formula ψ, that promises r, if $\neg\psi \in A$ or $r \in A$

Fulfilling path

A **path** $\pi = A_0, A_1, \ldots$ in T_φ is **fulfilling** if for every promising formula $\psi \in \Phi_\varphi$, π contains infinitely many atoms A_j which fulfill ψ (that is, either $\neg\psi \in A_j$ or $r \in A_j$ or both)

Fulfilling atoms and paths

Fulfilling atom

An **atom** A **fulfills** a formula ψ, that promises r, if $\neg\psi \in A$ or $r \in A$

Fulfilling path

A **path** $\pi = A_0, A_1, \ldots$ in T_φ is **fulfilling** if for every promising formula $\psi \in \Phi_\varphi$, π contains infinitely many atoms A_j which fulfill ψ (that is, either $\neg\psi \in A_j$ or $r \in A_j$ or both)

An example

The path A_7^ω is not fulfilling, because $F\neg p \in \Phi_\varphi$ promises $\neg p$, but $\neg p \not/ A_7$ and $\neg F\neg p \not/ A_7$

Additional examples

The path A_2^ω is fulfilling, because $F\neg p \in \Phi_\varphi$ promises $\neg p$, the path visits A_2 infinitely many times, and both $F\neg p$ and $\neg p$ belong to A_2

The path $(A_2 \cdot A_3)^\omega$ is fulfilling, because $F\neg p \in \Phi_\varphi$ promises $\neg p$, $\neg p \in A_2$, and the path visits A_2 infinitely many times

The path $A_4 \cdot A_5^\omega$ is fulfilling, because $F\neg p \in \Phi_\varphi$ promises $\neg p$, the path visits A_5 infinitely many times, $\neg p$ does not belong to A_5, but $\neg F\neg p (= Gp)$ belongs to A_5

From models to fulfilling paths

Proposition (models induce fulfilling paths)

If $\pi_\sigma = A_0, A_1, \ldots$ is a path induced by a model σ, then π_σ is fulfilling

From models to fulfilling paths

Proposition (models induce fulfilling paths)

If $\pi_\sigma = A_0, A_1, \ldots$ is a path induced by a model σ, then π_σ is fulfilling

Proof

Let $\psi \in \Phi_\phi$ be a formula that promises r. By the definition of model, σ contains infinitely many positions j such that $(\sigma, j) \models \neg\psi$ or $(\sigma, j) \models r$. By the correspondence between models and induced paths, for each of these positions j, $\neg\psi \in A_j$ or $r \in A_j$.

From fulfilling paths to models - 1

Proposition (fulfilling paths induce models)

If $\pi = A_0, A_1, \ldots$ is a fulfilling path in T_φ, then there exists a model σ inducing π, that is, $\pi = \pi_\sigma$ and for every $\psi \in \Phi_\varphi$ and every $j \geq 0$, $(\sigma, j) \Vdash \psi$ iff $\psi \in A_j$

From fulfilling paths to models - 1

Proposition (fulfilling paths induce models)

If $\pi = A_0, A_1, \ldots$ is a fulfilling path in T_φ, then there exists a model σ inducing π, that is, $\pi = \pi_\sigma$ and for every $\psi \in \Phi_\varphi$ and every $j \geq 0$, $(\sigma, j) \Vdash \psi$ iff $\psi \in A_j$

Proof

The proof is by induction on the structure of $\psi \in \Phi_\varphi$.

Base case. For all $j \geq 0$, we require the state s_j of σ to agree with A_j on the interpretation of propositions in Φ_φ, that is, $s_j[p] = \mathit{true}$ iff $p \in A_j$. The case of propositions is thus trivial.

Inductive case. The case of Boolean connectives is straightforward. Let consider the case of X and F.

Let $\psi = Xp$. We have that $(\sigma, j) \Vdash Xp$ iff (definition of X) $(\sigma, j+1) \Vdash p$ iff (inductive hypothesis) $p \in A_{j+1}$ iff (definition on the edges of the tableau) $Xp \in A_j$

From fulfilling paths to models - 2

Proof

Let $\psi = Fr$.

We first prove that $Fr \in A_j$ implies $(\sigma, j) \Vdash Fr$. Assume that $Fr \in A_j$. Since π is fulfilling, it contains infinitely many positions k beyond j such that A_k fulfills Fr. Let $k \geq j$ the smallest $k \geq j$ fulfilling Fr. If $k = j$, then, since Fr in A_j, $r \in A_j$ as well. If $k > j$, then A_{k-1} does not fulfill Fr, that is, it contains both Fr and $\neg r$. By R_β for Fr, $XFr \in A_{k-1}$ and thus $Fr \in A_k$. The only way A_k can fulfill Fr is to have $r \in A_k$. It follows that there always exists $k \geq j$ such that $r \in A_k$. By the inductive hypothesis, $(\sigma, k) \Vdash r$, which, by definition of Fr, implies $(\sigma, j) \Vdash Fr$.

We prove now that $(\sigma, j) \Vdash Fr$ implies $Fr \in A_j$. Assume that $(\sigma, j) \Vdash Fr$ and $Fr \notin A_j$. From $\neg Fr \in A_j$, it follows that $\{\neg r, \neg Fr\} \subseteq A_k$ for all $k \geq j$. By the inductive hypothesis, this implies that $(\sigma, k) \Vdash \neg r$ for all $k \geq j$ (which contradicts $(\sigma, j) \Vdash Fr$).

Satisfiability and fulfilling paths

Main proposition

A formula φ is satisfiable iff the tableau T_φ contains a fulfilling path $\pi = A_0, A_1, \ldots$ such that $\varphi \in A_0$

Proof

The direction from right to left follows from the last lemma (from fulfilling paths to models).

The direction from left to right follows from the previous lemma (from models to fulfilling paths) .

Applications

Is $\varphi : Gp \wedge F\neg p$ satisfiable?

φ is satisfiable if T_φ contains a fulfilling path $\pi = B_0, B_1, \ldots$ with $\varphi \in B_0$

Applications

Is $\varphi : Gp \wedge F\neg p$ satisfiable?

φ is satisfiable if T_φ contains a fulfilling path $\pi = B_0, B_1, \ldots$ with $\varphi \in B_0$

- A_7 is the only atom containing φ (φ-atom)
- A_7^ω is the only infinite path starting at A_7

Applications

Is $\varphi : Gp \wedge F\neg p$ satisfiable?

φ is satisfiable if T_φ contains a fulfilling path $\pi = B_0, B_1, \ldots$ with $\varphi \in B_0$

- A_7 is the only atom containing φ (φ-atom)
- A_7^ω is the only infinite path starting at A_7

Since A_7^ω is not fulfilling, φ is not satisfiable

Applications

Is $\varphi : Gp \wedge F\neg p$ satisfiable?

φ is satisfiable if T_φ contains a fulfilling path $\pi = B_0, B_1, \ldots$ with $\varphi \in B_0$

- A_7 is the only atom containing φ (φ-atom)
- A_7^ω is the only infinite path starting at A_7

Since A_7^ω is not fulfilling, φ is not satisfiable

Is $\neg\varphi : \neg Gp \vee \neg F\neg p$ satisfiable?

$\neg\varphi$ is satisfiable if $T_{\neg\varphi}$ ($= T_\varphi$) contains a fulfilling path $\pi = B_0, B_1, \ldots$ with $\neg\varphi \in B_0$

Applications

Is $\varphi : Gp \wedge F\neg p$ satisfiable?

φ is satisfiable if T_φ contains a fulfilling path $\pi = B_0, B_1, \ldots$ with $\varphi \in B_0$

- A_7 is the only atom containing φ (φ-atom)
- A_7^ω is the only infinite path starting at A_7

Since A_7^ω is not fulfilling, φ is not satisfiable

Is $\neg\varphi : \neg Gp \vee \neg F\neg p$ satisfiable?

$\neg\varphi$ is satisfiable if $T_{\neg\varphi}$ (= T_φ) contains a fulfilling path $\pi = B_0, B_1, \ldots$ with $\neg\varphi \in B_0$

Since A_5^ω is a fulfilling path and A_5 contains $\neg\varphi$, $\neg\varphi$ is satisfiable (model $\langle p : \top \rangle^\omega$)

Strongly connected subgraphs

How do we check the existence of fulfilling paths starting at a φ-atom?

Strongly connected subgraphs

How do we check the existence of fulfilling paths starting at a φ-atom?

Definition (strongly connected subgraph)

A subgraph $S \subseteq T_\varphi$ is a strongly connected subgraph (SCS) if for every pair of distinct atoms $A, B \in S$, there exists a path from A to B which only passes through atoms of S

Strongly connected subgraphs

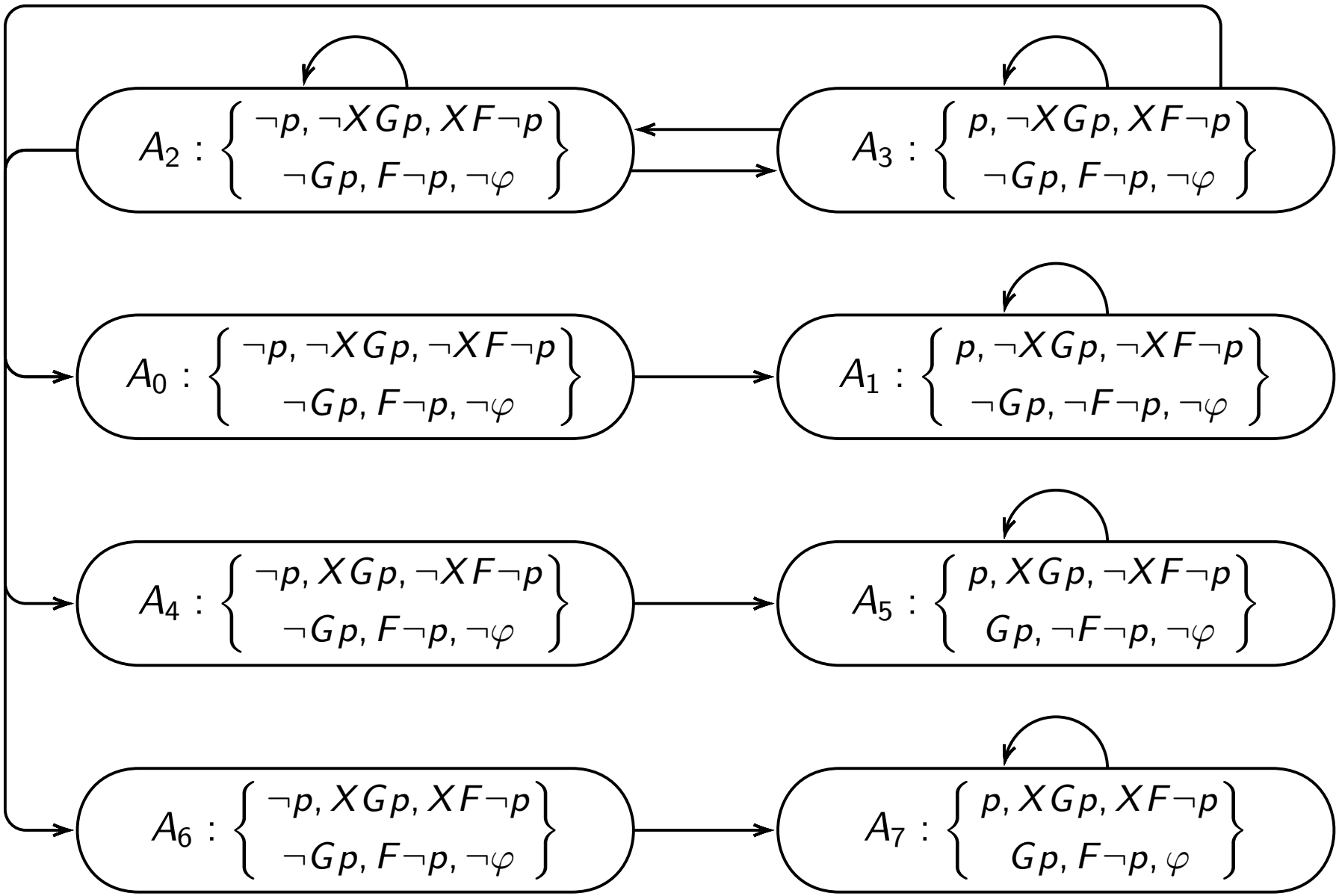
How do we check the existence of fulfilling paths starting at a φ-atom?

Definition (strongly connected subgraph)

A subgraph $S \subseteq T_\varphi$ is a strongly connected subgraph (SCS) if for every pair of distinct atoms $A, B \in S$, there exists a path from A to B which only passes through atoms of S

Definition (fulfilling SCS)

A non-transient **SCS** S is **fulfilling** if every formula $\psi \in \Phi_\varphi$ that promises r is fulfilled by some atom $A \in S$ (either $\neg\psi \in A$ or $r \in A$ or both), where a transient SCS is an SCS consisting of a single node not connected to itself


$A_2 : \{\neg p, \neg XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_3 : \{p, \neg XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_0 : \{\neg p, \neg XGp, \neg XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_1 : \{p, \neg XGp, \neg XF\neg p, \neg Gp, \neg F\neg p, \neg\varphi\}$
$A_4 : \{\neg p, XGp, \neg XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_5 : \{p, XGp, \neg XF\neg p, Gp, \neg F\neg p, \neg\varphi\}$
$A_6 : \{\neg p, XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_7 : \{p, XGp, XF\neg p, Gp, F\neg p, \varphi\}$

Examples

Positive examples

The two SCSs
$\{A_2, A_3\}$
$\{A_5\}$
are fulfilling SCSs.

Examples

Positive examples

The two SCSs
$\{A_2, A_3\}$
$\{A_5\}$
are fulfilling SCSs.

Negative examples

The two SCSs
$\{A_1\}$
$\{A_7\}$
are not fulfilling.

SCS and satisfiability

Definition (φ-reachable SCS)

An SCS is φ-reachable if there exists a finite path $B_0, B_1, \ldots, B_k$ such that $\varphi \in B_0$ and $B_k \in S$

SCS and satisfiability

Definition (φ-reachable SCS)

An SCS is φ-reachable if there exists a finite path $B_0, B_1, \ldots, B_k$ such that $\varphi \in B_0$ and $B_k \in S$

Proposition

The tableau T_φ contains a fulfilling path starting at a φ-atom iff T_φ contains a φ-reachable fulfilling SCS

SCS and satisfiability

Definition (φ-reachable SCS)

An SCS is φ-reachable if there exists a finite path $B_0, B_1, \ldots, B_k$ such that $\varphi \in B_0$ and $B_k \in S$

Proposition

The tableau T_φ contains a fulfilling path starting at a φ-atom iff T_φ contains a φ-reachable fulfilling SCS

Corollary

A formula φ is satisfiable iff T_φ contains a φ-reachable fulfilling SCS

An example

Is $\neg\varphi : \neg Gp \vee \neg F\neg p$ satisfiable?

The SCS $S = \{A_2, A_3\}$ is $(\neg\varphi)$-reachable fulfilling SCS because

$(A_2, A_3)^\omega$: $A_2, A_3, A_2, A_3, \ldots$

and

$\neg\varphi \in S$

Hence, $\neg\varphi$ is satisfiable ((model $(\langle p : \bot\rangle\langle p : \top\rangle)^\omega$)

One step more: maximal SCS

Definition (MSCS)

An SCS is maximal (MSCS) if it is not contained in any larger SCS (notice that there exist at most $|T_\varphi|$ MSCSs)

One step more: maximal SCS

Definition (MSCS)

An SCS is maximal (MSCS) if it is not contained in any larger SCS (notice that there exist at most $|T_\varphi|$ MSCSs)

Example

$\{A_2\}$ and $\{A_3\}$ are not MSCS, while $\{A_2, A_3\}$ is an MSCS

One step more: maximal SCS

Definition (MSCS)

An SCS is maximal (MSCS) if it is not contained in any larger SCS (notice that there exist at most $|T_\varphi|$ MSCSs)

Example

$\{A_2\}$ and $\{A_3\}$ are not MSCS, while $\{A_2, A_3\}$ is an MSCS

Proposition

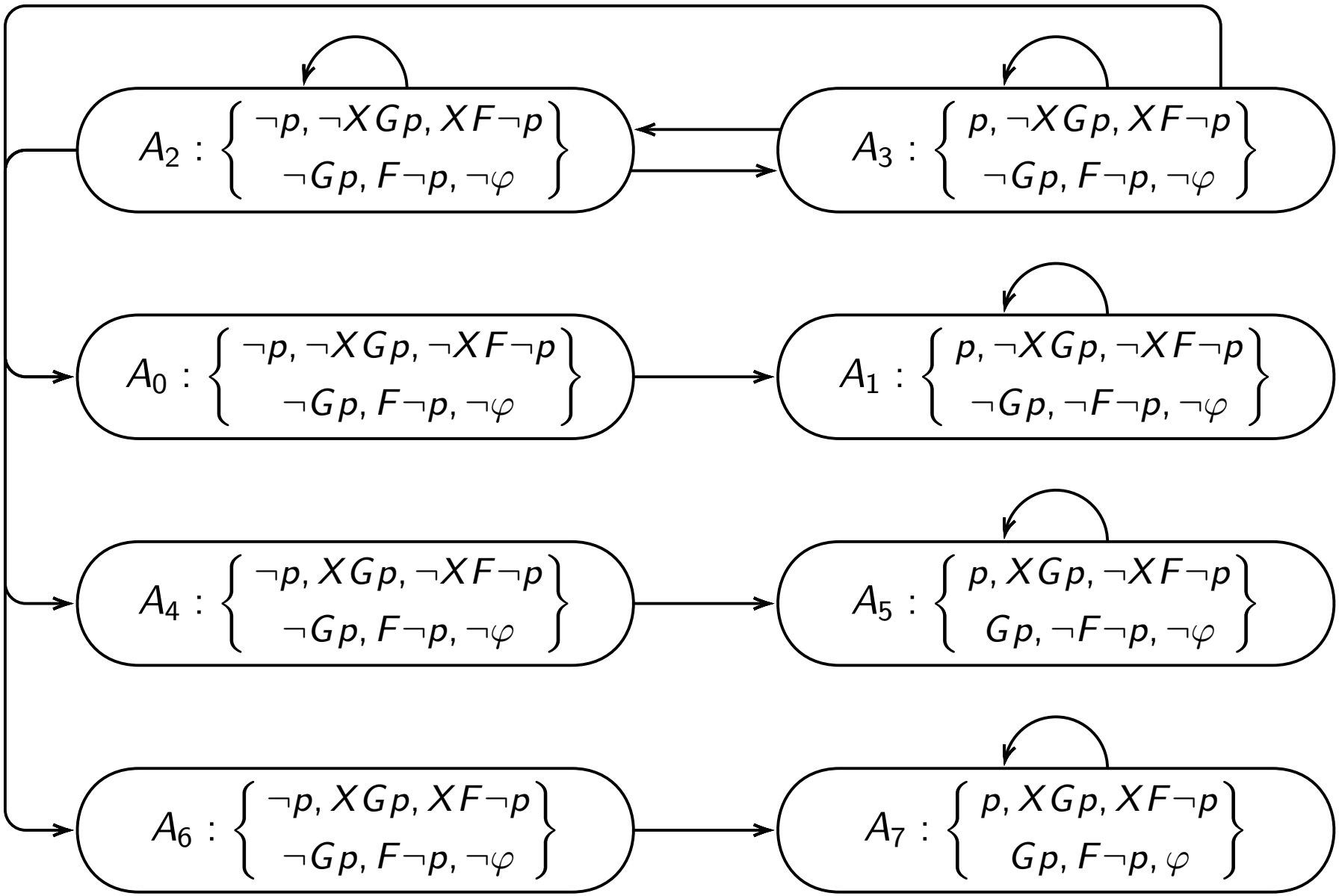
A formula φ is satisfiable iff the tableau T_φ contains a φ-reachable fulfilling MSCS (as a matter of fact, we can preliminarily remove all atoms which are not reachable from a φ-atom)

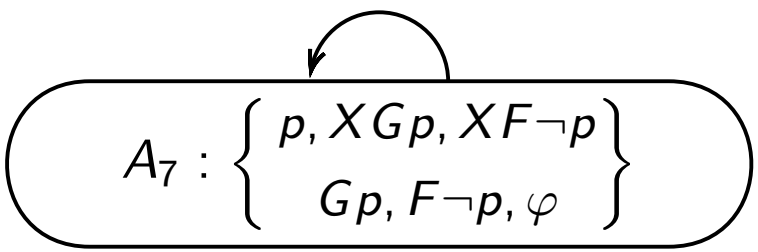
Example 1

Is $\varphi : Gp \wedge F\neg p$ satisfiable?

If we remove all atoms which are not reachable from a φ-atom, the resulting pruned graph (tableau) only includes A_7 connected to itself

The only MSCS is $\{A_7\}$; since it is not fulfilling, it immediately follows that φ is not satisfiable


$A_2 : \{\neg p, \neg XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_3 : \{p, \neg XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_0 : \{\neg p, \neg XGp, \neg XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_1 : \{p, \neg XGp, \neg XF\neg p, \neg Gp, \neg F\neg p, \neg\varphi\}$
$A_4 : \{\neg p, XGp, \neg XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_5 : \{p, XGp, \neg XF\neg p, Gp, \neg F\neg p, \neg\varphi\}$
$A_6 : \{\neg p, XGp, XF\neg p, \neg Gp, F\neg p, \neg\varphi\}$
$A_7 : \{p, XGp, XF\neg p, Gp, F\neg p, \varphi\}$


$A_7 : \left\{ \begin{array}{c} p, XGp, XF\neg p \\ Gp, F\neg p, \varphi \end{array} \right\}$

Example 2

Is $\neg\varphi : \neg Gp \vee \neg F\neg p$ satisfiable?

The removal of all atoms which are not reachable from a $(\neg\varphi)$-atom has not effect in this case: the pruned graph (tableau) coincides with the original one.

The MSCSs are $\{A_0\}$, $\{A_1\}$, $\{A_2, A_3\}$, $\{A_4\}$, $\{A_5\}$, $\{A_6\}$, and $\{A_7\}$

MSCSs $\{A_0\}$, $\{A_4\}$, and $\{A_6\}$ are transient and MSCSs $\{A_1\}$ and $\{A_7\}$ are not fulfilling. However, since both $\{A_2, A_3\}$ and $\{A_5\}$ are fulfilling, it follows that $\neg\varphi$ is satisfiable

Further pruning the tableau

Definition (terminal MSCS)

An MSCS S is terminal if there are no edges leading from atoms of S to atoms outside S

Further pruning the tableau

Definition (terminal MSCS)

An MSCS S is terminal if there are no edges leading from atoms of S to atoms outside S

Examples

$\{A_7\}$ and $\{A_5\}$ are terminal MSCSs, while $\{A_6\}$ and $\{A_2, A_3\}$ are not

Further pruning the tableau

Definition (terminal MSCS)

An MSCS S is terminal if there are no edges leading from atoms of S to atoms outside S

Examples

$\{A_7\}$ and $\{A_5\}$ are terminal MSCSs, while $\{A_6\}$ and $\{A_2, A_3\}$ are not

Pruning criteria

After constructing T_φ,

- remove any MSCS which is not reachable from a φ-atom
- remove any terminal MSCS which is not fulfilling

How can we check the validity of φ?

To check the validity of a formula φ, we can apply the proposed algorithm to $\neg\varphi$.

Possible outcomes:

- If the algorithm reports success, $\neg\varphi$ is satisfiable and thus φ is not valid (the produced model σ is a counterexample to the validity of φ)
- If the algorithm reports failure, $\neg\varphi$ is unsatisfiable and thus φ is valid

Tableau tool for testing satisfiability in LTL: Implementation and experimental analysis

Valentin Goranko[1]

DTU Informatics
Technical University of Denmark
Kgs. Lyngby, Denmark

Angelo Kyrilov[2]

School of Computer Science
University of the Witwatersrand
Johannesburg, South Africa

Dmitry Shkatov[3]

School of Computer Science
University of the Witwatersrand
Johannesburg, South Africa

Abstract

We report on the implementation and experimental analysis of an incremental multi-pass tableau-based procedure à la Wolper for testing satisfiability in the linear time temporal logic LTL. We describe the implementation and discuss the performance of the tool on several series of pattern formulae, as well as on some random test sets, and compare its performance with an implementation of Schwendimann's one-pass tableaux by Widmann and Goré on several representative series of pattern formulae, including eventualities and safety patterns. Our experiments have established that Schwendimann's algorithm consistently, and sometimes dramatically, outperforms the Wolper-style tableaux, despite the fact that the theoretical worst-case upper-bound of Schwendimann's algorithm, 2EXPTIME, is worse than that of Wolper's algorithm, which is EXPTIME. This shows, once again, that theoretically established worst-case complexity results do not always reflect truly the practical efficiency, at least when comparing decision procedures.

Keywords: LTL, satisfiability checking, incremental tableaux, implementation, one-pass tableaux.

1 Introduction

The multiple-pass incremental tableau-based decision procedure for the propositional linear-time logic LTL was first presented in print in [17]; the procedure builds on the ideas originally developed by Pratt for the propositional dynamic logic PDL in [11]. An analogous procedure was developed, at about the same time,

[1] Email: vfgo@imm.dtu.dk
[2] Email: angelo@cs.wits.ac.za
[3] Email: dmitry@cs.wits.ac.za

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs

for the branching-time temporal logic UB by Ben-Ari, Manna, and Pnueli [2]. Subsequently, a number of other decision procedures based on the incremental tableau technology were developed, including our recent work [5,8] on the multi-agent epistemic logics with common and distributed knowledge, on temporal-epistemic logics [6,7], and the logics of strategic ability [4].

The one-pass tableau procedure was first developed for LTL by Schwendimann in [13,14] and recently applied to CTL by Abate, Goré, and Widmann in [1].

It is well-known that the worst-case complexity for LTL is PSPACE [16]. Unless applying on-the-fly pruning, however, the incremental tableau works in EXPTIME, while the worst-case complexity of Schwendimann's method is 2EXPTIME.

In this paper we report on the implementation and preliminary experimental analysis of an incremental tableau-based procedure à la Wolper for LTL. The implementation is available online at `http://msit.wits.ac.za/ltltableau`. We describe the implementation and discuss the performance of the tool on several series of pattern formulae, as well as on some random test sets, and compare its performance with the implementation of Schwendimann's one-pass tableau by Widmann and Goré on several typical series of pattern formulae. Our experiments have shown that Schwendimann's algorithm consistently, and sometimes dramatically, outperforms the multiple-pass incremental algorithm, despite the theoretical advantage of the latter. Schwendimann's algorithm even succeeds on some apparently difficult cases, on which reportedly (see [15], p.9-10) most automata-based tools fail to produce corresponding automata in a reasonable time and our multiple-pass tableaux-based tool fails to establish non-validity, too. We note that neither of the two implementations compared herein is aided by any special optimization techniques; thus, we essentially compare the two algorithms in their "pure" form.

It is well-known, e.g., from research on description logics, that practice does not always comply with theory, viz., that sometimes algorithms for theoretically computationally hard problems can solve most of the practically significant problems efficiently, especially when augmented with optimization techniques. Apparently, we face a similar phenomenon in the case of LTL as well, confirming that theoretical worst-case complexity results should be taken with a grain of salt when determining the practical utility of algorithms.

This paper, being a system description, only reports on the experimental performance comparison between the two tableau methods mentioned above. An in-depth theoretical analysis of the results will be presented in a follow-up work.

2 Preliminaries on the incremental multiple-pass tableaux for LTL

In this section, we briefly sketch out the incremental tableau procedure for LTL whose implementation is reported in this paper. We assume that the reader is familiar with the syntax and semantics of LTL (otherwise, see e.g., [17] or [3]).

In a nutshell, the incremental tableau procedure for testing an LTL-formula θ for satisfiability attempts to construct a graph $\mathcal{T}^\theta$, called a *tableau*, representing sufficiently many *Hintikka structures* for θ in the sense that if any Hintikka structure satisfies θ, then there is at least one represented by $\mathcal{T}^\theta$ that satisfies that formula. A Hintikka structure for θ is, essentially, a finite partial representation of a model

for θ. It is not hard to prove that an LTL formula is satisfiable in a model iff it is satisfiable in a Hintikka structure (for details see, e.g., [6]). Thus, an LTL-formula θ is satisfiable iff the procedure for θ succeeds.

The tableau procedure consists of two major phases: *construction*, and *elimination*, the latter, in turn, consisting of *pre-state elimination*, and *state elimination*.

During the construction phase, a directed graph $\mathcal{P}^\theta$—referred to as the *pretableau* for θ—is produced. Its set of nodes properly contains the set of nodes of the tableau $\mathcal{T}^\theta$ that the procedure is ultimately trying to build. Nodes of $\mathcal{P}^\theta$ are sets of LTL-formulae, some of which—referred to as *states*— represent states of a Hintikka structure (and, therefore, states of a model), while others—referred to as *pre-states*—fulfill a technical role, in particular of helping to keep $\mathcal{P}^\theta$ finite.

During the pre-state elimination phase, a smaller graph $\mathcal{T}_0^\theta$ is created out of $\mathcal{P}^\theta$—referred to as the *initial tableau for* θ—by eliminating all the pre-states from $\mathcal{P}^\theta$, as they have already fulfilled their role, and redirecting the edges.

Lastly, during the state elimination phase, all, if any, states of $\mathcal{T}_0^\theta$ are removed that cannot be satisfied in a Hintikka structure, for one of the following reasons: either they are *patently inconsistent*, i.e., contain a complementary pair of formulae $\psi, \neg\psi$, or contain unrealizable eventualities (i.e., formulae of the form $\varphi \mathcal{U} \psi$ such that no state containing ψ can be reached along the states containing φ from the state in question), or do not have any successors (which is against the LTL-semantics), e.g, because all their successors may have been eliminated earlier.

Note, that the removal of "bad" states may have to be repeated many times until a stable configuration is reached, hence the term "multiple-pass" tableau.

The result of the overall procedure is a (possibly empty) subgraph $\mathcal{T}^\theta$ of $\mathcal{T}_0^\theta$, referred to as the *final tableau for* θ. Then, if there is some state Δ in $\mathcal{T}^\theta$ containing θ, the procedure pronounces θ satisfiable; otherwise, θ is declared unsatisfiable.

The completeness proof shows how to build a Hintikka structure, and thus, a model, out of a non-empty final tableau, while the soundness proof shows that the final tableau for any unsatisfiable formula will always be empty.

We will describe briefly the three stages mentioned above in the next section, while describing the implementation. Before that, we need to introduce some standard terminology and notation that will be used later on.

The tables below list the types of LTL formulae classified as α's (conjunctions) and β's (disjunctions), together with their respective conjuncts and disjuncts.

α	α_1	α_2	β	β_1	β_2
$\neg\neg\varphi$	φ	φ	$\neg(\varphi \wedge \psi)$	$\neg\varphi$	$\neg\psi$
$\varphi \wedge \psi$	φ	ψ	$\varphi \vee \psi$	φ	ψ
$\neg(\varphi \vee \psi)$	$\neg\varphi$	$\neg\psi$	$(\varphi \mathcal{U} \psi)$	ψ	$\varphi \wedge \mathcal{X}(\varphi \mathcal{U} \psi)$
$\neg\mathcal{X}\varphi$	$\mathcal{X}\neg\varphi$	$\mathcal{X}\neg\varphi$	$\neg(\varphi \mathcal{U} \psi)$	$\neg\psi \wedge \neg\varphi$	$\neg\psi \wedge \neg\mathcal{X}(\varphi \mathcal{U} \psi)$
$\mathcal{G}\varphi$	φ	$\mathcal{X}\mathcal{G}\varphi$	$\neg\mathcal{G}\varphi$	$\neg\varphi$	$\neg\mathcal{X}\mathcal{G}\varphi$

All the other formulae (propositional parameters and constants, as well as the formulae of the form $\mathcal{X}\varphi$) are called *primitive*. Unlike the case of α- and β-formulae, their truth at a state of a model cannot be reduced to the truth of simpler formulae *at the same state*. A set of LTL-formulae Σ is said to be *downwards-saturated* if,

first, $\alpha \in \Sigma$ implies that both $\alpha_1 \in \Sigma$ and $\alpha_2 \in \Sigma$, and second, if $\beta \in \Sigma$ implies that either $\beta_1 \in \Sigma$ or $\beta_2 \in \Sigma$. A set of formulae Δ is a *maximal downward saturated extension* of the set Γ if, first, Δ is downward-saturated, and second, there is no downward-saturated Δ' such that $\Gamma \subseteq \Delta' \subset \Delta$.

3 Description of the implementation

3.1 Syntax

The algorithm takes as input the formula to be tested (represented by a string), and returns the string `'satisfiable'` if the formula is found to be satisfiable or, otherwise, `'not satisfiable'`. The implementation supports all the usual Boolean and temporal connectives. These are `A` for $\wedge$, `O` for $\vee$, `I` for $\rightarrow$, `N` for $\neg$, `U` for 'Until', `F` for 'Sometime in the future', `G` for 'Always in the future', and `X` for 'Nexttime'. The formulae are inductively defined as follows:

(i) Every propositional variable, encoded here by lower-case Latin letter followed by a decimal, such as `a12`, is a formula.
(ii) If φ is a formula then `N`φ, `X`φ, `F`φ and `G`φ are formulae.
(iii) If φ and ψ are formulae then `(`φ `A` ψ`)`, `(`φ `O` ψ`)`, `(`φ `I` ψ`)` and `(`φ `U` ψ`)` are formulae.

3.2 Data Structures

The tableau is a directed graph, made up of states and pre-states. The generic term *node* will be used to refer to either states or pre-states when it is not important to distinguish between the two. The graph is implemented as a list of nodes. Each node is a record that contains the following fields:

`id`: A unique integer identifier for the node.
`parents`: A list of integers containing the ids of the parents of the node.
`children`: A list of integers containing the ids of the children of the node.
`type`: A string that specifies what type the node is. Possible values are `pre`, `proto` and `state`.
`formulae`: A list of strings that contains all the formulae that are true at the given node.
`marked`: A Boolean flag used for checking eventualities.
`succMarked`: A Boolean flag showing whether successor nodes are marked.

3.2.1 Construction Phase

As already explained, the construction phase produces a graph containing two kinds of node , states and pre-states. Technically, states, unlike pre-states, are required to be downward-saturated (see above). The graph also contains two kinds of edge. One kind of edge connects pre-states to states, and is denoted here by the double arrow $\Rightarrow$. The other kind of edge connects states to pre-states, and is denoted here by the single arrow $\longrightarrow$ (protostates mentioned above are part of the implementation, but do not feature in a high-level description of the procedure; essentially, they are "states in the making"). The construction procedure for a formula θ begins with creating a single pre-state $\{\theta\}$. Afterwards, the procedure alternates between creating states from pre-states using rule **SR** stated below, and pre-states from states using rule **PR** stated below, until we reach saturation.

SR Given a pre-state Γ such that **SR** has not been applied to Γ before, do the following:

(i) add all maximal downward-saturated extensions of Γ that are not patently inconsistent to the pretableau as *states*;
(ii) for each of the newly added states Δ, if Δ does not contain formulae of the form $\mathcal{X}\varphi$, add $\mathcal{X}\top$ to it; call the result Δ';
(iii) for each so created Δ', put $\Gamma \Rightarrow \Delta'$;
(iv) if, however, the part of the pretableau constructed so far already contains Δ', do not create a new copy of Δ', but simply put $\Gamma \Rightarrow \Delta'$.

PR Given a state Δ such that **PR** has not been applied to Δ before, do the following:

(i) add to the pretableau the set of the form $\Gamma = \{\varphi \mid \mathcal{X}\varphi \in \Delta\}$ as *pre-state*, provided it is not patently inconsistent;
(ii) for each so created Γ, put $\Delta \longrightarrow \Gamma$;
(iii) if, however, the pretableau already contains Γ, do not create a new copy of Γ, but simply put $\Delta \longrightarrow \Gamma$.

In the implementation, the construction algorithm starts off by creating the initial node of the tableau. This is the pre-state labelled with the input formula. The two construction rules are then applied continuously until no new nodes are added. The two construction methods, corresponding to the rules described above, are called *alphaBetaRules* and *nextTimeRule*. The *alphaBetaRules* method creates states from pre-states (the intermediate results are called proto-states) by a process of downward saturation and the *nextTime* method creates pre-states from states.

3.2.2 Elimination Phase

The elimination phase begins by removing all the pre-states and all the $\Rightarrow$ edges from the pre-tableau, and accordingly redirecting $\longrightarrow$ edges. The result is called the *initial tableau*. After that, we start eliminating "bad" states. Recall that these are states that are inconsistent, states that contain unfulfilled eventualities, and states that have no successors. The removal of prestates and inconsistent states is trivial. A naive way of checking whether eventualities have been fulfilled may cause the algorithm to run for extremely long time, so a more efficient ranking procedure, called *removeEventualities* is used to detect unfulfilled eventualities. It begins by finding all eventualities in the tableau and storing them in a list. For each eventuality in the list the algorithm does the following:

(i) For every state, set *marked* to false.
(ii) Find all states that fulfill that eventuality and set their *marked* property to true.
(iii) Mark all states whose successors are marked.
(iv) Repeat step (iii) until no more states can be marked.
(v) Remove all states that contain the eventuality and have not been marked.

The *removeNonSuccessors* procedure looks for states with no successors and removes them. The *removeEventualities* and *removeNonSuccessors* procedures are applied repeatedly until no more states can be removed from the tableau. The result

is the *final tableau* structure.

The last step is to check if the final tableau is open or closed. To do this we check if the tableau contains any of its initial states. These are states that contain the input formula. If the tableau is found to be open then the algorithm returns `'satisfiable'` otherwise it returns `'not satisfiable'`.

3.3 The Tableau Algorithm

The main tableau algorithm, as described above, is shown in figure 1 below.

```
Test(formula) {
    tableau = constructPretableau(formula)
    tableau = removepre-states(tableau)
    tableau = removeInconsistent(tableau)

    while(True) {
        current = tableau;
        tableau = removeEventualities(tableau)
        tableau = removeNonSuccessors(tableau)

        if(current == tableau){
            break
        }
    }
    return isOpen(tableau)
}
```

Fig. 1. The incremental tableau algorithm for testing satisfiability of LTL formulae

4 Testing and analysis

4.1 Correctness testing

A large set of random formulae was generated for the empirical testing of the correctness of the implementation. These were tested on our as well as other available tools, in particular, on R. Goré and F. Widmann's implementation of Schwendimann's one-pass tableau, also chosen for the performance comparison. After removing bugs in earlier versions, both tools returned consistent results on all tests. A comparison of the running times is presented in the rest of the paper. All tests were conducted on an Intel Xeon 8-core architecture with 8 GB RAM and the Mac OS X v10.5 operating system. Our tool was coded up in the Python programming language, while R. Goré and F. Widmann's tool was coded up in the OCaml language [4]. Memory usage by both tools was carefully monitored during all tests to ensure that it does not run out. That would cause the computer to start using virtual memory and greatly increase the running times. Virtual memory was not used in any of the tests reported below.

[4] A precise ratio between the performace speeds of the two languages is impossible to determine, as such a ratio depends on a particual computational task, but it is known that Ocaml can be up to 100 faster than Python, and that should be taken into account when comparing the runtimes of the two implementations.

Another way to test the correctness of the implementation was to generate and test large sets of formulae that we knew to be satisfiable, and formulae we knew to be not satisfiable. In particular, we have used sets of such formulae generated by M. Montali [10], and one of them detected a bug in an earlier version of the program.

4.2 Pattern Series

The first set of tests conducted were on pattern series. The patterns we used were taken from the paper of Rozier and Vardi [12], used there to test the performance of automata based tools for testing satisfiability of LTL formulae.

The diagrams in this section present the running times of the two tableau tools on the different patterns, to which we will refer hereafter as 'Wolper's tableau' and 'Schwendimann's tableau', because both implementations faithfully represent the respective algorithms, without any special features that may slow down or speed up the performance in specific cases.

For running times of automata-based tools on the patterns presented below, the reader is referred to Rozier and Vardi [12]. Later on, we briefly discuss the performance comparisons between the automata-based tools and the tableau tool presented here.

The first pattern is the *E-formulae* pattern, being a conjunction of eventualities, of the form $\mathcal{F}p1 \wedge \mathcal{F}p2 \wedge \ldots \wedge \mathcal{F}pn$. The pattern was tested on input sizes varying from $n = 1$ to $n = 10$. The running times are given on the graph in figure 2.

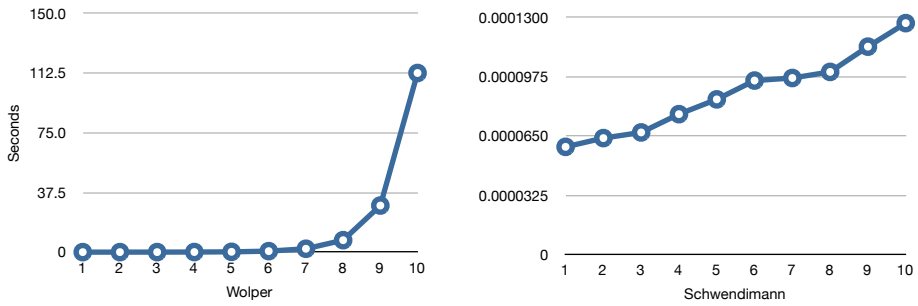


Fig. 2. Running time of E formulae

Wolper's tableau is not able to verify E formulae of more than 10 conjuncts within a reasonable time. As the input grows beyond $n = 7$, we see an exponentially sharp increase in running time. This increase is caused by the procedure that checks whether eventualities are realised. For example when $n = 10$ the program generates over 120 000 nodes in the tableau and 10 eventualities have to be checked. On the other hand, the running time of Schwendimann's tableau grows linearly because there is no separate procedure for checking eventualities.

The next pattern tested was the *S-formulae* pattern, of the form of $\mathcal{G}p1 \wedge \mathcal{G}p2 \wedge \ldots \wedge \mathcal{G}pn$. There are no eventualities in this formula pattern, so we should expect much better results, compared to the E-formulae patter. Indeed, the graph in figure 3 confirms this expectation.

The running time of Wolper's tableau is dominated by the procedure of removing pre-states, which is a costly procedure especially in highly connected graphs.

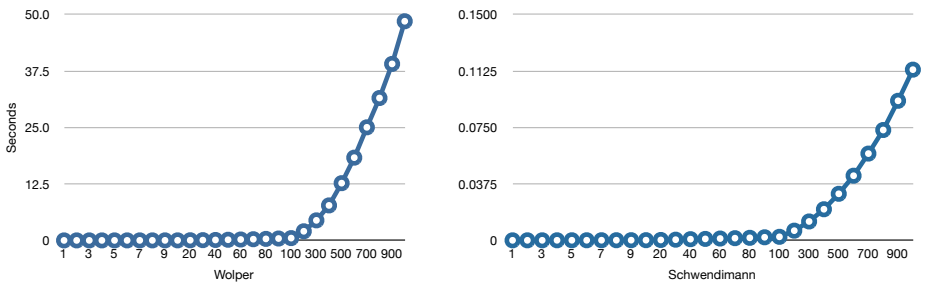


Fig. 3. Running time of S formulae

However, the difference between the running times of the two algorithms is only a constant factor.

The next two patterns involve nested *Until* operators. The first of them, the U_1-*formulae* pattern, is nesting in the first argument: $(((p1\ \mathcal{U}\ p2)\ \mathcal{U}\ p3)\ \mathcal{U}\ \ldots pn)$. The running times of both algorithms are shown in figure 4.

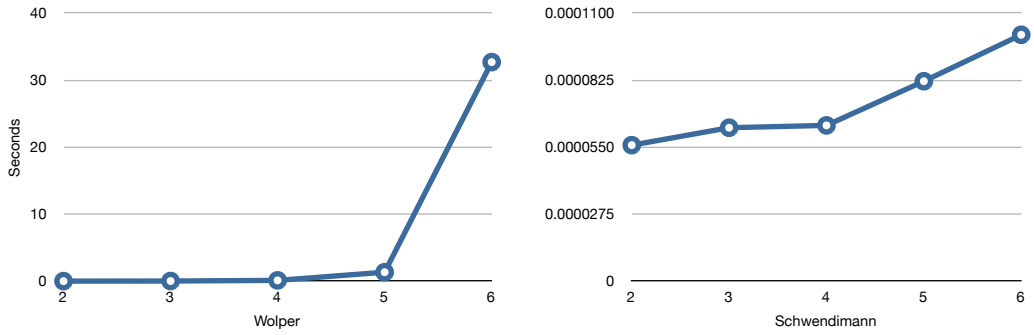


Fig. 4. Running time of U_1 formulae

Again, Wolper's tableau only manages to verify formulae with a very low n value. That is because, for a formula of size n, there are n eventualities to be checked. Also for $n = 7$ the program generates over 68 000 nodes. Again, Schwendimann's tableaux show vastly better behaviour here.

The U_2-*formulae* pattern has nesting on the second argument of *Until*, of the form $(p1\ \mathcal{U}\ (p2\ \mathcal{U}\ (p3\ \mathcal{U} \ldots pn))$. Formulae of this pattern contain n eventualities, too, but Wolper's tableau generates very few states compared to the U_1-formulae pattern. That is why the algorithm manages to verify much larger input formulae. The running times are shown in figure 5. Schwendimann's tableaux perform better again, but as can be seen from the graph, its running time curve grows at a similar rate to that for Wolper's algorithm.

The last pattern sets are the so called C-*formulae* patterns. They are made up of subformulae of the form $\mathcal{GF}pi$. The pattern C_1 is a disjunction of such subformulae, and C_2 is a conjunction of such subformulae. The running times of the algorithms on C_1-formulae are shown in figure 6.

Wolper's tableau manages to verify reasonably-sized formulae of the C_1-formulae pattern because very few nodes are generated. The small number of states allows the procedure to check all n eventualities in a reasonable time. The running time

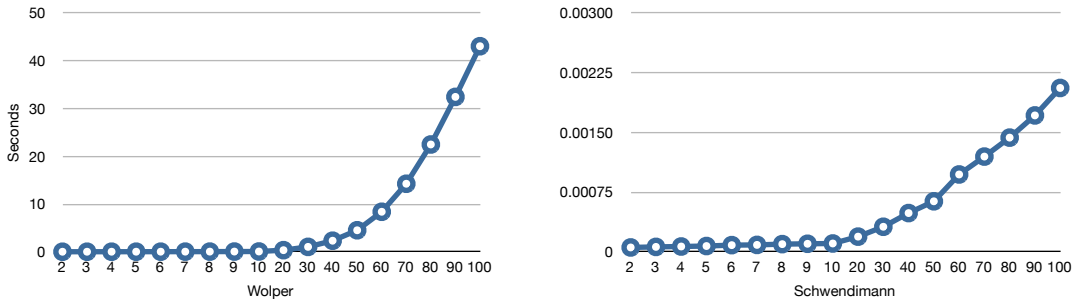


Fig. 5. Running time of U_2 formulae

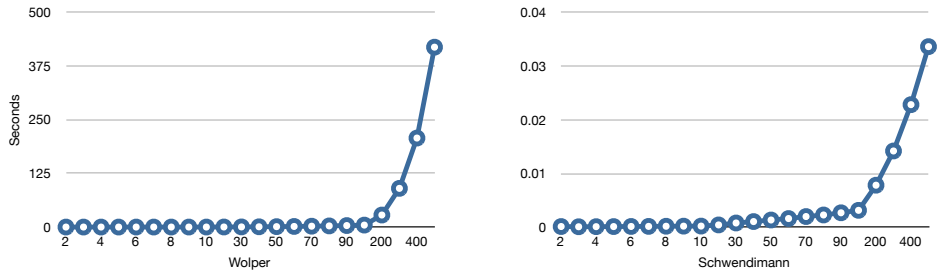


Fig. 6. Running time of C_1 formulae

of Schwendimann's tableau grows at a similar rate but again with a much lower constant factor.

The C_2 pattern is a conjunction of the form $\mathcal{GF}p1 \wedge \mathcal{GF}p2 \wedge \ldots \wedge \mathcal{GF}pn$. The running time of the algorithm on C_2 formulae is shown in figure 7.

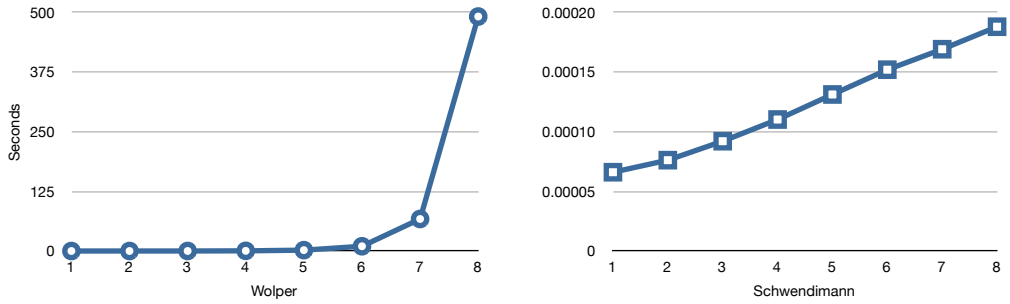


Fig. 7. Running time of C_2 formulae

The running time of Wolper's tableau increases sharply after $n = 7$ because the program begins to generate exponentially many nodes. The need to check whether a lot of eventualities are fulfilled, together with the high number of states, results in poor performance. Schwendimann's tableaux, where there is no elimination procedure, run in linear time for this formula pattern, too.

Other pattern series used to compare Wolper's tableau to Schwendimann's tableau were generated by M. Montalli [10]. These patterns use two parameters n, d, shown on the abscissa of the graphs on Fig. 8 and 9, where the running times for both tools are plotted. The first parameter is the number of propositional variables and the second is the depth of nesting of specific temporal patterns. For

instance, in one of the series the formula $\mathcal{F}(a_1 \wedge \mathcal{X}\mathcal{F}(a_1 \wedge \mathcal{X}Fa_1))$ has parameters $(1, 2)$, meaning that it contains 1 variable and the pattern $\mathcal{X}\mathcal{F}$ has a nesting depth 2.

For a description of the other patterns and further details on them, see [9].

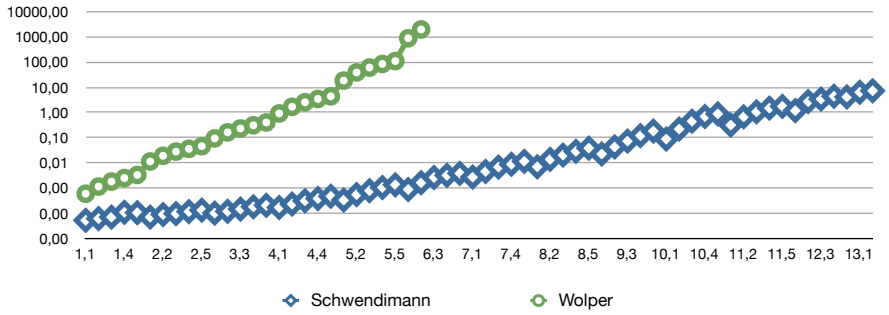


Fig. 8. Running time of Montali's satisfiable formulae

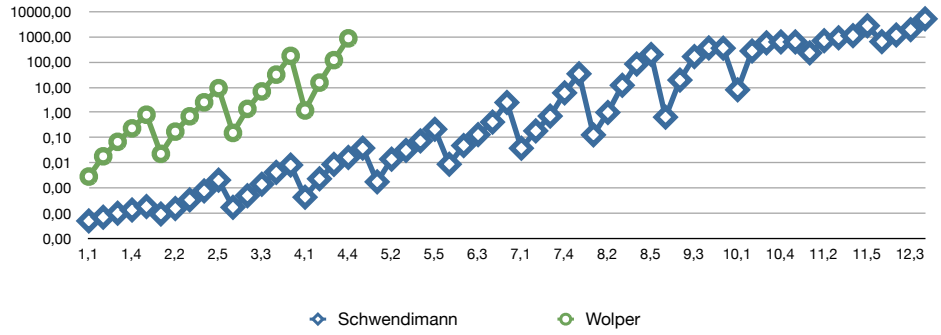


Fig. 9. Running time of Montali's unsatisfiable formulae

4.3 Random formulae

A random formula generator was used to generate random test formulae of different sizes. The parameters used were: n – the number of propositional variables, and d – the nesting depth for operators. Wolper's tableau manages to verify formulae with low nesting depth in a very reasonable time. When the depth is increased to 5 and beyond, the algorithm begins to struggle. As we can see from the graph in figure 10, which shows random formulae of two variables and nesting depth of 5, certain formulae go well beyond the 0.5 second mark. These are all the spikes in the graph, some of which reach times of over 100 seconds. For random formulae of more than 6 propositional variables and nesting depth over 5, Wolper's tableau has running times of over 1000 seconds while Schwendimann's procedure is consistently fast.

4.4 Performance comparisons with automata-based tools

In their paper, Rozier and Vardi tested both explicit and symbolic automata-based tools for LTL satisfiability checking. The implementation of Wolper's tableau compares well with the explicit tools, but is not as efficient as the symbolic ones. On

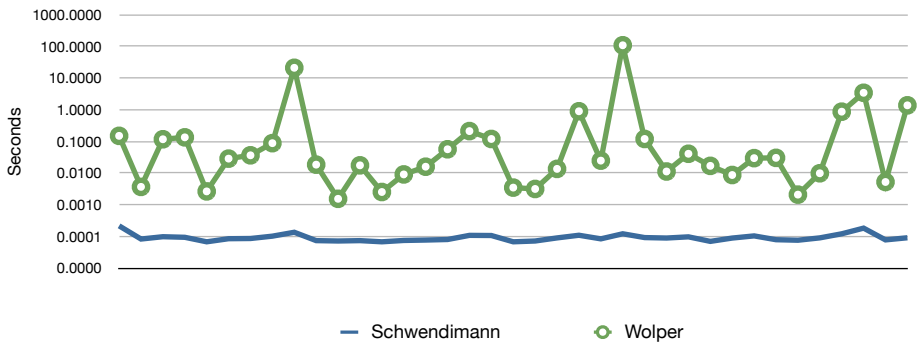


Fig. 10. Running time of random formulae

the other hand, Schwendimann's tableaux have proved to be much more efficient on some formulae patterns.

4.5 Summary of results

The purpose of doing the experimental analysis reported in this paper was twofold: to verify the correctness of the implementation, and to test the performance. The results of the performance testing can be used to determine the suitability of this tool for industrial use, at least for specific formulae patterns.

The correctness was successfully verified with practical certainty, as the last version of the implementation of Wolper's tableau returned correct answers for all the formulae that were tested on it. Also the individual sub-procedures of the tableau were tested independently to ensure their correctness.

As for performance, for formula patterns with no eventualities to be checked, the running times of Wolper's tableau and Schwendimann's tableaux grow at the same rate, typically the growth of the running time of Schwendimann's tableau having much lower constant factors. However, for the formula patterns described above that cause generation of many nodes and there are many eventualities, the running time of Wolper's tableau grows exponentially on the input size, whereas Schwendimann's remains linear.

5 Concluding remarks and future work

In future work we intend to analyze and compare theoretically the incremental multiple-pass, and the one-pass tableau methods and to provide a theoretical explanation of the superior performance of the latter, while identifying the scope of that superior performance and indicating the cases where the multi-pass tableaux perform better. We are also going to investigate optimization techniques of both methods, the ultimate intention being that of designing a "hybrid" procedure using the most optimal features of the both tableau procedures considered in this paper as well as optimization techniques.

6 Acknowledgments

This work is based on the Masters project of the second author (A. Kyrilov), supervised by the other two authors. It has been partly supported by the National

Research Foundation of South Africa.

We are indebted to Rajeev Goré and Florian Widmann for providing us with their implementation of Schwendimann's one-pass procedure and for many discussions on its merits – and this paper can be regarded, inter alia, as a vindication of Rajeev's sustained, but largely shrugged off by the modal logic community, claims on the superior practical performance of Schwendimann's one-pass tableau method.

We are also grateful to Marco Montali for testing and finding a bug in an earlier version of out tool, and for providing us with his benchmark formulae series. Finally, we thank the anonymous referees for their constructive and encouraging remarks.

References

[1] Pietro Abate, Rajeev Goré, and Florian Widmann. One-pass tableaux for Computation Tree Logic. In Nachum Dershowitz and Andrei Voronkov, editors, *Proceedings of LPAR 07*, volume 4790 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 2007.

[2] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. In *Proc. 8th ACM Symposium on Principles of Programming Languages, also appeared in Acta Informatica, 20(1983), 207-226*, pages 164–176, 1981.

[3] E. Allen Emerson. Temporal and modal logics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. MIT Press, 1990.

[4] Valentin Goranko and Dmitry Shkatov. Tableau-based decision procedures for logics of strategic ability in multi-agent systems. To appear in *ACM Transactions on Computational Logic*. Available at `http://tocl.acm.org/accepted.html`.

[5] Valentin Goranko and Dmitry Shkatov. Tableau-based decision procedure for the multi-agent epistemic logic with operators of common and distributed knowledge. In Antonio Cerone and Stefan Gunter, editors, *Proceedings of the sixth IEEE Conference on Sofware Engineering and Formal Methods (SEFM 2008)*, pages 237–246. IEEE Computer Society Press, 2008.

[6] Valentin Goranko and Dmitry Shkatov. Tableau-based decision procedure for full coalitional multiagent temporal-epistemic logic of linear time. In Decker, Sichman, Sierra, and Castelfranchi, editors, *Proc. of the 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009), May 2009, Budapest, Hungary*, 2009.

[7] Valentin Goranko and Dmitry Shkatov. Tableau-based decision procedure for the full coalitional multiagent temporal-epistemic logic of branching time. In *to appear in: Proc. of the Workshop on Formal Approaches to Multi-Agent Systems FAMAS'09, 7-11 September 2009, Torino, Italy*, 2009.

[8] Valentin Goranko and Dmitry Shkatov. Tableau-based procedure for deciding satisfiability in the full coalitional multiagent epistemic logic. In Sergei Artemov and Anil Nerode, editors, *Proc. of the Symposium on Logical Foundations of Computer Science (LFCS 2009)*, volume 5407 of *Lecture Notes in Computer Science*, pages 197–213. Springer-Verlag, 2009.

[9] M. Montali, P. Torroni, M. Alberti, F. Chesani, E. Lamma, and P. Mello. Abductive logic programming as an effective technology for the static verification of declarative business processes. *J. of Algorithms in Cognition, Informatics and Logic*, page to appear, 2009.

[10] Marco Montali. personal communication. 2009.

[11] Vaughan R. Pratt. A near optimal method for reasoning about action. *Journal of Computer and System Sciences*, 20:231–254, 1980.

[12] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In *14th Workshop on Model Checking Software (SPIN '07)*, volume 4595 of *Lecture Notes in Computer Science (LNCS)*, pages 149–167. Springer-Verlag, 2007.

[13] S. Schwendimann. *Aspects of Computational Logic*. PhD thesis, Universität Bern, Switzerland, 1998.

[14] Stefan Schwendimann. A new one-pass tableau calculus for pltl. In H. de Swart, editor, *Proceedings of TABLEAUX'98*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 277–291. Springer-Verlag, 1998.

[15] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: On hybrid approaches for ltl symbolic model checking. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 350–363. Springer, 2005.

[16] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. In *STOC*, pages 159–168. ACM, 1982.

[17] Pierre Wolper. The tableau method for temporal logic: an overview. *Logique et Analyse*, 28(110–111):119–136, 1985.