Generating Oracles from Your Favorite Temporal Logic Specifications*

L. K. Dillon
Computer Science Department
University of California
Santa Barbara, CA 93106

Y. S. Ramakrishna
Computer Science Department
State University of New York
Stony Brook, NY 11794

Abstract

This paper describes a generic tableau algorithm, which is the basis for a general customizable method for producing oracles from temporal logic specifications. A generic argument gives semantic rules with which to build the semantic tableau for a specification. Parameterizing the tableau algorithm by semantic rules permits it to easily accommodate a variety of temporal operators and provides a clean mechanism for fine-tuning the algorithm to produce efficient oracles.

The paper develops conditions to ensure that a set of rules results in a correct tableau procedure. It gives sample rules for a variety of linear-time temporal operators and shows how rules are tailored to reduce the size of an oracle.

Keywords: formal specification, verification, specification-based test oracles, tableau methods, propositional temporal logic, test validation.

1 Introduction

Temporal specifications describe constraints on the order in which events can occur in executions of a concurrent software system. Oracles produced from these specifications can verify that traces generated by executing a system conform to the specifications. Such oracles are needed during system testing to uncover temporal faults in test executions [16]. Proactive debuggers and run-time monitors can use temporal oracles to detect faults, notifying the user when they occur [8]. If efficient, temporal oracles can also be incorporated into robust, self-checking programs [1].

Several methods for checking that traces conform to temporal specifications have been designed for specific notations [2, 4, 12, 13]. The precise semantics of the operators that a notation provides motivate details of the methods. This practice makes it difficult to adapt a trace checker for use with additional operators or with different notations. A relatively minor change to the semantics of an operator may necessitate extensive revision of an oracle and of the method by which oracles are produced.

Notation-specific methods for constructing oracles must handle all operators that can appear in a specification. The introduction of additional operators complicates both the method for constructing oracles and the proof that oracles produced by the method are correct. Similar considerations impede specification-dependent optimizations which could result in more efficient oracles.

This paper describes a generic tableau algorithm, which is the basis for a general customizable method for producing oracles from temporal logic specifications. A procedure for producing semantic tableaux from specifications is obtained by instantiating the tableau algorithm with rules that express the semantics of the operators in a temporal logic. The semantic tableau generated from a specification describes a finite state automaton that accepts precisely those state sequences satisfying the specification. An oracle applies this automaton to finite state sequences that are produced from execution traces during testing.

Parameterizing the tableau algorithm by semantic rules permits it to easily accommodate a variety of temporal operators. This parameterization also provides a clean mechanism for fine-tuning the algorithm to produce automata that are efficiently simulated and from which an oracle can generate meaningful feedback about faults. Semantic rules are subject to logical criteria which guarantee that a tableau procedure terminates and that it produces a correct automaton, i.e., an automaton that accepts precisely those finite state sequences satisfying the input specification.

Section 2 presents an example that illustrates the definition and use of temporal oracles. Section 3 introduces notation and definitions that are needed, in Section 4, for presenting the generic tableau algorithm. This latter section also presents suitable correctness criteria, which ensure that the tableau algorithm produces only correct tableau procedures. Section 5 shows how to customize a tableau procedure. It gives sample rules for a variety of linear time temporal operators and shows how to tailor rules to reduce the size of an oracle. This section also illustrates how to eliminate nondeterminism in the tableau for a formula. In Section 6, we discuss the relationship between our work and the substantial body of work on tableau decision procedures. We briefly describe ongoing research in Section 7.

*This research was supported in part by NSF Grant CCR-9505392.

2 An Example

This section illustrates the definition and use of temporal oracles. In doing so, it helps motivate the algorithm, described in Section 4, for generating semantic tableaux from temporal specifications. The algorithm requires a specification that is written in a propositional linear-time temporal logic (PTL) [10], and must be parameterized by rules that express the semantics of the logical operators.

A PTL formula is evaluated at a *state* within a sequence of states, where a state assigns boolean values to primitive propositions. To interpret PTL formulas as expressing properties of program executions, the software engineer must define the relationship between values of primitive propositions and events that occur during executions. The definitions of propositions determine events that must be monitored when running tests in order to produce *execution traces*, which induce state sequences on which to evaluate PTL specifications. We use PTL formulas to describe properties that are to be verified of test executions. We therefore assume state sequences are

- Finite, because test executions are finite.
- Non-empty, because every test execution begins in an initial state.

The example in [7] illustrates methods for defining primitive propositions and for generating finite state sequences that represent test executions of an Ada tasking program.

For purposes of illustration here, we consider an implementation of a fair mutual exclusion algorithm in which processes that require mutually exclusive access to a shared resource are assigned turns for accessing it. To operate correctly, an implementation of the *distributed mutex algorithm* must assign turns to processes in a fair manner. Additionally, a process may enter a critical section (a section of code that accesses the resource) only in its turn and only when the resource is not being accessed by another process.

We use the following primitive propositions for specifying properties that are to be checked when testing a mutex program that contains two processes:

`turn1`, `turn2`: Process 1 / process 2 has priority to enter a critical section.

`in1`, `in2`: Process 1 / process 2 is in a critical section.

Figure 1 shows several state sequences that might be generated in testing an implementation of the mutex algorithm. Vertical lines denote events that modify the values of primitive propositions. The valuations of propositions are shown graphically, using shading to denote true. (The numbers beneath events are explained subsequently.)

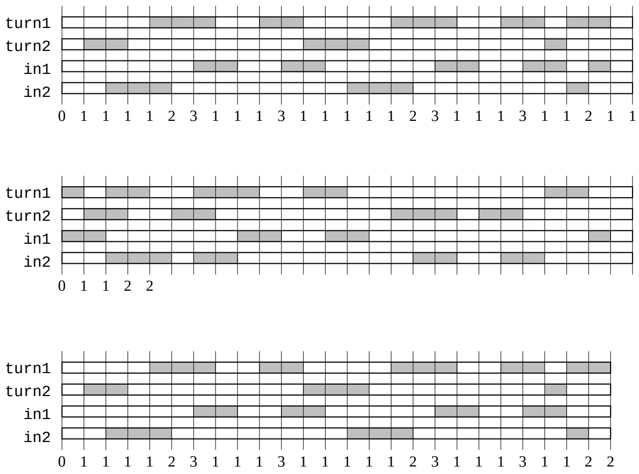


Figure 1: Some example state sequences.

An execution trace satisfies a PTL formula if the formula holds at the first state of the state sequence induced by the trace. The following are examples of two PTL formulas that should be satisfied by every state sequence generated in testing a correct implementation of the mutex algorithm.

$$\Box(\neg\texttt{in1} \vee \neg\texttt{in2}) \qquad (1)$$

$$\Box\,[\texttt{turn1} \Rightarrow ((\texttt{in2} \wedge \texttt{turn1})\,\mathcal{U}\,[(\neg\texttt{in2} \wedge \texttt{turn1})\,\mathcal{U}\,\texttt{in1}])] \qquad (2)$$

The temporal operators in these specifications denote *henceforth*, $\Box$, and *strong until*, $\mathcal{U}$. Informally, specification (1) asserts that the processes access the resource in a mutually exclusive manner (in every state at least one of `in1` or `in2` is false); and specification (2) asserts that, if process 1 has priority to enter a critical section (`turn1` is true), then (i) process 2 does not next enter a critical section before process 1 (when `in2` is next false, it remains false at least until `in1` is true), (ii) process 1 does not lose its turn before obtaining the resource (`turn1` remains true at least until `in1` is true), and (iii) process 1 does eventually access the resource (`in1` is eventually true). Appendix A contains formal definitions for all temporal operators used in this paper.

To obtain semantic tableaux from these specifications, we supply *reduction rules* which express the semantics of the operators in the specifications. A reduction rule associates a formula with a list of *options*, or sets of formulas that imply the associated formula. Table 1 gives reduction rules[1] for a variety of PTL formulas, including those required for building semantic tableaux from specifications (1) and (2). Informally, the options associated with a formula represent different ways in which the formula can be satisfied. For example, rule $[r\Box]$ associates a single option with formula (1), the option $\{\,\neg\texttt{in1} \vee \neg\texttt{in2},\ \bigcirc\Box(\neg\texttt{in1} \vee \neg\texttt{in2})\,\}$. Thus, specification (1) holds at a state in a finite sequence of states precisely if at least one of `in1` and `in2` is false in the given state and either specification (1) again holds at the next state in the sequence or the given state is the final state. Similarly, $[r\mathcal{U}]$ associates two options with $(\neg\texttt{in2} \wedge \texttt{turn1})\,\mathcal{U}\,\texttt{in1}$:

Option 1: It could be that `in1` is true in the given state.

Option 2: It could be that `in2` is false and `turn1` is true in the given state, that this state is not the final state of the sequence, and that $(\neg\texttt{in2} \wedge \texttt{turn1})\,\mathcal{U}\,\texttt{in1}$ holds at the next state in the sequence.

[1] More precisely, the table gives schemas for generating reduction rules. To obtain a reduction rule, the variables in a rule schema must be instantiated with actual formulas.

Because we use specifications in negation normal form (see Section 3), we do not need reduction rules for negation.

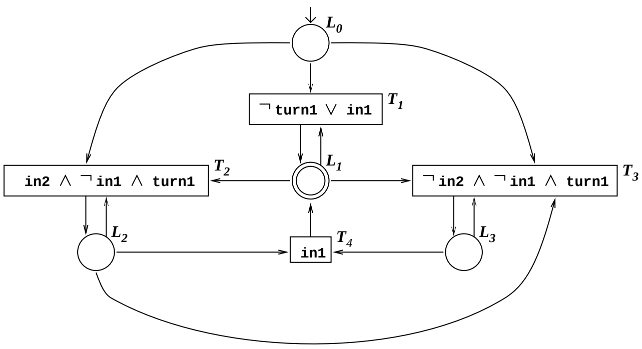
Label	Formula		Options
$[r\wedge]$	$f_1 \wedge f_2$	$\longrightarrow$	$\langle \{ f_1, f_2 \} \rangle$
$[r\vee]$	$f_1 \vee f_2$	$\longrightarrow$	$\langle \{ f_1 \}, \{ f_2 \} \rangle$
$[r\Rightarrow]$	$f_1 \Rightarrow f_2$	$\longrightarrow$	$\langle \{ \neg f_1 \}, \{ f_2 \} \rangle$
$[r\Box]$	$\Box f$	$\longrightarrow$	$\langle \{ f, \bigcirc \Box f \} \rangle$
$[r\Diamond]$	$\Diamond f$	$\longrightarrow$	$\langle \{ f \}, \{ \bigodot \Diamond f \} \rangle$
$[r\mathcal{U}]$	$f_1 \, \mathcal{U} \, f_2$	$\longrightarrow$	$\langle \{ f_2 \}, \{ f_1, \bigodot (f_1 \, \mathcal{U} \, f_2) \} \rangle$
$[r\mathsf{P}]$	$f_1 \, \mathsf{P} \, f_2$	$\longrightarrow$	$\langle \{ f_1, \neg f_2 \}, \{ \neg f_2, \bigcirc (f_1 \, \mathsf{P} \, f_2) \} \rangle$
$[r\mathsf{U}]$	$f_1 \, \mathsf{U} \, f_2$	$\longrightarrow$	$\langle \{ f_2 \}, \{ f_1, \bigcirc (f_1 \, \mathsf{U} \, f_2) \} \rangle$
$[r\mathcal{P}]$	$f_1 \, \mathcal{P} \, f_2$	$\longrightarrow$	$\langle \{ f_1, \neg f_2 \}, \{ \neg f_2, \bigodot (f_1 \, \mathcal{P} \, f_2) \} \rangle$

Table 1: Reduction rules for some common operators

The algorithm described in Section 4 produces a semantic tableau for specification (2) when parameterized with rules from Table 1. Figure 2 shows the tableau as a *flow graph.* A flow graph represents a finite state automaton (FSA) by a bipartite graph in a manner that accords equal status to states and transitions. *Locations* correspond to automaton states and are shown as circles. *Transitions* are denoted by rectangles. A pair of arcs (L, T) and (T, L'), with T a flow graph transition and L and L' flow graph locations, stands for the transition (L, T, L') in the conventional FSA representation. An arrow with no source designates the initial location, and a double circle designates a final one. The input alphabet of a flow graph for a semantic tableau is the set of possible system states, i.e., the set of valuations for the primitive propositions. Transitions are annotated with propositional formulas, which describe those inputs that enable the transitions. In the flow graph in Figure 2, for example, transition T_1 is taken upon checking the first state of a sequence if `turn1` is initially false or `in1` is initially true.

The flow graph for a formula accepts precisely those finite state sequences that satisfy the formula. Thus, an oracle runs the flow graph produced from a specification on the finite state sequences generated during testing to verify that the sequences satisfy the specification. In Figure 1, the numbers beneath events (vertical lines) show the indices of the locations that are encountered when the flow graph of Figure 2 is run on the respective state sequences. The oracle accepts the top sequence since the run ends at L_1, the only final location. The oracle rejects the middle sequence after the first four states because the fifth state does not enable any transition leaving L_2. The oracle rejects the last sequence in Figure 1 because the run does not end at a final location. Thus, the top sequence satisfies specification (2), but the middle and bottom sequences do not.

When a state sequence violates a specification, the semantic tableau contains information that can help in understanding the violation that occurred. As shown in Figure 2, we associate a set of formulas with each location in a flow graph; these are the formulas that must be verified of the remaining state sequence whenever a run of the flow graph enters the location. For example, at every location of the flow graph in Figure 2, the oracle must verify that the remaining input satisfies the invariant (2). In L_2, however, it



$L_0 = \{ \Box(\mathtt{turn1} \Rightarrow ((\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, ((\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}))) \}$

$L_1 = \{ \Box(\mathtt{turn1} \Rightarrow ((\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, ((\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}))) \}$

$L_2 = \{ (\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, ((\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}),$
$\quad \Box(\mathtt{turn1} \Rightarrow ((\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, ((\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}))) \}$

$L_3 = \{ (\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1},$
$\quad \Box(\mathtt{turn1} \Rightarrow ((\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, ((\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}))) \}$

Figure 2: Flow graph generated from (2).

must also check that the remaining input satisfies the subformula:

$$(\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, [(\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}] \qquad (3)$$

Consider then the middle sequence in Figure 1. Since the fifth state of this sequence does not enable any transition out of L_2, the valuation in this state contradicts one (or both) of the formulas to be verified at L_2. In fact, it is easily seen to contradict (3). If we negate (3) and push negations inward, we obtain a characterization of the problem that the oracle encountered: the remaining sequence, starting at the fifth state, satisfies $(\neg\mathtt{in2} \vee \neg\mathtt{turn1}) \, \mathsf{P} \, [(\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}]$. As `turn1` is false in the fifth state, which is the first state of the remaining sequence, we further conclude that the remaining sequence does not satisfy $(\neg\mathtt{in2} \wedge \mathtt{turn1}) \, \mathcal{U} \, \mathtt{in1}$, or equivalently, that the remaining sequence satisfies $(\mathtt{in2} \vee \neg\mathtt{turn1}) \, \mathsf{P} \, \mathtt{in1}$. This observation helps show the underlying problem: process 1 has lost its turn although it has not accessed the resource (`turn1` is false in the fifth state and `in1` is not yet true). Methods for analyzing failures to determine how a state sequence violates a specification are topics of ongoing research and are beyond this scope of this paper.

3 Notation and Definitions

This section establishes notation and terminology needed to describe the algorithm for generating flow graphs from PTL formulas. In the following, *Props* denotes the set of primitive propositions that appear in a formula (specification). The operators in Table 1 denote conjunction, $\wedge$; disjunction, $\vee$; implication, $\Rightarrow$; henceforth, $\Box$; eventually, $\Diamond$; strong until, $\mathcal{U}$; weak precedes, P; weak until, U; and strong precedes, $\mathcal{P}$.

Notation 3.1 We use upper case italic symbols (e.g., S, L) to denote sets; upper case bold symbols (e.g., $\mathbf{S}$, $\mathbf{L}$) to denote sequences; and lower case italic symbols (e.g., f, p)

to denote formulas. Additionally, we use a, b, and c to denote primitive propositions; p and q to denote propositional formulas; and f and g to denote arbitrary formulas. Any of these generic symbols may appear with or without subscripts. As is customary, we identify $\neg\neg f$ with f and *false* with $\neg$*true*.

A sequence $\mathbf{L}$ may be defined by explicitly listing its elements, so that $\mathbf{L} \in X^{\{0,\ldots,n-1\}}$ can be written $\langle \mathbf{L}(0), \ldots, \mathbf{L}(n-1) \rangle$, where $|\mathbf{L}| = n$ is the length of $\mathbf{L}$. The null sequence is denoted Λ, $|\Lambda| = 0$.

We also use S to denote a state, $S \in 2^{Props}$, and $\mathbf{S}$ to denote a sequence of states, $\mathbf{S} \in (2^{Props})^{\{0,\ldots,n-1\}}$.

For expediency, we adopt the convention of pushing negation into primitive propositions.

Definition 3.2 A formula is in *negation normal form* (NNF) if negations apply only to primitive propositions.

For example, if a and b belong to *Props*, then the NNF of $\neg\Box(a \wedge \neg b)$ is $\Diamond(\neg a \vee b)$. Using NNF is not necessary, but it helps streamline the presentation. Of course, this convention requires us to provide a dual (for negation) for each temporal operator.

Notation 3.3 $\mathrm{PTL}(o_1, o'_1, \ldots, o_k, o'_k)$ denotes the propositional temporal logic providing conjunction, disjunction, negation, and *true*, along with the temporal operators o_i and their duals o'_i, for $i = 1, \ldots, k$.

Primitive propositions, their negations, and formulas whose main connective is a next operator play a special role in the tableau construction.

Definition 3.4 We write $\neg$*Props* for the set of negated primitive propositions, $\neg Props = \{ \neg a \mid a \in Props \}$. An *atomic formula* is an element of $Props \cup \neg Props$. A *next formula* has the form $\bigcirc f$ or $\odot f$, for some formula f. A formula is *basic* if it is an atomic formula or a next formula; otherwise the formula is *nonbasic*.

Intuitively, atomic formulas are verified locally (in the current state of a state sequence), and next formulas are verified in the future (at the next state of a state sequence, if any).

It is useful to interpret sets of formulas as conjunctions and sequences of formulas as disjunctions. These conventions allow us to express operations on formulas very conveniently using operations on sets and sequences.

Notation 3.5 A set F of formulas represents the conjunction $\bigwedge_{f \in F} f$ and a sequence $\mathbf{F}$ of formulas represents the disjunction $\bigvee_{f \in \mathbf{F}} f$. Then, a sequence $\mathbf{F}$ of sets of formulas represents the disjunction $\bigvee_{F \in \mathbf{F}} (\bigwedge_{f \in F} f)$, the empty set $\emptyset$ represents *true*, and the empty sequence Λ represents *false*.

For example, the flow graph in Figure 2 associates the invariant being checked with L_0, and in Table 1 the right hand side of rule $[r\mathcal{U}]$ denotes the formula $f_2 \vee [f_1 \wedge \odot(f_1 \,\mathcal{U}\, f_2)]$.

A flow graph is formally defined as follows.

Definition 3.6 (Flow Graph) A *flow graph* G is a seven-tuple of the form

$$G = (Alph, Locs, Trans, Lab, Flow, Init, Fin)$$

where

- *Alph* is a finite input alphabet.
- *Locs* is a finite set of locations.
- *Trans* is a finite set of transitions.
- *Lab* is a transition-labeling function, $Lab\colon Trans \to 2^{Alph}$.
- *Flow* is a flow relation, $Flow \subseteq (Locs \times Trans) \cup (Trans \times Locs)$.
- *Init* is a set of initial locations, $Init \subseteq Locs$.
- *Fin* is a set of final locations, $Fin \subseteq Locs$.

Appendix A provides a formal semantics for flow graphs.

4 Basic Tableau Method

This section reviews the standard tableau algorithm for $\mathrm{PTL}(\Box, \Diamond)$, recasting it in terms of the specific algorithm used in the remainder of the paper. We then outline a correctness proof and identify the criteria that we must satisfy when modifying the algorithm for other logics and for better efficiency.

Given a NNF formula $f_0 \in \mathrm{PTL}(\Box, \Diamond)$, the tableau algorithm for $\mathrm{PTL}(\Box, \Diamond)$ constructs a flow graph with input alphabet 2^{Props} that accepts precisely those input strings satisfying f_0. Each location is identified with a set of formulas, which must be verified of the remaining input sequence.

We illustrate the general method using the formula $\Box(\Diamond\neg b \vee \Box a)$ before presenting the formal algorithm. (See also Example 4.2 below.) The first step creates the initial location $L_0 = \{ \Box(\Diamond\neg b \vee \Box a) \}$. Thus, L_0 is the formula that the flow graph must verify of an input sequence.

The next step *elaborates* a copy T of L_0 into one or more sets of basic formulas, which become the outgoing transitions of L_0. The set T is called a *proper* transition if it consists solely of basic formulas; otherwise we call it an *improper* transition. Elaboration of T preserves its semantics, recursively decomposing it into more basic transitions, until all formulas in each transition can be verified either in the current state or at a succeeding location. The proper transitions define the outgoing transitions of L_0.

Reduction rules perform the task of decomposing improper transitions. A reduction rule $f \longrightarrow \mathbf{F}$, with $\mathbf{F} = \langle F_1, \ldots, F_n \rangle$, is applicable to T if $f \in T$. If applicable, the rule specifies that T is to be replaced by the sets $(T - \{ f \}) \cup F_i$, for $1 \le i \le n$. The next operators are used as auxiliary operators in the right-hand sides of reduction rules to simplify the presentation. Intuitively, whenever $\bigcirc f'$ or $\odot f'$ is present in a transition, f' must be verified at the succeeding location. Moreover, if $\odot f'$ is present, then the succeeding location must not be final.

In the example here, the only rule applicable to $T = \{ \Box(\Diamond\neg b \vee \Box a) \}$ is $[r\Box]$, and its application replaces T with $\{ \Diamond\neg b \vee \Box a, \ \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$. Thus, T is enabled if an input sequence satisfies $\Diamond\neg b \vee \Box a$, and the remaining sequence must satisfy $\Box(\Diamond\neg b \vee \Box a)$. The former formula, which T must verify, is then reduced using $[r\vee]$. This rule splits the improper transition into $\{ \Diamond\neg b, \ \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$ and $\{ \Box a, \ \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$. We now apply $[r\Diamond]$ to the first of these sets and $[r\Box]$ to the second to obtain the sets $T_1 =$

$\{\neg b,\ \bigcirc\Box(\Diamond\neg b \vee \Box a)\}$, $T_2 = \{\odot\Diamond\neg b,\ \bigcirc\Box(\Diamond\neg b \vee \Box a)\}$, and $T_3 = \{a,\ \bigcirc\Box a,\ \bigcirc\Box(\Diamond\neg b \vee \Box a)\}$, which we install as proper transitions of the flow graph.

The transition T_1 can be taken on any input symbol (state) that does not contain b since $\neg b$ is the only atomic formula in T_1. Moreover, the location succeeding T_1 must verify $\{\Box(\Diamond\neg b \vee \Box a)\}$, the same formula to be verified by the initial location. We cannot, however, identify the successor of T_1 with L_0 because T_1's successor is accepting and L_0 is not. (The flow graph must accept a singleton state sequence in which b is false, but it must not accept the empty state sequence.) We therefore create a new location $L_1 = \{\Box(\Diamond\neg b \vee \Box a),\ \#\}$ to succeed T_1, where "#" is a special non-logical symbol used to mark final locations.

The transition T_2 can be taken on *any* input, since T_2 does not contain any propositional formulas, and its successor must verify $\{\Diamond\neg b,\ \Box(\Diamond\neg b \vee \Box a)\}$. Moreover, its successor must not be final, if $\Diamond\neg b$ is to hold at L_0. Thus, we create $L_2 = \{\Diamond\neg b, \Box(\Diamond\neg b \vee \Box a)\}$ to succeed T_2. Similarly, the transition T_3 is enabled by inputs (states) in which a is true, and its successor is $L_3 = \{\Box a, \Box(\Diamond\neg b \vee \Box a), \#\}$.

Having constructed the transitions for L_0 and their successors, the algorithm proceeds to elaborate each of the successors in turn and, in a recursive manner, to elaborate all locations that can be reached from the initial one. The presence of the final location marker # does not affect elaboration of a location, so that, for example, elaboration of L_1 produces the same three transitions as elaboration of L_0. Because the number of formulas that can be generated in this fashion is finite, only a finite number of locations can be produced, and the elaboration process eventually terminates.

The tableau algorithm is defined in Figure 3. The inputs to the algorithm give the formula f_0 that is to be verified, the set *Rules* to use for reducing improper transitions, the function *Applicable* that determines the rules applicable to a transition, and the function *Apply* that returns the result of applying a rule to a transition. The latter three inputs allow us to easily tailor the algorithm by supplying it with different reduction rules and with different methods for applying them. The algorithm returns the flow graph representation of a semantic tableau for f_0.

The components of the flow graph are constructed in the variables *Alph*, *Locs*, *Trans*, *Lab*, *Flow*, *Init*, and *Fin*. The stack (sequence) *L_stack* contains the locations whose outgoing transitions have yet to be elaborated. During elaboration of the outgoing transitions for a location, the stack *E_stack* contains the transitions that require additional elaboration. Lines 11 and 12 perform the elaboration of transitions leaving $L1$. Lines 13–19 are executed when a proper transition has been produced, to install it into the flow graph. The following conventions are assumed.

- *Props* denotes the set of primitive propositions in f_0.
- *Concat* returns the sequence formed by concatenating two sequences.
- *Successor* returns the set of formulas to be verified at a transition's successor:

$$\mathit{Successor}(T) = \begin{array}{l} \{\, f \mid \bigcirc f \in T \text{ or } \odot f \in T \,\} \cup \\ \{\, \# \mid \odot f \in T \,\} \end{array}$$

```
Input:   f0: formula (in NNF)
         Rules: set of reduction rules
         function Apply(reduction rule, set of formulas):
            sequence of sets of formulas
         function Applicable(set of formulas):
            set of reduction rules

Output:  (Alph, Locs, Trans, Lab, Flow, Init, Fin):
            flow graph

1    L0 := { f0 }
2    L_stack := ⟨L0⟩
3    Trans := Flow := Lab := ∅
4    Locs := { L0 }
5    while L_stack ≠ Λ do
6      L1 := pop(L_stack)
       ;; initialize stack of transitions to elaborate
7      E_stack := ⟨L1 − { # }⟩
8      while E_stack ≠ Λ do
9        T := pop(E_stack)
10       if T contains a nonbasic formula then
11         pick a rule R in Applicable(T)
12         E_stack:=
              Simplify(Concat(Apply(R, T), E_stack))
         else     ;;T is a proper transition
13         Trans := Trans ∪ { T }
14         L2 := Successor(T)
15         Flow := Flow ∪ { (L1, T), (T, L2) }
16         Lab := Lab[T ↦ Atoms(T)]
17         if L2 ∉ Locs then
18           Locs := Locs ∪ { L2 }
19           push(L2, L_stack)
           end if
         end if
       end while
     end while
20   Fin := { L ∈ Locs | # ∈ L }
21   Init := { L0 }
22   Alph := 2^Props
23   return (Alph, Locs, Trans, Lab, Flow, Init, Fin)
```

Figure 3: Basic Tableau Algorithm

- *Atoms* returns the set of atoms in a transition, $\mathit{Atoms}(T) = T \cap (\mathit{Props} \cup \neg\mathit{Props})$, which must be true in the current input-state for T to be enabled.
- The function $\mathit{Lab}[T \mapsto \mathit{Atoms}(T)]$ satisfies

$$\mathit{Lab}[T \mapsto \mathit{Atoms}(T)](T') = \begin{cases} \mathit{Atoms}(T), & \text{if } T' = T \\ \mathit{Lab}(T'), & \text{otherwise} \end{cases}$$

- To be *admissible*, the function *Simplify* must return a sequence of transitions that is equivalent to and at least as "simple" as the sequence to which it is applied. In particular, we require $\mathit{Simplify}(\mathbf{T}) \equiv \mathbf{T}$.

We discuss the simplification of transition sequences in Definition 4.10, when we consider termination of tableau procedures. For the examples in this paper, the only simplification performed by the *Simplify* function is removal of transitions that are either *locally inconsistent* or proper supersets of other transitions in the sequence. A transition is locally inconsistent if it contains *false* or if there is a formula f for which the transition contains both f and $\neg f$.

Definition 4.1 The tableau procedure $\mathrm{TPROC}_1(\Box, \Diamond)$ for $\mathrm{PTL}(\Box, \Diamond)$ is obtained by instantiating the algorithm in Figure 3 with the following rules and functions.

- The set $Rules_1$ produced from $[r\wedge]$, $[r\vee]$, $[r\Box]$, and $[r\Diamond]$.
- The function $Applicable_1$ defined by $Applicable_1(T) = \{ R \in Rules \mid R \text{ has the form } f \longrightarrow \mathbf{F} \text{ and } f \in T \}$
- The function $Apply_1$ defined by $Apply_1(R, T) = \langle T_1, \ldots, T_n \rangle$, where R has the form $f \longrightarrow \langle F_1, \ldots, F_n \rangle$ and $T_i = (T - \{ f \}) \cup F_i$, for $1 \le i \le n$.

Example 4.2 The locations, transitions, transition labeling function, and flow relation of the flow graph generated from $\Box(\Diamond\neg b \vee \Box a)$ by the procedure $\mathrm{TPROC}_1(\Box, \Diamond)$ are shown below. In this and in all subsequent examples, the initial location is the one labeled L_0, the final locations are those containing the $\#$ symbol, and the input alphabet is $Alph = 2^{Props}$, where *Props* denotes the primitive propositions that appear in the original formula. In this example, $Props = \{ a, b \}$. In the next section we show how to optimize $\mathrm{TPROC}_1(\Box, \Diamond)$ to reduce the number of locations and the number of transitions that it generates, and how to determinize the resulting flow graph.

Locs:

$$L_0 = \{ \Box(\Diamond\neg b \vee \Box a) \},\ L_1 = \{ \Box(\Diamond\neg b \vee \Box a), \# \},$$
$$L_2 = \{ \Diamond\neg b, \Box(\Diamond\neg b \vee \Box a) \},$$
$$L_3 = \{ \Box a, \Box(\Diamond\neg b \vee \Box a), \# \},$$
$$L_4 = \{ \Diamond\neg b, \Box a, \Box(\Diamond\neg b \vee \Box a) \}$$

Trans:

$$T_1 = \{ \neg b, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$$
$$T_2 = \{ \odot\Diamond\neg b, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$$
$$T_3 = \{ a, \bigcirc\Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$$
$$T_4 = \{ \neg b, a, \bigcirc\Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$$
$$T_5 = \{ a, \odot\Diamond\neg b, \bigcirc\Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$$
$$T_6 = \{ a, \neg b, \odot\Diamond\neg b, \bigcirc\Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$$

Lab:

$$[T_1 \mapsto \neg b;\ T_2 \mapsto true;\ T_3 \mapsto a;\ T_4 \mapsto \neg b \wedge a;\ T_5 \mapsto a;\ T_6 \mapsto a \wedge \neg b]$$

Flow:

$$\{ (L_0, T_1), (T_1, L_1), (L_0, T_2), (T_2, L_2), (L_0, T_3), (T_3, L_3),$$
$$(L_1, T_1), (L_1, T_2), (L_1, T_3),$$
$$(L_2, T_1), (L_2, T_2), (L_2, T_4), (T_4, L_3), (L_2, T_5), (T_5, L_4),$$
$$(L_3, T_3), (L_3, T_5), (L_3, T_6), (T_6, L_4),$$
$$(L_4, T_3), (L_4, T_5), (L_4, T_6) \}$$

Correctness of $\mathrm{TPROC}_1(\Box, \Diamond)$ depends on applying rules in a sound manner and on choosing appropriate final locations, thereby ensuring that each location verifies the formula that the algorithm identifies with it. The following definitions ensure that the application of reduction rules and the choice of final locations are sound.

Definition 4.3 (Valid Rules) Rule R is *valid* iff $R \in Applicable(T)$ implies $T \equiv Apply(R, T)$, for every set T of formulas.[2]

Observation 4.4 Assume the functions $Applicable_1$ and $Apply_1$ are used with a set of reduction rules of the form $f \longrightarrow \mathbf{F}$, where f is a formula and $\mathbf{F}$ is a sequence of sets of formulas. Then the rules are valid if every reduction rule $f \longrightarrow \mathbf{F}$ satisfies $f \equiv \mathbf{F}$ (i.e., $f \equiv \bigvee_{F\in\mathbf{F}}(\bigwedge_{f'\in F} f')$).

If instantiated with a set of valid rules, the algorithm in Figure 3 is (partially) correct, as demonstrated by the next lemma. The statement of the lemma requires the following shorthand for flow graphs.

Notation 4.5 Let $G = (Alph, Locs, Trans, Lab, Flow, Init, Fin)$ and $L \in Locs$ be given. We write $G[L]$ for the flow graph obtained from G by replacing *Init* with $\{ L \}$. That is, $G[L] = (Alph, Locs, Trans, Lab, Flow, Init', Fin)$, where $Init' = \{ L \}$.

Lemma 4.6 Let $G = (Alph, Locs, Trans, Lab, Flow, Init, Fin)$ be the flow graph returned by the algorithm in Figure 3. If *Rules* is valid, then for each location $L \in Locs$, the flow graph $G[L]$ accepts precisely those finite input sequences satisfying L (that is, satisfying $\bigwedge_{f\in L} f$).

A proof of this lemma appears in Appendix B.

To demonstrate that $\mathrm{TPROC}_1(\Box, \Diamond)$ terminates, we show that the number of transitions that can be generated from a given formula is finite and that the rules for reducing transitions elaborate a set of formulas into proper transitions. The following definitions assume that a set *Rules* for reducing transitions, a function *Applicable* for determining the rules that are applicable to a transition, and a procedure *Apply* for applying rules to transitions are given.

Definition 4.7 The *closure* of a formula f is the smallest set C_f of sets of formulas satisfying

- $\{ f \} \in C_f$, and
- if $T \in C_f$, $R \in Applicable(T)$, and $T' \in Apply(R, T)$, then $T' \in C_f$.

Definition 4.8 (Complete Rules) A set of rules is *complete* if the following criteria are met, where T and T' are sets of formulas, R is a rule, and f is a formula:

i) $Applicable(T)$ is empty iff T is proper.

ii) There exists a function m mapping sets of formulas to non-negative integers such that $R \in Applicable(T)$ and $T' \in Apply(R, T)$ imply $m(T') < m(T)$.

Observation 4.9

- Clearly, conditions (i) and (ii) of this definition guarantee that C_f is finite.

[2] More precisely, the validity of a rule is defined with respect to specific application functions. For simplicity of terminology, we refer to a given rule from a set *Rules* as valid or invalid, where the application functions are implicitly understood to be those that have previously been defined for *Rules*. A similar abuse of terminology is made in Definition 4.7 and Definition 4.8.

- The function m measures how far a transition is from being proper. We use the following measure for the examples in this paper:

$$m(T) = \begin{cases} 0, & \text{if } T = \emptyset \\ m(T - \{ f \}), & \text{if } f \in T \text{ and } f \text{ is basic} \\ m(T - \{ f \}) + \text{size}(f), & \\ & \text{if } f \in T \text{ and } f \text{ is nonbasic} \end{cases}$$

  Here, the size of f, size(f), is the sum of the numbers of operators and operands in f.

- Let the functions $\mathit{Applicable}_1$ and Apply_1 from Definition 4.1 be used with a set of reduction rules of the form $f \longrightarrow \mathbf{F}$, and let m be the measure defined above. Then criterion (ii) of Definition 4.8 holds if $m(F') < m(\{ f \})$, for each reduction rule $f \longrightarrow \mathbf{F}$ and each $F' \in \mathbf{F}$. This condition is easily seen to hold for the specific m defined above.

Finally, to prove termination, the procedure for simplifying sequences of transitions must be known to respect the measure in Definition 4.8.

Definition 4.10 The function *Simplify respects* the measure m if

$$\left[\sum_{T \in \mathit{Simplify}(\mathbf{T})} m(T) \right] \leq \left[\sum_{T \in \mathbf{T}} m(T) \right]$$

for all sequences $\mathbf{T}$ of sets of formulas.

The preceding definitions allow us to state sufficient conditions for termination of the tableau algorithm.

Lemma 4.11 The algorithm of Figure 3 terminates if *Rules* is complete and if *Simplify* respects the measure used to demonstrate completeness of *Rules*.

The proof of this lemma is straightforward, using standard techniques for demonstrating termination.

The following correctness theorem is an immediate consequence of Lemma 4.6 and Lemma 4.11.

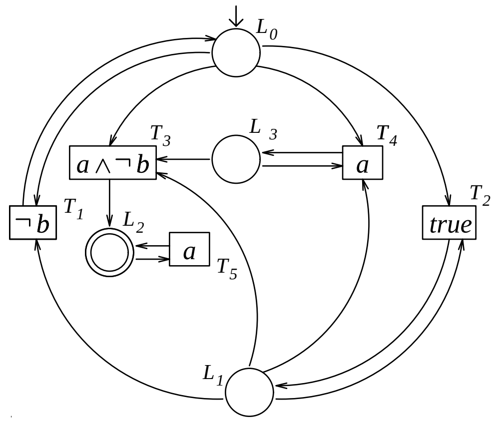
Theorem 4.12 (Correctness) The flow graph that is generated by the tableau algorithm of Figure 3 accepts precisely those finite state sequences satisfying f_0, provided that *Rules* is valid and complete and that *Simplify* respects the measure used to demonstrate completeness of *Rules*.

Corollary 4.13 The algorithm $\mathrm{TPROC}_1(\Box, \Diamond)$ generates a flow graph that accepts precisely those finite state sequences satisfying f_0.

5 Customizing the Tableau Method

To use the tableau method with additional temporal operators, we need only supply it with appropriate reduction rules.

Notation 5.1 We use $\mathrm{TPROC}_1(o_1, o_1', \ldots, o_k, o_k')$ to denote the tableau procedure for $\mathrm{PTL}(o_1, o_1', \ldots, o_k, o_k')$ obtained by instantiating the algorithm in Figure 3 with the appropriate reduction rules from Table 1 and with the functions $\mathit{Applicable}_1$ and Apply_1 defined in (4.1).[3]



$L_0 = \{ (\Box a)\ \mathcal{P}\ (\Box b) \}$, $L_1 = \{ \Diamond\neg b, (\Box a)\ \mathcal{P}\ (\Box b) \}$
$L_2 = \{ \Box a, \# \}$, $L_3 = \{ \Box a, \Diamond\neg b \}$

Figure 4: Flow graph for $(\Box a)\ \mathcal{P}\ (\Box b)$ that is generated by $\mathrm{TPROC}_1(\Box, \Diamond, \mathsf{U}, \mathcal{P})$.

That these procedures are correct follows routinely from Lemma 4.11 using the results in Observation 4.4 and Observation 4.9 to establish validity and completeness, respectively, of the rules.

Example 5.2 The procedure $\mathrm{TPROC}_1(\Box, \Diamond, \mathsf{U}, \mathcal{P})$ requires the rules $[r\vee]$, $[r\wedge]$, $[r\Box]$, $[r\Diamond]$, $[r\mathsf{U}]$, and $[r\mathcal{P}]$. Figure 4 shows the flow graph generated for the formula $(\Box a)\ \mathcal{P}\ (\Box b)$ by this procedure.

Any tableau procedure obtained by instantiating the algorithm in Figure 3 with valid and complete rules is correct. However, the specific rules provided for reducing formulas affect the efficiency of the resulting tableau procedure, the sizes of the flow graphs that the procedure generates, and the time that oracles require to check traces. For example, the rules $[r\vee]$ and $[r\Diamond]$ could be replaced with the following "deterministic" rules:

- $[r\vee]'$ $\quad f_1 \vee f_2 \longrightarrow \langle \{ f_1, \neg f_2 \}, \{ \neg f_1, f_2 \}, \{ f_1, f_2 \} \rangle$
- $[r\Diamond]'$ $\quad \Diamond f \longrightarrow \langle \{ f \}, \{ \neg f, \odot \Diamond f \} \rangle$

If these rules are used with the formula $f_0 = \Box(\Diamond\neg b \vee \Box a)$ of Example 4.2, the extra case split and the negated formulas, which introduce new formulas in C_{f_0}, more than double the size of the resulting flow graph.

We can often reduce the sizes of the flow graphs produced by a tableau procedure quite dramatically by simplifying improper transitions before they are fully elaborated. Consider, for instance, elaboration of the transitions leaving $L_2 = \{ \Diamond\neg b, \Box(\Diamond\neg b \vee \Box a) \}$ in Example 4.2. Application

[3] Negated formulas on the RHS of reduction rules are put in NNF when they require further reduction. No rules are needed for the next operators, as the semantics of the next operators are hard coded into the algorithm.

of $[r\Box]$ produces $\{\, \Diamond\neg b,\ \Diamond\neg b \vee \Box a,\ \bigcirc\Box(\Diamond\neg b \vee \Box a) \,\}$. When viewed as a formula to be verified at L_2, this transition is equivalent to $\{\, \Diamond\neg b,\ \bigcirc\Box(\Diamond\neg b \vee \Box a) \,\}$. Elaborating this latter transition, instead of the former, reduces the number of transitions leaving L_2. Similar simplifications apply to the transitions that are generated when elaborating the outgoing transitions of L_3. In addition to reducing the numbers of transitions leaving L_2 and L_3, the simplifications prevent generation of L_4.

The simplification of arbitrary formulas requires a theorem prover and can, in general, be very hard. The rules for reducing transitions, however, produce sets of formulas that admit fairly routine simplification. We incorporate the simplification of transitions into a tableau procedure by augmenting the rules for reducing formulas with special *replacement rules.*

Definition 5.3 Let each rule in a set of rules be classified as either a reduction rule, of the form $f \longrightarrow \mathbf{F}$, or a replacement rule, of the form $f \overset{F'}{\twoheadrightarrow} F$, where f is a formula, F and F' are sets of formulas, and $\mathbf{F}$ is a sequence of sets of formulas. The functions $Applicable_2$ and $Apply_2$ are then defined, for a rule R and a set T of formulas, as follows.

- $Applicable_2(T) = \{\, f \longrightarrow \mathbf{F} \mid f \in T \,\} \cup \left\{\, f \overset{F'}{\twoheadrightarrow} F \;\middle|\; f \in T \text{ and } F' \subseteq T \,\right\}$
- $Apply_2(R,T) = \begin{cases} Apply_1(R,T), & \text{if } R \text{ has the form } f \longrightarrow \mathbf{F} \\ \langle (T - \{\, f \,\}) \cup F \rangle, & \text{if } R \text{ has the form } f \overset{F'}{\twoheadrightarrow} F \end{cases}$

Intuitively, the rule $f \overset{F'}{\twoheadrightarrow} F$ specifies that f can be replaced by F in transitions that must also verify F'.

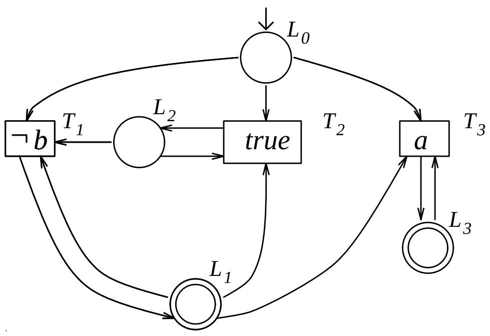
Observation 5.4 Let the application of rules be performed using $Applicable_2$ and $Apply_2$. Then a set of rules is valid if every reduction rule $f \longrightarrow \mathbf{F}$ satisfies $f \equiv \mathbf{F}$ and every replacement rule $f \overset{F'}{\twoheadrightarrow} F$ satisfies $F' \Rightarrow (f \equiv F)$. Moreover, criterion (ii) of Definition 4.8 holds if $m(F') < m(\{\, f \,\})$, for each reduction rule $f \longrightarrow \mathbf{F}$ and each $F' \in \mathbf{F}$, and if $m(F) < m(\{\, f \,\})$, for each replacement rule $f \overset{F'}{\twoheadrightarrow} F$, where m denotes the measure of Observation 4.9.

Notation 5.5 We use $\mathrm{TPROC}_2(o_1, o_1', \ldots, o_k, o_k')$ to denote the tableau procedure for $\mathrm{PTL}(o_1, o_1', \ldots, o_k, o_k')$ obtained by instantiating the algorithm in Figure 3 with the necessary reduction rules from Table 1, with the appropriate replacement rules from those specified below, and with the functions $Applicable_2$ and $Apply_2$ from Definition 5.3.

Example 5.6 The simplifications described above are performed by the replacement rules

$$[R\vee]\ \ f_1 \vee f_2 \overset{\{\, f_1 \,\}}{\twoheadrightarrow} \emptyset \qquad \text{and} \qquad [\vee R]\ \ f_1 \vee f_2 \overset{\{\, f_2 \,\}}{\twoheadrightarrow} \emptyset$$

Figure 5 shows the flow graph for $\Box(\Diamond\neg b \vee \Box a)$ that is produced by the procedure $\mathrm{TPROC}_2(\Box, \Diamond)$ using these rules and some simple heuristics to guide the selection of rules. In particular: replacement rules take precedence over reduction rules; if two formulas in a transition enable reductions and one formula is a proper subformula of the other, we choose the reduction enabled by the larger formula; other things being equal, we assign lower precedence to a reduction that creates a case split than to one that does not. Thus, for example, when elaborating the outgoing transitions for L_2 in Figure 5, we choose the rule obtained by using $[r\Box]$ to reduce $\Box(\Diamond\neg b \vee \Box a)$, rather than that obtained by using $[r\Diamond]$ to reduce $\Diamond\neg b$, since the latter formula is a proper subformula of the former. This reduction produces $\{\, \Diamond\neg b, \Diamond\neg b \vee \Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \,\}$, which is then simplified using $[R\vee]$. Finally, we use $[r\Diamond]$ to generate two proper transitions.



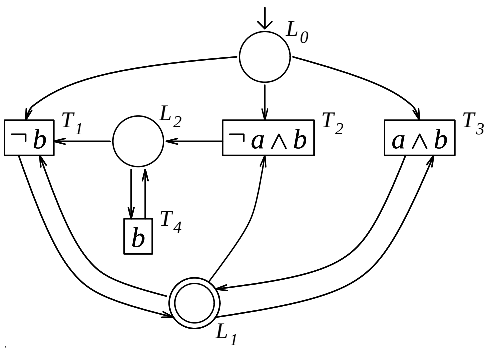
$L_0 = \{\, \Box(\Diamond\neg b \vee \Box a) \,\}$, $L_1 = \{\, \Box(\Diamond\neg b \vee \Box a), \# \,\}$
$L_2 = \{\, \Diamond\neg b, \Box(\Diamond\neg b \vee \Box a) \,\}$, $L_3 = \{\, \Box a, \Box(\Diamond\neg b \vee \Box a), \# \,\}$

Figure 5: Flow graph for $\Box(\Diamond\neg b \vee \Box a)$ that is generated by $\mathrm{TPROC}_2(\Box, \Diamond)$.

An oracle can simulate a nondeterministic flow graph online if a determinized flow graph is substantially larger than a nondeterministic one. However, in our experience, we can usually determinize the flow graph produced by a tableau procedure without unduly increasing its size.

We adapt the standard subset construction for determinizing finite state automata to determinize a flow graph representation of a semantic tableau. The disjunction of the locations that a nondeterministic flow graph (NFG) reaches on a given input defines the formula to be verified at the location of the determinized flow graph (DFG) reached by this input. To the extent practical, logical equivalences may be used to simplify these formulas, and thereby reduce the numbers of locations and transitions in the DFG. Space limitations preclude us from describing the determinization procedure more systematically here.

Example 5.7 Consider the NFG for $\Box(\Diamond\neg b \vee \Box a)$ shown in Figure 5. The formula to be identified with the initial location of the DFG is $\langle L_0 \rangle$, where, as before, we use sequences to express disjunction. The labels on the transitions leaving L_0 partition the input alphabet into those states satisfying $\neg b$, those states satisfying $\neg a \wedge b$, and those states satisfying $a \wedge b$. When the NFG is in L_0, an input satisfying $\neg b$ can take the NFG into any of the locations L_1, L_2, or L_3. Therefore, on inputs satisfying $\neg b$, the location in the DFG succeeding $\langle L_0 \rangle$ must verify $\langle L_1, L_2, L_3 \rangle$. Logical equivalences permit us to replace this latter formula (sequence) with $\langle L_1 \rangle$. The subset construction proceeds using the simplified formula rather than the original formula generated for this input. The table in Figure 6 summarizes the subset construction for this example. Each row of the table describes two edges in the DFG (the two edges that comprise a transition in an FSA representation of the flow graph): an edge from a source location to a transition and an edge from the transition to its successor. For the successor, we show both the disjunction produced from the NFG and the simplified formula that is identified with the successor. Thus, for example, the first three rows of the table correspond to the transitions leaving the initial location. Figure 6 also shows the corresponding DFG.



Trans.	Source	Label	Disj.	Succ.
T_1	$\langle L_0 \rangle$	$\neg b$	$\langle L_1, L_2, L_3 \rangle$	$\langle L_1 \rangle$
T_2	$\langle L_0 \rangle$	$\neg a \wedge b$	$\langle L_2 \rangle$	$\langle L_2 \rangle$
T_3	$\langle L_0 \rangle$	$a \wedge b$	$\langle L_2, L_3 \rangle$	$\langle L_1 \rangle$
T_1	$\langle L_1 \rangle$	$\neg b$	$\langle L_1, L_2, L_3 \rangle$	$\langle L_1 \rangle$
T_2	$\langle L_1 \rangle$	$\neg a \wedge b$	$\langle L_2 \rangle$	$\langle L_2 \rangle$
T_3	$\langle L_1 \rangle$	$a \wedge b$	$\langle L_2, L_3 \rangle$	$\langle L_1 \rangle$
T_1	$\langle L_2 \rangle$	$\neg b$	$\langle L_1, L_2 \rangle$	$\langle L_1 \rangle$
T_4	$\langle L_2 \rangle$	b	$\langle L_2 \rangle$	$\langle L_2 \rangle$

Figure 6: DFG produced from the NFG in Figure 5.

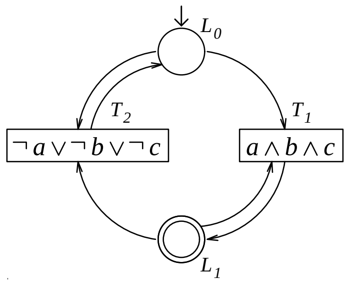
The simplifications that can be performed, both when elaborating transitions and when determinizing an automaton, depend on the formula given as input to a tableau procedure. Thus, the form of an input formula affects the replacement rules that should be considered when elaborating transitions and the equivalences to check when determinizing the flow graph produced by the tableau procedure.

Example 5.8 Example 5.6 admits optimization by $[R\vee]$ and $[\vee R]$ because the input formula contains an embedded disjunction. These rules are not applicable in transitions that are generated from a formula with no embedded disjunctions.

Example 5.9 We supply the replacement rule

$$\odot f \overset{\{\bigcirc\Box f\}}{\twoheadrightarrow} \{\odot \mathit{true}\}$$

when generating flow graphs for formulas containing subformulas of the form $\Box\Diamond f'$ because, when f has the form $\Diamond f'$,



$L_0 = \{\, \Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c \,\}$
$L_1 = \{\, \Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c, \# \,\}$

Figure 7: Flow graph for the formula $\Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c$.

reduction of $\Box f$ produces subformulas of the form $\odot f$ and $\bigcirc\Box f$.

The flow graph generated for $\Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c$ by $\mathrm{TPROC}_2(\Box, \Diamond)$ using this replacement rule contains 2 locations and 8 transitions. When determinized, the flow graph in Figure 7 is obtained. In general, the flow graph generated by $\mathrm{TPROC}_1(\Box, \Diamond)$ for the formula $\Box\Diamond a_1 \wedge \cdots \wedge \Box\Diamond a_n$ contains $2^n + 1$ locations and 2^n transitions. Using the above replacement rule, the algorithm $\mathrm{TPROC}_2(\Box, \Diamond)$ generates a flow graph with 2 locations and 2^n transitions; determinizing this flow graph reduces the number of transitions to 2.

Example 5.10 We check for instances of the equivalence $(f_1 \wedge \Box(f_1 \vee f_2)) \vee (f_2 \wedge \Box(f_1 \vee f_2)) \equiv \Box(f_1 \vee f_2)$ when determinizing a flow graph for a formula that contains a subexpression of the form $\Box(f_1 \vee f_2)$. This equivalence was used to produce the third and sixth rows of the table in Figure 6.

6 Related Work

The relationship between our work on test oracles and that of others engaged in testing research is discussed in [7]. Here, we concentrate on work related to tableau decision procedures in theorem proving and verification.

Most PTLs are interpreted over infinite state sequences, permitting formulas to specify properties of both terminating and nonterminating executions [10]. However, this general interpretation necessitates translation to more powerful (and costly) automata than simple FSA, e.g., Büchi automata [19]. Büchi automata are not required to check traces generated during testing since test traces are necessarily finite.

The first tableau methods for linear time temporal logics were designed for deciding the validity of formulas. An excellent discussion of these methods is found in [18]. The tableau algorithm described herein is very similar to the incremental tableau algorithm for full PTL, with "past" as well as "future" operators, that is presented briefly in [9], and in more detail in [11]; in fact, we borrow terminology from both of those works.

While virtually all tableau methods are driven by semantic rules, little attention has been devoted to correctness criteria for rules. Factoring the proof of correctness of a tableau procedure into a proof for the basic engine and separate proofs that semantic rules are valid and complete reduces the effort required to verify tableau procedures for different logics. Moreover, by isolating the correctness criteria for rules, we are able to generalize the semantic-rule concept from simple reduction rules, which correspond to the rules that drive most of the existing tableau methods, to contextual replacement rules, which facilitate customizing a tableau procedure in a specification-dependent manner. Existing tableau methods perform optimizations similar to those outlined in Section 5, but by hard coding them into the algorithm: Optimizations are always attempted, or never attempted, whereas the form of a specification may preclude any benefit from attempting certain optimizations.

The testing-based method described in Section 2 is related to, but nevertheless different from, both temporal logic-based deductive methods and model checking methods. Deductive methods use the specifications for a system as axioms for verifying critical correctness properties. Properties of the system correspond to theorems: A property p is valid if $(\wedge_i s_i) \Rightarrow p$ is valid, where s_i ranges over the specifications. Because large numbers of specifications are generally required to completely axiomatize the executions of a complex system, the verification of interesting properties of large systems tends to be intractable. But specifications of critical correctness properties tend to be short and concise, and therefore produce efficient oracles with which to check execution traces.

Model checking subjects a model of a system to exhaustive analysis and decides whether the model satisfies a given specification [5]. Current model-checking technology does not yet allow complete automated verification of general systems, because of their large state spaces. For this reason, model checking is used to verify an abstract model of the system, rather than the program(s) implementing it. The proof of correctness of the abstraction itself is often informal, and almost never automated. In contrast, the testing-based method considered in this paper subjects *a set of traces produced by executing a system* to analysis, and decides whether the traces in this set satisfy a given specification. This latter analysis generally does not constitute a verification of the system. However, a good test-generation method may exercise many significant behaviors of a system, and automated analysis of the resulting executions by means of oracles can be useful in revealing faults. (Test generation, and the extent of "coverage" afforded by specific tests, is beyond the scope of this paper.)

Much of the recent impressive progress in model checking can be traced to the development of symbolic techniques, for example binary decision diagrams (BDDs) [3], and of partial order techniques [17]. However, neither symbolic techniques nor partial order techniques would seem beneficial for constructing test oracles. While it is possible to represent the flow relation of the automaton for a formula as a BDD, elaborate encodings are usually not necessary since specifications are small and the resulting automata typically have no more than a few hundred states. Partial order techniques can be of little help since oracles examine actual linear execution traces, not the partial orderings of events representing sets of interleavings.

7 Ongoing Research

This paper has described a very general algorithm for generating efficient temporal oracles. The resulting oracles can check execution traces on-line and report a violation of a specification as early in a trace as possible. They can therefore support run-time monitoring and debugging, in addition to post-execution checking of logged traces. When a trace violates a specification, the oracle can provide feedback regarding the fault that was detected [7].

We have successfully applied the tableau algorithm in this paper with our own high-level temporal logic, Graphical Interval Logic (GIL), an interval temporal logic with a graphical representation [6]. The definition of a tableau procedure for a real-time extension of GIL appears without proof in [14]. Our current work was motivated in part by a desire to simplify the GIL tableau procedure and to facilitate exploration of methods for improving it.

For various reasons which are discussed in [7], the GIL tableau procedure that we implemented for the GIL proof checker does not readily fulfill the needs of an oracle. We are therefore currently implementing an oracle generator for GIL based on the tableau algorithm described in this paper. A parameterizable oracle generator for GIL will allow us to experiment with different reduction and replacement rules and with heuristics for choosing an order for applying rules that reduces the numbers of locations and transitions generated. The GIL oracle generator will also allow us to compare the efficiency of oracles generated by our tableau method with GIL-based oracles that are generated by other methods [13, 15].

Our future plans include the development of an oracle-generator generator (OGG) based on our investigations of tableau algorithms for various logics. The OGG will automate the construction of oracle generators for general temporal logics. It will permit experimental research directed at determining properties of "good" sets of semantic rules that can be used as input to the OGG. It may be possible, for example, to extract a set of semantic rules from a complete deductive system or from an automata-theoretic decision procedure, and then apply validity-preserving transformations to produce good sets of semantic rules.

Acknowledgments: This paper has benefited from Hugh McGuire's careful reading and helpful comments.

A Formal Semantics

The semantics of formulas is given by the satisfaction relation, $\models$, where $\models \subseteq 2^{Props} \times \{0, 1, \dots\} \times$ PTL: A state sequence $\mathbf{S}$ *satisfies* a formula f at state $\mathbf{S}(i)$ if $(\mathbf{S}, i) \models f$, for $0 \leq i \leq |\mathbf{S}| - 1$. The satisfaction relation gives primitive propositions and the classical boolean operators their usual interpretation. Thus, $(\mathbf{S}, i) \models p$ iff $\mathbf{S}(i) \in p$, for a propositional formula $p \in 2^{2^{Props}}$.

The following definition gives the semantics of the temporal operators that are used in this paper. It also shows the equivalences that we use to obtain NNF representations.

Definition A.1

- Weak next $\bigcirc$ and strong next $\odot$:
  $(\mathbf{S}, i) \models \bigcirc f$ iff $|\mathbf{S}| > i$ implies $(\mathbf{S}, i+1) \models f$
  $(\mathbf{S}, i) \models \odot f$ iff $|\mathbf{S}| > i$ and $(\mathbf{S}, i+1) \models f$
  NNF equivalence: $\neg \bigcirc f \equiv \odot \neg f$
- Henceforth $\Box$ and eventually $\Diamond$:
  $(\mathbf{S}, i) \models \Box f$ iff, for all j, $i \leq j < |\mathbf{S}|$ implies $(\mathbf{S}, j) \models f$
  $(\mathbf{S}, i) \models \Diamond f$ iff, for some j, $i \leq j < |\mathbf{S}|$ and $(\mathbf{S}, j) \models f$
  NNF equivalence: $\neg \Box f \equiv \Diamond \neg f$
- Weak until U and strong precedes $\mathcal{P}$:
  $(\mathbf{S}, i) \models f_1 \mathsf{U} f_2$ iff either
  - for some k, $i \leq k < |\mathbf{S}|$, $(\mathbf{S}, k) \models f_2$ and, for all j, $i \leq j < k$ implies $(\mathbf{S}, j) \models f_1$; or
  - for all k, $i \leq k < |\mathbf{S}|$ implies $(\mathbf{S}, k) \models f_1$

  $(\mathbf{S}, i) \models f_1 \mathcal{P} f_2$ iff
  - for all k, $i \leq k < |\mathbf{S}|$ and $(\mathbf{S}, k) \models f_2$ imply, for some j, $i \leq j < k$ and $(\mathbf{S}, j) \models f_1$; and
  - for some k, $i \leq k < |\mathbf{S}|$ and $(\mathbf{S}, k) \models f_1$

  NNF equivalence: $\neg(f_1 \mathsf{U} f_2) \equiv (\neg f_1) \mathcal{P} f_2$
- Strong until $\mathcal{U}$ and weak precedes P:
  $(\mathbf{S}, i) \models f_1 \mathcal{U} f_2$ iff
  - for some k, $i \leq k < |\mathbf{S}|$ and $(\mathbf{S}, k) \models f_2$ and, for all j, $0 \leq j < k$ implies $(\mathbf{S}, j) \models f_1$

  $(\mathbf{S}, i) \models f_1 \mathsf{P} f_2$ iff
  - for all k, $i \leq k < |\mathbf{S}|$ and $(\mathbf{S}, k) \models f_2$ implies, for some j, $i \leq j < k$ and $(\mathbf{S}, j) \models f_1$

  NNF equivalence: $\neg(f_1 \mathcal{U} f_2) \equiv (\neg f_1) \mathsf{P} f_2$

This interpretation can be viewed as specializing the more customary interpretation, defined for infinite state sequences; that is, if $\mathbf{S}$ is infinite, then the above interpretation agrees with the customary interpretation for PTL formulas. (Of course, strong and weak next operators are not distinct when formulas are interpreted over infinite sequences.)

The next definition relates to flow graphs.

Definition A.2 Let $G = (Alph, Locs, Trans, Lab, Flow, Init, Fin)$ denote a flow graph. Additionally, for a location $L \in Locs$, let $Out(L)$ and $In(L)$ denote the sets of outgoing and incoming transitions, respectively.

We say that an input symbol x, $x \in Alph$, *enables* a transition T if $x \in Lab(T)$.

A *move* is an element of $Locs \times Alph \times Trans \times Locs$. We write $L_1 \xrightarrow{x,T} L_2$ to denote the move (L_1, x, T, L_2). A move $L_1 \xrightarrow{x,T} L_2$ is *legal* iff $T \in Out(L_1) \cap In(L_2)$ and x enables T.

If $\mathbf{x} \in Alph^*$ is an input string of length n, a *run* of G from location L to location L' on $\mathbf{x}$ is any sequence of locations $\mathbf{L}$ of length $n+1$ for which $L = \mathbf{L}(0)$, $L' = \mathbf{L}(n)$, and there is some $T_i \in Trans$ for which the move $\mathbf{L}(i) \xrightarrow{\mathbf{x}(i), T_i} \mathbf{L}(i+1)$ is legal, for $0 \leq i < n$.

The flow graph G *accepts* the input string $\mathbf{x}$ iff there is a run $\mathbf{L}$ of G on $\mathbf{x}$ such that $\mathbf{L}(0) \in Init$ and $\mathbf{L}(n) \in Fin$, where $n = |\mathbf{x}|$.

B Partial Correctness Proof

For the proof of Lemma 4.6, assume that $G = (Alph, Locs, Trans, Lab, Flow, Init, Fin)$ denotes the flow graph produced by the tableau algorithm in Figure 3, that the rules used in the construction of G are valid, and that the procedure *Simplify* used in the construction of G is admissible. The following abbreviations are introduced.

Notation B.1 Let $T \in Trans$.

- $Wk\_Succ(T) = \{ f \mid \bigcirc f \in T \}$
- $St\_Succ(T) = \left\{ f \;\middle|\; \odot f \in T \right\}$

Clearly, $Successor(T) \equiv Wk\_Succ(T) \cup St\_Succ(T)$, if the final location marker # is identified with the formula *true* when viewing a location as a conjunction.

The proof of Lemma 4.6 requires the following observations. Informally, the first observation states that a location's outgoing transitions preserve its semantics, and the second says that the moves of the flow graph encode the semantics of transitions.

Observation B.2 If $L \in Locs$, then $L \equiv (\bigvee_{T \in Out(L)} T)$.

Proof. It is easy to see that $Out(L) = \emptyset$, if $L \in L\_stack$, and that the outgoing transitions for L are computed by the `while` loop at line 8 when $L1 = L$. We establish the following loop invariant for this `while` loop.

[*LI*] $L1 \equiv E\_stack \vee \langle T_1, \dots T_k \rangle$, where $\langle T_1, \dots T_k \rangle$ enumerates the transitions in $Out(L1)$.

Clearly, $Out(L1) = \emptyset$ following the assignment to E_stack in line 7. As this assignment ensures $E\_stack \equiv L1$, we conclude that [*LI*] holds when control enters the loop statement.

Validity of the rule R and admissibility of *Simplify* ensure that the body of the `while` loop (lines 9–19) preserves [*LI*] if the first branch (lines 10–12) of the `if` statement at line 11 is executed. Moreover, the body of the `while` loop is easily seen to preserve [*LI*] if the second branch (lines 13–19) is executed. Hence, [*LI*] is a loop invariant.

The observation now follows from [*LI*] and the loop exit condition, $E\_stack = \Lambda$. ∎

Observation B.3 If $L2$ is the location succeeding T, where $T \in Trans$ and $L2 \in Locs$, then

i) $L2 \in Fin$ iff $T \equiv Atoms(T) \wedge \bigcirc L2$

ii) $L2 \notin Fin$ iff $T \equiv Atoms(T) \wedge \odot L2$ and

iii) $(\mathbf{A}, 0) \models Atoms(T)$ iff $\mathbf{A}(0) \in Lab(T)$.

Proof. The algorithm adds T to *Trans* only if all formulas in T are basic, which implies that $T \equiv Atoms(T) \wedge \bigcirc Successor(T)$, if $St\_Succ(T) = \emptyset$, or $T \equiv Atoms(T) \wedge \odot Successor(T)$, if $St\_Succ(T) \neq \emptyset$. Since $L2 = Successor(T)$ by construction and since $L2 \in Fin$ iff $\# \in L2$, which holds precisely if $St\_Succ(T) = \emptyset$, we obtain (i) and (ii).

Observation (iii) holds by construction of Lab. ∎

The following restatement of Lemma 4.6 admits a proof by induction.

Lemma B.4 Let $\mathbf{A} = \langle \mathbf{A}(0), \ldots, \mathbf{A}(n-1) \rangle$ be a state sequence of length $n \geq 1$ and let $L \in Locs$ be a location of flow graph G. Then $G[L]$ accepts $\mathbf{A}$ iff $(\mathbf{A}, 0) \models L$.

Proof. Given $L \in Locs$, we show that there is some transition $T \in Out(L)$ such that $(\mathbf{A}, 0) \models T$ precisely if $G[L]$ accepts $\mathbf{A}$. The lemma then follows from Observation B.2.

For the base case assume $n = 1$. Then $G[L]$ accepts $\mathbf{A}$ iff there is some $T \in Out(L)$ such that $\mathbf{A}(0) \in Lab(T)$ and $L2 \in Fin$, where $(T, L2) \in Flow$. Now by Observation B.3(i), this latter statement is equivalent to $T \equiv Atoms(T) \wedge \bigcirc L2$ and $\mathbf{A} \models Atoms(T)$. Since $n = 1$, we conclude $\mathbf{A} \models T$.

For the induction step, assume $n > 1$, and let $\mathbf{A}' = \langle \mathbf{A}(1), \ldots, \mathbf{A}(n-1) \rangle$. Now $G[L]$ accepts $\mathbf{A}$ precisely if there is some transition $T \in Out(L)$ such that $\mathbf{A}(0) \in Lab(T)$ and such that $G[L2]$ accepts $\mathbf{A}'$, where $(T, L2) \in Flow$. Moreover, by Observation B.3(iii) and the induction hypothesis, this latter statement is equivalent to $(\mathbf{A}, 0) \models Atoms(\mathbf{A})$ and $(\mathbf{A}', 0) \models L2$. Finally, the semantics imply that, since $n > 1$, $(\mathbf{A}', 0) \models L2$ is equivalent to $(\mathbf{A}, 0) \models \bigcirc L2$, and also to $(\mathbf{A}, 0) \models \odot L2$. The induction step now follows from Observations B.3(i) and B.3(ii). ∎

References

[1] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *Proc. IEEE Symp. Foundations of Computer Science*, 1994.

[2] G. V. Bochmann, R. Dssouli, and J. R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Trans. Software Engineering*, 15(11):1347–1355, November 1989.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Computer Science*, pp. 428–439, 1990.

[4] S E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proc. IEEE Real-Time Systems Symp.*, pp. 74–83, San Antonio, TX, December 1991.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2):244–263, April 1986.

[6] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Trans. Software Engineering and Methodology*, 3(2):131–165, April 1994.

[7] L. K. Dillon and Qing Yu. Oracles for checking temporal properties of concurrent systems. In *Proc. 2nd ACM SIGSOFT Symp. Foundations of Software Engineering*, pp. 140–153, New Orleans, December 1994. ACM Press.

[8] V. J. Griswold. Core algorithms for autonomous monitoring of distributed systems. In *Proc. ACM/ONR Work. Parallel and Distributed Debugging*, pp. 23–34, May 1991.

[9] Y. Kesten, Z. Manna, H. McGuire, , and A. Pnueli. A decision algorithm for full propositional temporal logic. In C. Courcoubetis, ed., *Proc. 1993 Int. Conf. Computer-Aided Verification*, vol. 697 of *Lecture Notes in Computer Science*, pp. 97–109, Berlin, 1993. Springer-Verlag.

[10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[11] H. McGuire. *Two Methods for Checking Formulas of Temporal Logic*. PhD thesis, Dept. of Computer Science, Stanford University, Stanford, California, 1995.

[12] A. Morzenti, D. Mandrioli, and C. Ghezzi. A model parametric real-time logic. *ACM Trans. Programming Languages and Systems*, 14(4):521–573, October 1992.

[13] T. O. O'Malley, D. J. Richardson, and L. K. Dillon. Efficient specification-based oracles for critical systems. In Walter Scacchi and Richard Taylor, eds., *Proc. 1996 California Systems Symp.*, pp. 50–59, Los Angeles, CA, April 1996.

[14] Y. S. Ramakrishna. *Interval Logics for Temporal Specification and Verification*. PhD thesis, Dept. Computer and Electrical Engineering, University of California, Santa Barbara, August 1993.

[15] D. J. Richardson. TAOS: Testing with analysis and oracle support. In *Proc. Inter. Symp. Software Testing and Analysis*, pp. 138–153, Seattle, WA, August 1994.

[16] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proc. 14th Inter. Conf. Software Engineering*, pp. 105–118, Melbourne, AUS, May 1992.

[17] A. Valmari. A stubborn attack on state explosion. In Clarke and Kursham, editors, *Proc. 1990 Int'l. Conf. Computer-Aided Verification*, pp. 25–41, New Brunswick, NJ, June 1990. No. 3 in DIMACS Series in Discrete Mathematics and Theoretical, Providence, RI, 1991.

[18] P. Wolper. The tableau method for temporal logic: An overview. In *Logique et Analyse*, vol. 110–111, pp. 119–136, June–September 1985.

[19] P. Wolper. On the relation of programs and computations to models of temporal logic. In *Proc. Conf. Temporal Logic in Specification*, vol. 398 of *Lecture Notes in Computer Science*, pp. 75–123, Altrincham, England, April 1987. Springer-Verlag.