Generating Oracles from Your Favorite Temporal Specifications*

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

Y.S. Ramakrishna
Computer Science Group
T.I.F.R.
Bombay

Abstract

The 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 that ensure 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. To illustrate the versatility of this method, its application to a high level interval logic is discussed.

Keywords: formal specification and validation methods, specification-based test oracles, trace checking, tableau methods, temporal logic, interval logic, concurrent systems

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 [15]. Proactive debuggers and run time monitors can use temporal oracles to detect faults, and notify the user when they occur [8,10]. If efficient, temporal oracles can also be incorporated into robust self-checking programs [1].

Methods for checking that traces conform to temporal specifications are usually designed for specific notations [2, 7, 12]. Details of the methods are motivated by the specific semantics of the operators provided for expressing specifications. This makes it difficult to adapt the method for use with additional operators or different notations. A relatively minor change to the semantics of an operator may necessitate extensive revision of the method for producing oracles.

Notation specific methods for constructing oracles must treat all operators that can appear in a specification. Thus, the introduction of additional operators complicates both the method for constructing oracles and the proof that the method is sound and complete. Similar considerations

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

make it difficult to perform optimizations that, depending on the form of a specification, can 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 tableau procedure is obtained by instantiating the algorithm with rules expressing the semantics of the temporal operators. From a specification, the tableau procedure produces a semantic tableau, which describes a finite state automaton that accepts precisely those execution traces satisfying the specification. An oracle is obtained by simulating this automaton.

Parameterizing the tableau algorithm by semantic rules permits it to easily accommodate a variety of temporal operators. The instantiation of rules also provides a clean mechanism for fine-tuning the algorithm to produce efficient oracles and to permit the generation of meaningful feedback about faults. Logical criteria imposed on semantic rules guarantee that oracles produced by this method are sound and complete.

The next section explains how our work relates to the substantial body of work on tableau decision procedures in theorem proving. Section 3 introduces necessary notation and definitions. The generic tableau algorithm is then presented in Section 4. This section also develops sufficient conditions that ensure a set of rules results in a correct tableau procedure. Section 5 shows how a tableau procedure is customized. 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. This section also indicates how the tableau for a formula can be determinized without increasing its size. To illustrate the versatility of the generic tableau algorithm, its application to a high level interval logic is discussed in Section 6. Finally, conclusions and on-going research are presented.

2 Tableau Decision Procedures

Tableau procedures were invented for deciding the validity of logical formulas. Tableau decision procedures for linear time temporal logics are discussed in [18]; similar procedures for branching time temporal logics are presented in [5]; and a tableau decision procedure for the interval temporal logic of [16] appears in [13]. Differences between these procedures and the tableau procedures discussed below are motivated by demands of the different applications. Tableau decision procedures are intended for proving properties of concurrent systems, while we use a tableau to validate system executions. Because system executions are finite, we interpret temporal formulas over finite state sequences exclusively. This simplifies the tableau construction considerably.

When verifying properties of concurrent systems, the formulas in a set S of temporal logic specifications are viewed as axiomatizing the sequences of states through which the system can pass during executions. Thus, properties of the system correspond to theorems. Validity of a property is checked using a theorem prover: A property p is valid if $(\bigwedge_{s \in S} s) \wedge \neg p$ is unsatisfiable.

The practicality of verification is severely limited by problems of scaling decision procedures for use with large formulas. The tableau decision procedure for linear time propositional temporal logic (PTL) requires time and space exponential in the length of the input formula [17], while the decision problem for the branching time temporal logic CTL* is EXPTIME [6]. Individual specifications for a system and formulas expressing properties that require verification tend to be short and concise. However, complex systems generally require large numbers of specifications. Unless the user can identify a very small subset of specifications that imply a property, deciding if the property is valid quickly becomes intractable.

Tableau decision procedures often assume a small set of relatively low level primitive operators. Higher level operators are defined as abbreviations and are expanded into expressions over the primitive operators prior to building a tableau. This simplifies both the procedure for constructing a tableau and a proof that the procedure is correct. However, expanding the high level operators increases the size of a specification and, therefore, also increases the size of its tableau. A much smaller tableau can often be built directly from a high level specification using rules that express the semantics of the high level operators. Moreover, the direct approach makes it easier to provide the user with meaningful feedback about faults that are detected in system executions [4].

When constructing a tableau for use in validating execution traces, the size of the tableau is more critical than the time required to build it. The tableau is built once, but it is used repeatedly during testing and debugging to validate (typically, very long) execution traces. Thus, we are willing to invest time in building a tableau that can be efficiently simulated.

3 Preliminaries

We consider linear time propositional temporal logics in this paper. Executions of a system are represented as sequences of states for the evaluation of specifications. In this section, $\mathcal{P}$ denotes a set of primitive propositions, from which specifications (formulas) are formed. To streamline the presentation, we assume negation can be "pushed" into primitive propositions.

Definition 3.1 An *atomic* formula is an element of $\mathcal{P} \cup \neg\mathcal{P}$, where $\neg\mathcal{P} = \{\, \neg a \mid\ a \in \mathcal{P} \,\}$ denotes the set of negated primitive propositions.

Logical operators o and o', of arity k, are *duals* if every formula of the form $\neg o(f_1, \ldots, f_k)$ can be expressed in the form $o'(f'_1, \ldots, f'_k)$, where f'_i is either f_i or $\neg f_i$, $1 \leq i \leq k$.

A formula is in *negation normal form* (NNF) iff negations appear only in atomic formulas.

For example, conjunction and disjunction are duals and, if a and b belong to $\mathcal{P}$, then the NNF of $\neg(a \wedge \neg b)$ is $\neg a \vee b$.

Notation 3.2 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.

We also use S to denote a state, $S \in 2^{\mathcal{P}}$, and $\mathbf{S}$ to denote a sequence of states, $\mathbf{S} \in (2^{\mathcal{P}})^{\{\,0,\ldots,n-1\,\}}$, where $|\mathbf{S}| = n$ is the length of $\mathbf{S}$. Further, if $0 \leq k$, $k \leq j$, and $j \leq |\mathbf{S}|$, then $\mathbf{S}[k \ldots j)$ denotes the subsequence of $\mathbf{S}$ beginning with $\mathbf{S}(k)$ and of length $j-k$, $\mathbf{S}[k \ldots j) = \langle \mathbf{S}(k), \ldots, \mathbf{S}(j-1) \rangle$, and $\mathbf{S}^k$ abbreviates the suffix of $\mathbf{S}$ of length $n-k$, where $|\mathbf{S}| = n$, $\mathbf{S}^k = \mathbf{S}[k \ldots n)$. We write Λ for the null sequence, $|\Lambda| = 0$.

In the tableau construction, sets and sequences of formulas are interpreted as representing conditions that must be verified of a state sequence. The following conventions permit operations on formulas to be conveniently expressed in terms of operations on sets and sequences.

Notation 3.3 We use a set F of formulas to represent the conjunction $\bigwedge_{f \in F} f$ and a sequence $\mathbf{F}$ of formulas to represent the disjunction $\bigvee_{f \in \mathbf{F}} f$. Accordingly, a sequence $\mathbf{F}$ of sets of formulas represents the disjunction $\bigvee_{F \in \mathbf{F}}(\bigwedge_{f \in F} f)$. Additionally, $\emptyset \equiv \mathit{true}$ and $\Lambda \equiv \mathit{false}$.

The semantics of formulas is given by the satisfaction relation $\models$ between state sequences and formulas. This relation must give primitive propositions and the classical boolean operators their usual interpretation.

Definition 3.4 A non null state sequence $\mathbf{S}$ *satisfies* a formula f iff $\mathbf{S} \models f$, where we assume the relation $\models$ is defined so that

- $\mathbf{S} \models \mathit{true}$ and $\mathbf{S} \not\models \mathit{false}$
- $\mathbf{S} \models a$ iff $a \in \mathbf{S}(0)$
- $\mathbf{S} \models f_1 \wedge f_2$ iff $\mathbf{S} \models f_1$ and $\mathbf{S} \models f_2$
- $\mathbf{S} \models f_1 \vee f_2$ iff $\mathbf{S} \models f_1$ or $\mathbf{S} \models f_2$, and
- $\mathbf{S} \models \neg f$ iff $\mathbf{S} \not\models f$

where a is a primitive proposition; f, f_1 and f_2 are formulas; and $\mathbf{S}$ is a state sequence with $|\mathbf{S}| > 0$.

These conditions ensure that, for a propositional formula p, $\mathbf{S} \models p$ iff the initial state $\mathbf{S}(0)$ satisfies p under the classical interpretation, in which case we also write $\mathbf{S}(0) \models p$.

A variety of temporal operators have been proposed. The following definition gives the semantics of some common linear time operators, together with their duals, and shows the equivalences that are used to obtain NNF representations.

Definition 3.5

- Weak next $\bigcirc$ and strong next $\odot$:
  $\mathbf{S} \models \bigcirc f$ iff $|\mathbf{S}| > 1$ implies $\mathbf{S}^1 \models f$
  $\mathbf{S} \models \odot f$ iff $|\mathbf{S}| > 1$ and $\mathbf{S}^1 \models f$
  NNF equivalences: $\neg \bigcirc f \equiv \odot \neg f$ and $\neg \odot f \equiv \bigcirc \neg f$
- Henceforth $\Box$ and eventually $\Diamond$:
  $\mathbf{S} \models \Box f$ iff, for all j, $0 \leq j < |\mathbf{S}|$ implies $\mathbf{S}^j \models f$
  $\mathbf{S} \models \Diamond f$ iff, for some j, $0 \leq j < |\mathbf{S}|$ and $\mathbf{S}^j \models f$
  NNF equivalences: $\neg \Box f \equiv \Diamond \neg f$ and $\neg \Diamond f \equiv \Box \neg f$
- Weak until U and strong precedes $\mathcal{P}$:
  $\mathbf{S} \models f_1 \mathsf{U} f_2$ iff either
  - for some k, $0 \leq k < |\mathbf{S}|$, $\mathbf{S}^k \models f_2$ and, for all j, $0 \leq j < k$ implies $\mathbf{S}^j \models f_1$ or
  - for all k, $0 \leq k < |\mathbf{S}|$ implies $\mathbf{S}^k \models f_1$

  $\mathbf{S} \models f_1 \mathcal{P} f_2$ iff
  - for all k, $0 \leq k < |\mathbf{S}|$ and $\mathbf{S}^k \models f_2$ implies, for some j, $0 \leq j < k$ and $\mathbf{S}^j \models f_1$ and
  - for some k, $0 \leq k < |\mathbf{S}|$ and $\mathbf{S}^k \models f_1$

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

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

We use $\mathrm{PTL}(o_1, o'_1, \ldots, o_k, o'_k)$ to denote a propositional temporal logic providing conjunction, disjunction and negation, along with the temporal operators o_i and their duals o'_i, $i = 1, \ldots, k$.

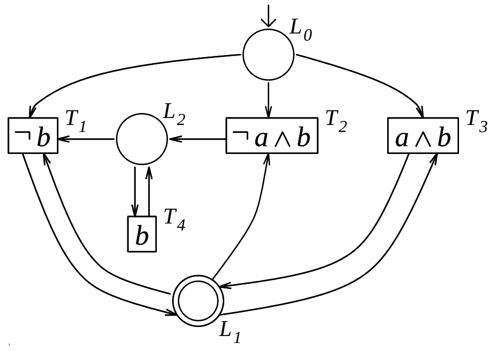
A formula of the form $\bigcirc f$ or $\odot f$ is said to be *deferred.* Atomic formulas and deferred formulas play a special role in the tableau construction. Intuitively, an atomic formula can be verified locally (in the "current" state of a state sequence) and a deferred formula must be verified in the future (at the "next" state of a state sequence).

The semantic tableau for a formula is a directed graph in which nodes are labeled with sets of formulas [18]. We find it useful to represent the tableau for a formula as a bipartite graph, called a flow graph, in a manner that accords equal status to transitions and locations (nodes of a finite state automaton).

Definition 3.6 (Flow Graph, Legal Moves and Runs) A *flow graph* G is a 7-tuple of the form $G = (\Sigma, \mathcal{L}, \mathcal{T}, \lambda, \Phi, I, F)$, where

- Σ is a finite input alphabet
- $\mathcal{L}$ is a finite set of locations
- $\mathcal{T}$ is a finite set of transitions
- λ is a transition labeling function $\lambda : \mathcal{T} \rightarrow 2^{\Sigma}$ associating boolean guards (predicates over Σ) with transitions
- Φ is a flow relation $\Phi \subseteq \mathcal{L} \times \mathcal{T} \cup \mathcal{T} \times \mathcal{L}$
- I is a set of initial locations $I \subseteq \mathcal{L}$
- F is a set of final locations $F \subseteq \mathcal{L}$.

Given a location $L \in \mathcal{L}$, the set $\mathrm{out}(L)$ of *outgoing* transitions is defined $\mathrm{out}(L) = \{\, T \mid (L, T) \in \Phi \,\}$ and the set $\mathrm{in}(L)$ of *incoming* transitions is defined $\mathrm{in}(L) = \{\, T \mid (T, L) \in \Phi \,\}$. An input symbol $x \in \Sigma$ *enables* a transition T if $x \in \lambda(T)$. A *move* is an element of $\mathcal{L} \times \Sigma \times \mathcal{T} \times \mathcal{L}$. We write $L_1 \stackrel{x,T}{\rightarrow} L_2$ to denote the move (L_1, x, T, L_2). The move $L_1 \stackrel{x,T}{\rightarrow} L_2$ is *legal* iff $T \in \mathrm{out}(L_1) \cap \mathrm{in}(L_2)$ and x enables T. Let $\mathbf{x}$ denote an input string of length n, $\mathbf{x} \in \Sigma^*$ and $|\mathbf{x}| = n$. 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 $T_i \in \mathcal{T}$ for which the move $\mathbf{L}(i) \stackrel{\mathbf{x}(i),T_i}{\rightarrow} \mathbf{L}(i+1)$ is legal, $0 \leq i < n$. The flow graph G *accepts* the input string $\mathbf{x}$ iff there is a run $\mathbf{L}$ of G on $\mathbf{x}$ such that $\mathbf{L}(0) \in I$ and $\mathbf{L}(n) \in F$, where $n = |\mathbf{x}|$.



$$
\begin{aligned}
\Sigma &= 2^{\{\,a,b\,\}}\\
\mathcal{L} &= \{\, L_0, L_1, L_2 \,\}\\
\mathcal{T} &= \{\, T_1, T_2, T_3, T_4 \,\}\\
\lambda &= [T_1 \mapsto \neg b, T_2 \mapsto \neg a \wedge b, T_3 \mapsto a \wedge b, T_4 \mapsto b]\\
\Phi &= \{\, (L_0,T_1),(L_0,T_2),(L_0,T_3),(T_1,L_1),(T_2,L_2),(T_3,L_1),\\
&\qquad (L_1,T_1),(L_1,T_2),(L_1,T_3),(L_2,T_4),(T_4,L_2) \,\}\\
I &= \{\, L_0 \,\}\\
F &= \{\, L_1 \,\}
\end{aligned}
$$

Figure 1: Flow graph for the formula $\Box(\Diamond\neg b \vee \Box a)$.

The input alphabet of the flow graph for a formula f is the set $2^{\mathcal{P}}$ of states and the flow graph accepts exactly those state sequences satisfying f. Figure 1 shows the conventions we use when drawing the flow graph for a formula. The flow graph in the figure accepts precisely those state sequences satisfying the formula $\Box(\Diamond\neg b \vee \Box a)$. Locations are represented as circles and transitions as boxes. Transition labels are shown as boolean guards over the primitive propositions appearing in the formula. Thus, for example, the label $\neg b$ on transition T_1 designates the set of states $\{\, \emptyset, \{\, a \,\} \,\}$. Arcs show the flow relation. A large arrow marks the initial location and final locations are marked by a double circle.

It is easy to check that the flow graph in Figure 1 is *deterministic* since it satisfies the following properties.

i) $|I| = 1$

ii) $T_i \in \mathrm{out}(L)$ and $T_j \in \mathrm{out}(L)$ implies that either $T_i = T_j$ or $\lambda(T_i) \cap \lambda(T_j) = \emptyset$, for $L \in \mathcal{L}$ and $T_i, T_j \in \mathcal{T}$, and

iii) $T \in \mathrm{in}(L_i)$ and $T \in \mathrm{in}(L_j)$ implies that $L_i = L_j$, for $L_i, L_j \in \mathcal{L}$ and $T \in \mathcal{T}$.

The tableau algorithm described below produces a flow graph that satisfies (i) and (iii) by construction; that is, the flow graph has a single initial location and each transition in the flow graph has a unique succeeding location. Thus, the flow graph is deterministic if it also satisfies property (ii). The flow graphs produced by the tableau algorithm are nondeterministic, but they can often be determinized without any increase in size.

4 Basic Tableau Method

We first review the standard tableau algorithm for PTL($\Box, \Diamond$), recasting it in terms of the specific algorithm used in the remainder of the paper. We then briefly discuss a correctness proof in order to identify the key properties that must be maintained to preserve correctness when modifying the algorithm for other logics and for increased efficiency.

Given a NNF formula $f_0 \in$ PTL($\Box, \Diamond$), the tableau algorithm for PTL($\Box, \Diamond$) constructs a flow graph $G_{f_0} = (\Sigma, \mathcal{L}, \mathcal{T}, \lambda, \Phi, I, F)$ with input alphabet $\Sigma = 2^{\mathcal{P}}$ that accepts precisely those input

Label	Rule		
$[r\wedge]$	$f_1 \wedge f_2$	$\longrightarrow$	$\langle \{ f_1, f_2 \} \rangle$
$[r\vee]$	$f_1 \vee f_2$	$\longrightarrow$	$\langle \{ f_1 \}, \{ f_2 \} \rangle$
$[r\Box]$	$\Box f$	$\longrightarrow$	$\langle \{ f, \bigcirc\Box f \} \rangle$
$[r\Diamond]$	$\Diamond f$	$\longrightarrow$	$\langle \{ f \}, \{ \odot\Diamond f \} \rangle$

Table 1: Reduction rules for PTL($\Box, \Diamond$)

strings satisfying f_0. Each location $L \in \mathcal{L}$ is identified with a set of *required conditions*, or formulas that the oracle must verify of the remaining trace that will be given to it.

We illustrate the tableau method for the example formula $\Box(\Diamond\neg b \vee \Box a)$ before presenting the formal algorithm. The algorithm begins with the creation of the initial location $L_0 = \{ \Box(\Diamond\neg b \vee \Box a) \}$. Intuitively, L_0 gives the conditions that the oracle must verify of an input trace. A copy T of L_0 is then created to be *elaborated* into outgoing transitions. Elaboration of T recursively breaks it down into one or more sets of conditions such that all conditions in the sets must either be verified locally (in the "current" state) or deferred to the future (verified at the succeeding location). These sets of "proper" required conditions then become the outgoing transitions of L_0.

The *reduction rules* of Table 1 carry out the task of breaking T down into proper transitions.[1] The reduction rule $f \longrightarrow \mathbf{F}$, with $\mathbf{F} = \langle F_1, \ldots, F_n \rangle$, is applicable in T if $f \in T$. If applicable, it specifies that T is to be replaced by the sets of conditions T_i, $1 \leq i \leq n$, for which $T_i = (T \setminus \{ f \}) \cup F_i$ is propositionally consistent. Any T_i containing a both an atomic formula and its negation is silently removed.[2] The operators $\bigcirc$ and $\odot$ are used as auxiliary operators in (the RHS) of reduction rules to simplify the presentation. Intuitively, whenever $\bigcirc f$ or $\odot f$ is present on a transition, f must be verified in the succeeding location. Moreover, if $\odot f$ is present, then the succeeding location must not be a final one.

In the example, the only rule that is applicable in T is $[r\Box]$, and its application replaces T with $\{ \Diamond\neg b \vee \Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$. Thus, the location L_0 must check that an input trace satisfies $\Diamond\neg b \vee \Box a$, and the succeeding locations must verify that the remaining trace satisfies $\Box(\Diamond\neg b \vee \Box a)$. The first of these conditions, which must be verified at the current location, is then reduced using $[r\vee]$. This splits the required conditions into two cases, represented by the sets $\{ \Diamond\neg b, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$ and $\{ \Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$, both of which contain simpler conditions to be verified at the current location. Now we apply the rule $[r\Diamond]$ to the first of these sets and the rule $[r\Box]$ 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 are installed as proper transitions of the flow graph.

The transition T_1 can be taken if the next input letter (state) does not contain b since $\neg b$ is the only local condition to be verified in T_1. Moreover the location succeeding T_1 must verify $\{ \Box(\Diamond\neg b \vee \Box a) \}$, which is precisely the condition to be verified by the initial location L_0. However, the location succeeding T_1 should be a final location, whereas the initial location must not. A new location, $L_1 = \{ \Box(\Diamond\neg b \vee \Box a), \# \}$, is therefore created 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

[1] More precisely, Table 1 gives schema for generating reduction rules. To obtain a reduction rule the variables in a rule schema must be instantiated with formulas from PTL($\Box, \Diamond$).

[2] As usual, we identify $\neg\neg f$ with f and $\neg \mathit{true}$ with false.

not contain any local conditions, and the location succeeding T_2 must verify $\{ \Diamond\neg b, \Box(\Diamond\neg b \vee \Box a) \}$. Moreover, the succeeding location must not be final, in order that $\odot\Diamond\neg b$ can be verified at L_0. Thus, $L_2 = \{ \Diamond\neg b, \Box(\Diamond\neg b \vee \Box a) \}$ is created to succeed T_2. Similarly, the transition T_3 is enabled by inputs containing a and the location succeeding T_3 is $L_3 = \{ \Box a, \Box(\Diamond\neg b \vee \Box a), \# \}$.

Having constructed the transitions and succeeding locations for L_0, the tableau algorithm proceeds to elaborate each of the succeeding locations in turn and, in a transitive manner, 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, L_1 is elaborated into the same three transitions as L_0. As the number of required conditions that can be generated in this fashion is finite, only a finite number of locations can be produced and the elaboration of locations terminates.

The tableau algorithm is formally defined in Figure 2. The inputs to the algorithm give the formula f_0 that is to be verified; the set $\mathcal{R}$ of rules to use for reducing required conditions; the function *Applicable* that determines the rules applicable in a set of conditions; and the function *Apply* that returns the result of applying a rule to a set of conditions. This permits the algorithm to be easily tailored by supplying it with different reduction rules and methods for applying them. The algorithm makes use of the following definitions.

Definition 4.1 The set T of formulas is a *proper transition* iff every formula in T is either atomic or deferred.

Definition 4.2 The set T of formulas is *consistent* iff *false* does not belong to T and there is no formula f for which T contains both f and $\neg f$.

Definition 4.3 The tableau procedure $\mathrm{TALG}_1(\Box, \Diamond)$ for $\mathrm{PTL}(\Box, \Diamond)$ is obtained by instantiating the algorithm in Figure 2 with the following rules and functions.

- The set $\mathcal{R}_1$ of reduction rules specified in Table 1
- The function $\mathit{Applicable}_1$ defined by
  $\mathit{Applicable}_1(T) \equiv \{ R \in \mathcal{R} \mid R \text{ has the form } f \longrightarrow \mathbf{F} \text{ and } f \in T \}$
- The function Apply_1 defined by $\mathit{Apply}_1(R, T) \equiv \langle T_1, \ldots, T_n \rangle$, where
  R has the form $f \longrightarrow \langle F_1, \ldots, F_n \rangle$ and $T_i = (T \setminus \{ f \}) \cup F_i$, $1 \leq i \leq n$.

Example 4.4 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 $\mathrm{TALG}_1(\Box, \Diamond)$ are shown below. The initial location in this and in all subsequent examples is L_0 and the final locations are those containing the # symbol. The input alphabet is $\Sigma = 2^{\mathcal{P}}$, where $\mathcal{P}$ denotes the primitive propositions that appear in the original formula. In this case, $\mathcal{P} = \{ a, b \}$. In the next section we show how $\mathrm{TALG}_1(\Box, \Diamond)$ is optimized to reduce the number of locations and transitions in the flow graph.

Locations: $\mathcal{L} = \{ L_0, L_1, L_2, L_3, L_4 \}$ where
$L_0 = \{ \Box(\Diamond\neg b \vee \Box a) \}$, $L_1 = \{ \Box(\Diamond\neg b \vee \Box a), \# \}$, $L_2 = \{ \Diamond\neg b, \Box(\Diamond\neg b \vee \Box a) \}$
$L_3 = \{ \Box a, \Box(\Diamond\neg b \vee \Box a), \# \}$, and $L_4 = \{ \Diamond\neg b, \Box a, \Box(\Diamond\neg b \vee \Box a) \}$

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

Input: f_0: formula (in NNF)
$\mathcal{R}$: 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: $(\Sigma, \mathcal{L}, \mathcal{T}, \lambda, \Phi, I, F)$: flow graph

```
Initialize: $L_0 := \{ f_0 \}$
            $L\_stack := \langle L_0 \rangle$        ;; stack of locations to be elaborated
            $\Sigma := 2^{\mathcal{P}}$, $\mathcal{L} := \{ L_0 \}$
            $\mathcal{T} := \Phi := \lambda := \emptyset$
            $I := \{ L_0 \}$
while $L\_stack \neq \Lambda$ do
    $L :=$ pop$(L\_stack) \setminus \{ \# \}$
    $E\_stack := \langle L \rangle$        ;; stack of sets being elaborated
    while $E\_stack \neq \Lambda$ do
        $T :=$ pop$(E\_stack)$
        if $T$ is not a proper transition then
            pick a rule $R$ in $Applicable(T)$
            for each $T_i \in Apply(R, T)$ do
                if $T_i$ is consistent then push$(T_i, E\_stack)$ end if
            end for
        else        ;;$T$ is a proper transition
            $\mathcal{T} := \mathcal{T} \cup \{ T \}$
            $L' := \{ f \mid \bigcirc f \in T$ or $\odot f \in T \}$
            if $\{ \odot f \mid \odot f \in T \} = \emptyset$ then $L' := L' \cup \{ \# \}$ end if
            $\Phi := \Phi \cup \{ (L, T), (T, L') \}$
            $Q := T \cap (\mathcal{P} \cup \neg\mathcal{P})$
            $\lambda := \lambda[T \mapsto W]$ where $W = \{ V \in 2^{\mathcal{P}} \mid \mathcal{P} \cap Q \subseteq V$ and $V \cup Q$ is consistent $\}$
            if $L' \notin \mathcal{L}$ then
                $\mathcal{L} := \mathcal{L} \cup \{ L' \}$
                push$(L', L\_stack)$
            end if
        end if
    end while
end while
$F := \{ L \in \mathcal{L} \mid \# \in L \}$
return $(\Sigma, \mathcal{L}, \mathcal{T}, \lambda, \Phi, I, F)$
```

Figure 2: Basic Tableau Algorithm

$T_5 = \{\, a, \odot\Diamond\neg b, \bigcirc\Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \,\}$, and $T_6 = \{\, a, \neg b, \odot\Diamond\neg b, \bigcirc\Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \,\}$

Transition labeling function:

$\lambda = [T_1 \mapsto \neg b;\ T_2 \mapsto \mathit{true};\ T_3 \mapsto a;\ T_4 \mapsto \neg b \wedge a;\ T_5 \mapsto a;\ T_6 \mapsto a \wedge \neg b]$

Flow relation:

$$\begin{aligned}\Phi = \{\, & (L_0,T_1),(T_1,L_1),(L_0,T_2),(T_2,L_2),(L_0,T_3),(T_3,L_3),\\ & (L_1,T_1),(L_1,T_2),(L_1,T_3),\\ & (L_2,T_1),(L_2,T_2),(L_2,T_4),(T_4,L_3),(L_2,T_5),(T_5,L_4),\\ & (L_3,T_3),(L_3,T_5),(L_3,T_6),(T_6,L_4),(L_4,T_3),(L_4,T_5),(L_4,T_6) \,\}\end{aligned}$$

Correctness of $\mathrm{TALG}_1(\Box,\Diamond)$ depends on applying rules in a sound manner and on choosing appropriate final locations, thereby ensuring that each location verifies the conditions associated with it. This is the thrust of the next lemma. The following definitions ensure that the application of reduction rules and the choice of final locations are sound.

Definition 4.5 (Valid Rules) A set $\mathcal{R}$ of reduction rules is *valid* iff $R \in \mathit{Applicable}(T)$ implies $T \equiv \mathit{Apply}(R,T)$, for every $R \in \mathcal{R}$ and set T of formulas.[3]

Observation 4.6 Let the functions $\mathit{Applicable}_1$ and Apply_1 be used with a set $\mathcal{R}$ of rules of the form $f \longrightarrow \mathbf{F}$, where f is a formula and $\mathbf{F}$ is a sequence of sets of formulas. Then $\mathcal{R}$ is valid if every rule $f \longrightarrow \mathbf{F}$ in $\mathcal{R}$ satisfies $f \equiv \mathbf{F}$ (that is, if $f \equiv \bigvee_{F\in\mathbf{F}}(\bigwedge_{f'\in F} f')$).

The appendix gives a proof of the lemma, which makes use of a shorthand for flow graphs.

Definition 4.7 Given a flow graph $G = (\Sigma,\mathcal{L},\mathcal{T},\lambda,\Phi,I,F)$ and a location $L \in \mathcal{L}$, we write $G[L]$ for the flow graph obtained from G by replacing I with $\{\, L \,\}$, $G[L] = (\Sigma,\mathcal{L},\mathcal{T},\lambda,\Phi,\{\, L \,\},F)$.

Lemma 4.8 Let $G = (\Sigma,\mathcal{L},\mathcal{T},\lambda,\Phi,I,F)$ be the flow graph returned by the algorithm in Figure 2. If $\mathcal{R}$ is valid then, for each $L \in \mathcal{L}$, the flow graph $G[L]$ accepts precisely those input sequences satisfying the required condition L (that is, the condition $\bigwedge_{f\in L} f$).

For termination, it suffices to show that the number of required conditions that can be generated from a formula is finite and that the rules for reducing formulas elaborates a set of required conditions into proper transitions. The following definitions assume that a set $\mathcal{R}$ of reduction rules, a function *Applicable* for determining the rules that are applicable to a set of conditions, and a procedure *Apply* for applying rules to sets of conditions are given.

Definition 4.9 The collection of required conditions that can be generated from a formula f is the smallest set W_f of sets of formulas satisfying

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

[3] More precisely, the validity of a set of rules is defined with respect to specific application functions. For simplicity in terminology, however, we refer to a given set $\mathcal{R}$ of rules as valid, with the application functions implicitly understood to be those that have previously been defined for $\mathcal{R}$. A similar abuse of terminology is used in (4.9) and (4.10).

Definition 4.10 (Complete Rules) A set $\mathcal{R}$ of rules is *complete* iff the following properties are met, where T and T' are sets of formulas, $R \in \mathcal{R}$, and f is a formula.

i) $\mathit{Applicable}(T) = \emptyset$ iff T is a proper transition.

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

iii) W_f is finite.

Observation 4.11

- Properties (i) and (ii) of this definition guarantee property (iii) provided that $\mathit{Applicable}(T)$ is finite for every finite set T of formulas.
- Intuitively, the function m measures how "far" a set of formulas is from being proper. A suitable measure, which is used for the examples below, is defined recursively as follows:

$$m(T) = \begin{cases} 0 & \text{if } T = \emptyset \\ m(T \setminus \{ f \}) & \text{if } f \in T \text{ and } f \text{ is atomic or deferred} \\ m(T \setminus \{ f \}) + \mathrm{size}(f) & \text{if } f \in T \text{ and } f \text{ is neither atomic nor deferred} \end{cases}$$

  where size(f), the size of a formula f, is the sum of the numbers of operators and operands in f (the number of nodes in the abstract syntax tree for f).
- Let the functions $\mathit{Applicable}_1$ and Apply_1 defined in (4.3) be used with a set $\mathcal{R}$ of rules of the form $f \longrightarrow \mathbf{F}$ and let m be the measure defined above. Then property (ii) of (4.10) holds if $m(F') < m(\{ f \})$, for each $f \longrightarrow \mathbf{F}$ in $\mathcal{R}$ and $F' \in \mathbf{F}$.

This definition allows the statement of sufficient conditions for termination of the tableau algorithm. The proof of the following lemma uses standard methods for demonstrating that an algorithm terminates.

Lemma 4.12 The algorithm of Figure 2 terminates if $\mathcal{R}$ is complete.

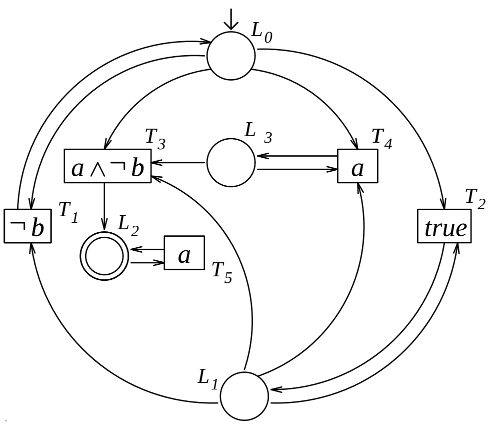
The following correctness theorem is an immediate consequence of (4.8) and (4.12), given construction of the initial location (line 5 of Figure 2).

Theorem 4.13 (Correctness) The flow graph generated by the tableau algorithm in Figure 2 accepts precisely those state sequences satisfying f_0 provided that $\mathcal{R}$ is valid and complete.

Corollary 4.14 The algorithm $\mathrm{TALG}_1(\Box, \Diamond)$ generates a flow graph that accepts precisely those state sequences satisfying f_0.

Label			Rule
$[r\mathsf{U}]$	$f_1 \mathsf{U} f_2$	$\longrightarrow$	$\langle \{ f_1, \bigcirc(f_1 \mathsf{U} f_2) \}, \{ f_2 \} \rangle$
$[r\mathcal{P}]$	$f_1 \mathcal{P} f_2$	$\longrightarrow$	$\langle \{ f_1, \neg f_2 \}, \{ \neg f_2, \odot(f_1 \mathcal{P} f_2) \} \rangle$
$[r\mathcal{U}]$	$f_1 \mathcal{U} f_2$	$\longrightarrow$	$\langle \{ f_1, \odot(f_1 \mathcal{U} f_2) \}, \{ f_2 \} \rangle$
$[r\mathsf{P}]$	$f_1 \mathsf{P} f_2$	$\longrightarrow$	$\langle \{ f_1, \neg f_2 \}, \{ \neg f_2, \bigcirc(f_1 \mathsf{P} f_2) \} \rangle$

Table 2: Reduction rules for the until and precedes operators



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

Figure 3: Flow graph generated by $\mathrm{TALG}_1(\Box, \Diamond, \mathsf{U}, \mathcal{P})$ for the formula $(\Box a) \mathcal{P} (\Box b)$.

5 Customizing the Tableau Method

To use the tableau method with other temporal logics, it is necessary to provide reduction rules that express the semantics of the temporal operators. Table 2 gives rules expressing the semantics of the until and precedes operators defined in (3.5). The negated formulas in the rules for the precedes operators are put in NNF when they require further reduction. The semantics of the next operators are hard coded into the tableau algorithm, so that no rules are needed for the next operators.

Definition 5.1 The tableau procedure $\mathrm{TALG}_1(o_1, o'_1, \ldots, o_k, o'_k)$ for $\mathrm{PTL}(o_1, o'_1, \ldots, o_k, o'_k)$ is that obtained by instantiating the algorithm in Figure 2 with the appropriate reduction rules from Tables 1 and 2 and with the functions $\mathit{Applicable}_1$ and Apply_1 defined in (4.3).

These procedures are easily shown to be correct using the methods in (4.6) and (4.11).

Example 5.2 The procedure $\mathrm{TALG}_1(\Box, \Diamond, \mathsf{U}, \mathcal{P})$ requires the rules $[r\vee]$, $[r\wedge]$, $[r\Box]$, $[r\Diamond]$, $[r\mathsf{U}]$, and $[r\mathcal{P}]$. Figure 3 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 2 with valid and complete rules is correct. However, the specific rules provided for reducing formulas affects the efficiency of the resulting tableau procedure, the sizes of the oracles that the tableau procedure generates, and the time that oracles generated by the tableau procedure take to check traces. For example, the rules $[r\vee]$ and $[r\Diamond]$ can be replaced with the "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 $\Box(\Diamond\neg b \vee \Box a)$ of Example 4.4, the extra case split and the negated formulas, which introduce new required conditions, more than double the size of the resulting flow graph.

The sizes of the flow graphs produced by a tableau procedure can often be reduced if the required conditions generated during the elaboration of locations are "simplified". Consider, for instance, elaboration of the location $L_2 = \{ \Diamond\neg b, \Box(\Diamond\neg b \vee \Box a) \}$ in Example 4.4. Application of the rule $[r\Box]$ transforms L_2 into the set $\{ \Diamond\neg b, \Diamond\neg b \vee \Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$. When viewed as a set of required conditions, this set is equivalent to $\{ \Diamond\neg b, \bigcirc\Box(\Diamond\neg b \vee \Box a) \}$. Elaborating this latter set, instead of the former, reduces the number of transitions leaving L_2. The required conditions generated during elaboration of L_3 can be similarly simplified. In addition to reducing the numbers of transitions leaving L_2 and L_3, the simplifications prevent L_4 from being generated.

The simplification of arbitrary logical conditions requires a theorem prover and can, in general, be very hard. The rules for reducing formulas, however, give rise to sets of conditions that admit fairly routine simplification. We incorporate the simplification of required conditions into a tableau procedure by augmenting the rules for reducing formulas with special "replacement rules" and appropriately modifying the application functions.

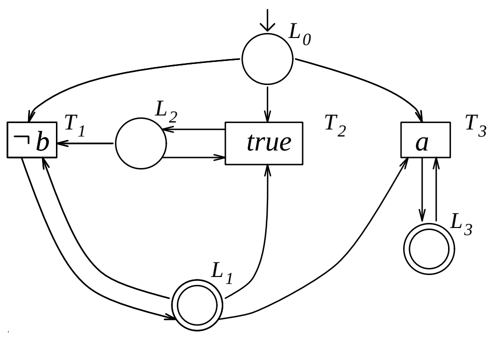
Definition 5.3 Let each rule in a set $\mathcal{R}$ of rules be classified as either a reduction rule, of the form $f \longrightarrow \mathbf{F}$, or a replacement rule, of the form $f \xrightarrow{F'}\!\!\!\!\rightarrow F$, where f is a formula, F and F' are sets of formulas, and $\mathbf{F}$ is a sequence of sets of formulas. The functions $\mathit{Applicable}_2$ and Apply_2 are then defined, for $R \in \mathcal{R}$ and a set T of formulas, as follows.

- $\mathit{Applicable}_2(T) = \{ f \longrightarrow \mathbf{F} \in \mathcal{R} \mid f \in T \} \cup \left\{ f \xrightarrow{F'}\!\!\!\!\rightarrow F \in \mathcal{R} \;\middle|\; f \in T \text{ and } F' \subseteq T \right\}$
- $\mathit{Apply}_2(R, T) = \begin{cases} \mathit{Apply}_1(R, T) & \text{if } R \text{ has the form } f \longrightarrow \mathbf{F} \\ \langle (T \setminus \{ f \}) \cup F \rangle & \text{if } R \text{ has the form } f \xrightarrow{F'}\!\!\!\!\rightarrow F \end{cases}$

Intuitively, the rule $f \xrightarrow{F'}\!\!\!\!\rightarrow F$ specifies that the required condition f is to be replaced by F in situations where F' is also required.

Observation 5.4 Assume the application of rules is carried out using the functions $\mathit{Applicable}_2$ and Apply_2. Then $\mathcal{R}$ is valid if every reduction rule $f \longrightarrow \mathbf{F}$ in $\mathcal{R}$ satisfies $f \equiv \mathbf{F}$ and every replacement rule $f \xrightarrow{F'}\!\!\!\!\rightarrow F$ in $\mathcal{R}$ satisfies $F' \Rightarrow (f \equiv F)$. Moreover, property (ii) of Definition 4.10 holds if $m(F') < m(\{ f \})$, for every $f \longrightarrow \mathbf{F}$ in $\mathcal{R}$ and $F' \in \mathbf{F}$, and $m(F) < m(\{ f \})$, for every $f \xrightarrow{F'}\!\!\!\!\rightarrow F$ in $\mathcal{R}$, where m is the measure defined in (4.11).

Definition 5.5 The tableau procedure $\mathrm{TALG}_2(o_1, o_1', \ldots, o_k, o_k')$ for $\mathrm{PTL}(o_1, o_1', \ldots, o_k, o_k')$ is obtained by instantiating the algorithm in Figure 2 with appropriate reduction rules from Tables 1 and 2, appropriate replacement rules from those specified below, and the functions $\mathit{Applicable}_2$ and Apply_2 defined in (5.3).



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

Figure 4: Flow graph generated by $\mathrm{TALG}_2(\Box, \Diamond)$ for the formula $\Box(\Diamond\neg b \vee \Box a)$.

Example 5.6 The simplifications described above are carried out by the replacement rules

$$[R\vee] \qquad f_1 \vee f_2 \xrightarrow{\{f_1\}} \emptyset \qquad \text{and} \qquad [\vee R] \qquad f_1 \vee f_2 \xrightarrow{\{f_2\}} \emptyset$$

Figure 4 shows the flow graph for $\Box(\Diamond\neg b \vee \Box a)$ that is produced by the procedure $\mathrm{TALG}_2(\Box, \Diamond)$ using these rules and some simple heuristics to guide the selection of rules. In particular, replacement rules take precedence over reduction rules, so that required conditions are simplified before being reduced. If two formulas can be reduced and one is a subformula of the other, the larger formula is selected for reduction. Other things being equal, a reduction that creates a case split has lower precedence than one that does not. For example, when elaborating the location L_2 in Figure 4, the rule $[r\Box]$ is selected to be applied to $\Box(\Diamond\neg b \vee \Box a)$, which contains $\Diamond\neg b$ as a subformula. This produces the set $\{\, \Diamond\neg b, \Diamond\neg b \vee \Box a, \bigcirc\Box(\Diamond\neg b \vee \Box a) \,\}$, which is then simplified using $[R\vee]$. Finally, $[r\Diamond]$ is applied, producing two outgoing transitions.

An oracle can simulate a nondeterministic flow graph on line if a determinized flow graph is substantially larger than a nondeterministic one. However, our experience suggests that the flow graph produced by a tableau procedure can often be determinized without increasing its size.

The standard "subset construction" is used to determinize the flow graph produced by a tableau procedure, with the disjunction of the locations that the nondeterministic flow graph can reach on a given input giving a *composite condition* that must be verified by a location of the determinized flow graph. Where possible, logical equivalences are used to simplify composite conditions and reduce the number of locations in the deterministic flow graph.

Example 5.7 Consider the nondeterministic flow graph (NFG) for $\Box(\Diamond\neg b \vee \Box a)$ shown in Figure 4. The composite condition for the initial location of the determinized flow graph (DFG) is $\langle L_0 \rangle$, where, as usual, we use sequences to express disjunction. The guards 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 $\langle L_1, L_2, L_3 \rangle$. But this is equivalent to verifying $\langle L_1 \rangle$. The subset construction proceeds using the simplified composite conditions, rather than the original composite conditions generated for each input. Table 3 summarizes the subset construction for this example. The determinized flow graph appears in Figure 1.

DFG Transition	Source	Guard	Successor	Simpl. Successor
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	$\langle L_2 \rangle$	$\langle L_2 \rangle$

Table 3: Subset construction for the flow graph in Figure 4.

Label	Rules		
$[R\Diamond]$	$\Diamond f \xrightarrow{\{f\}} \emptyset$	$\odot\Diamond f \xrightarrow{\{\bigcirc f\}} \{\odot true\}$	$\odot\Diamond f \xrightarrow{\{\odot f\}} \emptyset$
$[R\Box]$	$f \xrightarrow{\{\Box f\}} \emptyset$	$\bigcirc f \xrightarrow{\{\bigcirc\Box f\}} \emptyset$	$\odot f \xrightarrow{\{\bigcirc\Box f\}} \{\odot true\}$

Table 4: Replacement rules for formulas containing henceforth and eventually.

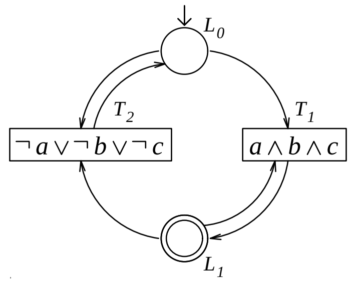
The simplifications that can be performed, both during the elaboration of locations and when determinizing an automaton, depend on the formula given as input to a tableau procedure. Thus, the form of an input formula affects the replacement rules that are to be used during the elaboration of locations and the equivalences that are to be used when determinizing the resulting flow graph.

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

Example 5.9 The replacement rules in Table 4 are useful when generating flow graphs for formulas containing subformulas of the form $\Box\Diamond f$ or $\Diamond\Box f$. The rules in the third and fourth columns are deferred versions of rules in the second column. Similar deferred versions of rules can be derived, as needed, from any replacement rule.

The flow graph generated for $\Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c$ by $\mathrm{TALG}_2(\Box, \Diamond)$ using these replacement rules contains 2 locations and 8 transitions. When determinized, the flow graph in Figure 5 is obtained. In general, the flow graph generated by $\mathrm{TALG}_1(\Box, \Diamond)$ 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 replacement rules in Table 4, $\mathrm{TALG}_2(\Box, \Diamond)$ generates a flow graph with 2 locations and 2^n transitions, and determinizing this flow graph reduces the number of transitions to 2.

Example 5.10 The equivalence $(f_1 \wedge \Box(f_1 \vee f_2)) \vee (f_2 \wedge \Box(f_1 \vee f_2)) \equiv \Box(f_1 \vee f_2)$ was used to simplify the composite condition in the third and sixth rows of Table 3. In general, this equivalence is useful when determinizing the flow graph for a formula containing a subexpression of the form $\Box(f_1 \vee f_2)$.



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

Figure 5: Flow graph for the formula $\Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c$.

6 Application to an Interval Logic

In this section we discuss higher level "interval" operators, which permit the scopes of properties to be restricted to subtraces of an execution trace. The following BNF summarizes the syntax of the new operators.

$$\begin{array}{rcl} \theta & ::= & \triangleright p \;\Big|\; \triangleright p\theta_1 \;\Big|\; \triangleright\!\!\triangleright p \;\Big|\; \triangleright\!\!\triangleright p\theta_1 \\ I & ::= & [-|\theta) \;\Big|\; [\theta_1|\theta_2) \;\Big|\; [\theta_1|\triangleright) \;\Big|\; [\![-\|\theta)\!) \;\Big|\; [\![\theta_1\|\theta_2)\!) \\ f & ::= & If_1 \end{array}$$

where p is a propositional formula and f and f_1 are formulas.

Intuitively, an interval I specifies a subsequence $\mathbf{S}'$ of states in a given sequence $\mathbf{S}$, and the formula If_1 asserts that $\mathbf{S}'$ satisfies f_1. The endpoints of $\mathbf{S}'$ are specified by *search patterns*, with the *trivial patterns* $-$ and $\triangleright$ designating, respectively, the right and left endpoints of $\mathbf{S}$, and a *nontrivial pattern* θ designating a sequence of searches to states at which propositional *target formulas* are satisfied. Searches start at a designated state and move forward to later states in the temporal order. A *weak search* $\triangleright p$ does not require that any "future" state satisfies p, while the *strong search* $\triangleright\!\!\triangleright p$ does. A *weak interval* $[\theta_1|\theta_2)$ may designate the null sequence and a *strong interval* $[\![\theta_1\|\theta_2)\!)$ cannot.

More precisely, the semantics of intervals make use of a *search locator function.* Starting from a designated state within a non null state sequence, the search locator function carries out a sequence of searches and returns the position of the final state located, if the searches succeed; $\bot$, if a weak search fails; or $\top$, if a strong search fails. The following equations, in which we assume $|\mathbf{S}| > 0$ and $i \in \{\, 0, \ldots, |\mathbf{S}| \,\}$, define the search locator function.

$$\begin{array}{lcl} \mathrm{loc}(\mathbf{S}, i, -) & = & i \\ \mathrm{loc}(\mathbf{S}, i, \triangleright) & = & |\mathbf{S}| \\ \mathrm{loc}(\mathbf{S}, i, *p\theta) & = & \begin{cases} \bot & \text{if } i = |\mathbf{S}| \text{ and } * \text{ is } \triangleright \\ \top & \text{if } i = |\mathbf{S}| \text{ and } * \text{ is } \triangleright\!\!\triangleright \\ i & \text{if } \mathbf{S}(i) \models p \text{ and } \theta \text{ is null} \\ \mathrm{loc}(\mathbf{S}, i, \theta) & \text{if } \mathbf{S}(i) \models p \text{ and } \theta \text{ is non null} \\ \mathrm{loc}(\mathbf{S}, i+1, *p\theta) & \text{otherwise} \end{cases} \end{array}$$

$[\theta 1]$	If	$\xrightarrow{p}$	$\langle \{ I'f \} \rangle$	where either I is $[\![*p\theta_1 \| \theta_2)\!)\!)$ and I' is $[\![\theta_1 \| \theta_2)\!)\!)$, I is $[\![*p \| \theta_2)\!)\!)$ and I' is $[\![- \| \theta_2)\!)\!)$ or I is $[\![- \| *p\theta_2)\!)\!)$ and I' is $[\![- \| \theta_2)\!)\!)$
$[\theta 2]$	$[\![*p \| \triangleright)\!)\!) f$	$\xrightarrow{p}$	$\langle \{ f \} \rangle$	
$[\theta 3]$	$[- \| *p) f$	$\xrightarrow{p}$	$\langle \emptyset \rangle$	
$[\theta 4]$	$[\![- \| *p)\!) f$	$\xrightarrow{p}$	$\langle \{ \mathit{false} \} \rangle$	
$[\theta 5]$	If	$\xrightarrow{\neg p}$	$\langle \{ \bigotimes If \} \rangle$	where I is $[\![*p\theta_1 \| \theta_2)\!)\!)$
$[Ap]$	If	$\xrightarrow{\mathit{true}}$	$\langle \{ p, If \}, \{ \neg p, If \} \rangle$	where I is $[\![*p\theta_1 \| \theta_2)\!)\!)$ or $[\![- \| *p\theta_2)\!)\!)$
$[Iq]$	Iq	$\xrightarrow{\neg p}$	$\langle \{ q, \bigotimes I^w \mathit{true} \}, \{ \bigotimes I^s \mathit{false} \} \rangle$	where I is $[\![- \| *p\theta_2)\!)\!)$
$[I\wedge]$	$I(f_1 \wedge f_2)$	$\xrightarrow{\neg p}$	$\langle \{ If_1, If_2 \} \rangle$	where I is $[\![- \| *p\theta_2)\!)\!)$
$[I\vee]$	$I(f_1 \vee f_2)$	$\xrightarrow{\neg p}$	$\langle \{ If_1 \}, \{ If_2 \} \rangle$	where I is $[\![- \| *p\theta_2)\!)\!)$
$[I\Box]$	$I\Box f$	$\xrightarrow{\neg p}$	$\langle \{ If, \bigotimes I^w \Box f \} \rangle$	where I is $[\![- \| *p\theta_2)\!)\!)$
$[I\Diamond]$	$I\Diamond f$	$\xrightarrow{\neg p}$	$\langle \{ If \}, \{ \bigotimes I^s \Diamond f \} \rangle$	where I is $[\![- \| *p\theta_2)\!)\!)$

Table 5: Sample reduction rules for PIL($\Box, \Diamond$)

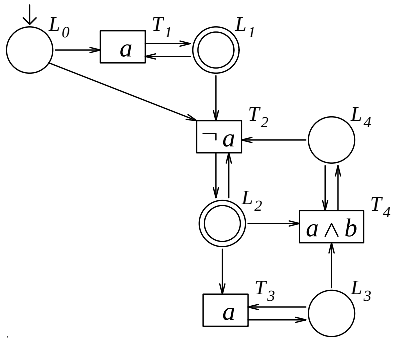
The semantics of the search and interval modalities are defined as follows.

- $\mathbf{S} \models [\theta_1 | \theta_2) f$ iff, for $l_0 = \mathrm{loc}(\mathbf{S}, 0, \theta_1)$ and $l_1 = \mathrm{loc}(\mathbf{S}, l_0, \theta_2)$, $l_1 \neq \top$ and, if $l_1 \neq \bot$ and $l_0 < l_1$, then $\mathbf{S}[l_0 \ldots l_1) \models f$
- $\mathbf{S} \models [\![\theta_1 \| \theta_2)\!) f$ iff, for $l_0 = \mathrm{loc}(\mathbf{S}, 0, \theta_1)$ and $l_1 = \mathrm{loc}(\mathbf{S}, l_0, \theta_2)$, $l_1 \neq \top$ and, if $l_1 \neq \bot$, then $l_0 < l_1$ and $\mathbf{S}[l_0 \ldots l_1) \models f$

We write PIL($o_1, o_1', \ldots, o_k, o_k'$) for the propositional temporal logic obtained by extending PTL($o_1, o_1', \ldots, o_k, o_k'$) with interval modalities. Space limitations do not permit a complete treatment of the tableau method for PIL($o_1, o_1', \ldots, o_k, o_k'$). Instead, we present semantic rules that permit construction of an efficient tableau for the formula $\Box[\triangleright\neg a \triangleright a | \triangleright\!\triangleright \neg a)\Diamond\Box b$ and indicate how these rules are generalized to obtain rules for nested intervals and for additional temporal operators.

Table 5 shows some reduction rules for PIL($\Box, \Diamond$). These tables and the subsequent discussion make use of the following abbreviations. We write $[\![\theta_1 \| \theta_2)\!)\!)$ to stand for either $[\theta_1 | \theta_2)$ or $[\![\theta_1 \| \theta_2)\!)$, and $*p$ to stand for either $\triangleright p$ or $\triangleright\!\triangleright p$. If I stands for $[\![\theta_1 \| \theta_2)\!)\!)$, then I^w denotes $[\theta_1 | \theta_2)$ and I^s denotes $[\![\theta_1 \| \theta_2)\!)$. Further, we write $\bigotimes If$ for $\bigcirc If$, if the leading search in I is weak, and for $\bigodot If$, if the leading search in I is strong. Thus, for example, $\bigotimes[\triangleright a | \triangleright\!\triangleright \neg a)\Diamond\Box b$ denotes $\bigcirc[\triangleright a | \triangleright\!\triangleright \neg a)\Diamond\Box b$ and $\bigotimes[- | \triangleright\!\triangleright \neg a)\Diamond\Box b$ denotes $\bigodot[- | \triangleright\!\triangleright \neg a)\Diamond\Box b$.

The reduction rules for PIL($\Box, \Diamond$) are parameterized by propositional formulas, which are used in determining when the rules are applicable. A rule of the form $f \xrightarrow{q} \mathbf{F}$ is applicable in a set T only if $f \in T$ and q is *deducible* in T. Informally, deductibility is determined by writing q in disjunctive NNF as a sequence $\mathbf{Q}$ of sets of atomic propositions. Then q is deducible in T if $Q \subseteq T$, for some $Q \in \mathbf{Q}$. For the rule $[Ap]$ to be applicable, we also require that neither p nor $\neg p$ are deducible in T. We need this additional stipulation in the case of $[Ap]$ to guarantee termination. As above, applying a rule $f \xrightarrow{q} \mathbf{F}$ to a set T replaces T with the sets T_i for which $T_i = (T \setminus \{ f \}) \cup F_i$ is consistent, $F_i \in \mathbf{F}$. Validity of the rules in Table 5 follows from the observation that $q \Rightarrow (f \equiv \mathbf{F})$, for a rule $f \xrightarrow{q} \mathbf{F}$ obtained from Table 5.



$$
\begin{aligned}
L_0 &= \{\, \Box[\triangleright\neg a\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b \,\}\\
L_1 &= \{\, \Box[\triangleright\neg a\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b, \# \,\}\\
L_2 &= \{\, [\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b, \Box[\triangleright\neg a\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b, \# \,\}\\
L_3 &= \{\, [\![-\|\triangleright\!\triangleright\neg a))\Diamond\Box b, \Box[\triangleright\neg a\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b \,\}\\
L_4 &= \{\, [-|\triangleright\!\triangleright\neg a)\Box b, \Box[\triangleright\neg a\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b \,\}
\end{aligned}
$$

Figure 6: Flow graph for $\Box[\triangleright\neg a\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b$

The rules $[\theta 1]$–$[\theta 5]$ express the semantics of searching. Rule $[Ap]$ is used when no other rules are applicable to introduce assumptions needed to evaluate search targets. The remaining rules are used upon locating the start of a non null interval.

The rules $[I\Box]$ and $[I\Diamond]$, which handle temporal operators that are embedded in an interval, are "derived" from the rules $[r\Box]$ and $[r\Diamond]$ using the following tautologies, in which I stands for $[\![-\| *p\theta_2)\!))$.

$$\neg p \Rightarrow (I\bigcirc f \equiv \bigotimes I^w f) \text{ and } \neg p \Rightarrow (I\bigodot f \equiv \bigotimes I^s f)$$

For example, $[r\Box]$ justifies $I\Box f \longrightarrow \langle\{\, If, I\bigcirc\Box f \,\}\rangle$, from which we obtain $[I\Box]$ by the first tautology. Likewise, $[I\Diamond]$ is obtained from the second tautology and $I\Diamond f \longrightarrow \langle\{\, If \,\}, \{\, I\bigodot\Diamond f \,\}\rangle$, which follows from $[r\Diamond]$. Rules for embedded precedes and until operators and for embedded intervals can be obtained, in an analogous fashion, as they are needed to elaborate a formula.

As above, the introduction of replacement rules can reduce the size of a flow graph and the time required to construct it. Figure 6 shows the flow graph generated for $\Box[\triangleright\neg a\triangleright a|\triangleright\!\triangleright\neg a)\Diamond\Box b$ using the reduction rules in Figure 5 and the following replacement rules.

$$
\begin{array}{rcll}
\bigcirc f & \xrightarrow{\{\, \bigcirc\Box f \,\}}\!\!\!\!\to & \emptyset & \\
\bigotimes I\, true & \xrightarrow{\{\, \bigotimes I f \,\}}\!\!\!\!\to & \emptyset & \\
\bigotimes[\![-\|\theta)\!) & \xrightarrow{\emptyset}\!\!\!\!\to & \mathit{false} & \text{if all searches in } \theta \text{ are strong}
\end{array}
$$

7 On-going Research

The paper has described a very general algorithm that can be used to automate the generation of efficient temporal oracles. The 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, as described in [4].

The tableau method described above is the basis for an oracle synthesizer that we are developing for use with specifications expressed in Graphical Interval Logic (GIL) [3]. The GIL oracle synthesizer will permit us to experiment with heuristics for determining an elaboration order for sets of formulas and with different reduction and replacement rules. It will permit us to compare the efficiency of the oracles generated by this method with GIL-based oracles generated by other methods. The straightforward method in [14] scans a trace to locate intervals and then recursively evaluates formulas within the intervals. This method does not support run time checking of execution traces and does not scale for use with long traces. A method for directly translating GIL into a hierarchical finite state machine (HFSM) formalism resembling Statecharts [9] is described in [12]. The HFSMs are designed for expressing the semantics of GIL and the translation is specific to GIL. It is not known if this method generalizes for use with other temporal notations.

Up to this point we have considered only linear time temporal logics, but the ideas presented above should generalize to branching time logics. The treatment in this paper is quite natural when a logic provides only "future" temporal operators. However, it is not immediately evident how to express the semantics of "past" operators [11] in rules that are valid by definition (4.5). We are therefore currently investigating a modified tableau method for linear time logics with "past" operators. This method constructs a flow graph in a manner symmetric to that described above. The construction begins with a single final location and proceeds by elaborating locations to generate incoming transitions. In this method, previous-state operators play the role of next operators.

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

A Correctness Proof

For the proof of (4.8), we assume that $G = (\Sigma, \mathcal{L}, \mathcal{T}, \lambda, \Phi, I, F)$ denotes the flow graph produced by the tableau algorithm in Figure 2 and that the rules $\mathcal{R}$ used in the construction of G are valid. The following abbreviations are introduced to simplify the presentation.

Notation A.1 Let $T \in \mathcal{T}$.

- $\mathrm{atom}(T) = T \cap (\mathcal{P} \cup \neg\mathcal{P})$
- $\mathrm{wk\_prop}(T) = \{\, f \mid \bigcirc f \in T \,\}$
- $\mathrm{st\_prop}(T) = \{\, f \mid \odot f \in T \,\}$
- $\mathrm{prop}(T) = \mathrm{wk\_prop}(T) \cup \mathrm{st\_prop}(T)$

The proof of (4.8) relies on 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. The final location marker # is identified with the formula *true* for the purpose of viewing a final location $L \in F$ as the condition $\bigwedge_{f \in L} f$.

Observation A.2 If $L \in \mathcal{L}$ then $L \equiv \bigvee_{T \in \mathrm{out}(L)} T$.

Proof. We establish the following loop invariant for the `while` loop at line 9 in Figure 2.

$[LI]$ $\quad L \equiv E\_stack \vee \langle T_1, \ldots T_k \rangle$, where $\langle T_1, \ldots T_k \rangle$ enumerates the transitions in $\mathrm{out}(L)$.

The invariant $[LI]$ holds initially, since $E\_stack = \langle L \rangle$ and $\mathrm{out}(L) = \{\, T_i \mid (L, T_i) \in \Phi \,\} = \emptyset$ following the assignment to E_stack (line 8). The validity of the replacement rules implies that the body of the `while` loop preserves $[LI]$ when the first branch (lines 12–14) of the `if` statement at line 11 is executed, and the body of the `while` loop is easily seen to preserve $[LI]$ when the second branch (lines 16–24) is executed. Thus, $[LI]$ is a loop invariant.

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

Observation A.3 If $T \in \mathcal{T}$ and $L' \in \mathcal{L}$ is the location succeeding T, then

i) $L' \in F$ iff $T \equiv \mathrm{atom}(T) \wedge \bigcirc L'$

ii) $L' \notin F$ iff $T \equiv \mathrm{atom}(T) \wedge \odot L'$ and

iii) $\mathbf{A} \models \mathrm{atom}(T)$ iff $\mathbf{A}(0) \in \lambda(T)$,

Proof. The algorithm enters T into $\mathcal{T}$ only if T is a proper transition. Thus, if $\mathrm{st\_prop}(T) = \emptyset$, then $T \equiv \mathrm{atom}(T) \wedge \bigcirc \mathrm{prop}(T)$ and, if $\mathrm{st\_prop}(T) \neq \emptyset$, then $T \equiv \mathrm{atom}(T) \wedge \odot \mathrm{prop}(T)$. Since $L' = \mathrm{prop}(T)$ by construction of L' (lines 17–18) and $L' \in F$ iff $\# \in L'$, which holds precisely if $\mathrm{st\_prop}(T) = \emptyset$, we obtain (i) and (ii).

Observation (iii) holds by construction of λ (line 20). ∎

The following restatement of (4.8) admits a proof by induction.

Lemma A.4 Let $n \geq 1$. If the length of a (non null) state sequence $\mathbf{A}$ is at most n, then $G[L]$ accepts $\mathbf{A}$ iff $\mathbf{A} \models L$, for $L \in \mathcal{L}$.

Proof. Given $L \in \mathcal{L}$, we show that there is some transition $T \in \mathrm{out}(L)$ such that $\mathbf{A} \models T$ precisely if $G[L]$ accepts $\mathbf{A}$ by induction on n. The lemma then follows from (A.2).

Assume $|\mathbf{A}| = 1$. Then $G[L]$ accepts $\mathbf{A}$ iff there is some $T \in \mathrm{out}(L)$ such that $\mathbf{A}(0) \in \lambda(T)$ and $L' \in F$, where L' denotes the location succeeding T. However, (A.3) implies that the latter is equivalent to $\mathbf{A} \models \mathrm{atom}(T)$ and $T \equiv \mathrm{atom}(T) \wedge \bigcirc L'$, and that this is equivalent to $\mathbf{A} \models T$ under the assumption $|\mathbf{A}| = 1$.

For the induction step, assume that $1 < n$ and $|\mathbf{A}| \leq n$. Then $G[L]$ accepts $\mathbf{A}$ precisely if there is some transition $T \in \mathrm{out}(L)$ such that $\mathbf{A}(0) \in \lambda(T)$ and $G[L']$ accepts $\mathbf{A}^1$, for the location L' succeeding T. Thus, the induction step follows from (A.3) and the induction hypothesis. ∎

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] 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.

[4] L. K. Dillon and Q. 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.

[5] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proc. 14th Symp. Theory of Computing*, pp. 169–180, San Francisco, May 1982.

[6] E. A. Emerson and A. P. Sistla. Deciding branching time logic. In *Proc. 16th Symp. Theory of Computing*, pp. 14–24, Washington, May 1984.

[7] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, February 1990.

[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] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] P. K. Harter, D. M. Heimbigner, and R. King. IDD: An interactive distributed debugger. In *IEEE Inter. Conf. Distributed Computer Systems*, pp. 498–506, 1985.

[11] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, pp. 196–218. Springer-Verlag, 1985. LNCS 193.

[12] T. O. O'Malley, D. J. Richardson, and L. K. Dillon. Efficient Specification-based Oracles for Critical Systems. Submitted, August 1995.

[13] D. Plaisted. A low level language for obtaining decision procedures for classes of temporal logics. In *An Interval Logic for Higher Level Temporal Reasoning*, LNCS 164, pp. 403–420. Springer-Verlag, 1983.

[14] 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.

[15] 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.

[16] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasoning. In *Proc. 2nd ACM Symp. Principles of Distributed Computing*, pp. 173–186, Montreal, Canada, August 1983.

[17] A. P. Sistla and E. M. Clarke. The complexity of propositional linear time temporal logic. In *Proc. 14th ACM Symp. Theory of Computing*, pp. 159–168, San Francisco, May 1982.

[18] P. Wolper. The tableau method for temporal logic: An overview. In *Logique et Analyse, Novelle Série* 28^e *Année, 110–111 Numero Special (Automated Reasoning in Non-Classical Logic)*, pp. 119–136, June–September 1985.