A Decision Algorithm for Full Propositional Temporal Logic*

Y. Kesten** , Z. Manna*** , H. McGuire*** , A. Pnueli**

Abstract
The paper presents an efficient algorithm for checking the satisfiability of a propositional linear time temporal logic formula, which may have past as well as future operators. This algorithm can be used to check validity of such formulas over all models as well as over computations of a finite-state program (*model checking*).
Unlike previous theoretical presentations of a decision method for checking satisfiability or validity, whose first step is to construct the full set of all possible atoms of a tableau (satisfaction graph) and immediately pay the worst case exponential complexity price, the algorithm presented here builds the tableau *incrementally*. This means that the algorithm constructs only those atoms that are reachable from a possible initial atom, satisfying the formula to be checked.
While incremental tableau construction for the future fragment of linear time temporal logic can be done in a single pass, the presence of past operators requires multiple passes that successively construct augmented versions of existing atoms, while still maintaining consistency and reachability.
The proof of correctness of the algorithm is based on showing that any model of the considered formula is *embedded* as a path in the tableau at all the construction stages, and can be delineated when the construction terminates.
The paper also describes an implementation of the algorithm with further attention to efficiency. This implementation is available as a support system for the book [8] under the name "temporal prover". It has been used to verify all the propositional temporal formulas and to model-check all the finite-state programs appearing in the book.

Keywords: temporal logic, satisfiability checking, validity checking, model checking, past and future operators, incremental tableau, automatic verification.

1 Introduction

We consider the full language of linear time temporal logic, as defined by Kamp [5]. This language includes operators that symmetrically allow references to the *future*

* This research was supported in part by the National Science Foundation under grant CCR-89-11512, by the Defense Advanced Research Projects Agency under contract NAG2-703, by the United States Air Force Office of Scientific Research under contract F49620-93-1-0139, by the European Community ESPRIT Basic Research Action Projects SPEC (3096) and REACT (6021), and by the France-Israel project for cooperation in Computer Science.

** Department of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel. E-mail: yonit@wisdom.weizmann.ac.il

*** Department of Computer Science, Stanford University, Stanford, CA 94305.
E-mail: manna@cs.stanford.edu

and *past* of any time instant. Temporal logic has been used successfully for specifying properties of reactive systems, including concurrent programs and hardware circuits.

One of the important advantages of temporal logic is that its propositional version (PTL), while being expressive enough to specify interesting properties of programs, circuits, and communication protocols, is decidable. This has been exploited extensively for automatic verification of such systems. All of these applications used only the future fragment of the temporal language.

The heart of many of these decision methods is an algorithm for checking the satisfiability of a given temporal formula φ. This algorithm is based on *tableau construction*. While the name *temporal tableau* is commonly used in the description of these algorithms, there are in fact two types of temporal tableaux, to which we may refer as *declarative* and *incremental* tableaux, respectively. In both cases, the algorithm constructs a graph (*tableau*) G, whose nodes (*atoms*) are labeled by sets of formulas derived from φ, such that every model of φ is represented as an infinite path in G. The difference between the two constructions is that all possible atoms are present in declarative tableaux, while only atoms reachable from initial atoms are present in incremental tableaux. Declarative tableaux are clearer and easier to understand and analyze. They are used to prove properties of the logic, such as upper bounds on the complexity of decision procedures [2], [11]. Incremental tableaux, on the other hand, are more efficient, and are obviously better for implementation [1], [10], [9].

The reason for this difference in efficiency is that the declarative construction starts by constructing all possible atoms, immediately realizing the worst case complexity which, as shown in [11], is exponential. The incremental construction, on the other hand, proceeds more conservatively, constructing only reachable atoms. As a result, in most cases, a much smaller number of atoms is ever explored.

This paper presents an implementation of an algorithm for deciding the satisfiability of a propositional linear time temporal formula in the full language that includes both past and future operators, using incremental tableaux.

For comparison, the decision procedure described in [7] is based on declarative tableaux and is, therefore, unsuitable for implementation. The algorithms described in [10] and [9] are incremental but are restricted to the future fragment of the language.

Other previous implementations of a decision procedure for the full language are presented in [4] and [3], but very little algorithmic detail is provided.

One can clearly use any satisfiability checking algorithm to check validity over all models, since a formula φ is valid iff $\neg\varphi$ is unsatisfiable. As shown in [6] and [11], a satisfiability (validity) checking algorithm for linear time temporal logic can be used to check validity of a formula φ over all computations of a given finite-state program P (*model checking*). This can be done either by checking the validity of the formula $Sem_P \rightarrow \varphi$, where Sem_P is a formula characterizing the computations of program P, or by forming a cartesian product of the program's transition graph with the tableau constructed by the satisfiability checking algorithm applied to $\neg\varphi$.

The implementation reported here has been used to verify all the propositional formulas appearing in the book [8] and to model-check all the finite-state programs appearing there. The implementation is available to readers of the book as a support tool.

Section 2 presents a high-level, simplified description of the algorithm. Section 3 provides a proof of correctness for the simplified algorithm. Section 4 provides additional details about our implementation, which is derived from the implementation reported in [12]. The section lists several points in which the implementation improves upon the simplified description of the algorithm by being more general and more efficient than the simplified algorithm. Section 5 presents an improved version of the basic algorithm which uses additional data structures removing some redundant and unnecessary construction steps. Section 6 summarizes the work, with a short discussion.

2 An Incremental Tableau Algorithm for PTL formulas

In this section, we present a simplified description of an algorithm for checking the satisfiability of a temporal formula φ.

2.1 The Language PTL

In what follows, we refer to the PTL language, with syntax and semantics as defined in [8].

For a simpler presentation of the algorithm, we consider only the following operators:

- Boolean operators: $\neg$, $\wedge$.
- Temporal operators: $\bigcirc$ – Next, $\mathcal{U}$ – Until, $\ominus$ – Previous, $\mathcal{S}$ – Since.

It is well known that all the other boolean and temporal operators, such as $\vee$, $\diamondsuit$, and $\square$, can be defined in terms of these basic ones. As described in Section 4, the implementation accepts formulas using the full complement of boolean and temporal operators.

We also use the formula *true*, which is always true, and the notations *false* and *first* as abbreviations for $\neg true$ and $\neg \ominus true$, respectively. We further identify $\neg\neg p$ with p and reduce double negations whenever they arise.

Basic and Nonbasic formulas A formula is called *basic* if it has one of the following forms:

Proposition, *true*, $\bigcirc p$, $\ominus p$,

or the negation of any of these forms. Otherwise, it is called *nonbasic*.

Every nonbasic formula has one or two *preconditions*, which are sets of formulas, according to the following table:

Formula	pre_1	pre_2
$p \wedge q$	$\{p, q\}$	—
$p \mathcal{U} q$	$\{q\}$	$\{p, \bigcirc(p \mathcal{U} q)\}$
$p \mathcal{S} q$	$\{q\}$	$\{p, \ominus(p \mathcal{S} q)\}$
$\neg(p \wedge q)$	$\{\neg p\}$	$\{\neg q\}$
$\neg(p \mathcal{U} q)$	$\{\neg q, \neg p\}$	$\{\neg q, \neg \bigcirc(p \mathcal{U} q)\}$
$\neg(p \mathcal{S} q)$	$\{\neg q, \neg p\}$	$\{\neg q, \neg \ominus(p \mathcal{S} q)\}$

Clearly, a nonbasic formula φ holds at a time-instant t if and only if all the formulas in at least one of its preconditions pre_i hold at t.

2.2 Atoms and Coverage

Let φ be a formula whose satisfiability we wish to check.

Definition: The *closure of* φ, $CL(\varphi)$, is the smallest set of formulas containing φ and satisfying:

- *first* $\in CL(\varphi)$.
- $p \in CL(\varphi)$ iff $\neg p \in CL(\varphi)$.
- If $\bigcirc p \in CL(\varphi)$ or $\ominus p \in CL(\varphi)$, then $p \in CL(\varphi)$.
- If a nonbasic formula $p \in CL(\varphi)$, then all the formulas appearing in the preconditions of p are in $CL(\varphi)$.

The closure of φ can be partitioned into $CL(\varphi) = CL^+(\varphi) \cup CL^-(\varphi)$, where $CL^-(\varphi)$ is the set of all formulas in $CL(\varphi)$ of the form $\neg p$, and $CL^+(\varphi)$ is the set of all other formulas in $CL(\varphi)$. It can be shown that $|CL^-(\varphi)| = |CL^+(\varphi)| \leq 1.5 \cdot |\varphi| + 2$, where $|\varphi|$ is the size of the formula φ.

Definition: A φ*-atom* is a set of formulas $A \subseteq CL(\varphi)$ satisfying:

- *false* $\notin A$.
- If $\neg p \in A$ then $p \notin A$.
- If *first* $\in A$ then $\ominus p \notin A$ for any p.
- If $p \in CL(\varphi)$ is a nonbasic formula, then $p \in A$ iff $pre_i \subseteq A$, for one of the preconditions of p.

It is possible to establish $3^{|CL^+(\varphi)|} \leq 3^{1.5 \cdot |\varphi|+2}$ as an upper bound on the number of φ-atoms.[4]

Definition: An atom A is said to *generalize* atom B, denoted $A \sqsubseteq B$, if:

- $A \subseteq B$.
- For every formula $p \mathcal{U} q \in CL(\varphi)$, if $p \mathcal{U} q \in A$ (hence also $p \mathcal{U} q \in B$) and $q \in B$, then also $q \in A$. Thus, if B manages to satisfy $p \mathcal{U} q$ by satisfying q, so should A.

Note that, while B has more formulas than A, it is more specific, in the sense that it commits itself to more formulas being true than A. Thus, A covers more cases than B does. Atom A is a *strict generalization* of B, denoted $A \sqsubset B$, if $A \sqsubseteq B$ and $A \neq B$ (implying $A \subset B$).

Definition: A set $\{A_1, \ldots, A_k\}$ of φ-atoms is said to be a (*complete*) *cover* of a set of formulas $S \subseteq CL(\varphi)$ if:

- Each A_i, for $i = 1, \ldots, k$, contains S; i.e., $S \subseteq A_i$.

[4] A better upper bound of $3^{|\varphi|+2}$ can be obtained if we restrict ourselves to *irreducible atoms*. A formula in $CL(\varphi)$ is called *extraneous* if it is neither *first* nor a subformula of φ. An atom A is called *irreducible* if there does not exist an extraneous formula $p \in A$ such that $A - \{p\}$ is also an atom. It can be shown that the algorithm works correctly, constructing a somewhat smaller tableau, if only irreducible atoms are used.

- Every atom B containing S is generalized by some A_i, $i = 1, \ldots, k$.

There are many ways to compute a complete cover for a given set of formulas $S \subseteq CL(\varphi)$. Section 4 describes one such algorithm, with attention to efficiency. Here, we assume only that we have one such an algorithm, and denote the complete cover for S by $Cover(S)$. Implementations usually attempt to construct a *minimal cover*, which is a cover such that $A_i \not\sqsubseteq A_j$ for all i and j.

The following property follows from the definition of a cover.

Property 2.1 (Monotonicity) *Given two sets of formulas $S_1 \subseteq S_2 \subseteq CL(\varphi)$, then each $B_2 \in Cover(S_2)$ is generalized by some $B_1 \in Cover(S_1)$, i.e., $B_1 \sqsubseteq B_2$.*

2.3 The Tableau Algorithm

Definitions:

A set of formulas S is called *locally consistent* if it satisfies the first three requirements of an atom. That is, it does not contain the formula *false*, it does not contain two formulas of the form p and $\neg p$, and it does not contain a formula of the form $\ominus p$ together with the formula *first*.

An atom is called *initial* if it contains the formula *first*.

For a set of formulas S, we define the following sets of formulas:

- $Next(S) = \{p \mid \bigcirc p \in S\} \cup \{\neg p \mid \neg \bigcirc p \in S\}$.
- $Prev(S) = \{p \mid \ominus p \in S\} \cup \{\neg p \mid \neg \ominus p \in S\}$.
- $Basic(S)$ is the set of basic formulas among S.

For two atoms A and B, we say that the pair (A, B) is *neighborly consistent* if $Next(A) \subseteq B$ and $Prev(B) \subseteq A$.

The algorithm constructs and manipulates a graph structure G consisting of vertices which are φ-atoms, and directed edges connecting them. We denote the set of vertices by $\mathcal{V}$, and the set of edges by $\mathcal{E}$. In addition, we maintain a set $\overline{\mathcal{E}}$ of *removed edges*. These are edges that have been once in $\mathcal{E}$ but have been removed from the graph, and their inclusion in $\overline{\mathcal{E}}$ is intended to ensure that they will not be recreated. We denote the union $\mathcal{E} \cup \overline{\mathcal{E}}$ by $\mathcal{E}^+$.

An edge connecting atom A to atom B indicates that B is a possible temporal successor of A. We refer to such an edge as either $\langle A, B\rangle$, $\langle A, B\rangle_{future}$, or $\langle B, A\rangle_{past}$. An edge $\langle A, B\rangle$ may be marked as *future-satisfactory*, if $Next(A) \subseteq B$, and *past-satisfactory*, if $Prev(B) \subseteq A$. If an edge $\langle A, B\rangle$ is satisfactory in both directions (the pair (A, B) is neighborly consistent), it is said to be *satisfactory*.

The algorithm comprises two construction phases:

- Phase I:
  Given a formula φ, we construct an initial graph $G : (\mathcal{V}, \mathcal{E}, \overline{\mathcal{E}})$, in which the set of vertices $\mathcal{V}$ comprises a set of initial (*root*) atoms, each containing the formulas φ and *first*, and a *generic future* atom F containing no formulas. The set of edges $\mathcal{E}$ comprises edges drawn from each element of $\mathcal{V}$ to F. The set of forbidden edges $\overline{\mathcal{E}}$ is initially empty.

– Phase II:
As long as some edge $\langle A, B\rangle$ in the graph is unsatisfactory in some direction, we consider a new atom that augments either A or B (according to the direction in which the edge is unsatisfactory) by additional formulas that are necessary to make this direction satisfactory. This atom may already exist in the graph or, if not, will be added to the graph. The unsatisfactory edge will be replaced by a new edge connecting to this new atom, and some additional connections may be duplicated. The unsatisfactory edge is moved to $\overline{\mathcal{E}}$.
This process continues until all edges in $\mathcal{E}$ are satisfactory in both directions.

Phase I of the algorithm is presented in procedure *construct-initial*, and phase II, in *correct-graph*.

construct-initial(φ)
  $F := \{\}$ /* F is the generic future atom */
  $\mathcal{V} := \{F\}$
  $\mathcal{E} := \{\langle F, F\rangle\}$ /* draw an edge from F to itself */
  $\overline{\mathcal{E}} := \{\}$ /* $\overline{\mathcal{E}}$ is initially empty */
  for each atom $A \in Cover(\{\varphi, \mathit{first}\})$ do
    add A to $\mathcal{V}$ /* create a new vertex for each initial atom */
    add edge $\langle A, F\rangle$ to $\mathcal{E}$ /* draw an edge from A to the generic future */
end construct-initial

correct-graph
  for each unsatisfactory edge $\langle A, B\rangle_d \in \mathcal{E}$ do
    [if $d = \mathit{future}$ then $\mathcal{A} := Cover(Next(A) \cup B)$
        else $\mathcal{A} := Cover(Prev(A) \cup B)$
    for each atom $B' \in \mathcal{A}$ do
      [add B' to $\mathcal{V}$
      add-edge($\langle A, B'\rangle_d$)
      for each edge $\langle B, Y\rangle_d \in \mathcal{E}^+$ do
        add-edge($\langle B', Y\rangle_d$)]
    move edge $\langle A, B\rangle_d$ from $\mathcal{E}$ to $\overline{\mathcal{E}}$]
end correct-graph.

Upon detecting an unsatisfactory edge $\langle A, B\rangle_d$, the algorithm considers an augmentation of the formulas in B by the formulas that A requires for neighborly consistency, and generates in $\mathcal{A}$ a set of atoms that forms a complete cover of the augmentation. Then, there is a loop which adds to $\mathcal{V}$ each atom $B' \in \mathcal{A}$ that is not already there. For each edge departing from atom B to some atom Y in direction d, we construct a duplicate edge, connecting B' to Y. This ensures that any previous path segment, traversing A, B, and Y in direction d, is now available as the segment A, B', Y.

add-edge($\langle X, Y\rangle_d$)
  if edge $\langle X, Y\rangle_d \notin \mathcal{E}^+$

$$\left[\begin{array}{l}\text{add } \langle X, Y\rangle_d \text{ to } \mathcal{E} \\ \text{for each edge } \langle Y, Z\rangle_d \in \overline{\mathcal{E}} \text{ do} \\ \qquad \text{move } \langle Y, Z\rangle_d \text{ from } \overline{\mathcal{E}} \text{ to } \mathcal{E} \\ \text{for each edge } (W, X)_d \in \overline{\mathcal{E}} \text{ do} \\ \qquad \text{move } \langle W, X\rangle_d \text{ from } \overline{\mathcal{E}} \text{ to } \mathcal{E}\end{array}\right]$$

end add-edge

The procedure *add-edge* constructs an edge from X to Y in direction d, provided it is not already there (not in $\mathcal{E}$) and has not been constructed before and deleted (as would be evident by belonging to $\overline{\mathcal{E}}$). The newly constructed edge is registered in $\mathcal{E}$. Also, since there is a new connection between X and Y, we move from $\overline{\mathcal{E}}$ back to $\mathcal{E}$ (reinstate) all the edges entering X and departing from Y in direction d. This is so that the next time these edges will be considered, and surely removed again because they still connect neighborly inconsistent atoms, the newly added edge from X to Y will also be duplicated to the replacement of X or the replacement of Y.

The entire algorithm:

satisfy(φ)
  construct-initial(φ)
  correct-graph
  strongly-connected-components-analysis
end satisfy.

The procedure *strongly-connected-components-analysis* analyzes the graph by decomposing it into maximal strongly connected components and identifying those which are *self-fulfilling* [6]. A strongly connected component $\mathcal{C}$ is called *self-fulfilling* if every atom $A \in \mathcal{C}$ has at least one successor, and for every formula $p\mathcal{U}q \in A$, there exists an atom $B \in \mathcal{C}$ such that $q \in B$.

If the procedure finds a self-fulfilling component which is reachable from an initial atom (an atom containing *first* and φ), the procedure reports success, claiming that φ is satisfiable. Otherwise, it reports failure, claiming that φ is unsatisfiable ($\neg\varphi$ is valid).

3 Proof of correctness for the Tableau Algorithm

In this extended abstract we omit all proofs. They are provided in the fuller version of this paper.

We fix our attention on a formula φ whose satisfiability we wish to check. The first proposition states that the algorithm always terminates.

Proposition 1 Termination. Algorithm *satisfy* terminates.

The proof of termination is based on the following points: there are only finitely many possible atoms, so the set of possible new edges is bounded; only new edges cause later processing; and the processing for each edge is bounded.

The main theorem of this paper is Theorem 9 stating that the algorithm succeeds iff formula φ is satisfiable. One direction of the proof, showing that if the algorithm succeeds then φ is satisfiable, is rather straightforward. The algorithm reports success only if there exists a self-fulfilling strongly connected subgraph $\mathcal{C}$ reachable from some initial atom I. It can be shown that the path starting at I, proceeding to $\mathcal{C}$, and then repeatedly following a closed path that traverses all vertices in $\mathcal{C}$, yields a model for φ.

In the other direction, we have to show that if there exists a model σ for φ, then the algorithm is guaranteed to find at least one self-fulfilling subgraph $\mathcal{C}$ reachable from some initial atom. This proof is based on the notion of *embedding*, showing that the model σ induces an infinite sequence of atoms (called a *fulfilling pre-model* in the proof below) which can be traced as a path within the tableau at all stages of its construction.

Definitions:

- We write $(A, B) \sqsubseteq (A', B')$ as abbreviation for the two generalizations $A \sqsubseteq A'$ and $B \sqsubseteq B'$.
- A *pre-model* (*for* φ) is an infinite sequence of φ-atoms $\pi : A_1, A_2, \ldots$, such that:
  - $\{\varphi, \mathit{first}\} \subseteq A_1$
  - For every $i \geq 1$, *first* $\notin A_{i+1}$, and the pair (A_i, A_{i+1}) is neighborly consistent.
- A *fulfilling pre-model* (*for* φ) is a pre-model for φ, $\pi : A_1, A_2, \ldots$, such that for every $i \geq 1$, if $p \mathcal{U} q \in A_i$, then there is a $j \geq i$ such that $q \in A_j$.

Theorem 2. *The formula* φ *is satisfiable iff there is a fulfilling pre-model satisfying* φ.

The proof of this theorem follows the lines of Proposition 1 of [7].

The application of the algorithm of the preceding section constructs a sequence of graphs, $\mathcal{G}_\varphi : G_0, G_1, \ldots$, where each G_i consists of the components $(\mathcal{V}_i, \mathcal{E}_i, \overline{\mathcal{E}}_i)$, for $i = 0, 1, \ldots$. The graph G_0 is the one obtained at the conclusion of procedure *construct-initial*. The graphs $G_1, G_2, \ldots$, are the ones observed at the end of each iteration of the main loop within procedure *correct-graph*.

Definitions:

- Let $\pi : A_1, A_2, \ldots$, be a pre-model. A *generalization of* π is an infinite sequence of atoms $\widetilde{\pi} : B_1, B_2, \ldots$, such that B_i generalizes A_i, i.e., $B_i \sqsubseteq A_i$, for all $i \geq 1$. We write $\widetilde{\pi} \sqsubseteq \pi$.
- A *path in* $G_i : (\mathcal{V}_i, \mathcal{E}_i, \overline{\mathcal{E}}_i)$, is a (possibly infinite) sequence of atoms $\sigma : B_1, B_2, \ldots$, where $B_k \in \mathcal{V}_i$ for all $k \geq 1$ and, for each pair of consecutive atoms B_k and B_{k+1} in σ, there is an edge $\langle B_k, B_{k+1} \rangle \in \mathcal{E}_i \cup \overline{\mathcal{E}}_i$.
- A path σ in G_i is called a *good path*, if for each pair of consecutive atoms B_k and B_{k+1} in σ, the edge $\langle B_k, B_{k+1} \rangle$ belongs to $\mathcal{E}_i$ (rather than to $\overline{\mathcal{E}}_i$).
- A *prefix* of $\sigma : B_1, B_2, \ldots$ is a finite sequence of atoms $B_1, B_2, \ldots, B_j$, for some $j \geq 1$, and is denoted by $\sigma[1..j]$.
  The infinite subsequence $B_j, B_{j+1}, \ldots$ is a *suffix* of σ and is denoted by $\sigma[j..]$.
- Let $\sigma : A_1, \ldots, A_k$ be a finite path, and $\sigma' : B_1, \ldots$ a possibly infinite path. We denote the concatenation of σ and σ' by $\sigma; \sigma' = A_1, \ldots, A_k, B_1, \ldots$.

Proposition 3. *Let* π *be a pre-model. For every* $G_i \in \mathcal{G}_\varphi$, *there is a path* σ *in* G_i *which generalizes* π.

Proposition 4. *Let $G_i : (\mathcal{V}_i, \mathcal{E}_i, \overline{\mathcal{E}}_i)$ be a graph in $\mathcal{G}_\varphi$. Let $\langle X, Y\rangle \in \overline{\mathcal{E}}_i$. Then, one of the following two cases holds:*

(a) $\langle X, Y\rangle$ is not past-satisfactory and, for each atom $X' \in Cover(X \cup Prev(Y))$,
  - *$X' \in \mathcal{V}_i$ and $\langle X', Y\rangle \in \mathcal{E}_i^+$ is past-satisfactory.*
  - *For every atom $Q \in \mathcal{V}_i$ such that $\langle Q, X\rangle \in \mathcal{E}_i^+$, $\langle Q, X'\rangle \in \mathcal{E}_i^+$.*

(b) $\langle X, Y\rangle$ is not future-satisfactory and, for each atom $Y' \in Cover(Y \cup Next(X))$,
  - *$Y' \in \mathcal{V}_i$ and $\langle X, Y'\rangle \in \mathcal{E}_i^+$ is future-satisfactory.*
  - *For every atom $Q \in \mathcal{V}_i$ such that $\langle Y, Q\rangle \in \mathcal{E}_i^+$, $\langle Y', Q\rangle \in \mathcal{E}_i^+$.*

Definition: We say that the pair of atoms (A', B') *clones* the pair (A, B) in G_i if

- $(A, B) \sqsubseteq (A', B')$.
- For every $X \in \mathcal{V}_i$, if $\langle X, A\rangle \in \mathcal{E}_i^+$ then $\langle X, A'\rangle \in \mathcal{E}_i^+$.
- For every $Y \in \mathcal{V}_i$, if $\langle B, Y\rangle \in \mathcal{E}_i^+$ then $\langle B', Y\rangle \in \mathcal{E}_i^+$.

Proposition 5. *Let $\pi : A_1, A_2, \ldots$ be a pre-model. For some $G_i \in \mathcal{G}_\varphi$, let the edge $\langle B_j, B_{j+1}\rangle \in \overline{\mathcal{E}}_i$ be such that $(B_j, B_{j+1}) \sqsubseteq (A_j, A_{j+1})$. Then, there exists an edge $\langle B'_j, B'_{j+1}\rangle \in \mathcal{E}_i^+$, such that (B'_j, B'_{j+1}) clones (B_j, B_{j+1}) in $\mathcal{V}_i$ and*

$$(B_j, B_{j+1}) \sqsubseteq (B'_j, B'_{j+1}) \sqsubseteq (A_j, A_{j+1}).$$

The following proposition improves on Proposition 5 by claiming the existence of a cloning edge, as above, but one that belongs to $\mathcal{E}_i$, rather than to $\mathcal{E}_i^+$.

Proposition 6. *Let $\pi : A_1, A_2, \ldots$ be a pre-model. For some $G_i \in \mathcal{G}_\varphi$, let the edge $\langle B_j, B_{j+1}\rangle \in \overline{\mathcal{E}}_i$ be such that $(B_j, B_{j+1}) \sqsubseteq (A_j, A_{j+1})$. Then, there exists an edge $\langle B'_j, B'_{j+1}\rangle \in \mathcal{E}_i$, such that (B'_j, B'_{j+1}) clones (B_j, B_{j+1}) in $\mathcal{V}_i$ and*

$$(B_j, B_{j+1}) \sqsubseteq (B'_j, B'_{j+1}) \sqsubseteq (A_j, A_{j+1}).$$

Proposition 7. *For every pre-model π and every $G_i \in \mathcal{G}_\varphi$, there is a good path σ_g in G_i such that σ_g generalizes π.*

The following corollary specializes the preceding claim to the graph G_∞ obtained at the termination of procedure *correct-graph*.

Corollary 8. *For every fulfilling pre-model π, there exists a good path σ_g in G_∞ such that σ_g is a fulfilling pre-model generalizing π.*

Theorem 9. *The algorithm reports success iff φ is satisfiable.*

Proof: First assume that φ is satisfiable. Then, from Theorem 2 and Corollary 8, there exists a good path $\sigma_g : B_1, \ldots$ in G_∞, such that σ_g is a fulfilling pre-model for φ. Since a pre-model is infinite and G_∞ is a finite graph, there exists a strongly-connected-component $\mathcal{C}$ in G_∞ reachable from B_1, such that all atoms appearing infinitely many times in σ_g are in $\mathcal{C}$. Moreover, since σ_g is self-fulfilling, $\mathcal{C}$ is self fulfilling.

In the other direction, assume that the algorithm reports success. Namely, the algorithm finds a self-fulfilling, strongly-connected-component $\mathcal{C}$, reachable from an initial atom B_1. Let $B_1, \ldots, B_k$ be a path in G_∞, such that B_k is contained in $\mathcal{C}$. Then, any infinite path $B_1, \ldots, B_k, \ldots$ is a fulfilling pre-model for φ. From Theorem 2, φ is satisfiable.

4 Implementation and Improvements

In this section, we consider several aspects in which the implementation improves on the simplified description of the algorithm, as presented in Section 2, by being more general and more efficient.

A More General Language The implementation accepts a much richer temporal language than the one described in Section 2. It recognizes the boolean operators:

$$\neg, \quad \vee, \quad \wedge, \quad \longrightarrow, \quad \longleftrightarrow,$$

and the temporal operators:

$$\bigcirc, \quad \square, \quad \diamondsuit, \quad \mathcal{U}, \quad \mathcal{W}\ \text{(waiting-for)}, \quad \ominus, \quad \boxminus, \quad \diamondminus, \quad \mathcal{S}, \quad \mathcal{B}\ \text{(back-to)}.$$

The additional operators are not translated into primitive ones; they are handled directly. Avoiding translation conserves formulas' sizes and keeps the implementation's outputs reasonably understandable. The notions of basic and nonbasic formulas extend to the richer language in an obvious way.

Incremental Cover Consider the situation in procedure *correct-graph* in which the edge $\langle A, B\rangle_{future}$ is found to be unsatisfactory (in direction *future*), and we construct in $\mathcal{A}$ a cover of the set of formulas $Next(A) \cup B$. As seen from Proposition 5 (and Proposition 6), the situation is that we have some atom Y, belonging to the pre-model π (called in these propositions A_{j+1}, while B is called B_{j+1}), such that $B \sqsubseteq Y$, and we are interested only in atoms B' such that $B \sqsubseteq B' \sqsubseteq Y$. Consequently, it is sufficient to construct the *incremental cover* of B and $Next(A)$.

Definition: A set $\{A_1, \ldots, A_k\}$ of φ-atoms is said to be an *incremental cover* of an atom B and a set of formulas $S \subseteq CL(\varphi)$ if:

- Each A_i, for $i = 1, \ldots, k$, contains $B \cup S$.
- Every atom Y generalized by B and containing S is generalized by some A_i, $i = 1, \ldots, k$; i.e., $B \sqsubseteq A_i \sqsubseteq Y$.

The following recursive procedure is used in the implementation to calculate the incremental cover of B and S.

incremental-cover(*base: atom, increment: set of formulas*) : *set of sets of formulas*
    $new := increment - base$
    if $new = \{\}$ then return *base*
    Let $p :=$ longest formula in *new*
    if p is basic then
        $\mathcal{A} := incremental\text{-}cover(base,\ new - \{p\})$
    Otherwise, if p is nonbasic with preconditions $pre_1, \ldots, pre_k$, then

$$\mathcal{A} := \bigcup_{i=1}^{k} incremental\text{-}cover(base,\ (new \cup pre_i) - \{p\})$$

    $\mathcal{B} := \{X \cup \{p\} \mid X \in \mathcal{A}\}$
    $\mathcal{D} := \{X \in \mathcal{B} \mid X \text{ is locally consistent}\}$
    return $\mathcal{D}$
end construct-initial

Additional Improvements Other improvements are:

- Unsatisfactory edges are truly deleted — not saved anywhere — except in certain circumstances.
- Atoms that lose all their edges in either direction (except the past, for initial atoms) are deleted.

It has been proven that correctness is maintained even with each of these changes of the basic algorithm.

5 Improved Algorithm

The basic algorithm described in the section 2 is sound and complete, yet contains some inefficiencies, for the following reasons:

- **Non selective inheritance of edges**: When an unsatisfactory edge $\langle X, Y \rangle_d$ is corrected (in direction d), for every $Y' \in \mathit{Cover}(Y \cup f(X))$, where $f \in \{\mathit{Prev}, \mathit{Next}\}$, it is ensured that $Y' \in \mathcal{V}_{i+1}$, and for every atom $Q \in \mathcal{V}_i$ such that $\langle Y, Q \rangle_d \in \mathcal{E}_i^+$, it is ensured that $\langle Y', Q \rangle_d \in \mathcal{E}_{i+1}^+$. We say that the edge $\langle Y, Q \rangle_d$ is *d-inherited* by the new atom Y'. Y' will inherit *all* $\langle Y, Q \rangle_d$ edges, from both $\mathcal{E}_i$ and $\overline{\mathcal{E}}_i$. This non selective inheritance is redundant, creating redundant atoms and edges.
- **Reinstatement of removed edges**: An unsatisfactory edge in $\mathcal{E}$, is corrected and moved from $\mathcal{E}$ to $\overline{\mathcal{E}}$. The same edge may me moved back to $\mathcal{E}$ at a later stage. We say that this edge has been *reinstated.* Reinstated edges, can not be distinguished from unsatisfactory edges in $\mathcal{E}$ which have never been corrected. Their correction involves all correction activities, most of which are redundant when performed for the second (or more) time.

Since both inheritance and reinstatement of edges are propagated through the graph, both redundancies are multiplied.

The Improved algorithm corrects both of these deficiencies. Two corrections are introduced:

- **Clone Lists**: For every atom $A \in \mathcal{E}^+$, we maintain two *clone lists*, denoted *future-clones*(A) and *past-clones*(A). These lists are constructed incrementaly as follows. Whenever a new atom B is created, both *d-clones*(B) are initialized to empty-lists. Whenever an edge $\langle A, B \rangle_d$ is corrected in direction d, all atoms $B' \in \mathit{Cover}(B \cup f(A))$ are added to *d-clones*(B).
- **Selective Inheritance**: Let $\langle A, B \rangle_d \in \mathcal{E}$ be an unsatisfactory edge currently being corrected in direction d. Then, for every edge $e = \langle B, Q \rangle_d \in \mathcal{E}$ and every $B' \in \mathit{Cover}(A \cup f(B))$, e is inherited by B' *only* if e is d-satisfactory. Namely, only d-satisfactory edges will be d-inherited. An edge which is unsatisfactory in both directions, will be inherited prior to being corrected, but only by one of the possible d-inheritance. Whenever a d-satisfactory edge $\langle A, B \rangle$ is being constructed, it is d-inherited recursively, starting with the atoms in the appropriate clone list of either A or B.

The clone lists, together with recursive propagation of newly created d-satisfactory edges, avoids the need for reinstatement of edges. An edge that has been removed from $\mathcal{E}$, will never be moved back to $\mathcal{E}$.
The improved algorithm:

correct-graph
for each unsatisfactory edge $\langle A, B\rangle_d \in \mathcal{E}$ do
if $d = \mathit{future}$ then $\mathcal{A} := \mathit{Cover}(\mathit{Next}(A) \cup B)$
else $\mathcal{A} := \mathit{Cover}(\mathit{Prev}(A) \cup B)$
$d' = \mathit{inverse}(d)$
if $\langle A, B\rangle_d$ is d'-unsatisfactory
for each clone-atom $B' \in d'\text{-}\mathit{clones}(B)$ do
$\mathit{add\text{-}edge}(\langle A, B'\rangle_d)$
for each $B' \in \mathcal{A}$ do
add B' to $\mathcal{V}$
add B' to $d\text{-}\mathit{clones}(B)$
$\mathit{add\text{-}edge}(\langle A, B'\rangle_d)$
for each $\langle B, Y\rangle_d \in \mathcal{E}$ do
if $\langle B, Y\rangle_d$ is d-satisfactory
$\mathit{add\text{-}edge}(\langle B', Y\rangle_d)$
move edge $\langle A, B\rangle_d$ from $\mathcal{E}$ to $\overline{\mathcal{E}}$
end correct-graph.

add-edge$(\langle X, Y\rangle_d)$
$d' = \mathit{inverse}(d)$
if edge $\langle X, Y\rangle_d \notin \mathcal{E}^+$
add edge $\langle X, Y\rangle_d$ to $\mathcal{E}$
if $\langle X, Y\rangle_d$ is d-satisfactory
for each $X' \in d\text{-}\mathit{clones}(X)$ do
$\mathit{add\text{-}edge}(\langle X', Y\rangle_d)$
if $\langle X, Y\rangle_d$ is d'-satisfactory
for each $Y' \in d'\text{-}\mathit{clones}(Y)$ do
$\mathit{add\text{-}edge}(\langle Y', X\rangle_{d'})$
end add-edge

The first phase of the algorithm, *construct-initial* remains unchanged. The proof of correctness of the algorithm proceeds along lines similar to the proof presented in section 3

6 Discussion

The paper described an algorithm for checking the satisfiability of a PTL formula that includes past and future operators. The algorithm is based on incremental tableau construction and is expected to perform better on the average than previously available algorithms for this problem.

As explained above, the algorithm can be used for checking general validity as well as for model checking of a temporal formula over a finite-state program.

At present, we are investigating the possibility of incorporating this algorithm as a component in a general deductive system for first-order temporal logic.

Possible generalizations that are currently being investigated consider classes of temporal logics with variables and equality that may still be decidable.

References

1. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
2. M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *J. Comp. Sys. Sci.*, 18:194–211, 1979.
3. G. D. Gough and H. Barringer. A semantic driven temporal verification. In *Proceedings of ESOP'88*, 1988.
4. G. D. Gough. Decision procedures for temporal logic, Master's thesis, University of Manchester, England, 1984.
5. J.A.W. Kamp. *Tense Logic and the Theory of Order*. PhD thesis, UCLA, 1968.
6. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.
7. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, volume 193 of *Lect. Notes in Comp. Sci.*, pages 196–218. Springer-Verlag, 1985.
8. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
9. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 6:68–93, 1984.
10. A. Pnueli and R. Sherman. Semantic tableau for temporal logic. Technical Report CS81 - 21, The Weizmann Institute, 1981.
11. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *J. ACM*, 32:733–749, 1985.
12. R. Sherman and A. Pnueli. Model checking for linear temporal logic: An efficient implementation. Technical report, Information Science Institute, USC, 1989.