A Model Checker for Linear Time Temporal Logic

Michael Fisher*

The TEMPLE Group
Department of Computer Science
University of Manchester
michael@uk.ac.man.cs ...ukc!man.cs!michael

Abstract

This report describes the design and implementation of a model checker for linear time temporal logic. The model checker uses a depth-first search algorithm that attempts to find a minimal satisfying model and uses as little space as possible during the checking procedure. The depth-first nature of the algorithm enables the model checker to be used where space is at a premium.

1 Introduction

Temporal logic has been widely used for the specification and verification of reactive systems. It has been successfully used to describe verifiable properties of state-machines derived from practical applications [CES83, BCDM84, GB88]. In this report we consider the verification of temporal properties of such state-machines through *model-checking* [CES83] (also known as satisfiability checking). Using this approach, a finite state-machine, often derived from some practical system, is checked to see if it satisfies various properties represented by temporal formulae. The satisfaction of these properties by the state-machine generally implies that the original system satisfies such properties.

In this report we consider a model checking algorithm for linear time temporal logic. We will begin by describing the type of logic and the definitions of satisfiability and model checking used here.

2 Temporal Logic and Model Checking

Temporal logic is a derivative of modal and tense logics that has been developed for use in the description of reactive computer systems [Pnu77]. Propositional temporal logic can be seen as classical propositional logic extended with various modalities. Commonly, these are $\Diamond$, $\Box$, and $\bigcirc$. The intuitive meaning of these connectives is as follows: $\Diamond\varphi$ is true if φ is true *sometime* in the future; $\Box\varphi$ is true if φ is true *always* in the future; and $\bigcirc\varphi$ is true if φ is true at the *next* moment in time.

Obviously, the model of time used will affect the meaning of such connectives. For example, if we consider time to be a dense ordering of 'moments', then we cannot use the $\bigcirc$

*This work was supported both by Alvey under grant PRJ/SE/054 (SERC grant GR/D/57942) and by ESPRIT under Basic Research Action 3096 (SPEC).

operator as there is no distinct 'next moment'. If we think of the future as being a branching structure, then the meaning of $\Diamond$ must be clarified so that we know whether $\Diamond\varphi$ means that φ must be true on some path in the future, or on all paths in the future. Thus there is a great variety of such logics, including dense logics [BKP86], branching logics [EH82] and past-time logics [LPZ85], many of which can be placed in a single framework [Fis89].

In this report we will only consider a model of time that is linear and discrete, and allows statements about the present and future. The model structure for such a logic can be given as

$$\mathcal{M} = \langle S, \mathcal{N}, \pi \rangle$$

where S is a set of states, $\mathcal{N}$ is an infix binary relation on states, and π is a mapping from states to sets of atomic propositions. Thus, given a set of basic propositions PROP, and a state s, $\pi(s)$ is the set of propositions (a subset of PROP) that are true in state s. The relation $\mathcal{N}$ is the 'next-time' relation. It is constrained to define a linear sequence of states such that any given state is only related by $\mathcal{N}$ to at most one 'next' state.

We give an interpretation for a temporal statement in a model $\mathcal{M}$, and at a particular state s. For example, the semantics of the $\bigcirc$ operator is simply

$$\langle \mathcal{M}, s \rangle \models \bigcirc\varphi \quad \text{iff} \quad \forall t \in S.\ \text{if } s\mathcal{N}t \text{ then } \langle \mathcal{M}, t \rangle \models \varphi$$

The particular temporal language we use in this report consists of the connectives of standard propositional logic together with the unary operators $\bigcirc$ and $\odot$, and the binary operators $\mathcal{U}$ and $\mathcal{W}$. The informal semantics of these operators are as follows: both $\bigcirc\varphi$ and $\odot\varphi$ mean that φ must occur in the next state, the difference being that if there is no next state, $\bigcirc\varphi$ is true, while $\odot\varphi$ is false; $\varphi\mathcal{W}\psi$ means that φ must be true up to a point in the future where ψ occurs, or always in the future if ψ never occurs; $\varphi\mathcal{U}\psi$ is similar $\varphi\mathcal{W}\psi$, the only difference being that with the $\mathcal{U}$ operator, ψ *must* occur sometime in the future. The semantics of these operators is given below. (Note that the relation $\mathcal{R}$ is the transitive closure of $\mathcal{N}$.)

$$\begin{array}{lll}
\langle \mathcal{M}, s \rangle \models p & \text{iff} & p \in \pi(s) \quad \text{for } p \in \text{PROP} \\
\langle \mathcal{M}, s \rangle \models \neg\varphi & \text{iff} & \text{not } \langle \mathcal{M}, s \rangle \models \varphi \\
\langle \mathcal{M}, s \rangle \models \varphi \wedge \psi & \text{iff} & \langle \mathcal{M}, s \rangle \models \varphi \text{ and } \langle \mathcal{M}, s \rangle \models \psi \\
\langle \mathcal{M}, s \rangle \models \varphi \vee \psi & \text{iff} & \langle \mathcal{M}, s \rangle \models \varphi \text{ or } \langle \mathcal{M}, s \rangle \models \psi \\
\langle \mathcal{M}, s \rangle \models \bigcirc\varphi & \text{iff} & \forall t \in S.\ \text{if } s\mathcal{N}t \text{ then } \langle \mathcal{M}, t \rangle \models \varphi \\
\langle \mathcal{M}, s \rangle \models \odot\varphi & \text{iff} & \exists t \in S.\ s\mathcal{N}t \text{ and } \langle \mathcal{M}, t \rangle \models \varphi \\
\langle \mathcal{M}, s \rangle \models \varphi\mathcal{U}\psi & \text{iff} & \langle \mathcal{M}, s \rangle \models \psi \\
 & & \quad \text{or} \\
 & & \exists u \in S.\quad s\mathcal{R}u \text{ and} \\
 & & \qquad \langle \mathcal{M}, u \rangle \models \psi \text{ and} \\
 & & \qquad \langle \mathcal{M}, s \rangle \models \varphi \text{ and} \\
 & & \qquad \forall t \in S.\ \text{if } s\mathcal{R}t \text{ and } t\mathcal{R}u \text{ then } \langle \mathcal{M}, t \rangle \models \varphi \\
\langle \mathcal{M}, s \rangle \models \varphi\mathcal{W}\psi & \text{iff} & \langle \mathcal{M}, s \rangle \models \varphi\mathcal{U}\psi \\
 & & \quad \text{or} \\
 & & \langle \mathcal{M}, s \rangle \models (\varphi \wedge \neg\psi) \text{ and} \\
 & & \quad \forall t \in S.\ (\text{ if } s\mathcal{R}t \text{ then } \langle \mathcal{M}, t \rangle \models \varphi \wedge \neg\psi)
\end{array}$$

The $\Diamond$ and $\square$ operators can be derived from the $\mathcal{U}$ and $\mathcal{W}$ operators respectively as follows:

$$\begin{array}{rcl}
\Diamond\varphi & \stackrel{\text{def}}{=} & \mathbf{true}\,\mathcal{U}\,\varphi \\
\square\varphi & \stackrel{\text{def}}{=} & \varphi\mathcal{W}\mathbf{false}
\end{array}$$

A temporal statement, φ, is satisfied in a model $\mathcal{M}$ if

$$\langle \mathcal{M}, s_0 \rangle \models \varphi$$

where s_0 is the initial state in the ordering defined by $\mathcal{N}$.

We now consider the model checking of finite state-machines. A state-machine is represented by a structure of the form $\langle S, \mathcal{N}_F, \pi \rangle$, where S and π are as above and $\mathcal{N}_F$ is a binary relation similar to $\mathcal{N}$, but without the linearity restriction. Rather than just one recognised start state s_0, we have a set of such states, so we can follow many linear paths through the state-machine starting at a variety of initial states. As each of these paths is a model of the $\langle S, \mathcal{N}, \pi \rangle$ form, we can consider the state-machine as a set of such models[1]. Thus, the definition of the model checking of a formula φ over a finite state machine FSM is as follows.

> A state-machine FSM satisfies a linear temporal formula φ if, and only if, for all linear models, $\mathcal{M}$, in FSM, starting from s_0, $\langle \mathcal{M}, s_0 \rangle \models \varphi$.

Note that if we were using a branching-time temporal logic such as CTL*[ES84], we would check the satisfiability of the formula $A\varphi$, meaning that for *all* paths, φ must be satisfied, over the whole state-machine.

Thus, our main aim is to develop an effective procedure for model checking linear temporal formulae over finite state-machines.

3 Standard Model-Checking Algorithms

Many of the standard methods for checking the satisfiability of a temporal formula over a finite state automaton are derived from decision procedures for the particular logic used. Consequently, as the filtration method is widely used in decision procedures for program logics[2], model checking algorithms often use a method based on labelling states with closure subsets [FL79]. An example of this is the seminal paper on model checking by Clarke et. al. [CES83]. Lichtenstein and Pnueli [LP85] describe a similar approach, but for linear time, rather than branching time, temporal logic.

To describe this general approach, we will give an outline (based on the description given in [CES83]) of the model checking of a linear time temporal formula f over the finite-state machine *FSM*. (Note that each state can be labelled by a set of temporal formulae, and the set of formulae labelling a particular state s is represented by *label*(s).)

We first calculate the closure of the formula to be checked, f. This set, $CL(f)$, contains all subformulae of f, and has cardinality n. The algorithm then makes n passes over the state machine. On each pass, every state s is relabelled as follows:

- if $p \in CL(f)$, p is a proposition, and p is true in state s, then add p to *label*(s).
- if $\varphi \notin label(s)$ and $\neg\varphi \in CL(f)$, then add $\neg\varphi$ to *label*(s).
- if $\varphi \in label(s)$ and $\varphi \vee \psi \in CL(f)$, then add $\varphi \vee \psi$ to *label*(s).
- if $\psi \in label(s)$ and $\varphi \vee \psi \in CL(f)$, then add $\varphi \vee \psi$ to *label*(s).
- if ($\varphi \in label(s)$ and $\psi \in label(s)$) and $\varphi \wedge \psi \in CL(f)$, then add $\varphi \wedge \psi$ to *label*(s).
- if $\odot\varphi \in CL(f)$ and there exists a state t, which is a successor of s where $\varphi \in label(t)$, then add $\odot\varphi$ to *label*(s).
- if $\bigcirc\varphi \in CL(f)$ and for all states t, which are successors of s, $\varphi \in label(t)$, then add $\bigcirc\varphi$ to *label*(s).

[1]Note that when linear models are extracted from the state machine, certain renaming of states may be necessary.
[2]At least filtration is used as a decision procedure for many of the simpler program logics.

- if $\varphi \mathcal{U} \psi \in CL(f)$ and $\psi \in label(s)$, then add $\varphi \mathcal{U} \psi$ to $label(s)$.
- if $\varphi \mathcal{U} \psi \in CL(f)$ and $\varphi \in label(s)$ and there exists a state t, which is a successor of s where $\varphi \mathcal{U} \psi \in label(t)$, then add $\varphi \mathcal{U} \psi$ to $label(s)$.
- if $\varphi \mathcal{W} \psi \in CL(f)$ and $\psi \in label(s)$, then add $\varphi \mathcal{W} \psi$ to $label(s)$.
- if $\varphi \mathcal{W} \psi \in CL(f)$ and $\varphi \in label(s)$ and there exists a state t, which is a successor of s where $\varphi \mathcal{U} \psi \in label(t)$, then add $\varphi \mathcal{W} \psi$ to $label(s)$.
- if $\varphi \mathcal{W} \psi \in CL(f)$ and $\varphi \in label(s)$ and there are no successors of s, then add $\varphi \mathcal{W} \psi$ to $label(s)$.

Thus, after n passes over the state machine, each state is labelled with a subset of $CL(f)$ representing the formulae satisfied in that state. To check whether a formula f is satisfied on the whole state-machine, we must simply check whether f is labelled in the appropriate initial state of the state-machine.

Though, the above is a generalisation of the approach it conforms to the standard algorithms for both branching time [CES83], and linear time [LP85] logics[3]. Differences obviously exist in the detail of the approach, for example, the checking of eventualities is carried out after the labelling in the Lichtenstein and Pnueli approach. Also, in linear time temporal model checking, we are checking whether a formula is satisfied on all paths, whereas, in the branching time case, the quantification over paths depends on the formula being checked.

The complexity of the above algorithm is generally $O(n \,.\, |S|)$, where n is the cardinality of the closure, $CL(f)$, and S is the set of states in the state-machine. For CTL [CES83] this is $O(|f| \,.\, |S|)$ and for a standard linear time temporal logic [LP85] it is $O(2^{|f|} \,.\, |S|)$. This difference reflects the relative strengths of the languages used. So, for the linear case, the complexity of the model checking algorithm stems from the labelling procedure.

Below we describe several problems with this approach.

1. *High Complexity*
   In [LP85], the upper bound for the size of the closure constructed for each formula to be checked is actually $5.\,|f|$. Thus, the overall complexity of the model checking algorithm is $O(2^{5.|f|} \,.\, |S|)$.

2. *The Checking Problem*
   This problem can be described as excess checking of states at times when it is not necessary. For example, even if a formula is unsatisfiable over a state-machine, the whole machine may need to be scanned several times before unsatisfiability can be shown. This problem can be split into two sub-problems that can be described as follows:

   (a) If the formula to be checked is satisfiable on all paths, the whole of the state-machine may need to be checked several times, especially if the formula represents some complex global property such as

   $$\Box \Diamond x \lor \Box \Diamond y$$

   However, in the simple case of checking $\bigcirc x$, where x is a proposition, the above method requires at least $|\ S\ |$ states to be checked where only s_n comparisons are strictly necessary (where s_n is the number of successors of the present state). Though both of these are constant for varying formulae, models are often quite large (see

[3]It should be noted that much of the paper by Lichtenstein and Pnueli is devoted to extending this simple algorithm to handle the checking of fairness properties.

Clarke et. al. [CG87]). For example, state-machines often contain more than 100 states while such states rarely have more than 10 successors. Thus, even for this simple example, an order of magnitude saving in time is possible. Obviously, though optimisation seems possible for such simple examples, any method taking advantage of this may turn out to be very inefficient for more complex examples. This question of balancing efficiency will be discussed in section 4.

(b) Another problem with the standard algorithm is that even when the temporal formula checked over the state machine is unsatisfied, the whole machine must be checked (and built—see 3 below).

In the case where a counter-example is often easily found, such as $\Box p$, the whole state-machine must be constructed, even if p is false on some path[4].

3. *The Space Problem*
Using the algorithm described above, the whole state-machine must be constructed every time a formula is checked. In real applications of model checking e.g., SMG [GB88], MCB [Bro86], and Hardware verification [BCDM84], the space consumed by the checking procedure during model checking is important. In fact, research is under way [CG87, Ban87] to reduce the size of the state machines that are checked at any one time.

Thus, for some specific problems, the 'standard' model checking algorithm is time-inefficient and for many other problems it is very space-inefficient. This leads us to describe some of the possible solutions to defects in the standard algorithm.

3.1 Solutions

To ease some of the problems described above, several strategies have been proposed:

- For the algorithm described in [LP85], it is possible to reduce the size of the closure constructed, and thus reduce the size of the exponent in the complexity measure. Such an algorithm has been described by Barringer et. al. [BFG89]. This uses a minimal form of closure developed by Gough [Gou84].
- To reduce the size of the state-machine checked at any one time, compositional and compression techniques for state-machines have been developed. (See, for example Clarke et. al. [CGB86, CG87].)
- Rather than checking a truly complete state-machine, 'degrees of freedom' such as input values can be left undefined. This reduces the number of states considerably, but obviously requires a more complex checking algorithm [Bro86].
- The algorithm described above represents a breadth-first global labelling approach. In section 4 we describe an algorithm that uses a depth-first search in order to to find, minimally, a single satisfying path through the state machine. It is also amenable to use in a 'rule-based' model checker, which minimises the space used (see section 6).

In the next section, we will describe our approach and the implementation of a practical model-checker based on this approach. A practical approach to the problem of checking very large (so large that the whole state machine cannot be held in memory) state machines will be described in section 6.

Note that automata-theoretic approaches to model checking have also been considered. Here, the temporal formula is translated into an automaton and an emptiness check is carried out on the product of the two automata (the other being the state-machine itself) [VW86].

[4]Obviously, if the formula is true, the whole state-machine may still have to be searched.

4 An Alternative Strategy

In this section we describe a practical algorithm which combines solutions to some of the problems described above. In special cases, especially when liveness properties or unsatisfiable formulae are checked, substantial time and space can be saved over the standard algorithm.

The main advantage of this method is that it can be used in cases where the whole state-machine cannot be held in memory at once and where the checking procedure does not require the complete model to be able to test satisfiability. Below we describe two aspects of the model checking strategy used. The first is the actual model checking algorithm, and as described it is designed to check a formula over an already constructed state-machine. The algorithm described is a recursive one (but not a full graph marking algorithm like the one described earlier) and in certain cases it solves the problems described in section 3, and reduces the number of states checked, thus alleviating some of the space problems.

The second part of this strategy consists of the description of a refinement of the previous algorithm to deal with state-machine generation rules rather than complete state-machines. Though this second part is particularly designed to work with SMG [GB88], it can, in principle, be adapted to any system which constructs state-machines via the use of generation rules. This extension is described, briefly, in section 6.

4.1 An Overview of the Algorithm

When we check the satisfiability of a linear time temporal formula over a set of temporal models represented by a finite state-machine, we actually check if the formula is satisfiable on all paths through the state-machine (i.e., on all linear models). Our strategy is to check if any one path through the state-machine satisfies the negation of the formula to be checked.

Given a formula f that we intend to check over the state-machine, and an initial state s_0 where we start checking from, we do the following:

1. negate the formula, giving $\neg f$;
2. push all negations in this formula down to the atoms (i.e., propositions, true and false) and simplify giving a new formula g;
3. check if a path that satisfies g exists through the state-machine starting at the initial state s_0.

Thus, we are effectively trying to (constructively) solve

$$\exists \mathrm{path}_{s_0}.\ \mathrm{path}_{s_0} \models g$$

The algorithm we use distinguishes different forms of g as follows:

- g is a proposition, true or false:
  simply check the satisfiability of g in the current state.
- g is negation, such as $\neg a$:
  as all negations have been pushed to the atoms of the formula and it has been simplified, a is guaranteed to be a proposition. Thus a can be checked in the current state and g can be satisfied only if a is not.
- g is a disjunction, such as $a \vee b$:
  as we are attempting to show

$$\exists \mathrm{path}_{s_0}.\ \mathrm{path}_{s_0} \models a \vee b$$

and this is equivalent to

$$\exists \text{path}_{s_0}.\ \text{path}_{s_0} \models a \quad \text{or} \quad \exists \text{path}_{s_0}.\ \text{path}_{s_0} \models b$$

we simply check each disjunct in turn.

- g is a conjunction, such as $a \wedge b$:
  unfortunately, $\exists \text{path}_{s_0}.\ \text{path}_{s_0} \models a \wedge b$ is not equivalent to

  $$\exists \text{path}_{s_0}.\ \text{path}_{s_0} \models a \quad \text{and} \quad \exists \text{path}_{s_0}.\ \text{path}_{s_0} \models b$$

  So, in checking $a \wedge b$, we must ensure that a and b are both satisfied along the same path. The detailed explanation of how this is achieved is given in Appendix A. Simply, both a and b are decomposed (checking those properties that can be checked in the current state along the way) until they are of the form $\bigcirc a'$ and $\bigcirc b'$ respectively[5]. Then we 'synchronise' the move to the next state with all the conjuncts, thus effectively checking the satisfiability of $\bigcirc(a' \wedge b')$.

- g is a next-time formula such as $\bigcirc x$ or $\odot x$:
  here, we check for x being satisfied in any of the successor states of s. If there are no successors, any formula of the form $\bigcirc x$ is satisfied while formulae of the form $\odot x$ are unsatisfied.

- g is an until formula[6], such as $a\mathcal{U}b$:
  we first check whether $a\mathcal{U}b$ has been checked previously in the same state and the commitment to make $a\mathcal{U}b$ true passed on. If so, $a\mathcal{U}b$ is unsatisfied. Otherwise, the fact that we have checked $a\mathcal{U}b$ in the current state is recorded and $a\mathcal{U}b$ is satisfiable if and only if $b \vee (a \wedge \odot a\mathcal{U}b)$ is satisfiable.

- g is an unless formula, such as $a\mathcal{W}b$:
  a similar marking scheme is used in the case of unless formulae as was described above for until formulae, the only difference being that if we find $a\mathcal{W}b$ has already been checked in the current state, then $a\mathcal{W}b$ is satisfiable. Otherwise, the state is marked and $a\mathcal{W}b$ is satisfiable if and only if $b \vee (a \wedge \bigcirc a\mathcal{W}b)$ is satisfiable.

  Note that the marking carried out by the $\mathcal{U}$ and $\mathcal{W}$ cases is eventually removed from the state-machine.

4.2 The Implementation

Having described the basic strategy we use in model checking, we now give a brief description of relevant aspects of the implementation.

4.2.1 State-Machine representation

The state-machine is modelled by a set of states together with a set of identified initial states. Each state consists of the following:

- the propositions that are true in that particular state;
- some representation of the successor states of that particular state;
- the until and unless formulae that are 'possibly' true in that state.

[5]Obviously, if either a or b have no temporal constituents, they will be completely decomposed simply by checking properties in the current state. Also, they may be decomposed to strong next formulae such as $\odot a'$ and $\odot b'$.

[6]During the simplification phase, formulae of the form $\lozenge x$ are translated to $\textbf{true}\,\mathcal{U}\,x$ and $\square x$ is translated to $x\mathcal{W}\textbf{false}$.

4.2.2 Formula negation and simplification

When a temporal formula is input to be checked, it is negated and all the negations in the formula are pushed to the atoms (i.e., the leaves of the formula if it is represented as a tree). The formula is then simplified using the following simplification rules:

original formula	simplified to
$\neg\mathbf{true}$	$\mathbf{false}$
$\neg\mathbf{false}$	$\mathbf{true}$
$\Diamond a$	$\mathbf{true}\,\mathcal{U}\,a$
$\Box a$	$a\mathcal{W}\mathbf{false}$
$\bigcirc\,\mathbf{true}$	$\mathbf{true}$
$\odot\,\mathbf{false}$	$\mathbf{false}$
$a \wedge \mathbf{false}$	$\mathbf{false}$
$a \wedge \mathbf{true}$	a
$a \vee \mathbf{false}$	a
$a \vee \mathbf{true}$	$\mathbf{true}$
$a \wedge a$	a
$a \vee a$	a
$a \wedge \neg a$	$\mathbf{false}$
$a \vee \neg a$	$\mathbf{true}$
$\mathbf{false}\,\mathcal{U}\,p$	p
$a\,\mathcal{U}\,\mathbf{true}$	$\mathbf{true}$
$a\,\mathcal{U}\,\mathbf{false}$	$\mathbf{false}$
$p\,\mathcal{U}\,p$	p
$\mathbf{false}\mathcal{W}p$	p
$a\mathcal{W}\mathbf{true}$	$\mathbf{true}$
$\mathbf{true}\mathcal{W}p$	$\mathbf{true}$
$p\mathcal{W}p$	p

4.2.3 Basic checking function

The function used to model-check a temporal formula is called *check*. This returns true if a path is found through the state machine that satisfies the formula it is given (i.e., the negation of the original formula to be checked). Thus the original formula is satisfied on all paths through the state machine if *check* returns false when applied to its negation.

The *check* function has the following signature (here TL is the set of temporal formulae, $\mathcal{P}$ is the powerset function, and FSM is the set of finite state machines):

$$check\colon\ \mathrm{TL} \times \mathcal{P}(\mathrm{TL}) \times \mathcal{P}(\mathrm{TL}) \times \mathcal{P}(\mathrm{TL}) \times \mathbb{B} \times \mathrm{state} \times \mathrm{FSM} \to \mathbb{B}$$

The first argument is the temporal formula to be checked. The second argument is the set of temporal formulae representing the conjuncts yet to be processed. This is needed to implement the particular way of dealing with the checking of formulae such as $a \wedge b$. The third argument is the set of formulae to be checked in all successive states. The set of formulae represented by the fourth argument is used to hold the set of 'possible' eventualities described above. These are accumulated when until or unless formulae are processed and are added to the possible set in the current state only when all checking in the current state has finished. (Note that the old possible set from the state is saved so that it can be replaced as the function calls unwind.) The fifth argument is a flag used to signify is at least one successor is necessary for satisfaction.

Finally, the sixth argument represents the current state and the seventh argument represents the whole state machine.

We give a VDM definition of this function along with its associated data types and auxiliary functions in Appendix A.

4.2.4 Speedups

Several practical steps have been taken to speed up the implementation. Briefly, these are:

1. When a disjunction or a conjunction is processed, the 'smallest' side is checked first, i.e., state-formulae (non-temporal formulae) are checked before temporal formulae. Note that this measure of 'smallest' is not simply derived from the length of the formula. The nesting and structure of the various temporal operators in the formula is also considered.

2. As the state-machines generated automatically are often cyclic, the same paths may need to be checked several times. In order to avoid this, another set has been added to each state in the state-machine, containing the temporal formulae that are unsatisfiable on all future paths from that state. Thus, once it has been established that, for a given state, all future paths do not satisfy a given formula, this formula is added to the 'unsatisfiable' set in that state. Thus, the same formula need never be checked again in such a state.

   This may seem like a slow process, but it speeds up the implementation considerably. This is because when the model checker is slow, it is usually due to the state-machine being large and cyclic or the formula being (temporally) complex. In either of these cases, the cost of adding formulae to a set and doing an inclusion test at every state is minimal compared to checking a complex (looping) formula over such a structure.

3. One other, more pragmatic, speedup is that the actual implementation of temporal formulae is efficient, and the whole model checker is written in an efficient high-level language (C++ [Str86]). The object-oriented nature of the language also enables the model checking algorithm to be given succinctly.

5 Correctness and Complexity of the Algorithm

We now consider the more theoretical analysis of the model checking algorithm, in particular its correctness and complexity. An overview of these results is given below. We also consider the effects of the speedups described in the previous section on these results.

5.1 Correctness

We need to establish the soundness and completeness of the model checking algorithm with respect to the temporal logic used. Thus, we need to show that for a state-machine, *FSM*, an initial state in this structure, s_0, and a formula, φ,

$$\exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s_0 \rangle \models \varphi \quad \text{iff} \quad \mathit{check}(\varphi, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s_0, FSM)$$

The proof of this result is given as theorem 1 in Appendix B.

5.2 Complexity

Sistla and Clarke [SC85] give the general result that model checking for the particular type of temporal logic used here is PSPACE-complete. For our basic algorithm, we can see that the worst case occurs when the same formula must be checked on every possible path through the state-machine. As this would also occur for every subformula of the original formula, we arrive at a complexity of $O(2^{x.|f|}.(|\ S\ |!))$. Thus, in the worst case, the complexity of our basic algorithm is higher than that of the standard one, which Lichtenstein and Pnueli [LP85] give as $O(2^{5.|f|}.\ |\ S\ |)$. However, we will show below that the speedups proposed modify the complexity of our algorithm so that it is in line with that of the standard algorithm. Also, for a significant number of special cases, our algorithm is faster than the standard one:

- When the formula being checked is disjunctive, the standard algorithm must, effectively, check each of the disjuncts, while our algorithm only continues checking until a satisfiable disjunct is found.

  A trivial example of this occurs when checking the formula

  $$a \vee b \vee c \vee \ldots\ldots \vee y \vee z$$

  The standard algorithm takes at least 26 steps to decide if the formula is satisfiable; in the worst case, our algorithm would take a similar amount of time. However, in the best case our algorithm would take just one step and on average it takes 13 steps.

- When the state-machine being checked contains a large branching factor, the standard algorithm must explore all the branches — again, our algorithm will only explore as many as it needs to in order to find a path satisfying the formula.

- When the formula being checked can be satisfied by several paths through the state-machine, our algorithm will terminate when it finds just one of these paths, whereas the standard algorithm will explore all the states in the state-machine.

Thus, though the situations where improved performance is possible are difficult to characterise, when such situations do occur, our algorithm performs significantly better than the standard one. However, as we have shown, in the worst case our basic algorithm has a worse performance than the standard algorithm. We will now show that one of the speedups proposed in section 4.2.4 ensures that the worst cases for both algorithms are similar.

5.3 Effect of Speedups

We also need to consider the effects that the speedups described in section 4.2.4 have on both the correctness and the complexity of the model checking algorithm. Ignoring the speedup described in 3 above, which is a pragmatic consideration, we can easily see that the other suggested improvements do not affect the correctness of the algorithm. In 1 above, we simply use a more intelligent choice function when choosing which side of a conjunct or disjunct to process first. In 2 above, a state is only labelled with a formula when a check of the satisfiability of that formula has been attempted from that particular state and false has been returned. This means that the formula has been found to be unsatisfiable on all linear paths from that state. Thus, we know that if we ever reach that state again and are checking the same formula then false can be returned immediately.

As to the effect of these speedups on the complexity of the algorithm, the speedups described in 1 and 2 both have a considerable effect. The more 'intelligent' choice of which conjunct to check first often speeds up the algorithm if the original formula is satisfiable, as one of the conjuncts may be found to fail more quickly than the other. The choice of disjunct can often speedup the algorithm when the original formula is unsatisfiable, as one of the disjuncts may be found to be satisfiable more quickly than the other. Obviously, these improvements are not generally applicable, but in many cases significant increases in efficiency can be made.

The labelling of states with formulae, as described in 2 above, though not significantly improving the performance of the algorithm for the special cases described in the last section, does reduce the worst case complexity to be in line with that of the standard algorithm. The labelling ensures that each formula is only ever checked at most once in each state. Thus, the worst case complexity of the algorithm occurs when all the subformulae of the original formula must be checked in each state. This gives

$$O(sub(f).\ |S|)$$

as the upper bound, where *sub*(*f*) is the number of subformulae of the original formula (Lichtenstein and Pnueli [LP85] give an upper bound for *sub*(*f*) of $2^{5.|f|}$).

So, we have seen that this last speedup ensures that the worst case complexity of our algorithm is comparable with that of the standard algorithm. Also, the properties of our basic algorithm, together with the heuristics for choosing which paths to explore give our algorithm better performance in certain cases.

One final note is that, with the basic algorithm, the maximum number of states that are required at any one time is given by the length of the longest (acyclic) path in the state-machine. However, if the labelling speedup is used, this property of minimal space usage is lost as we now need to keep information about all the states that have been visited. Thus, this produces a mechanism which, through the non-uniform labelling of states, approaches the marking effect of the standard algorithm, without necessarily requiring a global labelling process.

6 Rule Based Model Checking

In an attempt to save space during the model checking procedure, the algorithm described above may be used to develop a 'rule-based' model checker. Such a model checker would be linked closely to some system, such as SMG [GB88], which can produce state-machine generation rules rather than complete state-machines. These rules represent type of transitions between states and are analogous to rewrite rules. The normal approach is to use these rules, along with various evaluation and update functions supplied by SMG to generate a complete state-machine. This corresponds to the instantiation of rewrite rules in a system with particular ground terms.

However, as the state generation rules (represented as a transition table) together with associated evaluation and update functions represent all the information required to construct the complete state-machine, we intend to construct each part of the state-machine *on demand*. Thus, a modified version of our model checker would check a temporal formula over a transition table and would only construct new states when, for example, the temporal formula requires that the next state be checked. If the basic model checking algorithm were modified in this way, and if states were 'deleted' after they had been checked, then the maximum number of states required at any one time would be equal to the length of the maximum acyclic path in the state-machine.

This would give a potentially large saving in the space required by the algorithm and would be useful for model checking very large state-machines.

One final point to note about this proposed extension is that a thorough destruction of 'checked' states may not be possible in practice. For example, deleting states once they have been checked would invalidate the optimisation described above where states are labelled by 'unsatisfiable' sets. Clearly, some compromise between these two approaches would be useful.

7 Fairness

When the model checker is used in tandem with SMG, some form of fairness needs to be considered. At present the model checker checks all paths through the state-machine, making no distinction between fair and unfair paths. To make this distinction, extra information needs to be placed in the state-machine when it is constructed and the model-checker needs to be able to recognise and deal with various forms of fairness. Such support is available already in MCB [Bro86], a model checker for branching temporal logic. A more transparent approach is described by Barringer et. al. [BFG89] and it is intended that such a scheme will eventually be added to the model checker described in this report.

8 Conclusions

The algorithm described in this paper, though not significantly faster than standard method in the worst case, has the possibility of large savings in speed and space consumed over the standard algorithm in specific cases. One of the further advantages of such a depth first search is that if the model checker returns false, we automatically get a linear model, extracted from the state-machine, on which the formula is unsatisfiable. This helps in the debugging of systems as it is useful to have an example showing why the formula is unsatisfiable. Also, such a model is possibly useful for debugging the original specification.

The model checker has been implemented and has been tested on a variety of examples. Initially, its speed was found to be acceptable.

Work on the above extensions is under way. In particular, the modification of the model-checker to use state generation rules is being undertaken at present.

Work is also under way to try to find a more appropriate name for the model checker than `mdf`, which it is called for historical reasons.

9 Acknowledgements

Finally, I would like to thank both Howard Barringer and Graham Gough for their help and comments, and for finding some of the bugs in the initial implementation. I am also grateful to the two anonymous referees for their helpful comments and to Zhou Chaochen for his patience.

A A more formal description of the model checking function

The data types used in the specification of the model checking function are described below. We assume that *TL-Formula* is the set of well-formed temporal formulas of the temporal logic being used, *TL-Proposition* is the set of atomic propositions for this logic, and *State-Id* is a set of state identifiers.

$$
\begin{array}{lcrcl}
\mathit{State} & :: & \mathit{true\text{-}props} & : & \text{set of } \mathit{TL\text{-}Proposition} \\
 & & \mathit{successors} & : & \text{set of } \mathit{State\text{-}Id} \\
 & & \mathit{possibles} & : & \text{set of } \mathit{TL\text{-}Formula}
\end{array}
$$

$$
\begin{array}{lcrcl}
\mathit{FSM} & :: & \mathit{states} & : & \text{map } \mathit{State\text{-}Id} \text{ to } \mathit{State} \\
 & & \mathit{initial} & : & \text{set of } \mathit{State\text{-}Id}
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{inv\text{-}FSM}(\mathit{mk\text{-}FSM}(\mathit{state\text{-}map}, s_0)) \quad \triangleq \quad s_0 \in \text{dom}\, \mathit{state\text{-}map} \wedge \\
\quad (\forall s_1, s_2 \in \text{dom}\, \mathit{state\text{-}map} \cdot s_1 \neq s_2 \Rightarrow \mathit{state\text{-}map}(s_1) \neq \mathit{state\text{-}map}(s_2)) \wedge \\
\quad (\bigcup\{s(\mathit{successors}) \mid s \in \text{rng}\, \mathit{state\text{-}map}\}) \subseteq \text{dom}\, \mathit{state\text{-}map}
\end{array}
$$

The specification of the *check* function is as follows:

$$
\begin{array}{l}
\mathit{check} : \mathit{TL\text{-}Formula} \times \text{set of } \mathit{TL\text{-}Formula} \times \text{set of } \mathit{TL\text{-}Formula} \times \text{set of } \mathit{TL\text{-}Formula} \times \mathbb{B} \times \mathit{State\text{-}Id} \times \mathit{FSM} \\
\qquad \to \mathbb{B}
\end{array}
$$

$$
\begin{array}{l}
\mathit{check}(f, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \quad \triangleq \\
\quad \text{cases } f \text{ of} \\
\quad\quad \begin{array}{rcl}
\mathit{mk\text{-}TL\text{-}formula}(\texttt{true}) & \to & \mathit{check}(\text{NILFORM}, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{false}) & \to & \text{false} \\
\mathit{mk\text{-}TL\text{-}proposition}(p) & \to & \text{if } \mathit{is\text{-}true}(p, \mathit{current}, \mathit{fsm}) \\
 & & \text{then } \mathit{check}(\text{NILFORM}, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
 & & \text{else false} \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{not}, a) & \to & \text{if } \mathit{is\text{-}true}(a, \mathit{current}, \mathit{fsm}) \\
 & & \text{then false} \\
 & & \text{else } \mathit{check}(\text{NILFORM}, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{or}, a, b) & \to & \text{if } \mathit{check}(a, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
 & & \text{then true} \\
 & & \text{else } \mathit{check}(b, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{and}, a, b) & \to & \mathit{check}(a, \mathit{conjs} \cup \{b\}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{next}, a) & \to & \mathit{check}(\text{NILFORM}, \mathit{conjs}, \mathit{nexts} \cup \{a\}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{snext}, a) & \to & \mathit{check}(\text{NILFORM}, \mathit{conjs}, \mathit{nexts} \cup \{a\}, \mathit{poss\text{-}set}, \text{true}, \mathit{current}, \mathit{fsm}) \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{until}, a, b) & \to & \text{if } \mathit{is\text{-}possible}(f, \mathit{current}, \mathit{fsm}) \\
 & & \text{then false} \\
 & & \text{else if } \mathit{check}(b, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
 & & \quad \text{then true} \\
 & & \quad \text{else } \mathit{check}(a, \mathit{conjs}, \mathit{nexts} \cup \{f\}, \mathit{poss\text{-}set} \cup \{f\}, \text{true}, \mathit{current}, \mathit{fsm}) \\
\mathit{mk\text{-}TL\text{-}formula}(\texttt{unless}, a, b) & \to & \text{if } \mathit{is\text{-}possible}(f, \mathit{current}, \mathit{fsm}) \\
 & & \text{then } \mathit{check}(\text{NILFORM}, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
 & & \text{else if } \mathit{check}(b, \mathit{conjs}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
 & & \quad \text{then true} \\
 & & \quad \text{else } \mathit{check}(a, \mathit{conjs}, \mathit{nexts} \cup \{f\}, \mathit{poss\text{-}set} \cup \{f\}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
\text{NILFORM} & \to & \text{if } \mathit{conjs} \neq \{\,\} \\
 & & \text{then let } c = \mathit{choose\text{-}from}(\mathit{conjs}) \text{ in} \\
 & & \quad \mathit{check}(c, \mathit{conjs} - \{c\}, \mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm}) \\
 & & \text{else if } \mathit{nexts} = \{\,\} \\
 & & \quad \text{then true} \\
 & & \quad \text{else } \mathit{check\text{-}nil}(\mathit{nexts}, \mathit{poss\text{-}set}, \mathit{strong}, \mathit{current}, \mathit{fsm})
\end{array} \\
\quad \text{otherwise } \mathit{error}() \\
\quad \text{end}
\end{array}
$$

$check\text{-}nil : \text{set of } TL\text{-}Formula \times \text{set of } TL\text{-}Formula \times \mathbb{B} \times State\text{-}Id \times FSM \to \mathbb{B}$

$check\text{-}nil(nexts, poss\text{-}set, strong, current, fsm) \quad \triangleq$
if $get\text{-}successors(current, FSM) = \{\,\}$
then $\neg strong$
else let $new\text{-}fsm = add\text{-}possibles(poss\text{-}set, current, fsm)$ in
$\exists succ \in get\text{-}successors(current, fsm) \cdot$
$check(\textsc{Nilform}, nexts, \{\,\}, \{\,\}, \text{false}, succ, new\text{-}fsm)$

$choose\text{-}from : \text{set of } TL\text{-}formula \to TL\text{-}formula$

$choose\text{-}from(aset) \quad \triangleq$
let $choice \in aset$ in
$choice$

$is\text{-}true : TL\text{-}proposition \times State\text{-}Id \times FSM \to \mathbb{B}$

$is\text{-}true(p, current, fsm) \quad \triangleq$ let $state\text{-}map = fsm(states)$ in
let $current\text{-}state = state\text{-}map(current)$ in
let $true\text{-}set = current\text{-}state(true\text{-}props)$ in
$p \in true\text{-}set$

$is\text{-}possible : TL\text{-}formula \times State\text{-}Id \times FSM \to \mathbb{B}$

$is\text{-}possible(f, current, fsm) \quad \triangleq$ Returns TRUE if f is in *possible-set* for current state

$add\text{-}possibles : \text{set of } TL\text{-}formula \times State\text{-}Id \times FSM \to FSM$

$add\text{-}possibles(fset, current, fsm) \quad \triangleq$ Adds *fset* to *possible-set* for current state, returning updated *FSM*

$get\text{-}successors : State\text{-}Id \times FSM \to \text{set of } State\text{-}Id$

$get\text{-}successors(current, FSM) \quad \triangleq$ let $state\text{-}map = fsm(states)$ in
let $current\text{-}state = state\text{-}map(current)$ in
$current\text{-}state(successors)$

To give an idea of some of the subtleties of the *check* function, consider first the part dealing with formulas of the form $\neg a$. Ideally, we might like the *mk-TL-formula(*`"not", a)` case to simply be

$$\neg\ check(a\ , conjs, nexts, poss\text{-}set, strong, current, fsm)$$

Unfortunately, because *conjs* and *nexts* represent sets of formulas that must be true, the above rules would be incorrect. For example, try checking $(\neg a) \wedge a$ where a is false in the current state and the $\neg a$ side of the conjunction is evaluated first. When the conjunction is evaluated originally, a is added to *conjs* and $\neg a$ is checked. In the case of the rule above, we would check a, find it was false, negate this and return true from the whole check. In the original rule, a would still have to be checked. The failure of this simplification is due (mainly) to the fact that the *check* function returns immediately if false is found, without evaluating any more elements from *conjs* or *nexts*.

We should also note that the mechanism for adding formulas to the *possible-set* and checking if a given formula is in a *possible-set* for a particular state is more complex than is implied above. The reason for this is as follows. During model-checking, if we are checking $a\,\mathcal{U} b$ and we reach a state where $a\,\mathcal{U} b \in poss\text{-}set$, then we have two possibilities:

1. the same $a\mathcal{U}b$ caused the $a\mathcal{U}b$ to be put in the *possible-set* originally (i.e., earlier on in the checking);

2. the $a\mathcal{U}b$ being checked now is not the same one as the $a\mathcal{U}b$ in *possible-set*. For example, consider the checking of the formula $a\mathcal{U}b \wedge \bigcirc\bigcirc a\mathcal{U}b$ on a state-machine of two states, s_0 and s_1. These states are related by $s_0\mathcal{N}_F s_1$ and $s_1\mathcal{N}_F s_0$, and s_0 is the initial state of the state machine. Also $\pi(s_0) = \{a\}$ and $\pi(s_1) = \{b\}$.

   If we begin to check $a\mathcal{U}b \wedge \bigcirc\bigcirc a\mathcal{U}b$ at s_0, then $a\mathcal{U}b$ is added to the *possible-set* of s_0. In s_1 the original $a\mathcal{U}b$ is now satisfied, but we also need to check $\bigcirc a\mathcal{U}b$. Thus, we come round to s_0 again, checking $a\mathcal{U}b$. However, this is a different $a\mathcal{U}b$ to the one in the *possible-set* of s_0. If we did not recognise this distinction, then we would assume that $a\mathcal{U}b$ is unsatisfiable on the loop and return false.

We obviously need some mechanism to recognise that the eventuality we are processing is the same as the one found in the current *possible-set* (or is an ancestor of that formula). In practice this is carried out by the tagging of $\mathcal{U}$ and $\mathcal{W}$ formulas to enable such checks to be made[7]. Rather than complicate the specification of the *is-possible* and *add-possibles* functions above, we use those definitions, with the proviso that these functions have the 'correct' behaviour. In fact, we encapsulate these assumptions in Lemma 1 given in Appendix B. If this lemma if true, then the eventuality checking works as required.

[7] An alternative solution is to carry around, with the *check* function, a set of temporal formulas representing those eventualities that have been satisfied so far on the current path.

B Correctness of the Algorithm

The following lemma needs to be true in order that the possibility-checking has the desired effect.

Lemma 1 *If we are checking f, where $f = a\mathcal{U}b$ or $f = a\mathcal{W}b$, and is-possible(f, s_i, FSM) returns* TRUE, *then*[8]

$$\exists \sigma_n \in FSM.\ \sigma_n(0) = s_i \ \wedge\ \sigma_n(n) = s_i \ \wedge\ \sigma_n(0) \ldots \sigma_n(n) \models \Box(a \wedge \neg b)$$

Proof (outline) Since we know that if *is-possible* returns TRUE for f in state s_i, then f has not been found to be false, and the eventuality associated with f has not been satisfied. Thus, because of the first assumption, a must be satisfied throughout the loop, and because $\Diamond b$ has not occurred, b must be false throughout the loop. Consequently $\Box(a \wedge \neg b)$ is satisfied on the loop.

It follows from Lemma 1 that if we are checking $a\mathcal{U}b$ and *is-possible*($a\mathcal{U}b, s_i, FSM$) returns true, then $a\mathcal{U}b$ is unsatisfiable on the path we are exploring. Similarly, if we are checking $a\mathcal{W}b$ and *is-possible*($a\mathcal{W}b, s_i, FSM$) is true, then $a\mathcal{W}b$ is satisfied on the path we have found.

The following lemma is needed for the full proof of the correctness of the model checking algorithm (theorem 1).

Lemma 2 *If for any non-conjunct a,*

$$check(a, \{\,\}, \{\,\}, \{\,\}, \text{FALSE}, s, FSM) \quad \text{iff} \quad \exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s\rangle \models a$$

then

$$check(\varphi, \{\,\}, \{\,\}, \{\,\}, \text{FALSE}, s, FSM) \quad \text{iff} \quad \exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s\rangle \models \varphi$$

where, for some C such that $|C| > 0$,

$$\varphi = \bigwedge_{c \in C} c$$

and each c is not a conjunct or a disjunct.

Proof
We prove this result by induction on the size of the set C.

Bases Case

(1) $|C| = 1$
Let $C = \{c\}$, then by the assumption, it follows that

$$check(c, \{\,\}, \{\,\}, \{\,\}, \text{FALSE}, s, FSM) \quad \text{iff} \quad \exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s\rangle \models c$$

[8]Extending satisfaction to sequences in the obvious way.

Induction Cases We assume that

$$check(\varphi, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, FSM) \quad \text{iff} \quad \exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s\rangle \models \varphi$$

where, for some C such that $|C| = n$,

$$\varphi = \bigwedge_{c \in C} c$$

and each c is not a conjunct or a disjunct.
We construct a set C' such that $C' = C \cup \{c'\}$ where c' is not a conjunct or a disjunct, and attempt to prove the result for φ', where

$$\varphi' = \bigwedge_{c \in C'} c$$

We do case analysis on the formula c'

(2) c' is a literal, i.e., **true**, **false**, a proposition or the negation of a proposition.
To check φ', we call

$$check(c', C, \{\,\}, \{\,\}, \textsc{False}, s, FSM)$$

As c' is a literal it is easy to show that this is true if and only if

$$\exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s\rangle \models c' \wedge \bigwedge_{c \in C} c$$

(3) c' is a next-time formula
If there are any literals in the set C, then we can exchange any of these for c' and process them as above. As C cannot contain conjuncts or disjuncts, we are left with 2 possibilities for the contents of the set C.

(a) There is at least one next-time formula in C.
In this case, we chose a next-time formula, say l from C. We can combine l and c' to get a new formula l_{new} according to the following rules[9]

l	c'	l_{new}
$\bigcirc x$	$\bigcirc y$	$\bigcirc(x \wedge y)$
$\odot x$	$\bigcirc y$	$\odot(x \wedge y)$
$\bigcirc x$	$\odot y$	$\odot(x \wedge y)$
$\odot x$	$\odot y$	$\odot(x \wedge y)$

Thus, we can create a new set C'' such that $C'' = (C\text{-}\{l\}) \cup \{l_{new}\}$. As $|C''| = n$ then by the induction hypothesis, the result is proved for this case.

(b) There are only $\mathcal{U}$ and $\mathcal{W}$ formulas in C.
In this case, we can exchange one of these $\mathcal{U}$ or $\mathcal{W}$ formulas with c' and process this case in (4) below.

(4) c' is an $\mathcal{U}$ or an $\mathcal{W}$ formula and C contains only $\mathcal{U}$ or $\mathcal{W}$ formulae and at most one next-time formula.
(Outline proof:)
For all the $\mathcal{U}$ or $\mathcal{W}$ formulas, we check the appropriate *is-possible* in the current state. If this does not reduce the set C' or return false, then translate all $a\mathcal{U}b$ formulas to $b \vee (a \wedge$

[9] These transformations are allowable since the models we are interpreting these formulas over are linear [Fis89].

$\odot(a\mathcal{U}b)$) and all $a\mathcal{W}b$ formulas to $b \vee (a \wedge \bigcirc(a\mathcal{W}b))$. We then go through all the cases of a and b until we are left to check formulas of the form

$$\bigvee_i \bigwedge_j (\bigcirc \varphi_{ij1} \wedge \odot \varphi_{ij2})$$

We then check all the disjunctions separately. If each of these disjunctions contains a conjunction with less than $n+1$ conjuncts in, the result for this case is established. If any conjunction contains more than n conjuncts, then as these conjuncts are all next-time formulas, we use the rules above to merge them into one single next-time formula. This establishes the desired result as $1 < n+1$.

We now turn to the proof of correctness (soundness and completeness) of the model-checking algorithm.

Theorem 1

$$\forall \varphi \in \textit{TL-formula}. \left[\begin{array}{c} \textit{check}(\varphi, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, \textit{FSM}) \\ \text{iff} \\ \exists \mathcal{M} \in \textit{FSM}.\ \langle \mathcal{M}, s \rangle \models \varphi \end{array} \right]$$

Proof

In order to make the proof simpler, we assume that φ is in temporal disjunctive normal form, i.e.,

$$\bigvee_i \bigwedge_j \varphi_{ij}$$

where each φ_{ij} is a temporal formula that is not a conjunct or a disjunct. Note that any temporal formula can easily be rewritten to an equivalent formula of this form.

We prove the correctness theorem by induction on the structure of φ.

Base Cases

(1) $\varphi = \textbf{true}$
check(**true**, $\{\,\}, \{\,\}, \{\,\}$, FALSE, s, *FSM*) is true
iff
$\exists \mathcal{M} \in \textit{FSM}.\ \langle \mathcal{M}, s \rangle \models$ **true** is true, simply by taking $\mathcal{M}$ to be $\langle S, \mathcal{N}, \pi \rangle$, where $S = \{s\}$

(2) $\varphi = \textbf{false}$
check(**false**, $\{\,\}, \{\,\}, \{\,\}$, FALSE, s, *FSM*) is false
iff
$\langle \mathcal{M}, s \rangle \models$ **false** is false, for any $\mathcal{M}$ in *FSM*.

(3) $\varphi \in$ PROP
check(φ, $\{\,\}, \{\,\}, \{\,\}$, FALSE, s, *FSM*) iff *is-true*(φ,*s*,*fsm*), i.e, if φ is true in state s
iff
$\langle \mathcal{M}, s \rangle \models \varphi$ is true iff $\varphi \in \pi(s)$, i.e., φ is true in state s.

(4) $\varphi = \neg a$
(N.B., because all negations have been pushed to the atoms, a must be a proposition)
check($\neg a$, $\{\,\}, \{\,\}, \{\,\}$, FALSE, s, *FSM*) iff *is-true*(*a*,*s*,*fsm*) is false, i.e, if a is false in state s
iff
$\langle \mathcal{M}, s \rangle \models \neg a$ is true iff $a \notin \pi(s)$, i.e., φ is false in state s.

Induction Cases We assume that for any temporal formula (in the correct form) $a, b, \ldots$etc, the following holds

$$check(a, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, FSM) \text{ is true} \quad \text{iff} \quad \exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s \rangle \models a$$

(5) $\varphi = a \vee b$
$check(a \vee b, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, FSM)$ iff

$$check(a, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, FSM)$$
or
$$check(b, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, FSM)$$

By the induction hypothesis, this is true if and only if

$$\exists \mathcal{M}_1 \in FSM.\ \langle \mathcal{M}_1, s \rangle \models a \ \textbf{or}\ \exists \mathcal{M}_2 \in FSM.\ \langle \mathcal{M}_2, s \rangle \models b$$

This is true iff $\exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s \rangle \models a \vee b$.

(6) $\varphi = a \wedge b$
As φ was in temporal disjunctive normal form, a and b cannot be conjunctions or disjunctions. Thus, by the induction hypotheses, we can use Lemma 2 to prove this case.

(7) $\varphi = \bigcirc a$
$check(\bigcirc a, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, FSM)$ iff

if $get\text{-}successors(s, FSM) = \{\,\}$
then true
else $\exists t \in get\text{-}successors(s, FSM).\ check(a, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, t, FSM)$ is true.

i.e., if s has no successors,
then true,
else there is a successor of s, t such that $check(a, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, t, FSM)$ is true.
iff

$$\exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s \rangle \models \bigcirc a \ \text{ iff } \ \exists \mathcal{M} \in FSM.\ \forall t \in S.\ \text{ if } \ s\mathcal{N}t \ \text{ then } \ \langle \mathcal{M}, t \rangle \models a$$

As $\mathcal{M}$ is a linear model, there is at most one successor of s, thus the two definitions are equivalent.

(8) $\varphi = \odot a$
The proof of this case is similar to the previous case, except for the fact that if there are no successors, $\odot a$ is unsatisfied. Thus,
$check(\odot a, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, s, FSM)$ iff
if s has no successors,
then false,
else there is a successor of s, t such that $check(a, \{\,\}, \{\,\}, \{\,\}, \textsc{False}, t, FSM)$ is true.
iff

$$\exists \mathcal{M} \in FSM.\ \langle \mathcal{M}, s \rangle \models \odot a \ \text{ iff } \ \exists \mathcal{M} \in FSM.\ \exists t \in S.\ s\mathcal{N}t \ \text{ and } \ \langle \mathcal{M}, t \rangle \models a$$

Again, the two definitions are equivalent as $\mathcal{M}$ is a linear model.

(9) $\varphi = a\mathcal{U}b$
(Outline Proof:)
If *check* succeeds, $\mathcal{M}$ is guaranteed to have a state where b occurs and a has been satisfied in all the states up to that one.

check($a\mathcal{U}b$, …) can not continue recursing forever because the finiteness of S means that eventually we will reach a state where we've already been and $a\mathcal{U}b$ has been marked as *possible*.

From Lemma 1 we can see that if we reach a state, t, where we are checking $a\mathcal{U}b$ and *is-possible*($a\mathcal{U}b, t, FSM$) returns true, then $a\mathcal{U}b$ is unsatisfiable on the the circular path from t to t.

(10) $\varphi = a\mathcal{W}b$
The proof of the $\mathcal{U}$ case is similar to that of the $\mathcal{U}$ case above. The main difference is that if we reach a state, t, where we are checking $a\mathcal{W}b$ and *is-possible*($a\mathcal{W}b, t, FSM$) returns true, then by Lemma 1, $a\mathcal{W}b$ is unsatisfiable on the the circular path starting and finishing at t.

References

[Ban87] Behnam Banieqbal.
The Problem of Verifying Systems with Many Processes.
Temple project report, Department of Computer Science, University of Manchester, 1987.

[BCDM84] M.C. Browne, E.M. Clarke, D. Dill, and B. Mishra.
Automatic Verification of Sequential Circuits using Temporal Logic.
Technical Report CS–85–100, Department of Computer Science, Carnegie Mellon University, 1984.

[BFG89] H. Barringer, M. Fisher, and G. Gough.
Fair SMG and Linear Time Model Checking.
In *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.
(Published in *Lecture Notes in Computer Science* volume 407).

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli.
A Really Abstract Concurrent Model and its Temporal Logic.
In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, St. Petersberg Beach, Florida, January 1986.

[Bro86] Michael C. Browne.
An Improved Algorithm for the Automatic Verification of Finite State Systems using Temporal Logic.
Technical report, Department of Computer Science, Carnegie Mellon University, December 1986.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla.
Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach.
In *Proceedings of the Tenth ACM Symposium on the Principles of Programming Languages*, Austin, Texas, 1983.

[CG87] E. M. Clarke and O. Grümberg.
Avoiding the State Explosion Problem in Temporal Logic Model Checking Algorithms.
Technical report, Department of Computer Science, Carnegie Mellon University, 1987.

[CGB86] E. M. Clarke, O. Grümberg, and M. C. Browne.
Reasoning about Networks with many identical Finite-State Processes.
Technical report, Department of Computer Science, Carnegie Mellon University, December 1986.

[EH82] E. A. Emerson and J. Y. Halpern.
Decision Procedures and Expressiveness in the Temporal Logic of Branching Time.
In *Proceedings of the Fourteenth ACM Symposium on the Theory of Computing*, pages 169–180, 1982.

[ES84] E. A. Emerson and A. P. Sistla.
Deciding full branching time logic.
Information and Control, 61:175–201, 1984.

[Fis89] Michael Fisher.

Characterising Temporal Logic.
Technical Report UMCS-89-10-6, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K., October 1989.

[FL79] M. J. Fischer and R. E. Ladner.
Propositional Dynamic Logic of Regular Programs.
Journal of Computer and System Sciences, 18(2):194–211, 1979.

[GB88] G. D. Gough and H. Barringer.
A Semantics Driven Temporal Verification System.
In *Proceedings of ESOP '88*, March 1988.

[Gou84] G. D. Gough.
Decision Procedures for Temporal Logic.
Master's thesis, Department of Computer Science, University of Manchester, October 1984.

[LP85] O. Lichtenstein and A. Pnueli.
Checking that Finite State Concurrent Programs Satisfy their Linear Specification.
In *Proceedings of the Twelfth ACM Symposium on the Principles of Programming Languages*, New Orleans, Louisiana, January 1985.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck.
The Glory of the Past.
Lecture Notes in Computer Science, 193:196–218, June 1985.

[Pnu77] A. Pnueli.
The Temporal Logic of Programs.
In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, Providence, November 1977.

[SC85] A. P. Sistla and E. M. Clarke.
Complexity of propositional linear temporal logics.
ACM Journal, 32(3):733–749, July 1985.

[Str86] Bjarne Stroustrup.
The C++ Programming Language.
Addison-Wesley, 1986.

[VW86] Moshe Y. Vardi and Pierre Wolper.
An Automata-Theoretic Approach to Automatic Program Verification.
Technical Report RJ 5184 (53488), IBM Almaden Research Center, San Jose, California, January 1986.