"Direct" Model Checking of Temporal Properties

R. Bharadwaj
Communications Research Laboratory
McMaster University,
Hamilton, ON, Canada L8S 4K1.
E-mail: ramesh@triose.crl.mcmaster.ca

Keywords: Verification, Model Checking, Linear-time Temporal Logic, On-the-fly Methods, Combining Model Checking and Theorem Proving, Assertional Reasoning.

Abstract

In this paper, we address the problem of model checking temporal properties of finite-state programs. This problem is usually solved by modelling the program as well as the *negation* of the desired temporal property as automata on infinite words (Büchi automata) and checking for emptiness of the automaton resulting from the synchronous product of the program automaton and the negated property automaton. To check properties expressed in Linear-time Temporal Logic (LTL) [MP91b] involves a construction which produces an automaton with states exponential in the size of the formula [Wol89]. This poses a problem especially when model checking under fairness assumptions — the formulae tend to be very long in these cases. In this paper, we present a method to *directly* check properties expressed in LTL for programs written in a UNITY-like notation. We do this by adapting well-known LTL proof rules to model checking. In our approach, there is no need to construct equivalent Büchi automata. Another advantage is our ability to handle fairness in a straightforward way. Our method combines model checking and theorem proving in a unified framework, and is compatible with on-the-fly techniques such as bit-state hashing.

1 Introduction

Model checking has emerged as a successful method to verify that a (finite-state) program satisfies its correctness requirements [CES86, Hol91, Kur89, McM93]. It is based on the idea of expressing a program's requirement as a formula in temporal logic, and viewing the program as a structure that may be interpreted as the formula's model [Wol89]. In this paper, we shall only concern ourselves with the problem of model checking formulae expressed in Linear-time Temporal Logic (LTL) [MP83, MP91b]. Other popular approaches use branching-time logics such as CTL (see for example [CES86, McM93] for details). The usual approach to model check an LTL formula is to construct a finite-automaton on infinite words (a Büchi automaton) for the *negation* of the formula, take a synchronous product of the automaton representing the behaviour of the program

and the Buchi automaton representing the negation of the formula, and check that the language accepted by the product automaton is empty [VW86, Wol89]. Emptiness is checked by determining whether the set of accepting states is reachable from the initial set of states, and belongs to a cycle [CVWY92, Hol91]. An advantage of this approach is that using the negation of the formula may result in a smaller state space to be explored. A popular way to implement this algorithm is to compute the program automaton and the synchronous product in the same step, obviating the need to store the whole state-space graph, an approach known as *on-the-fly (OTF)* model checking [CVWY92, Hol91].

A major disadvantage of this approach is that to model check a property expressed in LTL requires the construction of an equivalent Büchi automaton that has, in general, an exponential number of states in the length of the original formula [Wol89]. Smaller equivalent automata can be constructed by hand (no minimization procedures for these automata exist); this approach, however, is error-prone and is generally not recommended. A common work around is a pre-computed or "cookbook" approach for the translation. These automated or pre-computed approaches may seem appealing, but they do not work well in practice because verifiers are left with no intuitive understanding of the generated automata (or the ones from cookbooks) — interpreting counterexamples produced by the model checker therefore becomes very difficult.

Another disadvantage of this method is that fairness assumptions [Fra86, MP91b] cannot be handled very well. A plausible approach to handle fairness is to express the fairness assumption as a formula $\mathcal{F}$, and model check the formula $\mathcal{F} \Rightarrow q$, where q is the property to be checked. This approach is not very feasible because even if the original formula q expressing the desired property is short, the new formula will, in general, no longer be so. Other approaches that build fairness directly into the state exploration algorithms [Pel93] have the disadvantage that they generate inordinately long counterexamples, making them hard to interpret.

In this paper, we explore a new approach to model check temporal properties. Our method adapts well-known proof rules used in theorem proving LTL formulae [MP91a], to model checking. Termed *direct* model checking, this approach has several advantages:

- There is no need to construct (or use) Büchi automata for model checking properties expressed in LTL.
- It unifies model checking with theorem proving methods, vastly increasing the set of tools available to practicing verifiers. In fact, it permits a verification problem to be tackled using a mix of model checking and theorem proving tools.
- Fairness (both weak fairness as well as strong fairness) is handled in a straightforward way, just as in theorem proving approaches.
- The method is compatible with on-the-fly model checking and the bit-state hashing algorithm of [Hol88].

Our approach requires some intuition on the part of verifiers, to be able to come up with auxiliary invariants (and other predicates) which may be necessary for model checking. In the opinion of

this author, this is actually an advantage because use of this approach is close to a genuine proof i.e., a chain of argument which will convince a human reviewer; the pitfalls of relying on the mere grunt of assent from an oracle (in this case, the model checking program) are well-known [RvH91].

2 Preliminaries

In the following discussion, we associate a set V of *variables* with every program $\mathcal{P}$. Each variable may take on values over a (finite) domain $\mathbf{D}$. A *state* is an interpretation of V, assigning to each variable a value from domain $\mathbf{D}$.

2.1 Notation for Programs

Without loss of generality, we represent programs as (finite) sets of transitions, as in UNITY [CM88]. Programs with constructs such as processes, channels, sequential and parallel composition, and repetition can be mechanically translated to this notation (see for example [SdR94, Bha95]). A program $\mathcal{P}$ is a tuple $\langle \Theta, \mathcal{T} \rangle$, consisting of an *initial predicate* Θ, characterizing the set of *initial states*, and a (finite) set of *transitions* $\mathcal{T}$. Each transition consists of a quantifier-free predicate, or *guard*, the symbol "$\Leftrightarrow$", followed by a finite set of assignments (separated by commas), called the *body*. Operationally, a transition is said to be *enabled* in a state s if its guard evaluates to "true" in that state. If a transition is enabled in state s, its body may be *executed* in that state. Assignments in the body are executed as a single multiple assignment. For example, the body "x:=y, y:=x" exchanges the values of variables x and y.

A *behaviour* of program $\mathcal{P}$ is a *linear sequence* of states $s_0, s_1, \ldots$ such that the first state s_0 of the sequence is an initial state of $\mathcal{P}$ (i.e., it satisfies the initial predicate Θ), and for each consecutive pair of states $\langle s_i, s_{i+1} \rangle$ in the sequence, there exists a transition τ of $\mathcal{P}$ such that its guard is *true* in state s_i, and state s_{i+1} results from executing τ's body in state s_i. For our purpose, we assume all behaviours to be **infinite** sequences, without loss of generality; for, we may extend a finite sequence ending in state s_n into an infinite one by repeating s_n at the end of the sequence (infinitely many times).

2.2 Notation for Properties

We assume an underlying first-order assertion language $\mathcal{L}$, with interpreted symbols for expressing the standard operations and relations over domain $\mathbf{D}$. We shall refer to formulae in language $\mathcal{L}$ as *state formulae* or *assertions*. We shall denote the proposition "true" by $\mathbf{tt}$.

We construct *temporal formulae* out of state formulae by applying the logical operators $\neg$ (negation) and $\vee$ (disjunction), and the temporal operator $\mathcal{U}$ (until). Other logical operators such as $\Rightarrow$

(implication), $\Leftrightarrow$ (equivalence), and $\wedge$ (conjunction), and temporal operators such as $\Diamond$ (eventually) and $\Box$ (henceforth) can be defined in terms of these elementary operators.

A temporal formula p is interpreted over an infinite sequence of states $\sigma : s_0, s_1, \ldots$, known as a *structure*, where each state s_i is an interpretation for the variables in p. For a structure σ, the general notion of a temporal formula p *holding* at position $j \geq 0$ in σ is written as $(\sigma, j) \models p$. In the following, we provide an inductive definition of a temporal formula p holding for sequence σ at position $j = 0$[1]:

- If p is a state formula (i.e., has no temporal operators),

  $$(\sigma, 0) \models p \Leftrightarrow s_0 \models p$$

  we evaluate p using the interpretation given by state s_0.
- $(\sigma, 0) \models \neg p \Leftrightarrow (\sigma, 0) \not\models p$.
- $(\sigma, 0) \models p \vee q \Leftrightarrow (\sigma, 0) \models p$ or $(\sigma, 0) \models q$.
- $(\sigma, 0) \models p \; \mathcal{U} \; q \Leftrightarrow$ for some $k \geq 0$, $(\sigma, k) \models q$ and for all i such that $0 \leq i < k$, $(\sigma, i) \models p$.

In addition, we define two other temporal operators:

- $\Diamond p = \mathbf{tt} \; \mathcal{U} \; p$ (eventually)
- $\Box p = \neg \Diamond \neg p$ (henceforth).

2.3 Program Validity of LTL Formulae

The behaviour of a program $\mathcal{P}$, an infinite sequence of states, can serve as a structure over which a temporal formula p may be interpreted.

We define the validity of a temporal formula p for a program $\mathcal{P}$ by interpreting p over all behaviours of program $\mathcal{P}$. if p is valid for all behaviours of program $\mathcal{P}$, we say program $\mathcal{P}$ *satisfies* the temporal property p. We may check this by using proof rules for establishing temporal properties of programs. In this paper, we outline how these proof rules may be adapted for model checking, and propose an algorithm to accomplish this task.

We shall restrict ourselves to the following groups of properties[2]. These proof rules suffice to establish most temporal properties and are the main working tools for a practicing verifier [MP90].

[1] this is known as a *canonical* interpretation of formula p

[2] the proof system may be extended to a richer system, which establishes validity of any LTL formula; for details see [MP91a, MP91b]

One group of rules establishes the validity of the *invariance* formulae $\Box q$ and $\Box(p \Rightarrow \Box q)$; these formulae express the invariance of a state formula q, either throughout a behaviour, or starting from the state in a behaviour in which formula p holds.

Another group of rules establishes the validity of the *eventuality* formulae $\Diamond q$ and $\Box(p \Rightarrow \Diamond q)$; these formulae express the guarantee that q will eventually be true in a behaviour, either once, or following each state in which formula p holds.

These proof rules reduce the problem of establishing an LTL property for a program to the problem of establishing the validity of a (finite) set of state formulae. Some of these are directly expressed as (first-order) predicates. Others, known as *annotated transitions*, are written in terms of individual transitions of the program. They are of the form $\{pre\}\tau\{post\}$, where *pre* and *post* are predicates and τ is a transition. The interpretation of $\{pre\}\tau\{post\}$ is that whenever the guard of τ is true in a state in which *pre* holds, then *post* holds in the state resulting from the execution of the body of τ.

We may translate each annotated transition into a first-order predicate. We do this by using the notion of *weakest precondition* [Dij76]. Let B denote the body $x_1 := e_1, \cdots, x_n := e_n$. The weakest precondition $wp(B, post)$, where *post* is a predicate, is defined as $post[e_1/x_1, \cdots, e_n/x_n]$, where $post[e_1/x_1, \cdots, e_n/x_n]$ denotes the predicate resulting from the simultaneous substitution of e_i for x_i in *post* ($i = 1, \cdots, n$). An annotated transition $\{pre\}\tau\{post\}$, where $\tau = g \Leftrightarrow B$, is equivalent to the predicate $(pre \wedge g) \Rightarrow wp(B, post)$.

2.4 Notation for Annotated Programs

An *annotated program* $\mathcal{P}^A$ is a tuple $\langle \Theta^A, \mathcal{T}^A \rangle$. It consists of an *annotated initial predicate* Θ^A and a (finite) set of annotated transitions $\mathcal{T}^A$. $\Theta^A = \langle \Theta, Init \rangle$, where Θ and *Init* are predicates, is written as $\Theta\{Init\}$. Each annotated transition $\tau^A \in \mathcal{T}^A$ is a triple $\langle pre, \tau, post \rangle$ written as $\{pre\}\tau\{post\}$. Here, *pre* and *post* are predicates and τ is a transition of the form $g \Leftrightarrow B$ with guard g and body B. We define the enabledness of an annotated transition τ^A in terms of the enabledness of its component transition τ.

Operationally, we may define a behaviour of an annotated program just as we did for a program: a behaviour $\sigma = s_0, s_1, \ldots$ starts from a state s_0 satisfying predicate Θ of Θ^A, and for each pair of consecutive states s_i and s_{i+1} in σ, execution of the transition part of an enabled annotated transition in state s_i results in state s_{i+1}.

Additionally, each behaviour must satisfy the following conditions:

- Predicate *Init* must hold in state s_0. If not, the behaviour is said to have an *assertion violation* in state s_0. In this event, the singleton sequence s_0 is called a *counterexample*.
- If state s_{i+1} results from state s_i by executing the transition part of an enabled annotated transition τ^A, and the predicate part *pre* of τ^A holds in state s_i, then predicate part *post* of τ^A

must hold in state s_{i+1}. If not, such a behaviour is said to have an *assertion violation* in state s_i. In such a case, and additionally if there are no assertion violations in states $s_0, \ldots, s_{i-1}$, the sequence of states $s_0, \ldots s_i$ is called a *counterexample*.

3 The Model Checking Algorithm

The model checking problem we consider is as follows: given an annotated program $\mathcal{P}^A$ check for assertion violations, if any, and provide a counterexample for the first encountered violation. We call this process *assertion checking*. The algorithm, adapted form [HS82], performs a depth-first search (DFS) of the behaviours of program $\mathcal{P}^A$. In the following, the notation $Pred(s)$ where $Pred$ is a predicate and s is a state is used to denote the evaluation of predicate $Pred$ in state s (yielding a boolean result). We denote the cardinality of set $\mathcal{T}^A$ by $\mid \mathcal{T}^A \mid$. We use the notation τ_i^A to denote the $i^{\mathtt{th}}$ annotated transition of $\mathcal{T}^A$. The components of τ_i^A are respectively denoted by $\tau_i^A.pre$, τ_i, and $\tau_i^A.post$. Predicate $enabled(\tau_i^A, s)$ denotes the enabledness condition of transition τ_i^A in state s. Function $succ(\tau, s)$ returns the state resulting from the execution of transition τ in state s.

The main data structures used are the following: Stack S holds the sequence of states of a behaviour, starting from the initial state up to (and excluding) the *current state* (cs). Elements of stack S are tuples $\langle s, n \rangle$ where $s \in \mathbf{D}$ is a state and $n \in \mathbf{Nat}$ is a natural number denoting the first unexplored transition of state s. We denote the empty stack by `empty`; stack operations *push*, *pop*, and *top* have the usual meaning. Set H holds all states that have been visited during the DFS. Set H is usually implemented as a hash-table. Set operations $\cup$ and $\in$ have the usual meaning. We denote the null set by Φ.

$H := \Phi$; $S :=$ `empty`
for each s_0 such that $\Theta(s_0)$ and $s_0 \notin H$ **do**
  $cs := s_0$; $assert(Init(s_0))$; $H := H \cup \{s_0\}$; $push(S, \langle s_0, 1 \rangle)$;
  while $S \neq$ `empty` **do**
    $\langle cs, i \rangle := top(S)$; $pop(S)$
    while $i \leq \mid \mathcal{T}^A \mid$ **do**
      if $enabled(\tau_i^A, cs)$ **then**
        $ns := succ(\tau_i, cs)$
        if $\tau_i^A.pre(cs)$ **then** $assert(\tau_i^A.post(ns))$ **end**
        if $ns \notin H$ **then**
          $H := H \cup \{ns\}$; $push(S, \langle cs, i+1 \rangle)$;
          $cs := ns$; $i := 1$
        else $i := i + 1$ **end**
      else $i := i + 1$ **end**
    end
  end
end

The algorithm uses the following auxiliary procedure *assert*. In this procedure, assume that routine $PrintStates(S)$ prints out the state component of elements of stack S and routine $PrintState(s)$ prints out state s.

procedure $assert(Predicate\ \ p)$
  if $\neg p$ **then**
    `/* report assertion failure */`
    $PrintStates(S)$; $PrintState(cs)$; **exit**
  end
end

4 Adapting LTL Proof Rules to Model Checking

In this section, we outline how we may interpret LTL proof rules in the context of model checking. We have adapted these proof rules from [MP83, MP90, MP91a, MP91b]. As we mentioned in section 2.3, we shall restrict ourselves to proof rules for invariance and eventuality formulae, as our intent is to convey to the reader our central idea. It should be easy to extend our work to the general case of model checking any LTL formula (on the lines of [MP91a, MP91b]).

Using a proof rule, one may infer an LTL property for a program from the validity of a set of formulae known as its *premises*. In the classical *theorem proving* approach, these validities are established individually by providing proofs in a formal axiomatic theory. In our "direct" model checking approach, their validity is established in one of two ways: some of them are translated, using similar proof rules, into other model checking problems; alternately, validity of a set of formulae is collectively inferred by assertion checking an annotated program using the algorithm outlined in section 3.

In the following, we denote by $\{pre\}T\{post\}$, where *pre* and *post* are predicates, and $T \subseteq \mathcal{T}$ is a set of transitions, the condition of requiring $\{pre\}\tau\{post\}$ to hold for every $\tau \in T$.

4.1 Rules for Invariance

A proof rule to establish, for a program $\mathcal{P}$, the validity of the temporal formula $\Box q$, where q is a state formula, is as below.

INV	I1.	$\Theta \Rightarrow \phi$
	I2.	$\phi \Rightarrow q$
	I3.	$\{\phi\}\mathcal{T}\{\phi\}$
		$\Box q$

This rule uses an auxiliary predicate ϕ to establish the invariance of q. Assertion ϕ holds initially (by premise I1) and is propagated from each state to the next (by premises I3). Therefore, ϕ is an invariant of program $\mathcal{P}$, i.e., it holds continuously over all behaviours of $\mathcal{P}$. It follows from premise I2 that q is also an invariant of program $\mathcal{P}$.

A stronger auxiliary predicate ϕ may be required only when assertional validity is established individually for each premise. The property we want to establish, q, may be too *weak* to ensure validity of premises I3 when used directly. This will not be the case when model checking is used to collectively establish the premises of rule **INV**. For this approach, we may set ϕ to be the property q. In this case, premise I2 reduces to a tautaulogy and needs no further checking.

To establish, for program $\mathcal{P}$, the invariance of a state property q (i.e., $\Box q$), it is sufficient to assertion check an annotated program $\mathcal{P}^A$ with annotated initial predicate $\Theta\{q\}$, and annotated transitions $\{q\}\mathcal{T}\{q\}$. We present this as a *model checking rule INV**:

*INV**	Θ^A:	$\Theta\{q\}$
	$\mathcal{T}^A$:	$\{q\}\mathcal{T}\{q\}$
	$\mathcal{P} \models$	$\Box q$

Let us now examine how one may model check the temporal formula $\Box(p \Rightarrow \Box q)$, where p and q are state formulae. A proof rule to establish its validity for a program $\mathcal{P}$ is given below.

TINV	T1.	$p \Rightarrow \phi$
	T2.	$\phi \Rightarrow q$
	T3.	$\{\phi\}\mathcal{T}\{\phi\}$
		$\Box(p \Rightarrow \Box q)$

As in rule **INV**, rule **TINV** requires an auxiliary predicate ϕ to establish the invariance of q, starting from a state s_i in which predicate p holds. Assertion ϕ holds in state s_i (by premise T1), and is propagated from each state to the next (by premises T3). It follows from premise T2 that q also holds starting from state s_i.

To start with, let us examine the special case of establishing $\Box(q \Rightarrow \Box q)$ i.e., when predicate p is set to q in the above formulae. Known as a *stability* property, this formula asserts that if q holds at any state in a behaviour, then it will continue to hold for the rest of the behaviour. As in the previous proof rule, we use predicate q for ϕ. We may establish this using the following model checking rule *STAB**:

*STAB**	Θ^A:	$\Theta\{\mathbf{tt}\}$
	$\mathcal{T}^A$:	$\{q\}\mathcal{T}\{q\}$
	$\mathcal{P} \models$	$\Box(q \Rightarrow \Box q)$

Now, to model check the general case $\Box(p \Rightarrow \Box q)$, it is sufficient if we additionally establish validity of the formula $\Box(p \Rightarrow q)$. We may do this either by theorem proving, or by establishing that it is an invariant property of program $\mathcal{P}$, by using the model checking rule INV^*.

4.2 Rules for Eventuality

Let us now consider the eventuality formula $\Box(p \Rightarrow \Diamond q)$. Known as the *basic response property*, it asserts that every state s_i where formula p holds is followed by a state s_j $(j \geq i)$ where formula q holds. Response properties are usually established under a *fairness assumption* [Fra86, MP91b]. A common fairness assumption, known as *weak fairness*, formalizes the concept of "finite progress" [Dij68] — if a program $\mathcal{P}$ has an enabled operation, then it will be eventually executed. The effect of a fairness assumption is to filter out behaviours that would otherwise be legal behaviours of program $\mathcal{P}$. The following rule relies on weak fairness to ensure that a helpful transition τ_k, leading to q, will be taken.

RESP	R1.	$\Box(p \Rightarrow (q \vee \phi))$
	R2.	$\{\phi\}\mathcal{T}\{q \vee \phi\}$
	R3.	$\{\phi\}\tau_k\{q\}$
	R4.	$\Box(\phi \Rightarrow (q \vee En(\tau)))$
		$\Box(p \Rightarrow \Diamond q)$

Premise R1 ensures that p implies q or ϕ. Premise R2 ensures that all transitions either lead from ϕ to q, or preserve ϕ. Premise R3 states that the helpful transition τ_k leads from ϕ to q. Finally, premise R4 ensures that τ_k is enabled as long as ϕ holds and q does not hold (here, predicate $En(\tau)$ stands for the enableness condition of transition τ). Therefore, if p holds at state s_i, but q does not hold, then ϕ must hold continuously past state s_i, and transition τ_k is not taken. However, by premise R4, this means that transition τ_k is continuously enabled (but never taken) in these states, which violates our fairness assumption.

To cast proof rule **RESP** into a model checking problem, observe that premises R1 and R4 may be established by using model checking rules INV^* or $STAB^*$ (or by using a combination of model checking and theorem proving methods). The remaining premises R2 and R3 may be established by assertion checking the following annotated program $TRIG^*$:

$TRIG^*$	Θ^A:	$\Theta\{\mathbf{tt}\}$
	$\mathcal{T}^A$:	$\{\phi\}\mathcal{T}\{q \vee \phi\}$
	$\mathcal{T}^A$:	$\{\phi\}\tau_k\{q\}$
	$\mathcal{P} \models$	$\Box(\phi \Rightarrow \phi \,\mathcal{U}\, q)$

Note that program $TRIG^*$ contains annotated transitions $\{\phi\}\tau_k\{q \vee \phi\}$ **and** $\{\phi\}\tau_k\{q\}$ (since $\tau_k \in \mathcal{T}$). Our model checking algorithm has been designed in such a way that assertions of both transitions will be checked. An important point to note here is that *behaviours that violate the weak*

fairness assumption may be generated during model checking. These behaviours, however, will not trigger assertion violations as we only check for violations under the assumption of weak fairness.

Let us now examine how the basic response property may be established under the assumption of *strong fairness*[3]. The following rule relies on strong fairness to ensure that a helpful transition τ_k, leading to q, will be taken.

CRESP	C1.	$\Box(p \Rightarrow (q \vee \phi))$
	C2.	$\{\phi\}\mathcal{T}\{q \vee \phi\}$
	C3.	$\{\phi\}\tau_k\{q\}$
	C4.	$\Box(\phi \Rightarrow \Diamond(q \vee En(\tau)))$
		$\Box(p \Rightarrow \Diamond q)$

The difference between this rule and the earlier rule is the fourth premise. Premise R4 requires that ϕ holding in a state s_i implies that q holds or the helpful transition τ_k is enabled in state s_i. Premise C4, however, only requires formula q to hold or transition τ_k to be enabled in a state s_j, $(j \geq i)$. As before, if p holds in a state, and is not followed by a state in which q holds, then ϕ must hold continuously, with transition τ_k never being taken. Premise C4 guarantees that τ_k is enabled infinitely many times, which violates our assumption of strong fairness.

This rule may be adapted to model checking on the lines of rule **RESP**, but for the fourth premise, an eventuality formula, which is established either by theorem proving or by using model checking rule *TRIG**.

Let us now examine how we may establish eventuality properties without relying on single helpful transitions. Known as *extended response properties*, they are established based on the principle of *well-founded induction*, by an argument similar to proofs of termination for sequential programs.

Assume a reflexive and transitive binary relation $\preceq$ over a set $\mathcal{A}$, denoted as the pair $(\mathcal{A}, \preceq)$. Such a relation is called a *preorder*, i.e, for elements $a, b, c \in \mathcal{A}$, $a \preceq a$ and if $a \preceq b$ and $b \preceq c$ then also $a \preceq c$. If $a \preceq b$ and $b \preceq a$, we write $a = b$. If $a \preceq b$ but not $b \preceq a$, we denote $a \prec b$. The relation $\prec$ is an *ordering*, i.e., it is irreflexive, antisymmetric, and transitive. We refer to relation $\prec$ as the ordering *induced* by the preorder $\preceq$.

We say that an ordering $\prec$ is *well founded* if there does not exist an infinite sequence $a_0, a_1, \ldots$, where $a_i \in \mathcal{A}$ and for all $i = 0, 1, \ldots$, $a_{i+1} \prec a_i$. In such a case, we refer to the preorder $(\mathcal{A}, \preceq)$ inducing the ordering $\prec$ to be well founded.

For a well-founded preorder $(\mathcal{A}, \preceq)$, define a *ranking function* δ which maps sequences of states into the domain $\mathcal{A}$. The following rule uses well-founded induction to establish eventuality properties:

[3]Intuitively, the requirement of strong fairness disallows computations in which a transition τ is enabled infinitely many times but taken only finitely many times

WRESP	W1.	$\Box(p \Rightarrow (q \vee \phi))$
	W4.	$\Box((\phi \wedge (\delta = \alpha)) \Rightarrow \Diamond(q \vee (\phi \wedge (\delta \prec \alpha))))$
		$\Box(p \Rightarrow \Diamond q)$

Let σ be a behaviour of program $\mathcal{P}$. For each position $j \geq 0$, we refer to $\delta(s_0, s_1, \ldots, s_j)$ as the *rank* at position j.

Premise W1 ensures, as before, that if predicate p holds at state s_i, then either q or ϕ hold at s_i. Premise W2 ensures that if ϕ holds with a certain rank α, then eventually we will reach a state in which either q holds, or ϕ is maintained but with a rank lower than α. Since the range of function δ is set $\mathcal{A}$, this ranking cannot keep decreasing indefinitely because set $\mathcal{A}$ is well-founded. Therefore, predicte q must hold eventually.

This principle may be used in a straightforward way in model checking. Premise W1, an invariance property, may be established as outlined in section 4.1. Premise W2, an eventuality formula, is usually established by one of the two previous rules.

5 Conclusion and Ongoing Work

We have presented a "direct" model checking approach for checking temporal properties of finite-state programs. The traditional approach requires the construction of equivalent automata on infinite words (Büchi automata), and checking for their emptiness. Such a construction results in an automaton with an exponential number of states in the length of the original formula, which is a problem especially when model checking under fairness assumptions. Another drawback is the difficulty in interpreting counterexamples produced by the model checker.

A well known problem of model checking is that it does not supply an argument which will convince human reviewers — the fact that a model checking algorithm has detected no counterexamples to a temporal claim is not sufficient evidence to convince reviewers of its validity. The approach outlined in this paper brings model checking closer to theorem proving, in that one may extract a chain of argument as a result of a model checking effort. In fact, our approach unifies model checking with theorem proving, and allows portions of a model checking problem to be verifed by theorem proving methods. This increases the array of tools at the disposal of practicing verifiers.

At the time of writing, we have a prototype implementation of the model checker available for experimentation. We have been using it for practical verifications, notably for the analysis of liveness violations in the DQDB protocol [HCM92]. Other applications include consistency checking of SCR-style *requirements specification documents* [AFB+88, PM91].

Work is also underway to integrate our model checking algorithm into *Snap*, a theorem proving system [BS94]. Performing model checking in the context of a theorem proving system such as Snap will enable verifiers to use the system's book-keeping functions to manage complex verifications.

It will also allow verifiers to perform reductions and transformations (using term rewriting) on the verification problem.

Acknowledgements

I thank the Telecommunications Research Institute of Ontario for support. I also thank Gerard Holzmann, David Parnas, Doron Peled and Frank Stomp for fruitful discussions.

References

[AFB+88] T. Alspaugh, S. Faulk, K. Britton, R. Parker and D. L. Parnas. *Software Requirements for the A-7E Aircraft.* Naval Research Laboratory, March 1988.

[Bha95] R. Bharadwaj. *Tools to Support a Formal Verification Method for Systems with Concurrency and Nondeterminism.* Ph.D. Thesis, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada 1995.

[BS94] R. Bharadwaj and F. A. Stomp. *Towards a Checker/Annotator for Proofs of Distributed Programs.* Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ 1994.

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla. *"Automatic verification of finite-state concurrent systems using temporal logic specifications."* ACM TOPLAS, 8(2): 244–263, 1986.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley, 1988.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. *"Memory efficient algorithms for the verification of temporal properties."* In *Formal Methods in System Design*, 1: 275–288, Kluwer Academic Publishers, 1992.

[Dij68] E. W. Dijkstra. *"Cooperating Sequential Processes."* In *Programming Languages*, F. Genuys (Ed.), Academic Press, NY 1968.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ 1976.

[Fra86] N. Francez. *Fairness.* Springer Verlag, 1986.

[HCM92] E. L. Hahne, A. K. Choudhury and N. F. Maxemchuk. *"DQDB networks with and without bandwidth balancing."* IEEE Transactions on Communications, 40(7): 1192–1204, July 1992.

[Hol88] G. J. Holzmann. *"An improved protocol reachability analysis technique."* Software Practice and Experience, 18(2): 137–161, 1988.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall Software Series 1991.

[HS82] E. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press 1984.

[Kur89] R. P. Kurshan. *"Analysis of discrete event coordination."* LNCS 430, pp. 414–453, Springer Verlag 1989.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers 1993.

[MP83] Z. Manna and A. Pnueli. *"Verification of concurrent programs: a temporal proof system."* Foundations of Computer Science IV, Distributed Systems: Part 2, Mathematical Centre Tracts 159, pp. 163–255, Center for Mathematics and Computer Science, Amsterdam 1983.

[MP90] Z. Manna and A. Pnueli. *"Tools and rules for the practicing verifier."* In *Carnegie Mellon Computer Science: A 25-year Commemorative*, ACM Press 1990.

[MP91a] Z. Manna and A. Pnueli. *"Completing the temporal picture."* Theoretical Computer Science, 83(1): 97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag 1991.

[PM91] D. L. Parnas and J. Madey. *Functional Documentation for Computer Systems Engineering.* T.R. 237, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada September 1991.

[Pel93] D. Peled. *"All from one, one for all: on model checking using representatives."* In *Proc. 5^{th} Workshop on Computer Aided Verification*, Elounda, Springer Verlag June 1993.

[RvH91] J. Rushby and F. von Henke. *"Formal verification of algorithms for critical systems."* Computer Science Laboratory, SRI International 1991.

[SdR94] F. A. Stomp and W.-P. de Roever. *"Principles for sequential reasoning about distributed algorithms."* To appear at the ftp-site of Formal Aspects of Computing (1994).

[VW86] M. Y. Vardi and P. Wolper. *"An automata-theoretic approach to automatic program verification."* In Proceedings of a *Symposium on Logic in Computer Science*, pp. 322–331, Cambridge June 1986.

[Wol89] P. Wolper. *"On the relation of programs and computations to models of temporal logic."* In *Proceedings of Temporal Logic in Specification*, LNCS 398 B. Banieqbal, H. Barringer and A. Pneuli (Eds.), pp. 75–123, Springer Verlag 1989.