

Deductive Model Checking *

HENNY B. SIPMA, TOMÁS E. URIBE AND ZOHAR MANNA

sipma,uribe,manna@cs.stanford.edu

Computer Science Department, Stanford University, Stanford, CA. 94305

Received ???, 1997

Abstract. We present an extension of classical tableau-based model checking procedures to the case of infinite-state systems, using deductive methods in an incremental construction of the behavior graph. Logical formulas are used to represent infinite sets of states in an abstraction of this graph, which is repeatedly refined in the search for a counterexample computation, ruling out large portions of the graph before they are expanded to the state-level. This can lead to large savings, even in the case of finite-state systems. Only local conditions need to be checked at each step, and previously proven properties can be used to further constrain the search. Although the resulting method is not always automatic, it provides a flexible, general and complete framework that can integrate a diverse number of other verification tools.

Keywords: Deductive verification, model checking, reactive systems

1. Introduction

We present a model checking procedure for verifying temporal logic properties of infinite-state systems. It extends the classical tableau-based model checking procedure for verifying *linear-time temporal logic* specifications of reactive systems described by *fair transition systems*. To verify that a system \mathcal{S} satisfies a temporal specification φ , the classical procedure checks whether the $(\mathcal{S}, \neg\varphi)$ *behavior graph* admits any counterexample computations. This behavior graph is the product of the temporal tableau for $\neg\varphi$ and the state transition graph for \mathcal{S} , which makes the procedure essentially applicable to finite-state systems only.

Our procedure starts with the temporal tableau for $\neg\varphi$ and repeatedly refines and transforms this graph until a counterexample computation is found or it is demonstrated that such a computation cannot exist. Even for finite-state systems, this can lead to significant savings, since portions of the behavior graph can be eliminated long before they are fully expanded to the state level.

For infinite-state systems, the procedure is not guaranteed to terminate, in general. However, we show that it is a complete method for proving general state-quantified temporal properties, relative to the underlying first-order reasoning. In Section 4, we illustrate the procedure by model checking an accessibility property for the Bakery algorithm. Expansion to 16 nodes suffices to verify this property

* This research was supported in part by the National Science Foundation under grant CCR-95-27927, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grant DAAH04-95-1-0317, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

over this infinite-state system. Even when the procedure does not terminate, partial results can still be valuable, giving a representation of all potential counterexample computations, which can be used for further verification or testing.

Like standard model checking, our procedure does not require user-provided auxiliary formulas and allows the construction of counterexamples; the process is guided by the search for such computations. Like deductive methods, it only needs to check local conditions, and allows the verification of infinite-state systems through the use of powerful representations to describe sets of states (e.g. first-order formulas). We also accommodate the use of previously established invariants and simple temporal properties.

We present our procedure in the framework of [34], where deductive methods are used to verify linear-time temporal logic specifications for reactive systems described by fair transition systems. However, the main ideas can be easily adapted to other temporal logics and system specification languages as well.

2. Related work

Model Checking: Most approaches to temporal logic model checking [12, 39] have used explicit state enumeration, or specialized data structures to represent the transition relation and compute fixpoints over it, as in BDD-based “symbolic” model checking [10, 35]. While automatic, and particularly successful for hardware systems, these approaches require that the system, or a suitable abstraction of it, conform to the particular data structure used. Most often, the system must be finite-state. Furthermore, even in the finite-state case these techniques can generate representations whose size is proportional to the reachable state space.

The “on-the-fly model checking” for CTL* presented in [1] constructs only a portion of the state-space as required by the given formula, but is still restricted to finite-state systems. Our procedure is similarly need-driven, but expands the state-space in a top-down manner as well, moving from an abstract representation to a more detailed one as necessary.

A method for generating an abstract representation of a possibly infinite state-space is presented in [5], using partitioning operations similar to the ones we describe below. However, the abstraction in [5] is independent of any particular temporal property to be verified. Finally, the local model checking algorithm for real-time systems in [43] can be seen as a specialized variant of our procedure; it too refines a finite representation of an infinite behavior graph, consecutively splitting nodes to satisfy constraints that arise from the formula and system being checked.

Deductive Methods: A complete deductive system for temporal verification of branching-time properties is presented in [22], while [6] presents a proof system for the modal mu-calculus. Manna and Pnueli [34] present a deductive framework for the verification of fair transition systems based on *verification rules*, which reduce temporal properties of systems to first-order premises. *Verification diagrams* [33, 7, 8] provide a graphical representation of the verification conditions needed to establish a particular temporal formula.

All of these methods apply to infinite-state systems and enjoy relative completeness, but can require substantial user guidance to succeed. These methods yield a direct proof of the system-validity of a property, but do not produce counterexample computations when the property fails.

A different deductive verification method based on transforming a graphical representation of transition systems is presented in [19]. In this approach, the system is transformed gradually until an abstraction is produced that can be model checked, or corresponds directly to the specification. Thus, it presents a direct proof, whereas we proceed by showing that counterexample computations cannot exist.

The procedure presented in [3] for automatically establishing temporal *safety* properties is based on an *assertion graph* similar to the *falsification diagram* we use, and can also produce counterexamples. Our approach is a dual one: instead of checking that all computations satisfy the temporal tableau of the formula ϕ being proved, we check that no computations satisfy the tableau for $\neg\phi$. In Section 4.6 we discuss how some of the abstraction and propagation techniques in [3] can be adapted to this setting.

Abstraction: Property-preserving abstractions for the μ -calculus are investigated in [31] and [15]. An algorithm for producing abstracted state spaces which preserve various fragments of CTL* is presented in [18]. Like our procedure, this algorithm constructs a quotient of the state space, splitting equivalence classes (sets of states) according to the value of different subformulas. The abstraction can be computed with respect to the particular formula to be model checked, enhancing the degree of abstraction. Dams *et. al.* [17, 16] show how this quotient construction can be combined with abstract interpretation [13] of the program being verified.

Similarly, [23] uses deduction to construct an abstract state graph that can be model checked. A given set of subformulas determines the sets of states in the abstraction; these subformulas are atomic assertions in the property being verified, obtained from the control structure of the program, or provided by the user. The abstraction can be refined, by adding new subformulas, if it fails to prove or disprove the property being verified. In contrast, we construct the abstraction dynamically, refining an abstract behavior graph.

Other Combinations: Other methods for combining theorem proving and model checking have been proposed. Most of them rely on decomposing the verification effort into subgoals which can be model checked, where the decomposition is justified using deductive methods. Hungar [25] and Kurshan and Lamport [28] use deductive modular decomposition to reduce the correctness of a large system to that of smaller components that can be model checked. In [40] and [21], abstraction is used to obtain subgoals that can be model checked; the correctness of the abstraction is proved deductively. A similar approach is advocated by Ostroff [36] in the case of real-time systems.

The PVS system [37] includes a decision procedure to model check propositional μ -calculus expressions, which can appear as subgoals in a verification effort. The STeP system [2] uses deductive verification rules, but can also model check subgoals over the given system specification whenever possible. Pnueli and Shahar [38]

use BDD-based tools to perform deductive verification, and show how deductively obtained invariants can constrain the BDD-based symbolic model checking process.

Finally, [14] presents a combination of model checking and deduction, similar in spirit to the one we present, for the first-order branching-time logic ACTL. Local model checking is carried out relative to first-order verification conditions, which makes the verification of infinite-state systems possible.

3. Preliminaries

Fair Transition Systems: The computational model, following [34], is a *fair transition system* (FTS). An FTS \mathcal{S} is a triple $\langle \mathcal{V}, \Theta, \mathcal{T} \rangle$, where \mathcal{V} is a set of variables, Θ is the *initial condition*, and \mathcal{T} is a finite set of *transitions*. A finite set of *system variables* $V \subset \mathcal{V}$ determines the possible states of the system. The *state-space*, Σ , is the set of all possible valuations of the system variables.

We use a first-order assertion language \mathcal{A} to describe Θ and the transitions in \mathcal{T} . (This language can be augmented with features such as interpreted symbols and constraints, or specialized to the finite-state case, e.g. using BDDs.) Θ is an assertion over the system variables V . A transition τ is described by a *transition relation* $\rho_\tau(\vec{x}, \vec{x}')$, an assertion over the set of system variables \vec{x} and a set of *primed variables* \vec{x}' indicating their values at the next state. \mathcal{T} includes an *idling transition*, *Idle*, whose transition relation is $\vec{x} = \vec{x}'$.

A *run* is an infinite sequence of states s_0, s_1, \dots such that s_0 satisfies Θ , and for each $i \geq 0$, there is some transition $\tau \in \mathcal{T}$ such that $\rho_\tau(s_i, s_{i+1})$ evaluates to true. We then say that τ is *taken* at s_i , and that state s_{i+1} is a τ -*successor* of s . A transition is *enabled* if it can be taken at a given state. Such states are characterized by the formula

$$\text{enabled}(\tau) \stackrel{\text{def}}{=} \exists \vec{x}'. \rho_\tau(\vec{x}, \vec{x}') .$$

As usual, we define the *strongest postcondition* $\text{post}(\tau, \phi)$ and the *weakest precondition* $\text{wpc}(\tau, \phi)$ of a formula ϕ relative to a transition τ as follows:

$$\begin{aligned} \text{post}(\tau, \phi) &\stackrel{\text{def}}{=} \exists \vec{x}_0. (\rho_\tau(\vec{x}_0, \vec{x}) \wedge \phi(\vec{x}_0)) \\ \text{wpc}(\tau, \phi) &\stackrel{\text{def}}{=} \forall \vec{x}'. (\rho_\tau(\vec{x}, \vec{x}') \rightarrow \phi(\vec{x}')) . \end{aligned}$$

We also use the notation $\{\phi\} \tau \{\psi\} \stackrel{\text{def}}{=} (\phi(\vec{x}) \wedge \rho_\tau(\vec{x}, \vec{x}')) \rightarrow \psi(\vec{x}')$.

Fairness: The transitions in \mathcal{T} can be optionally marked as *just* or *compassionate*. A just (or *weakly fair*) transition cannot be continually enabled without ever being taken; a compassionate (or *strongly fair*) transition cannot be enabled infinitely often but taken only finitely many times. A *computation* is a run that satisfies these fairness requirements.

EXAMPLE: Figure 1 shows a program that implements Lamport's [29] *bakery algorithm* for mutual exclusion. Each of the statements in the program corresponds to

a transition, denoted by its label; thus, $\mathcal{T} = \{Idle, \ell_0..l_4, m_0..m_4\}$. All transitions are just, except for m_0 and ℓ_0 , which have no fairness requirements.

This is an infinite-state program, since the value of the integer variables y_1 and y_2 can grow arbitrarily large. The program ensures *mutual exclusion*: control is never at locations ℓ_3 and m_3 at the same time; in Section 4 we will also see that it ensures *accessibility*: if control resides at ℓ_1 (resp. m_1), it will always eventually reach ℓ_3 (resp. m_3). Note that since the **noncritical** statement is not fair, we cannot claim the same if control is initially at ℓ_0 (resp. m_0). Another property of this program is *1-bounded overtaking*: if a process has expressed interest in entering the critical section, the other process can enter the critical section at most once before the first process can enter. \square

The STeP (Stanford Temporal Prover) verification system translates programs such as BAKERY into the corresponding fair transition systems [2]. To each process corresponds a *control variable*, which ranges over its distinct locations. The assertions ℓ_i and m_j indicate that control resides at locations i and j for each of the two processes.

local y_1, y_2 : integer where $y_1 = y_2 = 0$

$$\left[\begin{array}{l} \mathbf{loop\ forever\ do} \\ \left[\begin{array}{l} \ell_0: \mathbf{noncritical} \\ \ell_1: y_1 := y_2 + 1 \\ \ell_2: \mathbf{await} (y_2 = 0 \vee y_1 \leq y_2) \\ \ell_3: \mathbf{critical} \\ \ell_4: y_1 := 0 \end{array} \right] \end{array} \right] \\ \parallel \\ \left[\begin{array}{l} \mathbf{loop\ forever\ do} \\ \left[\begin{array}{l} m_0: \mathbf{noncritical} \\ m_1: y_2 := y_1 + 1 \\ m_2: \mathbf{await} (y_1 = 0 \vee y_2 < y_1) \\ m_3: \mathbf{critical} \\ m_4: y_2 := 0 \end{array} \right] \end{array} \right]$$

Figure 1. Program BAKERY for mutual exclusion.

Linear-time Temporal Logic: As specification language we use linear-time temporal logic (LTL) over the assertion language \mathcal{A} , where no temporal operator is allowed to appear within the scope of a quantifier. Such temporal formulas are called *state-quantified* [34]. We use the usual future and past temporal operators, such as $\square, \diamond, \bigcirc, \mathcal{U}, \mathcal{W}$ (future) and their past-time counterparts. A formula with no temporal operators is called a *state-formula* or an *assertion*. A *safety property* is one that can be expressed as $\square p$ for a past temporal formula p ; in the special case where p is a state-formula, the property is called an *invariance*. We say that

a formula φ is \mathcal{S} -*valid*, written as $\mathcal{S} \models \varphi$, if all the computations of \mathcal{S} satisfy φ . Mutual exclusion for BAKERY can be expressed as the invariance $\Box(\neg(\ell_3 \wedge m_3))$, and accessibility as $\Box(\ell_1 \rightarrow \Diamond \ell_3)$ (a *response* property). For details on LTL and tableau constructions, we refer the reader to [27, 34], and define only the basic concepts we need.

The Formula Tableau: Given an LTL formula ϕ , we can construct its *tableau* Φ_ϕ , a finite graph that describes all of its models [34]. Briefly, each node in the tableau is labeled with an *atom*, which is a set of state- and temporal formulas expected to hold whenever a model resides at this node. Two nodes A_1 and A_2 are connected with a directed edge $\langle A_1, A_2 \rangle$ if the formulas in A_2 can hold at a state following one that satisfies the formulas in A_1 .

An atom is called *initial* if its formulas can hold at the initial state of a model. ϕ is satisfiable only if there is a *strongly connected subgraph* (SCS) (a subgraph with at least one edge in which each node is reachable from every other node) in Φ_ϕ that is reachable from an initial atom. Furthermore, if a model satisfies, e.g., $\Diamond p$ at some point, it must in fact satisfy p at this or another point later on. A *fulfilling* SCS is one where all such eventualities are satisfied.

PROPOSITION 1 *ϕ is satisfiable iff there is a fulfilling, reachable SCS in Φ_ϕ .*

The atomic formulas in the tableau are those appearing in ϕ ; normally, these are treated as propositional constants. However, we will combine them together into a single formula f of the assertion language \mathcal{A} , which is always rewritten and simplified as much as possible.

The size of the tableau is exponential in the size of the formula. We can often reduce the size of the tableau (although it remains exponential in the worst case) by introducing nondeterminism, where sequences of states may be mapped into multiple paths in the tableau, resulting in a so-called *particle tableau* [34].

The classic model checking algorithm for finite-state systems builds the product between the state transition graph of \mathcal{S} and $\Phi_{\neg\phi}$, and checks whether this *behavior graph* contains an SCS that is fulfilling and also satisfies the fairness requirements associated with \mathcal{S} . If such an SCS is reachable from a node that is initial with respect to both \mathcal{S} and $\Phi_{\neg\phi}$, a counterexample computation can be produced; otherwise, φ is \mathcal{S} -valid.

4. Deductive Model Checking

We now present an alternate approach to the verification of temporal properties of reactive systems, *deductive model checking*.

Instead of building the $(\mathcal{S}, \neg\varphi)$ behavior graph explicitly, we will start with a general skeleton of the behavior graph and progressively refine it. This graph will always contain all the possible computations of \mathcal{S} that violate φ , and we will call it the *falsification diagram* for \mathcal{S} and φ . (In [42], this graph is called the *\mathcal{S} -refined tableau*.)

Definition 1. (falsification diagram) Given an FTS \mathcal{S} and a temporal property φ , a *falsification diagram* for \mathcal{S} and φ is a directed graph \mathcal{G} whose nodes are labeled with pairs (A, f) , where A is a temporal tableau atom for $\neg\varphi$ and f is a state-formula. Edges of \mathcal{G} are labeled with subsets of \mathcal{T} , the set of transitions of \mathcal{S} . For nodes M and N , we write $\tau \in \langle M, N \rangle$ if transition τ is in the label of the edge from M to N , or simply say that τ *labels* $\langle M, N \rangle$. A subset of the nodes in \mathcal{G} is marked as *initial*.

A falsification diagram can be seen as a finite abstraction of the behavior graph. The state-formula f in a node (A, f) describes a superset of the states reachable at that node; similarly, the transitions that label an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ are a superset of those that can be taken from an f_1 -state to reach an f_2 -state. We will see that any path through a falsification diagram corresponds to a path through the corresponding temporal tableau. That is, for any path $(A_0, f_0), (A_1, f_1), \dots$ through \mathcal{G} , the *underlying path* A_0, A_1, \dots will be a path in $\Phi_{\neg\varphi}$.

4.1. The DMC Procedure

Starting with the tableau graph $\Phi_{\neg\varphi}$, we construct an initial falsification diagram \mathcal{G}_0 as follows:

1. Replace each node label A by (A, f_A) , where f_A is the conjunction of the state-formulas in the tableau atom A .
2. For each node $N : (A, f)$ such that A is initial in the tableau $\Phi_{\neg\varphi}$, add a new node $N_0 : (A, f \wedge \Theta)$, with no incoming edges, whose outgoing edges go to exactly the same nodes as those of N . A self-loop $\langle N, N \rangle$ becomes an edge $\langle N_0, N \rangle$ in the new graph. These new nodes are the *initial nodes* in the falsification diagram.
3. Label each edge in \mathcal{G}_0 with the entire set of transitions \mathcal{T} .

EXAMPLE: Figure 2 presents an example of an initial falsification diagram to prove accessibility, $\varphi : \Box(\ell_1 \rightarrow \Diamond\ell_3)$, for the BAKERY program of Figure 1. This diagram is based on the particle tableau for $\neg\varphi : \Diamond(\ell_1 \wedge \Box\neg\ell_3)$. Nodes 3 and 4 correspond to the initial nodes in the $\neg\varphi$ tableau. Node 1 results from adding the initial condition to node 4. The initial node derived from node 3 is immediately pruned since $\ell_1 \wedge \Theta$ is unsatisfiable. The SCS $\{4\}$ is not fulfilling, but $\{2\}$ is. \square

Subsequent examples will show how this property is verified with a DMC implementation based on the STeP system [2]. The simplification mechanisms provided by STeP are used to transform state-formulas into simpler, logically equivalent ones, using decision procedures for built-in theories such as linear arithmetic, equality, datatypes and finite domains.

STeP also provides general first-order validity checking procedures [4], as well as a complete, interactive theorem-proving environment. However, these tools were not

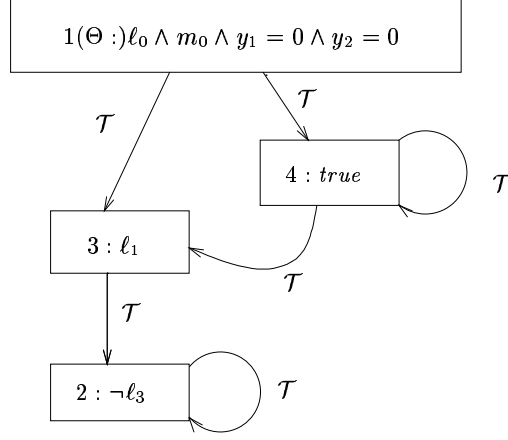


Figure 2. Initial falsification diagram for BAKERY accessibility.

needed in the examples we give below, since the required validities were all proved automatically by STeP's simplifier.

The main data structure maintained by the procedure is a falsification diagram \mathcal{G}_i and a list L_i of strongly connected subgraphs (SCS's) of this graph. We present the deductive model checking (DMC) procedure as a set of transformations on this pair. Initially, the SCS list contains all the *maximal strongly connected subgraphs* (MSCS's) of \mathcal{G}_0 .

Deductive model checking proceeds by repeatedly applying one of transformations 1–13 described below. The process stops if we find a *reachable, fulfilling and adequate* SCS (see Section 4.5), in which case we have a counterexample computation, or if we obtain an empty SCS list, indicating that φ is \mathcal{S} -valid.

Basic Transformations:

- **1 (remove edge label).** If an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ is labeled with a transition τ and the assertion

$$f_1(\vec{x}) \wedge f_2(\vec{x}') \wedge \rho_\tau(\vec{x}, \vec{x}')$$

is unsatisfiable, remove τ from the edge label.

- **2 (empty edge).** If an edge is labeled with the empty set, remove the edge.
- **3 (unsatisfiable node).** If f is unsatisfiable for a node (A, f) , or if a node has no successors, remove the node.
- **4 (unreachable node).** Remove a node unreachable from an initial node.

- **5 (unfulfilling SCS).** If an SCS is not fulfilling, remove it from the SCS list. (An SCS is fulfilling if its underlying tableau SCS is fulfilling.)
- **6 (SCS split).** If an SCS becomes disconnected (because a node or an edge is removed from the graph), replace it by its constituent MSCS's.

These basic transformations should be applied whenever possible. Except for (1) and (3), which require checking the satisfiability of formulas in the assertion language, they can all be easily automated (see Section 7).

Node Splitting: In the following, we will have the opportunity to replace a node N by new nodes N_1 and N_2 . Any incoming edge $\langle M, N \rangle$ is replaced by edges $\langle M, N_1 \rangle$ and $\langle M, N_2 \rangle$ with the same label, for $M \neq N$. Similarly, any outgoing edge $\langle N, M \rangle$ is replaced by edges $\langle N_1, M \rangle$ and $\langle N_2, M \rangle$ with the same label as the original edge.

If a self-loop $\langle N, N \rangle$ was present, we add edges $\langle N_1, N_1 \rangle$, $\langle N_2, N_2 \rangle$, $\langle N_1, N_2 \rangle$ and $\langle N_2, N_1 \rangle$, all with the same label as $\langle N, N \rangle$. If an initial node is split, the two new nodes are also labeled as initial. If the split node was part of an SCS in the SCS list, this SCS is updated accordingly.

Basic Refinement Transformations:

- **7 (postcondition split).** Consider an edge from node $N_1 : (A_1, f_1)$ to $N_2 : (A_2, f_2)$ whose label includes transition τ . If $f_2 \wedge \neg post(\tau, f_1)$ is satisfiable (that is, f_2 does not imply $post(\tau, f_1)$), then replace (A_2, f_2) by the two nodes

$$\begin{aligned} N_{2,1} &= (A_2, f_2 \wedge post(\tau, f_1)) \\ N_{2,2} &= (A_2, f_2 \wedge \neg post(\tau, f_1)) \quad . \end{aligned}$$

We can immediately apply the **remove edge label** transformation to the edge between N_1 and $N_{2,2}$, removing transition τ from its label.

Nodes N_1 and N_2 need not be distinct. If $N_1 = N_2$ then we split the node into two new nodes as above, but now the self-loop for $N_{2,2}$ and the edge from $N_{2,1}$ to $N_{2,2}$ will not contain the transition τ .

- **8 (precondition split).** Consider an edge

$$\langle N_1, N_2 \rangle = \langle (A_1, f_1), (A_2, f_2) \rangle$$

labeled with transition τ . If $f_1 \wedge \neg(enabled(\tau) \wedge wpc(\tau, f_2))$ is satisfiable, then replace (A_1, f_1) by the two nodes

$$\begin{aligned} N_{1,1} &= (A_1, f_1 \wedge enabled(\tau) \wedge wpc(\tau, f_2)) \\ N_{1,2} &= (A_1, f_1 \wedge \neg(enabled(\tau) \wedge wpc(\tau, f_2))) \quad . \end{aligned}$$

EXAMPLE: Figure 3 presents the result of applying a precondition split on edge $\langle 2, 2 \rangle$ and transition ℓ_0 of the initial falsification diagram, replacing node 2 by nodes 6 and 5. Basic transformations have been applied where applicable, for instance, removing all transitions except ℓ_4 from edge $\langle 6, 5 \rangle$. The underlying simplifications are performed automatically by STeP. \square

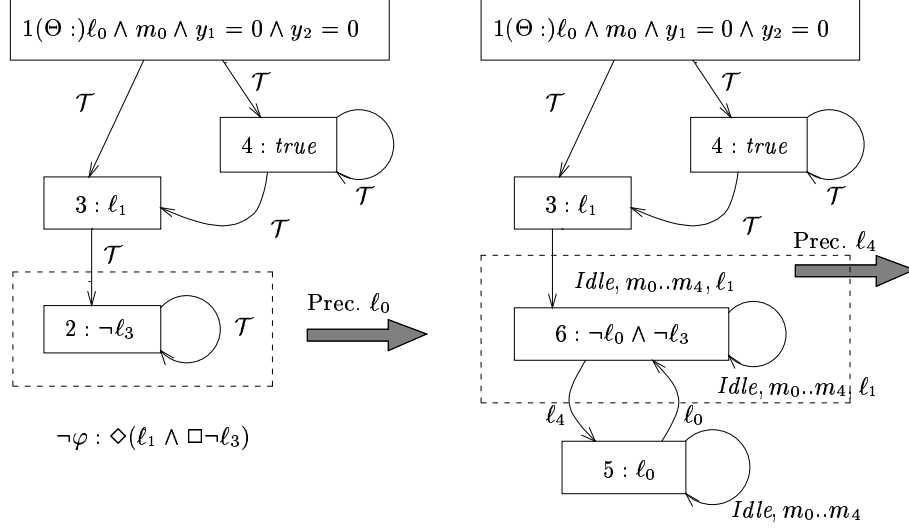


Figure 3. First splitting step for BAKERY accessibility.

The conditions for applying these transformations can be weakened if the required satisfiability checks are too expensive (see Section 7).

Variants of these transformations, such as n -ary splits according to possible control locations, are convenient in practice. In general, arbitrary conditions can be used to split nodes: if ϕ_1, \dots, ϕ_n are assertions, one can split a node $\langle A, f \rangle$ into the $n + 1$ nodes

$$\langle A, f \wedge \phi_1 \rangle, \dots, \langle A, f \wedge \phi_n \rangle, \langle A, f \wedge \neg(\phi_1 \vee \dots \vee \phi_n) \rangle .$$

For example, ϕ_1, \dots, ϕ_n can correspond to the set of relevant control locations, and the n -ary split would be a shortcut for a sequence of (binary) precondition splits. Our basic refinement transformations are motivated by the general structure of the system and property being checked, and can be automated as well.

Known invariants can be used to strengthen all (or only some) of the assertions in the falsification diagram; if $\Box p$ is a known invariant, for a state-formula p , then we can replace any node $\langle A, f \rangle$ by the node $\langle A, f \wedge p \rangle$. This use of invariants can also be very valuable in finite-state symbolic model checking [38]. Similarly, simple temporal properties of \mathcal{S} can be used to rule out paths in the diagram. For example, if we know that $\Box(p \rightarrow \diamond q)$ is \mathcal{S} -valid, then we can require that any candidate SCS featuring a state-formula that implies p also contain a state-formula compatible with q . (Before model checking begins, $\neg\varphi$ could always be conjoined with all other known temporal properties of \mathcal{S} , but at the risk of greatly increasing the size of the temporal tableau.)

4.2. Fairness Transformations

Together, transformations 1-8 are sufficient for the analysis of temporal safety properties, which do not depend on the fairness constraints of the system. If an *adequate* SCS is found (see Section 4.5), a counterexample is produced. If the set of SCS's (all of which are actually MSCS's, in this case) becomes empty, then we know there is no counterexample computation. However, to account for just and compassionate transitions we need the following extra transformations:

- **9 (enabled split).** Consider a just or compassionate transition τ containing a node $N : (A, f)$ such that $f \wedge \neg \text{enabled}(\tau)$ is satisfiable.

Then replace N by the two nodes

$$\begin{aligned} N_1 &= (A, f \wedge \text{enabled}(\tau)) \\ N_2 &= (A, f \wedge \neg \text{enabled}(\tau)) \quad . \end{aligned}$$

Definition 2. A transition τ is *fully enabled* at a node (A, f) if $f \rightarrow \text{enabled}(\tau)$ is valid; τ is *fully disabled* at a node (A, f) if $f \rightarrow (\neg \text{enabled}(\tau))$ is valid. A transition is *taken* on an SCS S if it is included in an edge-label in S . An SCS S is *just* (resp. *compassionate*) if every just (resp. compassionate) transition is either taken in S or not fully enabled at some node (resp. all nodes) in S .

That is, an SCS S is unjust (resp. uncompassionate) if some just (resp. compassionate) transition is never taken in S and fully enabled at all nodes (resp. some node) in S .

The motivation for these definitions is the following: if an SCS S is unjust or uncompassionate, then we know that a computation cannot visit each node in S infinitely often, never leaving S , since a fairness constraint would be violated. Note that if an SCS is not just, then all of its subgraphs are not just; but an SCS that is not compassionate may have a compassionate subgraph. This leads us to the next two transformations, which, like the basic ones, should be applied whenever possible:

- **10 (unjust SCS).** If an SCS is not just, remove it from the SCS list.
- **11 (uncompassionate SCS).** If an SCS S is not compassionate, then let τ_1, \dots, τ_n be all the compassionate transitions that are not taken in S . Replace S by all the MSCS's of the subgraph resulting from removing all the nodes in S where one of these transitions is fully enabled.

Note that these transformations do not change the underlying graph \mathcal{G} , but only the SCS's under consideration. However, unjust or unfulfilling SCS's can be fully removed from the graph if they have no outgoing edges, since they will never be part of a counterexample computation.

EXAMPLE: Figure 4 shows the result of an ℓ_4 -precondition split on $\langle 6, 5 \rangle$ of Figure 3, which yields nodes 8 and 7. At this point, nodes 5 and 7 are unreachable from the

initial node and can be removed. The only candidate SCS is thus $\{8\}$. An ℓ_1 -postcondition split for $\langle 8, 8 \rangle$ yields nodes 9 and 10. The only fulfilling SCS is $\{10\}$, since $\{9\}$ is unjust for ℓ_1 . \square

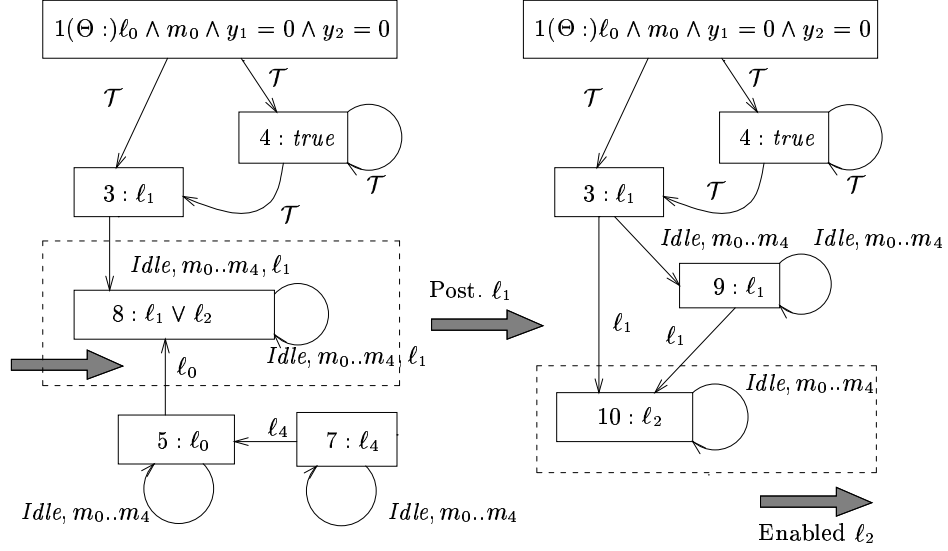


Figure 4. Second and third splitting steps for BAKERY accessibility.

EXAMPLE: Figure 5 shows the final three splits for the BAKERY verification. An enabled split for node 10 and transition ℓ_2 produces nodes 11 and 12. The SCS $\{11\}$ is unjust for ℓ_2 . Node 12 is now strengthened with the invariant $(y_2 \neq 0) \rightarrow (m_2 \vee m_3 \vee m_4)$. (Such invariants are generated automatically by STeP based on the program text.) An m_3 -precondition split on $\langle 12, 12 \rangle$ produces nodes 13 and 14. SCS $\{13\}$ is unjust for m_3 . Finally, an m_2 -precondition split on $\langle 14, 13 \rangle$ results in 15 and 16. SCS $\{16\}$ is unjust for m_4 , while $\{15\}$ is unjust for m_2 . Since no candidate SCS's remain, we have established that φ is \mathcal{S} -valid. \square

4.3. Termination Transformation

Transformations 1-11 are sufficient for the analysis of transition systems that have finite abstractions relative to the property to be proved. For example, the BAKERY program has a finite abstraction relative to accessibility and mutual exclusion. However, consider the property

$$\varphi : \diamond \square \neg((\ell_0 \vee \ell_1) \wedge (m_0 \vee m_1)) \rightarrow \diamond(\max(y_1, y_2) > N)$$

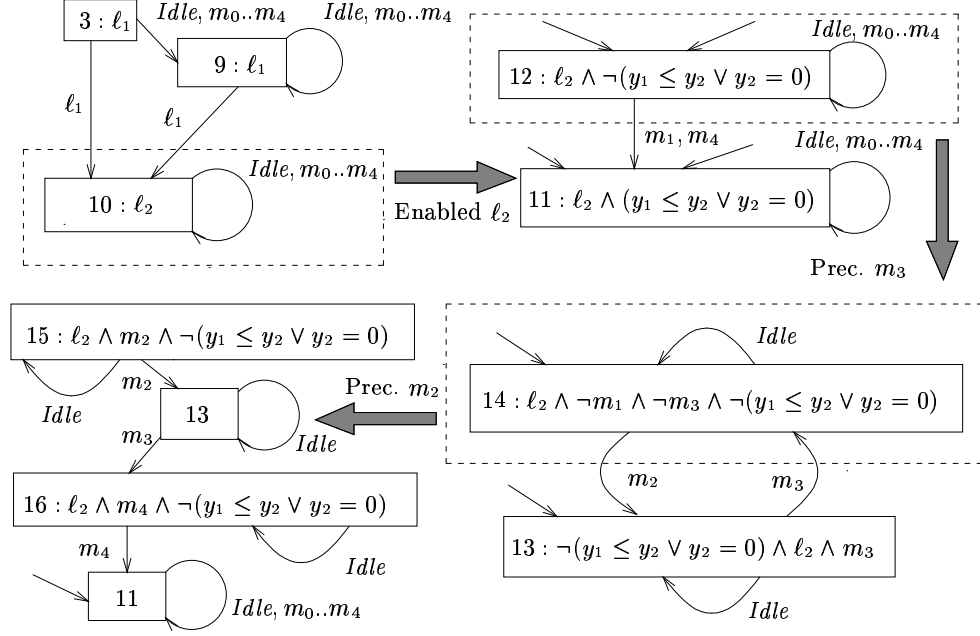


Figure 5. Last splitting steps for BAKERY accessibility.

where N is an arbitrary integer. This property states that $\max(y_1, y_2)$ will grow beyond N , provided the two processes are never at locations ℓ_0 or ℓ_1 (resp. m_0 or m_1) at the same time. BAKERY has no finite-state abstraction that can prove φ for an arbitrary N .

To verify such properties, we introduce a transformation that allows the user to assign a well-founded order to an SCS of the graph to prove, deductively, that infinite computations cannot forever reside in this SCS and visit all nodes.

Definition 3. (terminating SCS) An SCS S is *terminating* if there is no run that visits every node of S infinitely often.

We say that a binary relation \succ is *well-founded* over some domain \mathcal{D} if there are no infinite descending chains, that is, no infinite sequences of elements e_1, \dots, e_n, \dots in \mathcal{D} such that $e_1 \succ e_2 \succ \dots \succ e_n \succ \dots$. We write $x \succeq y$ iff $x \succ y$ or $x = y$.

Definition 4. (terminating edge) An edge e_t in an SCS S is *terminating* if there exist a well-founded domain \mathcal{D} and a *ranking function* $\delta(n, s)$, mapping pairs of nodes and states into \mathcal{D} , such that

1. for every edge $e = \langle N_1, N_2 \rangle = \langle (A_1, f_1), (A_2, f_2) \rangle$ in S and every $\tau \in e$

$$f_1(\vec{x}) \wedge \rho_\tau(\vec{x}, \vec{x}') \wedge f_2(\vec{x}') \rightarrow \delta(N_1, \vec{x}) \succeq \delta(N_2, \vec{x}')$$

and

2. for every $\tau \in e_t$, with $e_t = \langle N_1, N_2 \rangle = \langle (A_1, f_1), (A_2, f_2) \rangle$

$$f_1(\vec{x}) \wedge \rho_\tau(\vec{x}, \vec{x}') \wedge f_2(\vec{x}') \rightarrow \delta(N_1, \vec{x}) \succ \delta(N_2, \vec{x}') .$$

Clearly, no computation that resides in the SCS from some point onwards can traverse e_t infinitely many times. This leads to the following transformation (which was not included in [42]):

- **12 (terminating SCS).** If e is a terminating edge in an SCS S , then replace S by all the MSCS's of the subgraph obtained by removing e from S .

Again, this transformation does not change the underlying graph, but only the SCS's that may participate in a counterexample. However, as with unjust and unfulfilling SCS's, a terminating SCS may be removed from the graph if it has no outgoing edges.

The ranking functions can be part of an extended assertion language that is not necessarily the state-assertion language \mathcal{A} , e.g. it could include the μ -calculus [44]. \mathcal{D} can be the set of ordinals or a lexicographic combination of well-founded orders, for example.

4.4. Unfolding SCS's

Unfortunately, the **terminating SCS** transformation is not applicable to every terminating SCS, as illustrated by the following example.

EXAMPLE: Consider the transition system XYZ shown in Figure 6. Starting with $x > 0$, $y > 0$ and z equal to 0 or 1, this system repeatedly decreases x or y by 1, until $x = 0$ if $z = 0$, or until $y = 0$ if $z = 1$. At this point, the system moves to $loc = 2$. We want to prove $\diamond loc = 2$ for this system. Figure 7 shows the initial falsification diagram and the result of performing a split on the disjunction ($t = 1 \vee t = 2 \vee t = 3$), followed by the basic transformations. Clearly, the SCS $S = \{4, 5, 6\}$ is terminating. However, we cannot apply the terminating edge transformation, since (ignoring fairness) the loops (4, 6), and (5, 6) can each be taken infinitely many times, depending on the value of z . \square

To obtain completeness, we add one more transformation. The purpose of this transformation is to reduce a terminating SCS S with n nodes, in which no single edge can be proved to decrease the ranking function for every run, to a ring of n SCS's, S_1, \dots, S_n . Each S_i is a copy of S , with node i removed. If S is terminating,

$$\begin{aligned}
V &= \{x, y, z, t, loc\} \\
\Theta &= x > 0 \wedge y > 0 \wedge (z = 0 \vee z = 1) \wedge loc = 0 \wedge t = 0 \\
\mathcal{T} &= \{\tau_1, \tau_2, \tau_3, \tau_{idle}\} \text{ where} \\
\rho_{\tau_1} &= loc = 0 \wedge loc' = 1 \wedge x' = x - 1 \wedge t' = 1 \\
\rho_{\tau_2} &= loc = 0 \wedge loc' = 1 \wedge y' = y - 1 \wedge t' = 2 \\
\rho_{\tau_3} &= loc = 1 \wedge t' = 3 \wedge \\
&\quad ((z = 0 \wedge x = 0) \vee (z = 1 \wedge y = 0) \wedge loc' = 2) \vee \\
&\quad (\neg((z = 0 \wedge x = 0) \vee (z = 1 \wedge y = 0)) \wedge loc' = 0) \\
\mathcal{J} &= \{\tau_3\} \\
\mathcal{C} &= \{\tau_1, \tau_2\}
\end{aligned}$$

Figure 6. Fair transition system XYZ.

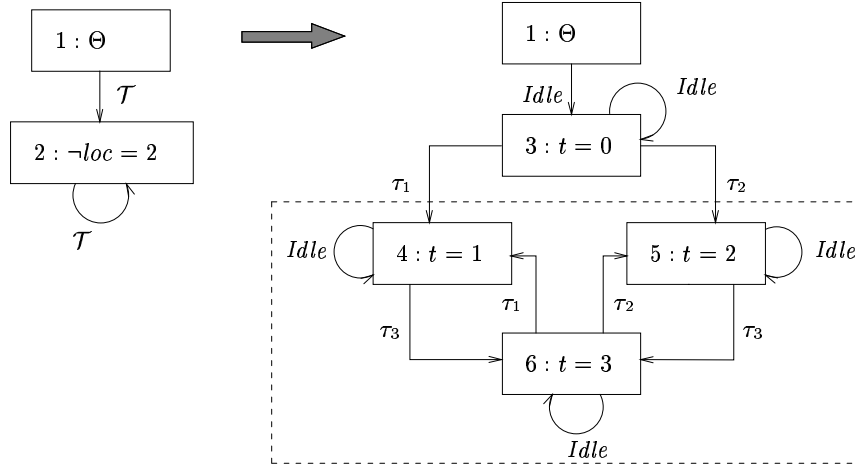


Figure 7. First two falsification diagrams for XYZ example.

then the edges constituting the ring can be taken only finitely many times. Thus, the **terminating SCS** transformation is applicable. When applied, it disconnects the ring and leaves n new SCS's on the SCS list, all of which are strictly smaller than S .

- **13 (unfold SCS).** Consider an SCS $S : \{N_1, \dots, N_n\}$. Remove S from the diagram and replace it with S_1, \dots, S_n , constructed as follows, where $N_i(S_j)$ refers to the node that is a copy of $N_i \in S$ and is part of S_j :
 1. S_1 is identical to S except that every edge $e : \langle N_k, N_1 \rangle \in S$ is replaced by an edge $e^* : \langle N_k(S_1), N_1(S_2) \rangle$. For all edges $e : \langle N_x, N_k \rangle$ where $N_x \notin S$

and $N_k \in S$, there exists an edge $e^* : \langle N_x, N_k(S_1) \rangle$, and for all edges $e : \langle N_k, N_x \rangle$, where $N_k \in S$ and $N_x \notin S$, there exists an edge $\langle N_k(S_1), N_x \rangle$.

2. $S_i, 1 < i \leq n$ is identical to S except that node N_i is removed along with every edge entering and leaving N_i , and for every edge $e : \langle N_k, N_i \rangle, 1 \leq k \leq n, k \neq i$, there is an edge $e^* : \langle N_k(S_i), N_i(S_{i \oplus 1}) \rangle$, where $i \oplus 1 = i + 1$ if $i < n$ and 1 otherwise. All edges $e : \langle N_x, N_k \rangle, N_x \notin S, N_k \in S$ are removed. For all edges $e : \langle N_k, N_x \rangle, N_k \in S, N_x \notin S$, there exists an edge $\langle N_k(S_i), N_x \rangle$.

The above transformation also replaces S on the SCS list with the single SCS containing S_1, \dots, S_n .

EXAMPLE: Figure 8 illustrates the application of the **unfold SCS** transformation to the terminating SCS $\{4, 5, 6\}$ of Figure 7. The result is a ring of three components: in the first component, node 6 was removed from the SCS and nodes 7 and 8 correspond to nodes 4 and 5 in Figure 7. In the second and third components, nodes 4 and 5 are removed, respectively. We can then apply the **terminating SCS** transformation to this ring, with edge $\langle 9, 11 \rangle$ as the terminating edge, as justified by the ranking function shown in Figure 8. It is easy to show that $\square((\text{if } z = 0 \text{ then } x \text{ else } y) \geq 0)$ is an invariant of system XYZ, so this ranking function is well-founded.

After thus breaking the ring, eliminating the remaining SCS's, namely $\{8\}, \{9, 10\}$ and $\{11, 12\}$, is straightforward using the fairness transformations. \square

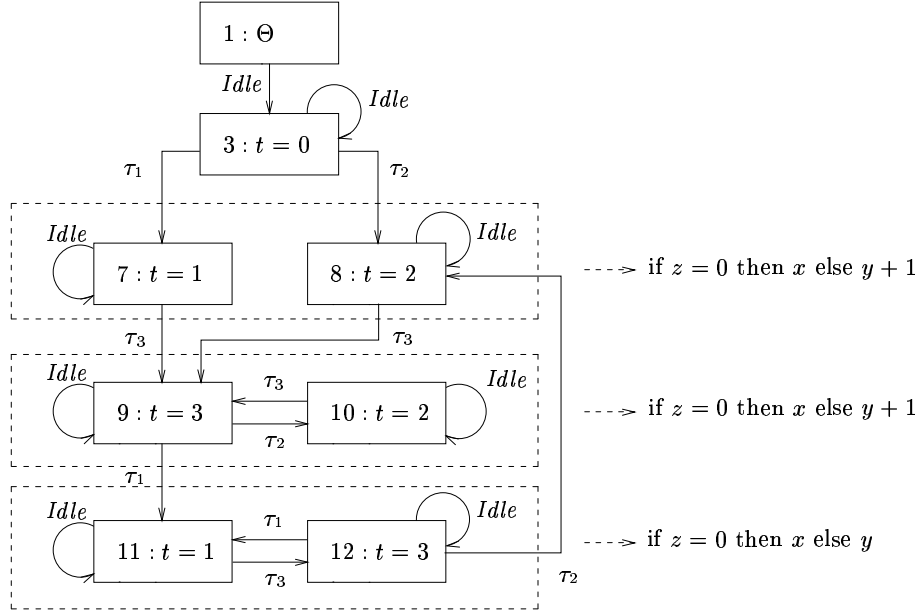


Figure 8. Result of the **unfold SCS** rule and ranking function for **terminating SCS** rule.

The **unfold SCS** and **terminating SCS** transformations can be combined, yielding transformations that use additional verification conditions and auxiliary assertions, and do not generate intermediate nodes that are to be discarded later. Such a transformation is presented in [20]. However, the two transformations given here are sufficient for completeness and facilitate the completeness proof itself (Section 6).

4.5. Obtaining Counterexamples

The process of transforming the falsification diagram can continue until there are no SCS's under consideration, in which case the original property φ is guaranteed to hold for the system \mathcal{S} .

In the case that \mathcal{S} does not satisfy φ , finding a counterexample computation requires some additional work. The above transformations remove from consideration SCS's that are known to be unreachable because they are disconnected from an initial node. However, no provisions ensure that a node is indeed reachable in an actual computation, or that a (infinite) computation can in fact reside indefinitely within an SCS. To identify portions of the behavior graph that are guaranteed to be reachable, we do some additional bookkeeping:

- **(executable transition).** Given an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ labeled with transition τ , mark τ as *executable* if the following formula is valid:

$$f_1(\vec{x}) \rightarrow \exists \vec{x}'. (\rho_\tau(\vec{x}, \vec{x}') \wedge f_2(\vec{x}')) .$$

That is, τ is labeled as executable if it can always be taken at states satisfying f_1 to reach a state that satisfies f_2 . For example, the idling transition can be marked as executable on all self-loops. (A stronger condition, the validity of $(f_1 \rightarrow \text{enabled}(\tau)) \wedge (\{f_1\} \tau \{f_2\})$, can be used if existential quantifiers are to be avoided.)

Definition 5. (fully just and compassionate) A transition is *fully taken* at an SCS if it is marked as executable for an edge in the SCS. An SCS S is *fully just* (resp. *fully compassionate*) if every just (resp. compassionate) transition is either fully taken in S or fully disabled at some node (resp. all nodes) in S .

If S is a fully just and compassionate SCS, a computation that resides indefinitely within S cannot violate any fairness requirements. If we can guarantee that such a computation exists, we have a counterexample:

Definition 6. (adequate SCS) An SCS S is *adequate* if after removing all edges not marked with executable transitions we obtain a subgraph S' where:

1. S' is still strongly connected;
2. S' is fully just and fully compassionate;

3. there is a path of executable transitions from a satisfiable initial node to a node in S' ;
4. the state-formulas in S' and the path that leads to S' are satisfiable.

An adequate SCS guarantees the existence of a computation of \mathcal{S} that satisfies $\neg\varphi$ (see Section 5).

4.6. Propagation and Strengthening

It is possible to obtain a reachable SCS that includes a counterexample computation, such that not all of its edges can be labeled with an executable transition. In this case, we are in danger of refining the SCS *ad infinitum* (e.g. with forwards propagation from an initial state), finding ever longer fragments of the counterexample computation. In such cases, the methodology presented by Bjørner, Browne and Manna [3] to automatically establish invariants can be used as a heuristic to strengthen a candidate SCS into an adequate one.

We say that S' *strengthens* an SCS S if S' can be obtained by replacing each node (A, f) in S by a node (A, f') such that f' implies f . By computing the greatest fixpoint of an appropriate monotonic operator defined on S based on weakest preconditions, we can find formulas ϕ_n for each node n such that (1) ϕ_n implies f_n for each node (A, f_n) , and (2) each transition τ in N can be marked as executable if we replace each f_n by ϕ_n . That is, the strengthened SCS S' is adequate, provided it is just and compassionate and one of its nodes is known to be reachable.

Note that justice and compassion for an SCS are preserved by strengthening, but reachability is not. To check if a strengthened node is reachable, backward propagation through the graph to reach an initial state can be carried out, using the weakest precondition operator.

To make fixpoint computations more feasible and efficient, different *abstraction domains* can be used, such as linear equations or convex polyhedra, based on abstract interpretation techniques [13]. While the above use of strengthening is meant to obtain a counterexample as a final step in model checking, abstract backward and forward propagation through the graph can *always* be used to soundly strengthen the formulas in the falsification diagram, whenever such propagation converges. Forward propagation can be used to strengthen the formulas while preserving the computations of \mathcal{S} , while backward propagation preserves those that satisfy $\neg\varphi$. Unlike the counterexample computation above, such strengthening can be done in a purely automatic and incremental fashion, interleaved with our transformations on the falsification diagram. See [3] for more details.

On the other hand, we may choose to approximate nodes by *weakening* them. Here, we replace an assertion f in a node by an assertion f' such that $f \rightarrow f'$ is valid. If the procedure terminates after finding no adequate SCS in the weakened graph, we are sure that the original property was \mathcal{S} -valid. However, a counterexample found in the weakened graph may not be an actual computation of the original

system or a model of $\neg\varphi$. (We may also choose to weaken a node if the particular representation of f becomes too expensive to maintain.)

5. Soundness

We now outline the proof of soundness for our deductive model checking procedure. It is easy to see that our requirements for a counterexample are sound: at any point, any fulfilling path through the falsification diagram is also a fulfilling path through the temporal tableau, and thus a model of $\neg\varphi$. That it is a computation of \mathcal{S} follows from the justice and compassion requirements on adequate SCS's, and by the executable transition labels, which ensure consecution.

Thus, we now show that if the procedure discards all SCS's, then the original property φ is \mathcal{S} -valid:

Definition 7. (computation follows \mathcal{G}) A computation s_0, s_1, \dots of \mathcal{S} follows a falsification diagram \mathcal{G} iff there is a path N_0, N_1, \dots in \mathcal{G} such that

1. N_0 is an initial node in \mathcal{G} ,
2. for each node $N_i = (A_i, f_i)$, s_i satisfies f_i , for all $i \geq 0$, and
3. (*consecution*) each edge $\langle N_i, N_{i+1} \rangle$ is labeled by a transition τ such that s_{i+1} is a τ -successor of s_i .

THEOREM 1 *Let \mathcal{G}_0 be the initial falsification diagram. Then any computation of \mathcal{S} that satisfies $\neg\varphi$ follows \mathcal{G}_0 .*

Proof: Any sequence that satisfies $\neg\varphi$ corresponds to a path in the underlying temporal tableau. If it is a computation of \mathcal{S} , then it must satisfy the initial condition; all other states are constrained only by the state-formulas in the tableau atom. Since it is a computation, the consecution requirement holds, since all edges in \mathcal{G}_0 are labeled by all transitions in \mathcal{S} . ■

THEOREM 2 *Let \mathcal{G}' be the result of applying any of the DMC transformations to a falsification diagram \mathcal{G} , and let c be a computation of \mathcal{S} . Then c follows \mathcal{G} if and only if c follows \mathcal{G}' .*

Proof: Of the basic transformations, only **remove edge label**, **empty edge**, **unsatisfiable node** and **unreachable node** change the underlying graph. But the conditions for removing an edge label, an edge, or a node ensure that the removed component could not have participated in any computation of \mathcal{S} that followed \mathcal{G} .

The remaining transformations are the three splitting rules. For these, any computation in \mathcal{G} that reaches the split node can be converted to a computation in \mathcal{G}' by choosing which of the two new nodes it satisfies. Conversely, any computation in \mathcal{G}' can be converted to a computation in \mathcal{G} by replacing any use of the two new nodes by the original one in \mathcal{G} . ■

Note that for the soundness of the procedure we only need the “only if” direction of Theorem 2. Approximation transformations can add new computations and still be useful for proving the \mathcal{S} -validity of φ , at the risk of reporting false counterexamples.

COROLLARY 1 *At any point of the procedure, any computation of \mathcal{S} that satisfies $\neg\varphi$ follows the current falsification diagram \mathcal{G}_i .*

Definition 8. (fully adequate) An SCS in a falsification diagram \mathcal{G} is *fully adequate* iff there is a computation of \mathcal{S} that follows \mathcal{G} that eventually remains within the nodes of the SCS.

Note that testing whether an SCS is fully adequate may be a non-trivial endeavor; our definition of an adequate SCS in Section 4.5 gives sufficient, but not necessary, criteria that can be established incrementally.

THEOREM 3 *There is a computation that follows \mathcal{G} if and only if \mathcal{G} has a fully adequate SCS.*

Proof: The “if” direction is part of the definition of fully adequate. For the “only if”, consider a computation that follows \mathcal{G} , and consider the nodes that occur infinitely often in \mathcal{G} . These must form an SCS. ■

THEOREM 4 *Any fully adequate SCS in the initial graph must be a subgraph of an SCS in the initial SCS list.*

Proof: A computation that follows the initial graph must be a model of $\neg\varphi$. If an SCS is fully adequate, then it must be an SCS in the underlying temporal tableau, and therefore a subset of one of its MSCS’s. ■

THEOREM 5 *Let \mathcal{G}, L be the graph and SCS list before a transformation is applied, and \mathcal{G}', L' be those afterwards. Then any fully adequate SCS in \mathcal{G} that is subgraph of an element of L corresponds to a fully adequate SCS in \mathcal{G}' that is a subgraph of an element of L' .*

Proof: If τ does not label an edge, then we can be sure that it is not taken when that edge is traversed, which justifies our definition of when a transition is not taken in an SCS.

Any adequate SCS must be fulfilling, just and compassionate. If an SCS is not fulfilling or just, none of its subgraphs is, so this justifies the **unfulfilling SCS** and **unjust SCS** rules. For the **uncompassionate SCS** rule, the removed nodes could not have been used infinitely often in a computation of \mathcal{S} , since the untaken transition is known to be always enabled at that point. Similarly, for the **terminating SCS** rule, the removed edge could not have been traversed infinitely often in a computation of \mathcal{S} .

The **SCS split** rule preserves the property since any SCS which is a subgraph of the split SCS must be a subgraph of one of its MSCS’s. The other basic rules

and the node-splitting rules preserve the computations that follow the graph and do not split an SCS.

The **unfold SCS** transformation is sound since it preserves the set of computations that follows the graph; that is, if a computation follows the original graph, then it also follows the graph after the SCS has been unfolded. This is clear from the construction of the unfolded SCS. ■

We can now conclude the following:

COROLLARY 2 *If the list of SCS's becomes empty, then no computation of S satisfies $\neg\varphi$.*

6. Completeness

With the addition of the **terminating SCS** and **unfold SCS** transformations, the DMC procedure is complete, as expressed by the following theorem:

THEOREM 6 *If S satisfies φ , then a falsification diagram whose SCS list is empty can be obtained from the initial falsification diagram by a finite number of transformations (possibly using strengthening with invariants and augmentation of the program).*

This completeness result is, as usual, relative to the underlying assertional reasoning. The proof is adapted from the completeness proof for generalized verification diagrams [7], and incorporates ideas from [32].

Definition 9. (Deterministic diagram) A falsification diagram \mathcal{G} is *deterministic* if any sequence of states follows at most one path in \mathcal{G} .

Before we prove the main theorem we present some supporting lemmas.

LEMMA 1 *Let S be an FTS and \mathcal{G} a falsification diagram. Then there exists a mapping acc from nodes to assertions such that $s \models acc(N)$ iff there exists a computation segment $\sigma : s_0, \dots, s_k = s$ of S and a path segment $\pi : N_0, \dots, N_k = N$ in \mathcal{G} such that $s_0 \models \Theta$, N_0 is an initial node, and σ follows π . That is, $acc(N)$ characterizes the states that are accessible at node N .*

Justification. Such an assertion can always be constructed in an assertion language that is sufficiently expressive to encode finite sequences [34].

THEOREM 7 (from [30]) *Let \succ be a well-founded ordering over a set W . Then there exists a function δ into the ordinals, $\delta : W \rightarrow \mathcal{O}$, such that:*

(W1) $w \succ w'$ implies $\delta(w) > \delta(w')$.

(W2) If $w \succ w''$ implies $w' \succ w''$ for every $w'' \in W$, then $\delta(w) \leq \delta(w')$.

LEMMA 2 *Let S be an FTS and S a reachable SCS in a deterministic falsification diagram, such that there is no run of S that follows a path that traverses edge e_t in S infinitely often. Then there exists a well-founded domain \mathcal{D} and a ranking function $\delta(n, s)$, mapping pairs of nodes and states into \mathcal{D} , such that:*

1. *for every edge $e = \langle N_1, N_2 \rangle = \langle (A_1, f_1), (A_2, f_2) \rangle$ and every $\tau \in e$,*

$$f_1(\vec{x}_1) \wedge f_2(\vec{x}_2) \wedge \rho_\tau(\vec{x}_1, \vec{x}_2) \rightarrow \delta(N_1, \vec{x}_1) \succeq \delta(N_2, \vec{x}_2) ;$$

2. *for edge $e_t = \langle N_1, N_2 \rangle = \langle (A_1, f_1), (A_2, f_2) \rangle$ and for all $\tau \in e_t$,*

$$f_1(\vec{x}_1) \wedge f_2(\vec{x}_2) \wedge \rho_\tau(\vec{x}_1, \vec{x}_2) \rightarrow \delta(N_1, \vec{x}_1) \succ \delta(N_2, \vec{x}_2) .$$

Proof: Let \succ be the order on $S \times \Sigma(\vec{x})$ given by $(N_a, \vec{x}_a) \succ (N_b, \vec{x}_b)$ iff there is a computation segment $\vec{x}_a, \dots, \vec{x}_b$ that follows a path N_a, \dots, N_b that traverses e_t .

\succeq is a **partial order**. It is clearly reflexive and transitive. To show that it is also antisymmetric, suppose that $(N_a, \vec{x}_a) \succeq (N_b, \vec{x}_b)$, $(N_b, \vec{x}_b) \succeq (N_a, \vec{x}_a)$ and $(N_a, \vec{x}_a) \neq (N_b, \vec{x}_b)$. By Lemma 1 there exists a computation segment s_0, \dots, \vec{x}_a that follows n_0, \dots, N_a . But then there exists an infinite run that follows a path

$$n_0, \dots, (N_a, \dots, N_b, \dots, N_{a-1})^\omega$$

that traverses e_t infinitely often, a contradiction.

\succ is **well-founded**. Suppose that $(N_a, \vec{x}_a) \succ (N_b, \vec{x}_b) \succ \dots$. Then, as before, we can construct a run of S (that goes through $\vec{x}_a, \vec{x}_b, \dots$) which follows a path that traverses e_t infinitely often.

By Theorem 7, there exists a function $\delta : S \rightarrow \mathcal{O}$ that satisfies W1 and W2. We have to show that δ satisfies conditions 1 and 2.

(1) Suppose $\langle N_1, N_2 \rangle \in S$ and $\rho_\tau(\vec{x}_1, \vec{x}_2) \wedge f_1(\vec{x}_1) \wedge f_2(\vec{x}_2)$. For any (N_c, \vec{x}_c) , if $(N_2, \vec{x}_2) \succ (N_c, \vec{x}_c)$ then also $(N_1, \vec{x}_1) \succ (N_c, \vec{x}_c)$, since if the path N_2, \dots, N_c traverses e_t then the path N_1, \dots, N_c also traverses e_t . Thus, by W2, $\delta(N_1, \vec{x}_1) \geq \delta(N_2, \vec{x}_2)$.

(2) Suppose $e_t = \langle N_1, N_2 \rangle$ and $f_1(\vec{x}_1) \wedge f_2(\vec{x}_2) \wedge \rho_\tau(\vec{x}_1, \vec{x}_2)$ holds. Then $(N_1, \vec{x}_1) \succ (N_2, \vec{x}_2)$, and thus by W1,

$$f_1(\vec{x}_1) \wedge f_2(\vec{x}_2) \wedge \rho_\tau(\vec{x}_1, \vec{x}_2) \rightarrow \delta(N_1, \vec{x}_1) \succ \delta(N_2, \vec{x}_2) . \quad \blacksquare$$

LEMMA 3 *Every terminating SCS S on the list can be reduced to a set of SCS's on the list that are all strictly smaller than S .*

Justification. Suppose SCS S is terminating, i.e. there is no run that visits all nodes of S infinitely often. We consider two cases: (1) If S has a terminating edge, by Lemma 2 there exists a well-founded domain \mathcal{D} and a ranking function $\delta(n, s)$ that justify the application of the **terminating SCS** transformation, resulting in the replacement of S by strict subgraphs of S on the list. (2) If S has no terminating edge, the application of **unfold SCS** to S results in an SCS S' that consists of a

unidirectional ring of SCS's, each of which is a copy of S minus one node, such that the size of each constituent SCS S_i is at least one less than the size of the original SCS. By construction, a path that follows the entire ring must visit every node in the original SCS S . Therefore, none of the edges connecting the S_i 's can be taken infinitely often, and hence they can be removed by the terminating edge transformation, and S can be replaced by the strictly smaller SCS's S_i .

Finally, we can now outline the proof of the completeness claim:

Proof of Theorem 6: For an FTS S and property φ , assume $S \models \varphi$. From an initial falsification diagram \mathcal{G}_0 construct a falsification diagram as follows:

1. Apply enabled splits (followed by basic transformations) to each node of the diagram to obtain a diagram such that every node of the initial diagram is split into $2^{|\mathcal{T}|}$ nodes, one node for each subset of transitions $T \subseteq \mathcal{T}$ and labeled by

$$\bigwedge_{\tau \in T} \text{enabled}(\tau) \wedge \bigwedge_{\tau \in \mathcal{T} - T} \neg \text{enabled}(\tau) .$$

2. Apply postcondition splits to obtain a diagram in which each edge is labeled by exactly one transition. This assumes that for two transitions τ_1, τ_2 , $\text{post}(\tau_1) \wedge \text{post}(\tau_2)$ is unsatisfiable. This may be obtained by augmenting the original transition system with a new variable a and adding to each transition the conjunct $a' = i_\tau$, where i_τ is unique for every transition τ . (Clearly, this augmentation will not affect the validity of the property to be proved). Note that the resulting diagram has the property that each node can be reached by only one transition.
3. Strengthen every node of the diagram with $\text{acc}(N)$.
4. Remove all SCS's that are not fulfilling from the list.
5. Repeat application of unfold to terminating SCS's (followed by application of terminating edge) and removal of unfair SCS's (possibly decomposing the remaining part of an uncompassionate or terminating SCS into new MSCS's) until no SCS's can be removed.

Claim. The SCS list for the resulting falsification diagram is empty.

Justification. Suppose not, i.e., there exists an SCS S that is fair and not terminating. From nontermination we can conclude that there exists a run of S that follows a path π that visits every node in S infinitely often. Since $S \models \varphi$, this run cannot be fair. Suppose it is not fair with respect to some just transition τ , that is, τ is always enabled beyond a certain point but never taken. This means that τ cannot label any edge in S , since the run visits every node in S , and by the property that each node is reached by only one transition, every transition must be taken in order to reach every node. But if τ is always enabled beyond a certain point then it must be fully enabled at every node of the SCS (because of the way

the enabled splits were done). But then the SCS is not just and would have been removed. A similar argument can be given for a compassionate transition.

Note that as with most completeness results, the above is not intended as a guideline to be used in the construction of all proofs, but rather as an indication of what machinery is sufficient to obtain them in the worst case.

7. Analysis

As presented, the model checking procedure constructs the entire formula tableau before the exploration of the state space begins. The procedure can be made incremental in the tableau construction itself. The tableau Φ_φ can be exponential in the size of φ ; however, properties to be model checked are usually simple, so the tableau is small when compared with the system's state space (even for finite-state systems).

We can use *encapsulation conventions* such as those used in verification diagrams [33, 24]: if f_1 implies f_2 , for two nodes $(A_1, f_1), (A_2, f_2)$, then we can make them subnodes (A_1, f_1) and $(A_2, true)$ of a larger node labeled with f_2 .

In the case of a temporal safety property $\varphi : \Box p$, the only fulfilling SCS in the tableau for $\neg\varphi$ is a trivial one, labeled with *true*, which follows $\neg p$ -states. The verification question then reduces to reachability, where fairness constraints are not relevant. The basic transformations (1 through 8), together with strengthening with auxiliary assertions, are sufficient in this case (the equivalent of the *assertion graph* of [3] can be built). If φ is an invariance, the trivial tableau for $\neg\varphi$ is not of much help; however, for general safety properties (where p is a past formula), the tableau does provide greater constraints on the state-space to be explored.

For progress properties, where fairness is relevant, we can incrementally identify transitions that are known to be enabled and disabled at each node. Similarly, we can incrementally mark the transitions that are known to be executable at each edge. Some of this work (but not all) may have to be repeated after each split.

PROPOSITION 2 *For a finite-state system \mathcal{S} , the exhaustive application of transformations 1–11 terminates, deciding the \mathcal{S} -validity of φ .*

Proof: If \mathcal{S} is finite-state, we can use a finite-state assertion language \mathcal{A} with normal forms (such as that provided by OBDDs). The satisfiability tests required at the splitting steps are now all decidable, and there will only be a finite number of distinct nodes. Since every transformation reduces the size of the graph or replaces a node with more specific nodes (that is, nodes covering strictly fewer states), the process must terminate. If the SCS list is empty, the original property φ is \mathcal{S} -valid; otherwise, any remaining SCS must be adequate, and thus provide a counterexample. ■

In this case, the formulas can be encoded using OBDDs [9] or any other finite-domain constraint language. Representations that combine OBDDs with other constructs can be used if the OBDD size is problematic.

In the general case of infinite-state systems, the model checking problem is undecidable. Indeed, we are faced with a potentially undecidable satisfiability check when applying many of the DMC transformations. However, the satisfiability test used in the DMC rules does not have to be complete. The available theorem-proving and simplification techniques are not required to give a definite answer at any given time: if the validity of a formula is hard to decide, additional splits can make subsequent questions easier. (This was, in effect, the approach taken in our example, where formulas were only known to be unsatisfiable when STeP's simplification mechanisms reduced them to *false*.)

The specialized constraint languages used in Constraint Logic Programming [26] are usually designed to facilitate such an incremental satisfiability test. Concurrent constraint programs are reactive programs based on such constraint languages [41], so they may be particularly amenable to our verification framework. We expect constraint-solving and propagation techniques to play a central role in the deductive model checking of large systems, which is yet to be explored in practice.

Deductive Support

The DMC procedure benefits from user guidance in two forms (apart from providing ranking functions). First, the choice of which refinement transformation to perform next determines how the state space is explored. Second, the process can be sped up considerably by refinement steps based on auxiliary formulas provided by the user, using the general node splitting described in Section 4.1. However, many of the steps can be performed automatically, with user guidance required only when the size of the falsification diagram becomes a problem. Decision procedures for particular domains, such as those used in the PVS [37] and STeP [2] systems, are very useful in this setting.

Partial Results

Even when the model checking effort is not completed, the resulting falsification diagram can be used to restrict the search for a counterexample. As is clear from the soundness proof of Section 5, at any given point the falsification diagram contains all the possible counterexample computations to the property being checked. The system can be executed (or simulated) following only paths through the falsification diagram, in the search for a computation that violates the property. Backward propagation (possibly approximated—see Section 4.6) can be used to find a smaller set of initial states that can generate a counterexample computation. A similar approach is used in [11] to generate test cases for processor designs.

Acknowledgments

We thank Nikolaj Bjørner, Anca Browne, Michael Colón, Luca de Alfaro, Arjun Kapur and the anonymous reviewers for their comments. We thank Luca de Alfaro for pointing out a flaw in an earlier version of the completeness proof.

References

1. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Proc. 10th IEEE Symp. Logic in Comp. Sci.*, pages 388–397, 1995.
2. N.S. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. 8th Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 415–418. Springer-Verlag, July 1996.
3. N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comp. Sci.*, 173(1):49–87, February 1997.
4. N.S. Bjørner, M.E. Stickel, and T.E. Uribe. A practical combination of first-order reasoning and decision procedures. In *14th Intl. Conference on Automated Deduction*, volume 1249 of *LNCS*, pages 101–115. Springer-Verlag, July 1997.
5. A. Bouajjani, J-C. Fernandez, and N. Halbwegs. Minimal model generation. In *Proc. 2nd Intl. Conference on Computer Aided Verification*, volume 531 of *LNCS*, pages 197–203, 1990.
6. J. C. Bradfield. *Verifying Temporal Properties of Systems*. Birkhäuser, 1992.
7. A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.
8. A. Browne, Z. Manna, and H.B. Sipma. Hierarchical verification using verification diagrams. In *2nd Asian Computing Science Conf.*, volume 1179 of *LNCS*, pages 276–286. Springer-Verlag, December 1996.
9. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
10. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, pages 428–439. IEEE Computer Society Press, 1990.
11. A.K. Chandra, V.S. Iyengar, R.V. Jawalekar, M.P. Mullen, I. Nair, and B.K. Rosen. Architectural verification of processors using symbolic instruction graphs. In *Intl. Conference on Computer Design: VLSI in Computers and Processors*, pages 454–459. IEEE Press, 1994.
12. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. Princ. of Prog. Lang.*, pages 238–252. ACM Press, 1977.
14. W. Damm, O. Grumberg, and H. Hungar. What if model checking must be truly symbolic. In *First Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 95)*, volume 1019 of *LNCS*, pages 230–244. Springer-Verlag, May 1995.
15. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
16. D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, Eindhoven, July 1996.
17. D.R. Dams, R. Gerth, G. Döhmen, R. Herrmann, P. Kelb, and H. Pargmann. Model checking using adaptive state and data abstraction. In *Proc. 6th Intl. Conference on Computer Aided Verification*, volume 818 of *LNCS*, pages 455–467. Springer-Verlag, June 1994.
18. D.R. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proc. 5th Intl. Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 479–490. Springer-Verlag, June 1993.

19. L. de Alfaro and Z. Manna. Temporal verification by diagram transformations. In *Proc. 8th Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 287–299, July 1996.
20. L. de Alfaro, Z. Manna, H.B. Sipma, and T.E. Uribe. Visual verification of reactive systems. In *Third Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 97)*, volume 1217 of *LNCS*, pages 334–350. Springer-Verlag, April 1997.
21. J. Dingel and T. Filkorn. Model checking of infinite-state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc. 7th Intl. Conference on Computer Aided Verification*, volume 939 of *LNCS*, pages 54–69, July 1995.
22. L. Fix and O. Grümberg. Verification of temporal properties. *J. Logic and Computation*, 6(3):343–362, 1996.
23. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th Intl. Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.
24. D. Harel. Statecharts: A visual approach to complex systems. Technical Report CS84-05, Dept. of Applied Mathematics, Weizmann Institute of Science, 1984.
25. H. Hungar. Combining model checking and theorem proving to verify parallel processes. In *Proc. 5th Intl. Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 154–165. Springer-Verlag, 1993.
26. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, pages 111–119, January 1987.
27. Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proc. 5th Intl. Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 97–109. Springer-Verlag, 1993.
28. R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proc. 5th Intl. Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 166–179. Springer-Verlag, 1993.
29. L. Lamport. A new solution of Dijkstra’s concurrent programming problem. *Communications of the ACM*, 17(8):435–455, 1974.
30. D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. 8th Int. Colloq. Aut. Lang. Prog.*, volume 115 of *LNCS*, pages 264–277. Springer-Verlag, 1981.
31. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.
32. Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.
33. Z. Manna and A. Pnueli. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.
34. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
35. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Pub., 1993.
36. J.S. Ostroff. A visual toolset for the design of real-time discrete event systems. *IEEE Trans. on Control Systems Technology*, May 1997.
37. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. 8th Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 411–414. Springer-Verlag, July 1996.
38. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *Proc. 8th Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 184–195. Springer-Verlag, 1996.
39. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Intl. Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.
40. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In *Proc. 7th Intl. Conference on Computer Aided Verification*, volume 939 of *LNCS*, pages 84–97, July 1995.

41. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
42. H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *Proc. 8th Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 208–219. Springer-Verlag, 1996.
43. O.V. Sokolsky and S.A. Smolka. Local model checking for real-time systems. In *Proc. 7th Intl. Conference on Computer Aided Verification*, volume 939 of *LNCS*, pages 211–224, July 1995.
44. F.A. Stomp, W.-P. de Roever, and R.T. Gerth. The μ -calculus as an assertion language for fairness arguments. *Inf. and Comp.*, 82:278–322, 1989.