Model Checking and Deduction for Infinite-State Systems

Henny B. Sipma, Tomás E. Uribe, Zohar Manna
Computer Science Department
Stanford University
Stanford, CA. 94305
`sipma|uribe|manna@CS.Stanford.EDU` *

Abstract

Two well-known approaches to the verification of reactive systems are *deductive verification* and *model checking*. After briefly reviewing them, we present *deductive model checking*, which combines these two approaches. The new procedure uses deduction to extend the classical tableau-based model checking algorithms to the case of infinite-state systems.

1 Introduction

Reactive systems are those with an ongoing interaction with their environment. While the relationship between any two given system states can be expressed in first-order logic or a suitable assertion language, properties of non-terminating computations, which are infinite sequences of states, are best expressed using some form of temporal logic.

The two main approaches to verifying temporal properties of reactive systems are *deductive verification*, on the one hand, and *model checking* on the other. In the deductive approach, the validity of a given temporal property over a given program is reduced to the general validity of a set of first-order formulas, through the use of a given set of *verification rules*. In *model checking*, the state space of the program is systematically explored in search for a computation that violates the property being verified, until such a counterexample is found or it is shown that it cannot exist.

Model checking procedures are usually automatic, while deductive verification often relies on user interaction to identify suitable lemmas and auxiliary assertions. However, model checking is usually applicable only to systems with a finite, fixed number of states, while the deductive approach can verify infinite-state systems and parameterized finite-state systems of arbitrary size.

*This research was supported in part by the National Science Foundation under grant CCR-92-23226, the Advanced Research Projects Agency under NASA grant NAG2-892, the United States Air Force Office of Scientific Research under grant F49620-93-1-0139, the Department of the Army under grant DAAH04-95-1-0317, and a gift from Intel Corporation.

After presenting the basic concepts and notation in Section 2, we present a brief overview of model checking (Section 3) and deductive verification (Section 4). We then present *deductive model checking* (Sipma et al. 1996), which attempts to combine the two approaches into a single procedure that retains some of their complementary strengths. We discuss this procedure in Section 6, and present related work in Section 7.

2 Preliminaries

2.1 Fair Transition Systems

To carry out formal verification, we need a precise mathematical model of the systems being analyzed. For this, we use *fair transition systems* (Manna and Pnueli 1991) as our computational model for reactive systems. A fair transition system (FTS) $\mathcal{S}$ is a triple $\langle \mathcal{V}, \Theta, \mathcal{T} \rangle$, where $\mathcal{V}$ is a set of variables, Θ is an *initial condition*, and $\mathcal{T}$ is a finite set of *transitions*. A finite set of *system variables* $\vec{x} \subset \mathcal{V}$ determines the possible states of the system. The *state space*, Σ, is the set of all possible valuations of the system variables, each defining a *system state*.

Fair transition systems allow the description of a wide range of systems, including hardware and software, using the full expressive power of first-order logic. We use a first-order assertion language $\mathcal{A}$ to describe the initial condition Θ and the transitions in $\mathcal{T}$. This assertion language can be augmented with interpreted function symbols and constraints, or specialized to the finite-state case, depending on the the system being specified—see Section 3. A formula over the system variables V describes the set of states for which it is true; if a state s satisfies the formula φ, we say that s is a *φ-state*. The initial condition Θ is such an assertion over $\vec{x}$, describing the set of *initial states*. A transition τ is described by a *transition relation* $\rho_\tau(\vec{x}, \vec{x}')$. This is an assertion over the set of system variables $\vec{x}$, indicating their values at a given state, and a set of *primed variables* $\vec{x}'$, which indicates their values at the next state. $\mathcal{T}$ includes an *idling transition*, *Idle*, whose transition relation is $\vec{x} = \vec{x}'$.

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

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

For a formula φ and a transition τ, we define the *weakest precondition* $pre(\tau, \varphi)$ (Dijkstra 1975) and the *strongest postcondition* $post(\tau, \varphi)$ as follows:

$$post(\tau, \varphi) \stackrel{\text{def}}{=} \exists \vec{x}_0 .\, (\rho_\tau(\vec{x}_0, \vec{x}) \wedge \varphi(\vec{x}_0))$$
$$pre(\tau, \varphi) \stackrel{\text{def}}{=} \forall \vec{x}' .\, (\rho_\tau(\vec{x}, \vec{x}') \rightarrow \varphi(\vec{x}')) \ .$$

Intuitively, the weakest precondition of φ with respect to transition τ describes the states that must reach a φ-state after τ is taken. (This is sometimes called the *weakest liberal precondition*, since it is true of those states where τ cannot be taken at all.) Similarly, the strongest postcondition of φ with respect to transition τ describes the set of states that can be reached by taking transition τ from a state that satisfies φ; if τ cannot be taken, then the postcondition is the formula *false*. We also use the *Hoare triple* notation,

$$\{\varphi\}\, \tau\, \{\psi\} \stackrel{\text{def}}{=} (\varphi(\vec{x}) \wedge \rho_\tau(\vec{x}, \vec{x}')) \rightarrow \psi(\vec{x'}) \ .$$

This formula is valid if taking transition τ from a φ-state always leads to a state that satisfies ψ. Note that it holds if τ is not enabled at φ-states.

In a *parameterized* transition system, the set of transitions is indexed by one or more parameters (usually an integer i in a range $1 \ldots N$), whose range can be underspecified; in this way, systems with an arbitrary number of similar processes can be analyzed (Manna and Pnueli 1990).

Fairness

Modelling concurrency with interleaving enables a simple encoding of non-determinism, and abstracts away the relative speed of different processes. However, one then wants to exclude computations where a given process or non-deterministic action is forever ignored (the process never advances or an action never occurs even though it is allowed). To this effect, 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. Note that the presence of the idling transition ensures that all computation prefixes can be extended to an infinite computation (even though there are no fairness requirements associated with *Idle*).

Example 1 Figure 1 shows a program that implements Lamport's (1974) *bakery algorithm* for mutual exclusion. Each of the statements in the program corresponds to a transition, denoted by its label; thus, $\mathcal{T} = \{Idle, \ell_0..\ell_4, m_0..m_4\}$. 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; 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*, namely, that each process can enter its critical section at most once before the other process does.

The STeP (Stanford Temporal Prover) verification system translates programs such as BAKERY into the corresponding fair transition systems (Bjørner et al. 1996). 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. In Section 5, we show how accessibility can be checked for this program.

$$
\begin{array}{l}
\textbf{local} \quad y_1, y_2 \quad : \textbf{integer where } y_1 = y_2 = 0 \\[1ex]
\left[\begin{array}{l}
\textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_0\text{: } \textbf{noncritical} \\
\ell_1\text{: } y_1 := y_2 + 1 \\
\ell_2\text{: } \textbf{await } (y_2 = 0 \vee y_1 \leq y_2) \\
\ell_3\text{: } \textbf{critical} \\
\ell_4\text{: } y_1 := 0
\end{array}\right]
\end{array}\right] \\
\| \\
\left[\begin{array}{l}
\textbf{loop forever do} \\
\quad \left[\begin{array}{l}
m_0\text{: } \textbf{noncritical} \\
m_1\text{: } y_2 := y_1 + 1 \\
m_2\text{: } \textbf{await } (y_1 = 0 \vee y_2 < y_1) \\
m_3\text{: } \textbf{critical} \\
m_4\text{: } y_2 := 0
\end{array}\right]
\end{array}\right]
\end{array}
$$

FIGURE 1 Program BAKERY for mutual exclusion

2.2 Temporal Logic

As first pointed out by Pnueli (1977), temporal logic is a convenient formalism for specifying and verifying properties of reactive systems. A formula in temporal logic describes a set of infinite sequences, also known as a *temporal property*. A given system satisfies a property if all of its computations belong to this set (see Section 3.2).

A model of *linear-time temporal logic* (LTL) is an infinite sequence of states where each point in time has a unique successor. We use the *anchored* version of LTL, where the truth-value of a formula for a given model is evaluated at the initial position of the model.

In the model checking procedure of Section 5, we will use LTL over the assertion language $\mathcal{A}$ as our specification language. Given a model $s = s_0, s_1, \ldots$, the *future* temporal operators are defined as follows:

$\Box p$ holds at state s_j iff p is true for all states s_i, $i \geq j$;

$\Diamond p$ holds at state s_j iff p is true for some state s_i, $i \geq j$;
$\bigcirc p$ holds at state s_j iff p holds at state $j+1$;
$p\,\mathcal{U}\,q$ holds at state s_j iff for some $k \geq j$, q holds at state s_k, and p holds at every state s_i, $j \leq i < k$.
$p\mathcal{W}q$ holds at state s_j iff $(p\,\mathcal{U}\,q)$ or $\Box p$ hold at state s_j.

Past versions of these temporal operators are similarly defined. For simplicity, we will restrict our notation and examples to the future fragment of LTL. We do not allow temporal operators to appear within the scope of a quantifier, so all quantification can be thought of as *rigid*. (System variables are also called *flexible* variables, as their values can change over time; however, we only quantify over system variables to identify particular classes of states). A formula containing no temporal operators is called a *state-formula* or an *assertion*.

Example 2 Mutual exclusion for the BAKERY program of Figure 1 can be expressed with the LTL formula

$$\Box(\neg(\ell_3 \wedge m_3)) \ .$$

Accessibility is expressed with the LTL formula

$$\varphi : \Box(\ell_1 \rightarrow \Diamond \ell_3) \ .$$

That is, it is always the case that if control is at ℓ_1, then it will eventually reach ℓ_3. 1-bounded overtaking is expressed with the property

$$\Box(\ell_3 \rightarrow \neg m_4 \ \mathcal{W} \ m_4 \ \mathcal{W} \ \neg m_4 \ \mathcal{W} \ l_4) \ .$$

In *branching-time temporal logic*, points in time can have multiple successors, understood as possible future states. Thus, branching-time models are infinite trees, rather than the linear sequences of LTL models. The logic CTL (*computation tree logic*) of Clarke and Emerson (1981) is interpreted over such models, using the *path quantifiers* **E** and **A**, indicating that the *path formula* that follows holds for some or all the branches that start at a given state. Path formulas correspond to LTL formulas, with the operators **X**,**F** and **G** corresponding to $\bigcirc$,$\Diamond$ and $\Box$ of LTL. Thus, CTL includes modal operators such as $\mathbf{AX}p$ (p is true in all possible next states), $\mathbf{EF}p$ (p is true in a future state of some branch), and $\mathbf{EG}p$ (p is globally true for some branch). The logic CTL* (Emerson and Halpern 1986) combines the branching-time and linear-time operators, subsuming both LTL and CTL.

A detailed exposition of linear-time temporal logic as a specification language for fair transition systems is found in (Manna and Pnueli 1991). Emerson (1990) presents a survey of branching and linear-time temporal logic.

3 Model Checking

The state space of a transition system can be thought of as a *Kripke structure*, where the accessibility relation corresponds to transitions being taken from one state to another. Verifying that a temporal logic formula is valid for a program is equivalent to checking that the formula holds of the given Kripke structure (also called the *model*).

In a *finite-state system*, all system variables have finite domains, so the state space (and Kripke structure) is finite. Model checking for finite-state systems was introduced by Clarke and Emerson (1981) and independently by Queille and Sifakis (1982). The classic model checking algorithms for branching-time logic recursively label each system state with the formulae they satisfy, starting with the atomic propositions occurring in the subformula φ in question and analyzing more complex subformulas of φ at each step. For example, once the states that satisfy a subformula f have been identified, the states that satisfy $\mathbf{EG}(f)$ (namely, the ones where f is always true in some possible computation) are identified by finding all the strongly connected components of the state space where f is always true that are reachable by a path where f also holds (see (Clarke and Grümberg 1987)).

"On-the-fly" model checking algorithms, such as the one for CTL* presented in (Bhat et al. 1995), construct only the portion of the state space required by the given formula, rather than labeling the entire state space.

3.1 Symbolic Model Checking

For a finite-state system $\mathcal{S}$, the complexity of model checking depends on the size of the formula being checked (linear for CTL and exponential for LTL and CTL*) and the size of the state space of $\mathcal{S}$ (linear for all three). While the temporal formulas of interest are usually small, the size of the state space can grow exponentially in the size of its description, say, as a fair transition system. This is known as the *state explosion problem*, where the number of possible states limits the practical applicability of model checking algorithms.

Symbolic model checking combats the state-explosion problem by using a specialized data structure to describe sets of states, rather than explicitly enumerate them. Ordered Binary Decision Diagrams (OBDDs), introduced by Bryant (1986), are an efficient data structure for representing boolean functions and relations. McMillan (1993) used OBDDs to represent the transition relation of finite-state systems, as well as subsets of the system's state space. The efficient algorithms for manipulating OBDD's (Brace et al. 1990) can be used to compute predicate transformations, such as pre- and post-condition operations, over the transition relation and large, implicitly represented sets of states.

The resulting model checking algorithms can be described in terms of the *propositional μ-calculus* (Park 1974, Kozen 1983), which expresses fixpoint relations over finite domains. The temporal operators can be given a fixpoint characterization; for instance, $\mathbf{EX}p$ can be seen as the least fixed point of the function $f(y) = p \vee \mathbf{EX}y$ (Burch et al. 1990). (Berezin et al. 1996) is a recent survey of model checking for the μ-calculus.

Symbolic model checking algorithms for CTL can be adapted to LTL as well (Clarke et al. 1994).

Symbolic model checking extends the size of finite-state systems that can be analyzed, and is particularly successful for hardware systems. However, it is still restricted to finite-state systems of fixed size; thus, for instance, a system may be model checked for 128 bits, but not for 256. The size of the OBDD representation of particular functions can grow exponentially in the number of boolean variables (Bryant 1991), leading to what can be called the *OBDD explosion problem*, where the model checker can run out of memory before the user runs out of time. In these cases, efficient explicit-state model checkers are preferred (Dill 1996). To further extend the range of symbolic model checking, extensions of BDDs are being used to move from bit-level to word-level representations (Clarke et al. 1995).

Partial order reductions (Godefroid and Wolper 1991, Peled 1994) are used to avoid the state explosion that arises from the interleaving of independent transitions. The "partial model checking" of Andersen (1995), performs consecutive transformations of the system and its μ-calculus specification. Other approaches to combat the state explosion for finite-state systems include exploiting *symmetry* in the state space (Clarke et al. 1993, Emerson and Sistla 1993) and *abstraction*, which is discussed in Section 7.1.

3.2 The Automata-Theoretic Point of View

An alternate approach to model checking was introduced by Kurshan (1986, 1994) and Vardi and Wolper (1986). The system $\mathcal{S}$ can be identified with an ω-automaton that recognizes the language $\mathcal{L}(\mathcal{S})$, the set of all its computations. Similarly, the property to be verified can be expressed as an automaton φ. The language $\mathcal{L}(\varphi)$ is the set of all (infinite) computations accepted by φ. To prove that φ holds for S, we need to show that $\mathcal{L}(\varphi) \subseteq \mathcal{L}(\mathcal{S})$. If both φ and $\mathcal{S}$ can be expressed as finite automata, this can be decided by purely algorithmic means.

In the case of LTL, the automata of interest are over infinite words; for CTL and CTL*, automata over infinite trees can be used (Kupferman 1995). For a survey on such automata, see (Thomas 1990).

3.2.1 The Formula Tableau

Given an LTL formula φ, we can construct its *tableau* Φ_φ, a finite graph that describes all of its models; this graph is equivalent to an ω-automaton for φ. The nodes in the tableau are labeled by *atoms*, which are sets of assertions and temporal formulas. 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 *initial atom* is one whose formulas can hold at the initial state of a model; in particular, it should contain the formula φ. Thus, the atom labeling a node contains the state- and temporal formulas that should hold whenever a model of φ reaches this node.[1]

φ is satisfiable only if there is a *strongly connected subgraph* (SCS) in Φ_φ that is reachable from an initial atom. Furthermore, if a given model satisfies, e.g., $\Diamond q$ or $p\,\mathcal{U}\,q$ at some point, it must in fact satisfy q at this or another point later on. A *fulfilling* SCS is one where all such eventualities are satisfied.

Proposition 1 *φ is satisfiable if and only if there is a fulfilling, reachable SCS in Φ_φ.*

Example 3 Figure 2 shows a tableau for $\psi : \Diamond(\ell_1 \wedge \Box\neg\ell_3)$. Nodes 1 and 2 are initial, since they contain the formula φ. The only fulfilling SCS is $\{3\}$; the SCS $\{1\}$ is not fulfilling, since it contains an eventuality, φ, which is not satisfied in that SCS.

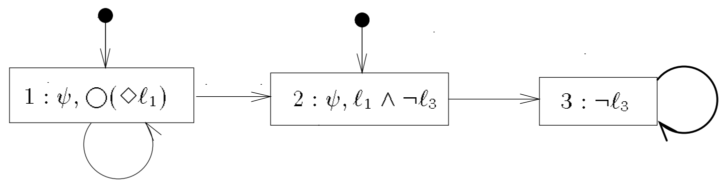

FIGURE 2 Tableau for $\psi : \Diamond(\ell_1 \wedge \Box\neg\ell_3)$.

For details on LTL tableau construction, see, e.g. (Kesten et al. 1993) or (Manna and Pnueli 1995).[2] The atomic formulas in a tableau are those appearing in φ; they are usually treated as propositional constants. However, we will combine all of the state-formulas in a given atom into a single formula f of the assertion language $\mathcal{A}$, which is always simplified as much as possible.

3.2.2 The Product Graph

One way to show that $\mathcal{L}(\varphi) \subseteq \mathcal{L}(\mathcal{S})$ is to show that $\mathcal{L}(\neg\varphi) \cap \mathcal{L}(\mathcal{S})$ is empty. This can be done by constructing the *product graph* between the system's state space graph (the Kripke structure) and the temporal tableau for $\neg\varphi$.

Definition 1 (product graph) Given a tableau $\Phi_{\neg\varphi}$ and a system $\mathcal{S}$, the *product graph* is a directed graph whose edges are labeled with pairs (A, s), for an atom A and a state s, such that:

1. For each node (A, s), state s satisfies the state-formulas in A.

[1] The set of atoms relevant to a given formula is sometimes called the *Fischer-Ladner closure* of the formula, after Fischer and Ladner (1979), who defined it as part of a decision procedure for propositional dynamic logic.

[2] The tableau of Figure 2 is a *particle tableau*, following a construction that minimizes the number of atoms that need to be considered.

2. There is an edge from (A, s) to (A', s') if and only if s' is a τ-successor of state s for some transition $\tau \in \mathcal{T}$, and A' is a successor of A in $\Phi_{\neg\varphi}$.

If an infinite path through the product graph satisfies the fairness constraints on the transitions and is fulfilling with respect to its tableau atoms, then it is both a model of $\neg\varphi$ and a computation of $\mathcal{S}$, and thus a counterexample to φ. On the other hand, if it can be proved that no such path exists, then we can conclude that φ holds for $\mathcal{S}$.

We should note that the size of the formula tableau $\Phi_{\neg\varphi}$ can be exponential in the size of φ; however, specifications are relatively small, and the size of the state space is the main obstacle in practice. *Incremental* and *particle tableau* constructions (Manna and Pnueli 1995) reduce the expense of building $\Phi_{\neg\varphi}$. If necessary, the construction of $\Phi_{\neg\varphi}$ can be interleaved with the enumeration of the state space.

The product graph is finite if $\mathcal{S}$ is a finite-state system. However, if the state space of $\mathcal{S}$ is very large or infinite, the explicit construction of the product graph is not feasible. In Section 5, we present an extension of this procedure that makes it applicable to infinite-state systems as well.

4 Deductive Verification

The classical deductive framework for the verification of fair transition systems is based on *verification rules*, which reduce temporal properties of systems to first-order premises (Manna and Pnueli 1995).

$$
\begin{array}{ll}
 & \text{For assertions } \psi \text{ and } p, \\
1. & \psi \rightarrow p \\
2. & \Theta \rightarrow \psi \\
3. & \text{For all transitions } \tau \in \mathcal{T}, \\
 & \{\psi\}\ \tau\ \{\psi\} \\
\hline
 & \Box p
\end{array}
$$

FIGURE 3 General invariance rule.

Figure 3 presents the *general invariance rule*, which can be used to establish formulas of the form $\Box p$ for an assertion p. The premises of the rule are a set of first-order (non-temporal) *verification conditions*; if they can be proved to be generally valid, the temporal conclusion must hold for the system $\mathcal{S}$.

The invariance rule relies on finding an *inductive* auxiliary assertion ψ that strengthens p; the invariant p may not be inductive (that is, it may not be preserved by all transitions), but ψ should be. This rule is *relatively complete*, in the sense that if p is an invariant of $\mathcal{S}$, then the strengthened assertion ψ always exists (Manna and Pnueli 1995). However, note that while model checking for finite-state systems is decidable, verifying temporal properties of infinite-state systems is undecidable in general.

Other verification rules can be used to verify different classes of temporal formulas, ranging from *safety* to *progress* properties. These rules are implemented, for example, in the STeP verification system (Bjørner et al. 1996). Fix and Grumberg (1996), present a complete deductive system for the temporal verification of branching-time temporal properties. Bradfield (1992) presents a deductive proof system for the modal μ-calculus.

These deductive methods are relatively complete and yield a direct proof of any valid property, but can require substantial user guidance to succeed and do not produce counterexample computations when the property fails.

4.1 Graphical Formalisms

Graphical formalisms can facilitate the task of guiding and understanding a deductive proof. *Verification diagrams* (Manna and Pnueli 1994, Browne et al. 1995) provide a graphical representation of the verification conditions needed to establish a particular temporal formula. These have been extended in (Browne et al. 1996) to verify properties expressible by arbitrary ω-automata over infinite strings[3]. In Section 5, we present *deductive model checking*, in which a diagram is constructed to show that the system does *not* satisfy the negation of the property to be proved. A third diagram-based formalism is presented by De Alfaro and Manna (1996), combining the direct and indirect proof approaches. Here, a graphical representation of the transition system is transformed incrementally until a representation is obtained that can be shown to satisfy the specification by algorithmic means. General verification conditions must be proved to justify each transformation step.

(Browne et al. 1996) summarizes the common features of verification diagrams, fairness diagrams, and deductive model checking. They all can describe and verify infinite-state systems using a finite representation which can be incrementally constructed, where verification conditions are local and where global properties can be checked algorithmically.

5 Deductive Model Checking

We now present an alternate approach to the verification of temporal properties of reactive systems, *deductive model checking*, closely following our presentation in (Sipma et al. 1996).

Instead of building the $(\neg\varphi, \mathcal{S})$ product graph explicitly, we will start with a general skeleton of the product 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 (Sipma et al. 1996), this graph is called the $\mathcal{S}$-*refined tableau*.)

[3] Properties expressible in LTL are a strict subset of those expressible by ω-automata.

Definition 2 (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 in $\mathcal{S}$. 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 product 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 $(A_0, f_0), (A_1, f_1), \ldots$ through $\mathcal{G}$, the *underlying path* $A_0, A_1, \ldots$ will be a path in $\Phi_{\neg\varphi}$.

5.1 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 $N_0 = (A, f \wedge \Theta)$, which has 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 4 Figure 4 presents an example of an initial falsification diagram to prove accessibility for the Bakery algorithm of Figure 1. This diagram is based on the particle tableau for $\neg\varphi : \Diamond(\ell_1 \wedge \Box\neg\ell_3)$ shown in Figure 2. 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 originating in node 3 is immediately pruned since $\ell_1 \wedge \Theta$ is unsatisfiable. The SCS $\{4\}$ is not fulfilling, but $\{2\}$ is. The examples in this section show how this property is verified with a DMC implementation based on the STeP system (Bjørner et al. 1996).

The main data structure maintained by the procedure is a falsification diagram $\mathcal{G}_i$ and a list 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 $\mathcal{G}_0$.

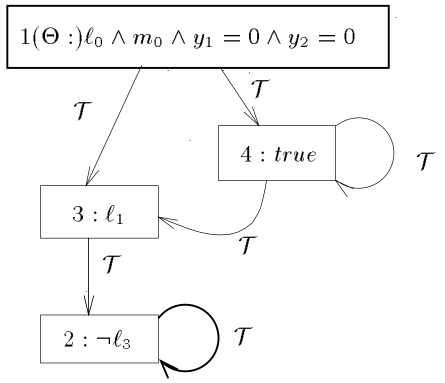

FIGURE 4 Initial falsification diagram for BAKERY accessibility

Deductive model checking proceeds by repeatedly applying one of transformations 1–11 described below. The process stops if we find a *reachable*, *fulfilling* and *adequate* SCS (see Section 5.3), 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 a satisfiability check for formulas in the assertion language, they can all be easily automated (see Section 6.1).

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 to N_2, $\langle N_1, N_2\rangle = \langle (A_1, f_1), (A_2, f_2)\rangle$, whose label includes transition τ. If $f_2 \wedge \neg \mathit{post}(\tau, f_1)$ is satisfiable (that is, f_2 does not imply $\mathit{post}(\tau, f_1)$), then replace (A_2, f_2) by the two nodes

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

  We can immediately apply the **remove 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 $f_1 \wedge \neg(\mathit{enabled}(\tau) \wedge \mathit{pre}(\tau, f_2))$ is satisfiable, then replace (A_1, f_1) by the two nodes

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

  Here, transition τ can be removed from the $\langle N_{1,2}, N_2\rangle$ edge.

Example 5 Figure 5 presents the result of applying a 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.

The conditions for applying these transformations can be weakened if the required satisfiability checks are too expensive (see Section 6). 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 $\phi_1, \ldots, \phi_n$ are assertions, one can split a node $\langle A, f\rangle$ into the $n+1$ nodes

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

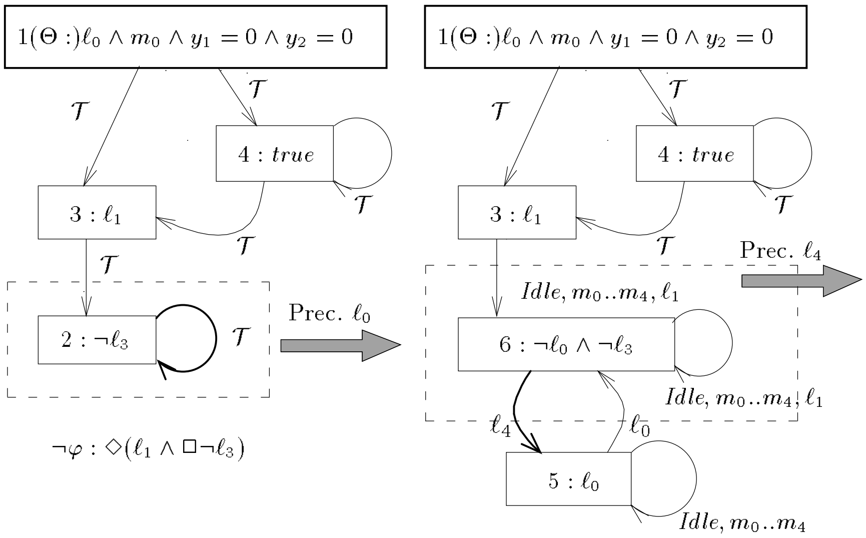

FIGURE 5 First splitting step for BAKERY accessibility

$\phi_1, \ldots, \phi_n$ could be the set of relevant control locations at a given point, and the n-ary split a shortcut for a sequence of (binary) precondition splits. However, 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 (A, f) by the node $(A, f \wedge p)$. This use of invariants can also be very valuable in finite-state symbolic model checking (Pnueli and Shahar 1996). 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.)

5.2 Fairness Transformations

Together, transformations 1-8 are sufficient for the analysis of transition systems with no fairness requirements. If an *adequate* SCS is found (see Section 5.3), 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 $f \wedge \neg enabled(\tau)$ is satisfiable. Then replace N by the two nodes

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

Definition 3 A transition τ is *fully enabled* at a node (A, f) if $f \rightarrow enabled(\tau)$ is valid; τ is *fully disabled* at a node (A, f) if $f \rightarrow (\neg enabled(\tau))$ 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 subsets are not just; but an SCS that is not compassionate may have a compassionate subset. This leads us to the last two transformations, which, like the basic ones, should be applied whenever possible:

- **10 (uncompassionate SCS).** If an SCS S is not compassionate, then let $\tau_1, \ldots, \tau_n$ 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.
- **11 (unjust SCS).** If an SCS is not just, remove it from the SCS list.

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

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

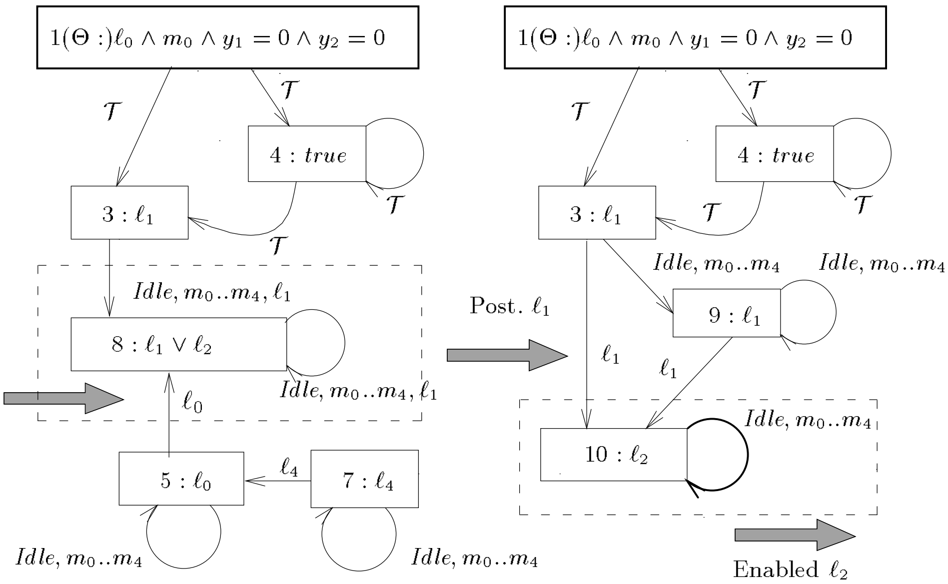

FIGURE 6 Second and third splitting steps for BAKERY accessibility

5.3 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 φ fails, 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 an infinite computation can in fact reside indefinitely within an SCS. To identify portions of the product 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:

$$(f_1 \rightarrow enabled(\tau)) \wedge (\{f_1\}\, \tau \,\{f_2\}) \quad .$$

That is, τ is labeled as executable if it can be taken at all states satisfying f_1 and always reaches a state that satisfies f_2. For example, the idling transition can be marked as executable on all self-loops.

Definition 4 (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 an SCS S is fully just and compassionate, then we can be sure that a computation that resides indefinitely within S will not violate any fairness requirements. If we can guarantee that such a computation exists, then we have a counterexample:

Definition 5 (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 $\neg\varphi$.

5.4 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, in what follows we show that if the procedure discards all SCS's, then the original property φ is $\mathcal{S}$-valid:

Definition 6 (computation follows $\mathcal{G}$) A computation $s_0, s_1, \ldots$ of $\mathcal{S}$ *follows* a falsification diagram $\mathcal{G}$ iff there is a path $N_0, N_1, \ldots$ 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 2 *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 3 *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 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 ones. 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 3. 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 4 *At any point of the procedure, any computation of $\mathcal{S}$ that satisfies $\neg\varphi$ follows the current falsification diagram $\mathcal{G}_i$.*

Definition 7 (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 that an SCS is fully adequate may be a non-trivial endeavour; our definition of an adequate SCS in Section 5.3 gives sufficient, but not necessary, criteria that can be established incrementally.

Theorem 5 *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 6 *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 $\neg\varphi$; for the SCS to be fully adequate, then, it must be an SCS in the underlying temporal tableau, and therefore a subset of one of its MSCS's. □

Theorem 7 *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 always be enabled at that point.

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 apart an SCS. □

Corollary 8 *If the list of SCS's becomes empty, then no computation of $\mathcal{S}$ satisfies $\neg\varphi$.*

Example 7 Figure 7 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 $(y_2 \neq 0) \rightarrow (m_2 \vee m_3 \vee m_4)$.[4] 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. Now, SCS $\{16\}$ is unjust for m_4, while $\{15\}$ is unjust for m_2. Since there are no candidate SCS's left, we have established that φ is $\mathcal{S}$-valid.

5.5 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*, finding ever longer fragments of the counterexample computation. In such cases, the methodology presented by Bjørner, Browne and Manna (1995) 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,

[4] Such invariants are generated automatically by STeP based on the program text.

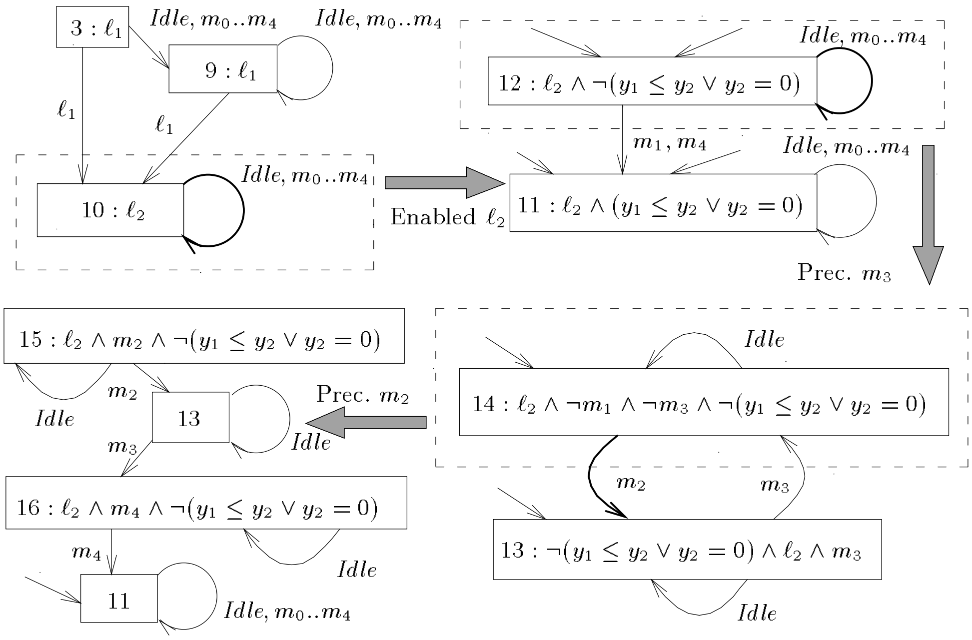

FIGURE 7 Last splitting steps for BAKERY accessibility

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 (see Section 7.1). 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 incremental fashion, and interleaved with our transformations on the falsification diagram. See (Bjørner et al. 1995) 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.)

6 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).

Proposition 9 *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 $\mathcal{A}$. 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. □

The formulas could be encoded using BDDs or any other finite-domain constraint language. Representations that combine BDDs with other constructs can be used if the BDD 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.

The specialized constraint languages used in Constraint Logic Programming (Jaffar and Lassez 1987) are usually designed to facilitate such an incremental satisfiability test. Concurrent constraint programs are reactive programs based on such constraint languages (Saraswat 1993), 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.

6.1 Deductive Support

The DMC procedure can benefit from user guidance in two forms: first, the choice of refinement transformation to perform next determines how the state space is explored. Second, the process can be speeded up considerably by refinement steps based on auxiliary formulas provided by the user. 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 (Owre et al. 1996) and STeP (Bjørner et al. 1996) systems are very useful in this setting.

6.2 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 5.4, at any given point the falsification diagram contains all the possible counterexample computations to the property being checked. Backward propagation (possibly approximated—see Section 5.5) can be used to find the set of initial states that can generate a counterexample computation. A similar approach is used in (Chandra et al. 1994) to generate test cases for processor designs.

7 Related Work

7.1 Abstraction

One way to model check a property φ for a large system $\mathcal{S}$ is to find a simpler system $\mathcal{S}_A$ such that if φ is valid for $\mathcal{S}_A$ then it is also valid for $\mathcal{S}$ (Long 1993). Using *abstract interpretation*, one can require the abstract program $\mathcal{S}_A$ to operate over a domain where the property can be efficiently checked. As a simple example, one could want to reduce an infinite-state program to a finite-state one.

In (Loiseaux et al. 1995), property-preserving abstractions for the μ-calculus are investigated. Dams, Grumberg and Gerth (1993, 1994) present an algorithm for producing abstracted state spaces that preserve various fragments of CTL*. A quotient of the state space is constructed, splitting equivalence classes (sets of states) according to the value of different subformulas. The goal is to obtain an abstracted state space that preserves a given set of formulas; the abstraction is maximized if this set contains only the property to be model checked. (Dams et al. 1994) shows how this quotient construction can be combined with abstract interpretation. An exceedingly clear and complete description of this line of research is (Dams 1996).

One can think of the DMC procedure as combining the abstraction and model checking phases within a general first-order setting. For instance, it may be possible to do enough splitting and simplification that the model checking of an infinite-state system is reduced to a finite set of finite-state model checking tasks, to which fully automatic methods can be applied.

Example 8 An example of a finite-state abstraction for the Bakery program is presented in Figure 8 The correspondence between the BAKERY program and the abstracted version is as follows:

BAKERY	ABSTRACT
$y_1 = 0$	t_1
$y_2 = 0$	t_2
$y_1 \leq y_2$	t_3

Accessibility, mutual exclusion and 1-bounded overtaking for the abstracted program can be easily verified using a finite-state model checker.

$$\textbf{local}\quad t_1, t_2, t_3\quad : \textbf{boolean where } t_1, t_2, t_3$$

$$\left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}\ell_0\text{: } \textbf{noncritical}\\ \ell_1\text{: } (t_1, t_3) := (\mathrm{F}, \mathrm{F})\\ \ell_2\text{: } \textbf{await } (t_2 \vee t_3)\\ \ell_3\text{: } \textbf{critical}\\ \ell_4\text{: } (t_1, t_3) := (\mathrm{T}, \mathrm{T})\end{array}\right]\end{array}\right] \quad || \quad \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}m_0\text{: } \textbf{noncritical}\\ m_1\text{: } (t_2, t_3) := (\mathrm{F}, \mathrm{T})\\ m_2\text{: } \textbf{await } (t_1 \vee \neg t_3)\\ m_3\text{: } \textbf{critical}\\ m_4\text{: } (t_2, t_3) := (\mathrm{T}, t_1)\end{array}\right]\end{array}\right]$$

FIGURE 8 A finite-state abstraction of the BAKERY program

A similar method for generating an abstract representation of a possibly infinite state space is presented in (Bouajjani et al. 1990). However, this is done independently of any particular formula to be verified.

Real-time systems are always infinite-state. The local model checking algorithm for real-time systems presented by Sokolsky and Smolka (1995) can be seen as a specialized variant of our procedure. It refines a finite representation of an infinite product graph, splitting nodes to satisfy constraints arising from the formula and system being checked.

7.2 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 (1993) and Kurshan and Lamport (1993) use (deductive) modular decomposition to reduce the correctness of a large system to that of smaller components that can be model checked. Abstraction is used in (Rajan et al. 1995) and

(Dingel and Filkorn 1995) to obtain subgoals that can be model checked, while the correctness of the abstraction is proved deductively.

The PVS system (Owre et al. 1996) includes a decision procedure to model check propositional μ-calculus expressions, which can appear as subgoals in a verification effort. The STeP system (Bjørner et al. 1996) uses deductive verification rules, but can also model check subgoals over the given system specification whenever possible. Pnueli and Shahar (1996) show how deductively obtained invariants can be used to constrain the BDD-based symbolic model checking process.

Acknowledgements: We thank Nikolaj Bjørner, Anca Browne, Luca de Alfaro and Arjun Kapur for their comments.

Bibliography

Andersen, H. 1995. Partial model checking. In *Proc. 10th IEEE Symp. Logic in Comp. Sci.*, 398–406.

Berezin, S., E. Clarke, S. Jha, and W. Marrero. 1996. Model checking algorithms for the μ-calculus. Technical Report CMU TR CMU-CS-96-180, School of Computer Science, Carnegie-Mellon University, September.

Bhat, G., R. Cleaveland, and O. Grumberg. 1995. Efficient on-the-fly model checking for CTL*. In *Proc. 10th IEEE Symp. Logic in Comp. Sci.*, 388–397.

Bjørner, N., A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. 1996. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, Vol. 1102 of *LNCS*, 415–418. Springer-Verlag, July.

Bjørner, N., A. Browne, and Z. Manna. 1995. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, Vol. 976 of *LNCS*, 589–623. Springer-Verlag, September.

Bouajjani, A., J.-C. Fernandez, and N. Halbwachs. 1990. Minimal model generation. In *Proc. 2^{nd} Intl. Conference on Computer Aided Verification*, Vol. 531 of *LNCS*, 197–203.

Brace, K. S., R. L. Rudell, and R. E. Bryant. 1990. Efficient implementation of a BDD package. In *Proc. of the 27^{th} ACM/IEEE Design Automation Conference*, 40–45.

Bradfield, J. C. 1992. *Verifying Temporal Properties of Systems.* Birkhäuser.

Browne, A., L. de Alfaro, Z. Manna, H. Sipma, and T. E. Uribe. 1996. Diagram-based formalisms for the verification of reactive systems. In *CADE-14 Workshop on Visual Reasoning.*

Browne, A., Z. Manna, and H. Sipma. 1995. Generalized verification diagrams. In *15th Conference on the Foundations of Software Technology* *and Theoretical Computer Science*, Vol. 1026 of *LNCS*, 484–498, December.

Browne, A., Z. Manna, and H. Sipma. 1996. Hierarchical verification using verification diagrams. In *Asian Computing Science*, LNCS. Springer-Verlag, December. To appear.

Bryant, R. E. 1991. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers* 40(2):205–213.

Bryant, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35(8):677–691.

Burch, J., E. Clarke, K. McMillan, D. Dill, and L. Hwang. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, 428–439. IEEE Computer Society Press.

Chandra, A., V. Iyengar, R. Jawalekar, M. Mullen, I. Nair, and B. Rosen. 1994. Architectural verification of processors using symbolic instruction graphs. In *International Conference on Computer Design: VLSI in Computers and Processors*, 454–459. IEEE Press.

Clarke, E., and E. Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, Vol. 131 of *LNCS*, 52–71. Springer-Verlag.

Clarke, E., T. Filkorn, and S. Jha. 1993. Exploiting symmetry in temporal logic model checking. In *Proc. 5^{th} Intl. Conference on Computer Aided Verification*, Vol. 697 of *LNCS*, 450–462. Springer-Verlag, June.

Clarke, E., M. Fujita, and X. Zhao. 1995. Hybrid decision diagrams. Overcoming the limitations of MTBDDs and BMDs. In *IEEE/ACM International Conference on Computer-Aided Design*, 159–163, November.

Clarke, E., and O. Grümberg. 1987. Research on automatic verification of finite-state concurrent systems. In J. Traub, B. Grosz, B. Lampson, and N. Nilsson (Eds.), *Annual Review of Computer Science*, Vol. II, 269–290. Annual Reviews.

Clarke, E., O. Grumberg, and K. Hamaguchi. 1994. Another look at LTL model checking. In D. L. Dill (Ed.), *Proc. 6^{th} Intl. Conference on Computer Aided Verification*, Vol. 818 of *LNCS*, 415–427. Springer-Verlag.

Dams, D. 1996. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, Eindhoven, July.

Dams, D., R. Gerth, G. Döhmen, R. Herrmann, P. Kelb, and H. Pargmann. 1994. Model checking using adaptive state and data abstraction. In *Proc. 6^{th} Intl. Conference on Computer Aided Verification*, Vol. 818 of *LNCS*, 455–467. Springer-Verlag, June.

Dams, D., O. Grumberg, and R. Gerth. 1993. Generation of reduced models for checking fragments of CTL. In *Proc. 5^{th} Intl. Conference on Com-*

puter Aided Verification, Vol. 697 of *LNCS*, 479–490. Springer-Verlag, June.

Dams, D., O. Grümberg, and R. Gerth. 1994. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$ECTL*, CTL*. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*, 573–592, June.

de Alfaro, L., and Z. Manna. 1996. Temporal verification by diagram transformations. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, Vol. 1102 of *LNCS*, 287–299, July.

Dijkstra, E. 1975. Guarded commands, nondeterminancy, and formal derivation of programs. *Communications of the ACM* 18(8):453–457.

Dill, D. 1996. The Murφ verification system. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, Vol. 1102 of *LNCS*, 390–393. Springer-Verlag, July.

Dingel, J., and T. Filkorn. 1995. Model checking of infinite-state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc. 7^{th} Intl. Conference on Computer Aided Verification*, Vol. 939 of *LNCS*, 54–69.

Emerson, E., and A. Sistla. 1993. Symmetry and model checking. In *Proc. 5^{th} Intl. Conference on Computer Aided Verification*, Vol. 697 of *LNCS*, 463–478. Springer-Verlag, June.

Emerson, E. 1990. Temporal and modal logic. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Vol. B, 995–1072. Elsevier Science Publishers (North-Holland).

Emerson, E., and J. Halpern. 1986. 'Sometimes' and 'not never' revisited: On branching time versus linear time. *J. ACM* 33:151–178.

Fischer, M., and R. Ladner. 1979. Propositional dynamic logic of regular programs. *J. Comp. Sys. Sci.* 18:194–211.

Fix, L., and O. Grumberg. 1996. Verification of temporal properties. *J. Logic and Computation* 6(3):343–362.

Godefroid, P., and P. Wolper. 1991. A partial approach to model checking. In *Proc. 6th IEEE Symp. Logic in Comp. Sci.*, 406–415. IEEE Computer Society Press.

Hungar, H. 1993. Combining model checking and theorem proving to verify parallel processes. In *Proc. 5^{th} Intl. Conference on Computer Aided Verification*, Vol. 697 of *LNCS*, 154–165. Springer-Verlag.

Jaffar, J., and J.-L. Lassez. 1987. Constraint logic programming. In *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, 111–119, January.

Kesten, Y., Z. Manna, H. McGuire, and A. Pnueli. 1993. A decision algorithm for full propositional temporal logic. In *Proc. 5^{th} Intl. Conference on Computer Aided Verification*, Vol. 697 of *LNCS*, 97–109. Springer-Verlag.

Kozen, D. 1983. Results on the propositional μ-calculus. *Theor. Comp. Sci.* 27(3):333–354.

Kupferman, O. 1995. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, Technion—Israel Institute of Technology, Haifa, Israel, June.

Kurshan, R., and L. Lamport. 1993. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis (Ed.), *Proc. 5^{th} Intl. Conference on Computer Aided Verification*, Vol. 697 of *LNCS*, 166–179. Springer-Verlag.

Kurshan, R. P. 1986. Testing containment of ω-regular languages. Technical Report 1121-861010-33, Bell Labs.

Kurshan, R. P. 1994. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press.

Lamport, L. 1974. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM* 17(8):435–455.

Loiseaux, C., S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. 1995. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design* 6:1–35.

Long, D. E. 1993. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July.

Manna, Z., and A. Pnueli. 1990. An exercise in the verification of multi-process programs. In W. Feijen, A. van Gasteren, D. Gries, and J. Misra (Eds.), *Beauty is Our Business*, 289–301. Springer-Verlag.

Manna, Z., and A. Pnueli. 1991. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York: Springer-Verlag.

Manna, Z., and A. Pnueli. 1994. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software*, Vol. 789 of *LNCS*, 726–765. Springer-Verlag.

Manna, Z., and A. Pnueli. 1995. *Temporal Verification of Reactive Systems: Safety*. New York: Springer-Verlag.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Pub.

Owre, S., S. Rajan, J. Rushby, N. Shankar, and M. Srivas. 1996. PVS: Combining specification, proof checking, and model checking. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, Vol. 1102 of *LNCS*, 411–414. Springer-Verlag, July.

Park, D. 1974. Finiteness is μ-ineffable. Theory of Computation Report 3, University of Warwick.

Peled, D. 1994. Combining partial order reductions with on-the-fly model-checking. In *Proc. 6^{th} Intl. Conference on Computer Aided Verification*, Vol. 818 of *LNCS*, 377–390.

Pnueli, A. 1977. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, 46–57. IEEE Computer Society Press.

Pnueli, A., and E. Shahar. 1996. A platform for combining deductive with algorithmic verification. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, Vol. 1102 of *LNCS*, 184–195. Springer-Verlag.

Queille, J., and J. Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari (Eds.), *Intl. Symposium on Programming*, Vol. 137 of *LNCS*, 337–351. Springer-Verlag.

Rajan, S., N. Shankar, and M. Srivas. 1995. An integration of model checking with automated proof checking. In *Proc. 7^{th} Intl. Conference on Computer Aided Verification*, Vol. 939 of *LNCS*, 84–97.

Saraswat, V. A. 1993. *Concurrent Constraint Programming*. MIT Press.

Sipma, H., T. Uribe, and Z. Manna. 1996. Deductive model checking. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, Vol. 1102 of *LNCS*, 208–219. Springer-Verlag, July.

Sokolsky, O. V., and S. A. Smolka. 1995. Local model checking for real-time systems. In *Proc. 7^{th} Intl. Conference on Computer Aided Verification*, Vol. 939 of *LNCS*, 211–224.

Thomas, W. 1990. Automata on infinite objects. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Vol. B, 133–191. Elsevier Science Publishers (North-Holland).

Vardi, M., and P. Wolper. 1986. Automata-theoretic techniques for modal logics of programs. *J. Comp. Sys. Sci.* 32:183–221.

Index