In 8th *International Conference on Computer Aided Verification*
Vol. 1102 of Lecture Notes in Computer Science, pp. 208–219
Springer-Verlag, July 1996.

Deductive Model Checking*

Henny B. Sipma, Tomás E. Uribe, Zohar Manna

Computer Science Department
Stanford University
Stanford, CA. 94305
sipma|uribe|manna@CS.Stanford.EDU

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 and general framework that can be used to integrate a diverse number of other verification tools.

1 Introduction

We present a model checking procedure for verifying temporal logic properties of general 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 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 state transition graph for $\mathcal{S}$ and the temporal tableau for $\neg\varphi$, 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 product graph can be eliminated long before they are fully expanded to the state level. For instance, in the verification of accessibility for the Peterson mutual-exclusion algorithm, expansion to 12 nodes suffices to demonstrate that no counterexample exists, whereas the full behavior graph contains 76 nodes.

* 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.

For infinite-state systems, the procedure will terminate in many cases. In Section 5 we illustrate the procedure by model checking an accessibility property for the Bakery algorithm. Expansion to 16 nodes suffices to verify this property 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 that can be used for further verification or testing.

We present our procedure in the framework of [14], 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.

Related work

Model Checking: Most approaches to temporal logic model checking [10, 16] 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 [8, 15]. 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 are limited by the size of the specialized representation, which is still ultimately limited by the number of reachable states.

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 [4], using partitioning operations similar to the ones we describe below. However, in [4] this is done independently of any particular formula to be verified. Finally, the local model checking algorithm for real-time systems in [18] can be seen as a specialized variant of our procedure; it too refines a finite representation of an infinite product graph, consecutively splitting nodes to satisfy constraints arising from the formula and system being checked.

Deductive Methods: A complete deductive system for temporal verification of branching-time properties is presented in [11], while [5] presents a proof system for the modal mu-calculus. Manna and Pnueli [14] 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* [13, 6] 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.

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.

The procedure presented in [3] for automatically establishing temporal *safety* properties is based on an *assertion graph* similar to the $\mathcal{S}$*-refined tableau* 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 $\neg\varphi$.

2 Preliminaries

Fair Transition Systems: The computational model, following [14], 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[2] assertion language $\mathcal{A}$ to describe Θ and the transitions in $\mathcal{T}$. Θ is an assertion over the system variables V. A transition τ is described by a *transition relation* $\rho_\tau(\mathbf{x}, \mathbf{x}')$, an assertion over the set of system variables $\mathbf{x}$ and a set of *primed variables* $\mathbf{x}'$ indicating their values at the next state. $\mathcal{T}$ includes an *idling transition*, *Idle*, whose transition relation is $\mathbf{x} = \mathbf{x}'$.

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 $\rho_\tau(s_i, s_{i+1})$ 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 with the formula

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

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

$$\begin{aligned} post(\tau, \varphi) &\stackrel{\text{def}}{=} \exists \mathbf{x}_0. \left(\rho_\tau(\mathbf{x}_0, \mathbf{x}) \wedge \varphi(\mathbf{x}_0)\right) \\ pre(\tau, \varphi) &\stackrel{\text{def}}{=} \forall \mathbf{x}'. \left(\rho_\tau(\mathbf{x}, \mathbf{x}') \rightarrow \varphi(\mathbf{x}')\right) \ . \end{aligned}$$

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

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

[2] Although it can be augmented with features such as interpreted symbols and constraints, or specialized to the finite-state case, e.g. using BDDs.

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.

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. We use the usual future and past temporal operators, such as $\Box, \Diamond, \bigcirc, \mathcal{U}, \mathcal{W}$ (future) and $\boxminus, \diamondminus, \ominus, \mathcal{B}, \mathcal{S}$ (past). A formula with no temporal operators is called a *state-formula* or an *assertion*. For details on LTL and tableau constructions, we refer the reader to [14], 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 [14]. Briefly, each node in the tableau is identified 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) in Φ_φ that is reachable from an initial atom. Furthermore, if a given 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.*

3 Deductive Model Checking

The classical approach to model checking [10, 16] verifies a property φ by constructing the *product graph* between the system's reachable-state graph and the temporal tableau for $\neg\varphi$. Any infinite path through the product graph that satisfies the fairness constraints on the transitions and is fulfilling with respect to its tableau atoms is a counterexample to φ.

The explicit construction of the state-graph restricts the method to finite-state systems. The procedure we present works in a top-down fashion, starting with a general skeleton of the product graph and refining it until a counterexample is found, or the impossibility of such a counterexample is demonstrated.

Definition 2 ($\mathcal{S}$-refined tableau). Given an FTS $\mathcal{S}$ and a temporal property φ, an *$\mathcal{S}$-refined tableau* is a directed graph $\mathcal{G}$ whose nodes are labeled with pairs (A, f), where A is an atom for the temporal tableau for $\neg\varphi$ and f is a state-formula, and whose edges are labeled with subsets of $\mathcal{T}$. For nodes M, 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*.

The $\mathcal{S}$-refined tableau can be viewed 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 labeling 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 an $\mathcal{S}$-refined tableau 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 $\Phi_{\neg\varphi}$.

3.1 The DMC Procedure

We begin with the tableau graph $\Phi_{\neg\varphi}$, from which we construct an initial $\mathcal{S}$-refined tableau $\mathcal{G}_0$ as follows:

1. Replace each node label A by (A, f_A), where f_A is the conjunction of the state-formulas in 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, and 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 $\mathcal{S}$-refined tableau.
3. Label each edge in $\mathcal{G}_0$ with the entire set of transitions $\mathcal{T}$.

Figure 2 in Section 5 presents an example of an initial $\mathcal{S}$-refined tableau.

The main data structure maintained by the procedure is an $\mathcal{S}$-refined tableau graph $\mathcal{G}_i$ and a list of strongly connected subgraphs 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 1–11 described below. The process stops if we find a *reachable*, *fulfilling* and *adequate* SCS (see Section 3.3) or obtain an empty SCS list.

Basic Transformations:

- **1 (remove label).** If an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ is labeled with a transition τ and $f_1(\mathbf{x}) \wedge f_2(\mathbf{x}') \wedge \rho_\tau(\mathbf{x}, \mathbf{x}')$ 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.

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 $f_2 \wedge \neg post(\tau, f_1)$ is satisfiable (that is, f_2 does not imply $post(\tau, f_1)$), then replace (A_2, f_2) by the two nodes

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

  Note that 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, only now the self-loop for $N_{2,2}$ as well as the edge from $N_{2,1}$ to $N_{2,2}$ do 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 $f_1 \wedge \neg(enabled(\tau) \wedge pre(\tau, f_2))$ is satisfiable, then replace (A_1, f_1) by the two nodes

$$N_{1,1} = (A_1, f_1 \wedge enabled(\tau) \wedge pre(\tau, f_2))$$
$$N_{1,2} = (A_1, f_1 \wedge \neg(enabled(\tau) \wedge pre(\tau, f_2))) \quad .$$

The conditions for applying these transformations can be weakened if the required satisfiability checks are too expensive (see Section 4). 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. However, our refinement transformations account for the structure of the system and property being checked, and can be automated as well.

3.2 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 3.3), 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 and an SCS containing a node $N = (A, f)$ such that $f \wedge \neg enabled(\tau)$ is satisfiable. Then replace N by the two nodes

$$N_1 = (A, f \wedge enabled(\tau))$$
$$N_2 = (A, f \wedge \neg enabled(\tau)) \quad .$$

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.

We now present 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 by 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.)

3.3 Reachability and Termination

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

Finding a counterexample computation in the case that φ fails, however, requires some additional work. Whereas the above transformations remove SCS's from consideration that are known to be unreachable because they are disconnected from an initial node, no provisions ensure that a node is indeed reachable in an actual computation, or that a computation can in fact reside indefinitely within an SCS.

To identify portions of the product graph known to be reachable, we do some additional book-keeping:

- **(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.

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 $\neg\varphi$ (but the reverse does not hold).

Using Previously Proven Properties: Known invariants can be used to strengthen all (or only some) of the assertions in the $\mathcal{S}$-refined tableau; if $\Box\, p$ is a known invariant for a state-formula p, then we can replace any node (A, f) by the node $(A, f \wedge p)$.

Similarly, simple temporal properties of the system can be used to rule out paths in the tableau. 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 which implies p also contain a state-formula compatible with q.[3]

4 Analysis

The soundness of the procedure is based on the fact that each transformation preserves the set of computations through the $\mathcal{S}$-refined tableau. Since this is equal to the $\neg\varphi$ computations of $\mathcal{S}$ for the initial graph $\mathcal{G}_0$, the procedure reports success only if there are no such computations. On the other hand, a computation that is obtained by reaching and then residing in an adequate SCS must indeed be a model of $\neg\varphi$ and a computation of $\mathcal{S}$, and thus a counterexample.

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). *Incremental* and *particle tableau* constructions [14] reduce the expense of building Φ_φ.[4]

Proposition 6. *For a finite-state system $\mathcal{S}$, the exhaustive application of transformations 1–11 terminates, deciding the $\mathcal{S}$-validity of φ.*

If the system $\mathcal{S}$ is finite-state, we can use a finite-state assertion language $\mathcal{A}$. Note that the satisfiability tests required at the splitting steps are now decidable, and there will only be a finite number of distinct nodes. Since every transformation reduces the size of the graphs under consideration or replaces a node with more specific ones (that is, nodes covering strictly fewer states), the process must terminate. If the SCS list is empty, the original property φ is

[3] $\neg\varphi$ could always be conjoined with all other known temporal properties of $\mathcal{S}$, but at the risk of further increasing the size of the temporal tableau.

[4] If necessary, this construction can be interleaved with the state-space refinement.

$\mathcal{S}$-valid; otherwise, any remaining SCS must be adequate, and thus provide a counterexample.

The node formulas may well be encoded using binary decision diagrams (BDDs) [7] or, in general, any finite-domain constraint language. The efficient tests for implication between BDDs can be used to maintain encapsulation conventions. Hybrid representations (including first-order constructs) can be used if the BDD size becomes problematic.

In the general case of infinite-state systems, the model checking problem is undecidable. However, we point to several features of our approach:

• The test for satisfiability used in the splitting rules need not be complete; we can change the condition "if X is satisfiable then..." to be "if X is not *known* to be unsatisfiable then..." without compromising soundness. Thus, the available theorem-proving and simplification techniques are not required to give a definite answer at any given time. When the validity of a formula is hard to decide, additional splits can make subsequent satisfiability questions easier.

This lazy evaluation of satisfiability makes specialized constraint languages such as those used in Constraint Logic Programming [12] well-suited to the task. Reactive programs based on such constraint languages, such as concurrent constraint programs [17], may be specially amenable to such a verification framework. We expect constraint-solving and propagation techniques, as in [3], to play a central role in the deductive model checking of large systems.

• Even when the model checking effort is not completed, the resulting $\mathcal{S}$-refined tableau can be used to restrict the search for a counterexample, since all such computations must follow the $\mathcal{S}$-refined tableau. Backward propagation (possibly approximated) [3] can be used to find sets of initial states that can generate a counterexample computation. A similar approach is used in [9] to generate test cases for processor designs.

• 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.

Inductive and well-foundedness arguments can also be used: for example, if a transition decreases a well-founded relation that is known to hold across an SCS, then we can remove it from all the edges in the SCS (but still account for it for reachability). Adding support for well-founded relations and ranking functions similar to those used in Verification Diagrams [13, 6] could make the method relatively complete and further the combination of theorem-proving and model checking we propose.

5 Example

We illustrate deductive model checking by proving accessibility for the BAKERY program, an infinite-state program implementing a mutual exclusion protocol, shown in Figure 1. 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. Accessibility can be expressed in LTL by the formula $\varphi : \Box(\ell_1 \rightarrow \Diamond \ell_3)$, i.e., always if control is at ℓ_1 it will eventually reach ℓ_3. The following describes the output of our DMC implementation based on the STeP system [2]. The splits are chosen by the user, but the underlying simplification and pruning are performed automatically.

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

Fig. 1. Program BAKERY

The initial $\mathcal{S}$-refined tableau for $\neg\varphi : \Diamond(\ell_1 \wedge \Box\neg\ell_3)$, based on its *particle tableau*, is shown in the left of 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 from node 3 is pruned since $\ell_1 \wedge \Theta$ is unsatisfiable. The SCS $\{4\}$ is not fulfilling, but $\{2\}$ is. We now perform a precondition split on edge $\langle 2, 2\rangle$ and transition ℓ_0, replacing node 2 by nodes 6 and 5. An ℓ_4-precondition split on $\langle 6, 5\rangle$ 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 $\{8\}$.

An ℓ_1-postcondition split for $\langle 8, 8\rangle$ yields nodes 9 and 10 in Figure 3. The only fulfilling SCS is $\{10\}$, since $\{9\}$ is unjust for ℓ_1. 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. Now, SCS $\{16\}$ is unjust for m_4, while $\{15\}$ is unjust for m_2. Since there are no candidate SCS left, we have established that φ is $\mathcal{S}$-valid.

Note that when model checking progress properties it may be profitable to concentrate on splitting and eliminating candidate SCS's, as done in this example. However, in general it may be necessary to show that certain parts of the state-space are unreachable through forward propagation from the initial nodes or backward propagation from the unreachable ones. We model checked mutual exclusion for BAKERY ($\Box\neg(\ell_3 \wedge m_3)$) using 3 splits (including a user-provided one) and automatically generated invariants.

We also model checked accessibility for the infinite-state 3-process version of BAKERY, expanding to 27 nodes. This included one user-provided case split according to the priority between processes (4 cases), together with 5 trivial location splits and one enabled split.

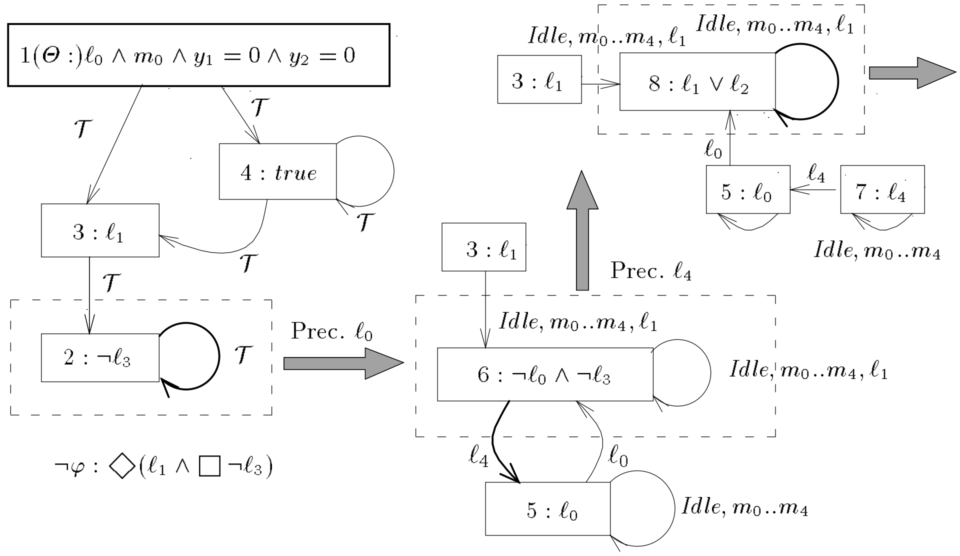

Fig. 2. Initial $\mathcal{S}$-refined tableau and first 2 refinement steps

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

References

1. Bhat, G., Cleaveland, R., and Grumberg, O. Efficient on-the-fly model checking for CTL*. In *Proc. 10th IEEE Symp. Logic in Comp. Sci.* (1995), pp. 388–397.
2. Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H., and Uribe, T. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification* (July 1996), Springer-Verlag.
3. Bjørner, N., Browne, A., and Manna, Z. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming* (Sept. 1995), vol. 976 of *LNCS*, Springer-Verlag, pp. 589–623.
4. Bouajjani, A., Fernandez, J.-C., and Halbwachs, N. Minimal model generation. In *Proc.* 2^{nd} *Intl. Conference on Computer Aided Verification* (1990), vol. 531 of *LNCS*, pp. 197–203.
5. Bradfield, J. C. *Verifying Temporal Properties of Systems.* Birkhäuser, 1992.
6. Browne, A., Manna, Z., and Sipma, H. Generalized verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science* (Dec. 1995), vol. 1026 of *LNCS*, pp. 484–498.
7. Bryant, R. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug. 1986), 677–691.
8. Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.* (1990), IEEE Computer Society Press, pp. 428–439.

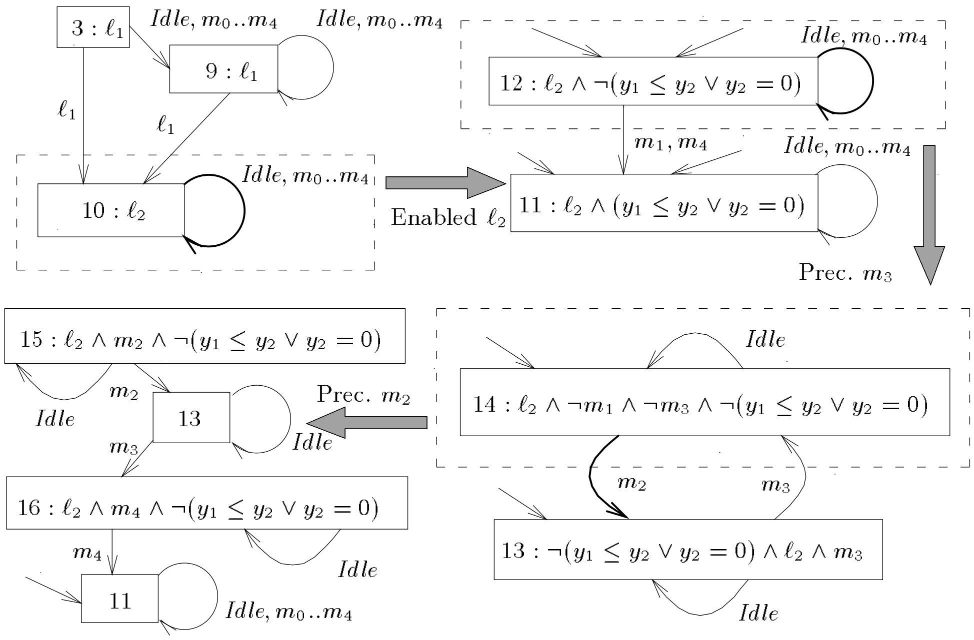

Fig. 3. Final 3 refinement steps to model check BAKERY

9. Chandra, A., Iyengar, V., Jawalekar, R., Mullen, M., Nair, I., and Rosen, B. Architectural verification of processors using symbolic instruction graphs. In *International Conference on Computer Design: VLSI in Computers and Processors* (1994), IEEE Press, pp. 454–459.
10. Clarke, E., and Emerson, E. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs* (1981), vol. 131 of *LNCS*, Springer-Verlag, pp. 52–71.
11. Fix, L., and Grumberg, O. Verification of temporal properties. *J. Logic and Computation 6*, 3 (1996), 343–362.
12. Jaffar, J., and Lassez, J.-L. Constraint logic programming. In *Proc. 14th ACM Symp. Princ. of Prog. Lang.* (Jan. 1987), pp. 111–119.
13. Manna, Z., and Pnueli, A. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software* (1994), vol. 789 of *LNCS*, Springer-Verlag, pp. 726–765.
14. Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
15. McMillan, K. L. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.
16. Queille, J., and Sifakis, J. Specification and verification of concurrent systems in CESAR. In *Intl. Symposium on Programming* (1982), M. Dezani-Ciancaglini and U. Montanari, Eds., vol. 137 of *LNCS*, Springer-Verlag, pp. 337–351.
17. Saraswat, V. A. *Concurrent Constraint Programming.* MIT Press, 1993.
18. Sokolsky, O. V., and Smolka, S. A. Local model checking for real-time systems. In *Proc. 7^{th} Intl. Conference on Computer Aided Verification* (1995), vol. 939 of *LNCS*, pp. 211–224.