To appear in *TACAS-97*,
LNCS, Springer-Verlag, April 1997.

Visual Verification of Reactive Systems*

Luca de Alfaro, Zohar Manna,
Henny B. Sipma and Tomás E. Uribe
luca|manna|sipma|uribe@cs.stanford.edu

Computer Science Department, Stanford University
Stanford, CA. 94305

Abstract. We describe diagram-based formal methods for verifying temporal properties of finite- and infinite-state reactive systems. These methods, which share a common background and tools, differ in the way they use automatic procedures within an interactive setting based on deduction. They can be used to produce a static proof object, or to perform incremental analysis of systems and specifications.

1 Introduction

Reactive systems have an ongoing interaction with their environment. They include distributed and concurrent algorithms, hardware systems, and control programs. *Temporal logic* has proved to be a convenient language for expressing *safety*, *progress* and *response* properties of reactive systems [MP91, MP95].

The classic approach to verifying temporal properties of reactive systems is based on *verification rules*, which reduce the system validity of a temporal property to the general validity of a set of first-order *verification conditions*. While this methodology is complete, relative to the underlying first-order reasoning, the proofs do not always reflect an intuitive understanding of the system and its specification; without this intuition, the proofs can be difficult to construct.

We address the need for a more intuitive approach to verification by using *diagram-based formalisms*. Our diagrams are graphs whose vertices are labeled with first-order formulas (*assertions*), representing sets of system states, and whose edges represent possible system transitions. This paper presents an overview of three diagram-based formalisms for the verification of temporal properties of systems, which correspond to different methods of proof construction. *Verification diagrams* [MP94, BMS95] represent a completed direct proof, and offer a compact representation of the necessary verification conditions. *Fairness diagrams* [dAM96] represent abstractions of the system, and the proof of a specification is constructed by stepwise diagram transformations. *Deductive model checking* [SUM96] uses diagrams to conduct an exhaustive search for a counterexample, giving a semi-automatic proof procedure.

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

Some features shared by these formalisms are:

- Diagrams (or sequences of diagrams) are formal proof objects, which succinctly represent verification conditions that replace a combination of textual verification rules.
- Diagrams can describe and verify infinite-state systems using a finite and often compact representation.
- The construction of a diagram can be incremental, starting from a high-level outline and then filling in details as necessary. The diagrams for a given system can serve as documentation; they can also be re-used when similar proofs are carried out for similar systems, or when a system is refined.
- The verification conditions are *local* to the diagram; failed verification conditions point to missing edges or vertices, or possible bugs in the system. The necessary global properties of diagrams can be proved algorithmically.
- An incomplete or failed proof provides feedback that can be used to establish the existence of counterexamples or to guide the simulation of the system along the potentially problematic computations.
- By expressing temporal properties as automata, the methods allow the direct verification of general temporal logic specifications, eliminating the need to express the formulas in the normal forms required by a fixed set of classic deductive verification rules.

A related formalism is used in [BBM95], as part of a procedure for automatically generating invariants and establishing temporal safety properties based on *assertion graphs* similar to the diagrams in this paper. To highlight the similarities and relationships between these verification methods, we have adopted as much as possible a uniform notation, deviating occasionally from that used in the original proposals. However, their essential features have been preserved.

These methods have been recently extended to hierarchical verification [BMS96] and modular verification [dAMS97]. The fairness diagrams method has also been successfully extended to the study of hybrid systems [dAKM97].

Section 2 summarizes the definitions and notation used in the ensuing presentation. Section 3 presents the basic definition of diagrams. Section 4 presents an overview of the three proof methods. Section 5 introduces the basic transformations that can be used in all cases, which are sufficient for the verification of *safety* properties. Section 6 discusses the role of fairness, which is treated differently by each formalism.

2 Background

Preliminary notation. Given a set $\mathcal{V}$ of variables, we let $\mathcal{V}' = \{x' \mid x \in \mathcal{V}\}$ be the set obtained by priming all the variables in $\mathcal{V}$, and let $\mathit{form}(\mathcal{V})$ be the set of well-formed first-order formulas whose free variables are among $\mathcal{V}$. Given $\varphi \in \mathit{form}(\mathcal{V})$, φ' is the formula of $\mathit{form}(\mathcal{V}')$ obtained from φ by replacing each free $x \in \mathcal{V}$ with $x' \in \mathcal{V}'$.

2.1 Fair Transition Systems

We represent reactive systems as *fair transition systems*. A fair transition system $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ includes a finite set of *state variables* $\mathcal{V}$, an *initial condition* $\Theta \in \mathit{form}(\mathcal{V})$, and a finite set of *transitions* $\mathcal{T}$. The *state space* Σ of the system is the set of all possible type-consistent assignments to the state variables. Each transition $\tau \in \mathcal{T}$ is described by its *transition relation* $\rho_\tau(\mathcal{V}, \mathcal{V}')$, where the unprimed variables describe the state before the transition, and the primed variables describe the following state. The set $\mathcal{T}$ always includes the *idling transition*, defined by the transition relation $\bigwedge_{x \in \mathcal{V}} (x = x')$. We distinguish two disjoint subsets $\mathcal{J}, \mathcal{C} \subseteq \mathcal{T}$ of transitions: the set $\mathcal{J}$ of *just* (weakly fair) transitions, and the set $\mathcal{C}$ of *compassionate* (strongly fair) transitions.

A *run* of $\mathcal{S}$ is an infinite sequence of states $s_0, s_1, \ldots$, where $s_0 \vDash \Theta$, and for every $i \geq 0$ there is a transition $\tau \in \mathcal{T}$ that is *taken* at i, that is, such that $(s_i, s_{i+1}) \vDash \rho_\tau$. A transition is *enabled* at a given state if it can be taken. This condition is expressed by the assertion $\mathit{enabled}(\tau) \stackrel{\text{def}}{=} \exists \mathcal{V}' . \rho_\tau(\mathcal{V}, \mathcal{V}')$. Just transitions cannot be continuously enabled without ever being taken; compassionate transitions cannot be enabled infinitely often without being taken infinitely often. A *computation* of $\mathcal{S}$ is a run that satisfies these fairness requirements, and $\mathcal{L}(\mathcal{S})$ is the set of all computations of $\mathcal{S}$.

Example. Figure 1 shows program BAKERY, an infinite-state system that implements a mutual exclusion protocol. Translating this program into a fair transition system is straightforward.[2] Each program statement corresponds to a transition, denoted by its label; thus, $\mathcal{T} = \{\mathit{Idle}, \ell_0..\ell_4, m_0..m_4\}$. All transitions are just, except for m_0 and ℓ_0, which have no fairness requirements (from the semantics of the **noncritical** statement). To each process corresponds a *control variable*. For BAKERY, the control variables for the two processes range over the locations $\{\ell_0, \ldots, \ell_4\}$ and $\{m_0, \ldots, m_4\}$. Assertions at_m_i, at_ℓ_j (abbreviated to m_i, ℓ_i when there is no confusion between assertions and transitions) indicate that control resides at locations m_i, m_j. □

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

Fig. 1. Program BAKERY

[2] The STeP system [BBC+96] parses such SPL programs into fair transition systems, or can take transition systems directly as input.

2.2 Temporal Logic

Specifications are expressed as formulas in *linear-time temporal logic*. *Assertions*, or state-formulas, are first-order formulas with no temporal operators, and can include quantifiers. Temporal formulas are constructed from assertions, boolean connectives, and the usual *future* ($\Box, \Diamond, \bigcirc, \mathcal{U}, \mathcal{W}$) and *past* ($\boxminus, \diamondminus, \ominus, \mathcal{B}, \mathcal{S}$) temporal operators [MP91]. For instance, the formula $\Box\, \neg(at\_\ell_3 \wedge at\_m_3)$ expresses mutual exclusion for BAKERY. Given a formula φ, $\mathcal{L}(\varphi)$ is the set of all infinite sequences that satisfy φ.

The formula tableau. Given a temporal formula φ, it is possible to construct the *tableau* $T_\varphi = \langle V, V_{in}, E, a\rangle$ of φ, a finite automaton that describes all the sequences in $\mathcal{L}(\varphi)$ [KMMP93, MP95]. The vertices V of this automaton correspond to the *atoms* of φ, which are consistent sets of subformulas of φ: to $v \in V$ is associated atom $a(v)$. The formulas in the atom $a(v)$ are expected to hold whenever the automaton is at vertex v. The set $E \subseteq V^2$ of edges contains edge (u, v) iff the formulas in $a(v)$ can hold at a state following one that satisfies the formulas in $a(u)$. The set $V_{in} \subseteq V$ of *initial vertices* contains all the vertices whose atom formulas can hold at the initial state of a sequence in $\mathcal{L}(\varphi)$.

A subformula of φ of the form $\Diamond\, \psi$ is called an *eventuality*. We say that the eventuality *arises* at vertex $v \in V$ if $\Diamond\, \psi \in a(v)$, and that it is *satisfied* at v if $\psi \in a(v)$. An infinite computation of T_φ is *fulfilling* if all the eventualities that arise are later satisfied. It is easy to see that the set of vertices of T_φ that appear infinitely often along a fulfilling computation must satisfy the following properties:

1. it forms a strongly connected subgraph (SCS) $U \subseteq V$ in the graph (V, E);
2. U is reachable from V_{in} in the graph (V, E);
3. if an eventuality arises at $u \in U$, there must be $v \in U$ where it is fulfilled.

The SCS's with these properties are called *fulfilling*. Their presence in T_φ indicates that formula φ is satisfiable, as stated by the following theorem.

Theorem 1 (fulfilling SCS's and satisfaction). *A linear-time temporal logic formula φ is satisfiable iff T_φ contains a fulfilling SCS.*

2.3 Algorithmic and Deductive Verification

Given a a system $\mathcal{S}$ and a temporal specification φ, $\mathcal{S}$ satisfies φ if all computations of $\mathcal{S}$ satisfy φ. Thus, verification consists of showing $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$. If $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$ we say that φ is *$\mathcal{S}$-valid*. The two main approaches to verification are *deductive* and *algorithmic*.

Deductive verification. Deductive verification uses *verification rules* to reduce the validity of a given temporal property over a system to the general validity of a set of first-order *verification conditions*. Each rule is applicable to a certain class of properties. For example, the *general invariance rule* reduces the proof of a property of the form $\Box\, \psi$ for an assertion ψ over a system $\mathcal{S}$, to finding an assertion φ that implies ψ, that is preserved by all transitions of $\mathcal{S}$, and is implied by the initial condition of $\mathcal{S}$, all of which are first-order conditions. Other verification rules are available for proving different classes of temporal properties, e.g. *nested wait-for formulas* and *response formulas*. The textbook [MP95] presents a set of rules that is complete for proving safety properties.

Algorithmic verification. Algorithmic verification, or *model checking*, can automatically decide whether a given system $\mathcal{S}$ satisfies its temporal specification φ [CE81, QS82]. These algorithms are based on explicit state enumeration or specialized data structures to represent the transition relation and compute fixpoints over it, as in BDD-based symbolic model checking [McM93]. Automata-based model checking methods [Kur94, VW86] represent both the system and property as ω-automata, and then use algorithmic procedures to test that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$.

Model checking has been particularly successful for the verification of hardware systems. However, it is essentially applicable to finite-state systems only, whereas deductive methods are applicable to infinite-state systems as well.

Our diagrams combine deductive and algorithmic verification. In general, the relationship between the diagram and the system is proved deductively, and that between the diagram and the property is checked algorithmically.

3 Diagrams

Given a fair transition system $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, we study the temporal properties of its set of computations by means of first-order *diagrams*. A diagram $A = \langle V, \mu, \theta, \nu \rangle$ for $\mathcal{S}$ contains the following components:

1. A set V of vertices.
2. Two labelings $\mu, \theta : V \mapsto form(\mathcal{V})$, which label each vertex $v \in V$ with a first-order assertion $\mu(v)$, describing the system states associated with v, and an initial condition $\theta(v)$.
3. A labeling $\nu : V^2 \mapsto 2^{\mathcal{T}}$, associating with each edge $e \in V^2$ a set $\nu(e)$ of transitions of $\mathcal{S}$.

A *location* of diagram A is a pair (v, s) for a vertex $v \in V$ and a state s that satisfies assertion $\mu(v)$. A *trail* of diagram A is an infinite sequence of locations $(v_0, s_0), (v_1, s_1), \ldots$ such that $s_0 \vDash \theta(v_0)$, and for each $i \geq 0$ there is $\tau \in \nu(v_i, v_{i+1})$ such that $(s_i, s_{i+1}) \vDash \rho_\tau$. We say that this τ is *taken* at position i of the trail. To each trail $(v_0, s_0), (v_1, s_1), \ldots$ corresponds a run $s_0, s_1, \ldots$ with the sequence of states traversed by the trail. Given a diagram A, *Trails*(A) and *Runs*(A) are its set of trails and runs, respectively.

Finally, diagrams have an associated set of *acceptance conditions* that encode fairness. Since these conditions differ for the three formalisms presented, we postpone their exact definition to Section 6. However, we define the set of *computations* of a diagram, $\mathcal{L}(A)$, as the set of all runs that correspond to trails that satisfy the acceptance conditions of the diagram.

To make diagrams more succinct, we often use *encapsulation conventions*, based on those of Statecharts [Har87]. The assertion labeling a compound vertex is added, as a conjunct, to its subvertices. Edges leaving (entering) a compound vertex are interpreted as leaving (entering) all of its subvertices.

We omit from our diagrams all edges (u, v) such that $\nu(u, v) = \emptyset$. Moreover, for the figures in this paper we adopt an additional convention concerning self-loops: for every vertex u, we assume that the edge (u, u) is labeled with all the transitions that can lead from $\mu(u)$ to $\mu(u)$, so that $\nu(u, u) = \{\tau \in \mathcal{T} \mid \rho_\tau \wedge \mu(u) \wedge \mu'(u) \text{ is satisfiable}\}$, and omit all self-loops from our figures.

Checking temporal properties of diagrams. Given a diagram A for an infinite-state system and a property φ, there is no algorithmic procedure to decide whether $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$ in the general case. However, we can give an algorithm that checks a *sufficient* condition for $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$.

The first step is to compute the automata-theoretic product between the diagram A and the tableau $T_{\neg\varphi}$ for the negation of φ. For each $u \in V_A$ and $v \in V_T$, the product $A \otimes T_{\neg\varphi}$ contains a vertex (u, v) labeled by assertion $\mu(u)$ and atom $a(v)$. There is an edge from (u, v) to (u', v') in the product iff $\nu(u, u') \neq \emptyset$ and (v, v') is an edge in $T_{\neg\varphi}$. A vertex (u, v) is *initial* in the product iff $\theta(u) \neq$ *false* and v is an initial vertex in $T_{\neg\varphi}$.

For a set of temporal formulas r, let $state(r)$ be the set of state-formulas in r. The next step removes from the product all vertices (u, v) such that

$$\mu(u) \wedge \bigwedge_{\psi \in state(a(v))} \psi$$

is a substitution instance of an unsatisfiable propositional formula. This last condition ensures that the test is algorithmic. These vertices represent pairs of diagram and tableau vertices whose labels are not simultaneously satisfiable.

In the resulting product $A \otimes T_{\neg\varphi}$ we can still talk about fulfilling SCS's, by considering each vertex (u, v) as labeled by the atom $a(v)$. The following result gives us the desired algorithmic test.

Theorem 2. *If there are no fulfilling SCS's in $A \otimes T_{\neg\varphi}$, then $Runs(A) \subseteq \mathcal{L}(\varphi)$.*

Recall that $\mathcal{L}(A) \subseteq Runs(A)$. Since we have not yet discussed the acceptance conditions that encode fairness for diagrams, we will describe in Section 6 more refined tests that take fairness into account.

4 Proof Methods

While they share the same basic notion of diagram, the three methodologies differ in the way they construct the proof of a temporal specification. We now present an outline of each method.

4.1 Verification Diagrams

Given a fair transition system $\mathcal{S}$ and a specification φ, to verify that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$ it suffices to find a diagram A such that

$$\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(A) \qquad \text{and} \qquad \mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$$

where the first containment can be proved by deductive methods, and the second can be checked algorithmically. Such a diagram is called a *verification diagram* for $\mathcal{S}$ and φ: it represents a finite abstraction of $\mathcal{S}$ that can be algorithmically shown to satisfy φ, and provides a proof of $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$.

To show $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(A)$ it suffices to prove the following first-order conditions:

1. *Initiality:* Each computation of $\mathcal{S}$ has a starting point in A, expressed by the verification condition
$$\Theta \rightarrow \bigvee_{u \in V} (\mu(u) \wedge \theta(u)) \enspace .$$
2. *Consecution:* Every computation prefix can always be extended, i.e., for every $\tau \in \mathcal{T}$ and $u \in V$, the following implication must hold:
$$(\mu(u) \wedge \rho_\tau) \rightarrow \bigvee_{v \in \tau(u)} \mu'(v) \enspace ,$$
where $\tau(u) \stackrel{\text{def}}{=} \{v \in V \mid (u, v) \in V^2 \wedge \tau \in \nu(u, v)\}$ is the set of vertices that can be reached from u by following an edge labeled with τ.

In the original proposal for verification diagrams [MP94], property-dependent well-formedness constraints on the diagram ensure that A satisfies φ. This was generalized in [BMS95] to an algorithmic language inclusion check $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$.

Example. Mutual exclusion for program BAKERY is expressed by the formula $\varphi_{mut} : \Box\neg(\ell_3 \wedge m_3)$. Figure 2 presents a verification diagram that proves that $\mathcal{L}(\text{BAKERY}) \subseteq \mathcal{L}(\varphi_{mut})$. The set of vertices of the diagram is $V = \{0, 1, 2, 3, 4\}$, and the initial labeling is $\theta(0) = \mathit{true}$, $\theta(i) = \mathit{false}$ for $i = 1, 2, 3, 4$. $\Box$

4.2 Fairness Diagrams

Verification diagrams provide a concise graphical way of presenting a proof of a specification. However, a trial-and-error process is necessary to find a suitable diagram. When a candidate verification diagram A is proposed, the two conditions $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(A)$ and $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$ must be checked; if either test fails, the user must provide a new candidate diagram. Note that since algorithmic tests provide only a sufficient condition for the second containment, not all diagrams such that $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$ will pass the test. The verification method based on fairness diagrams gives a way of guiding and formalizing the search for a suitable verification diagram.

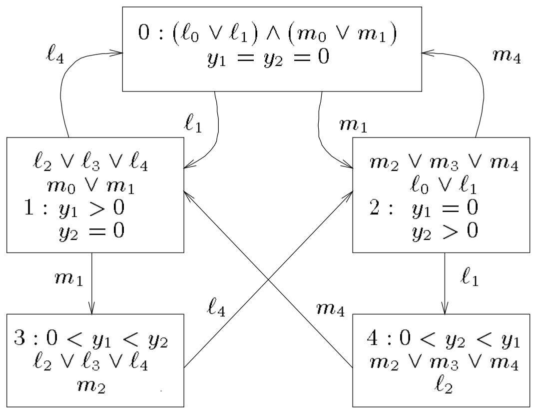

Fig. 2. Verification diagram A_0 for property $\Box \neg(\ell_3 \wedge m_3)$ of BAKERY.

In this method, the proof of $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$ is a sequence of diagrams $A_0 \Rightarrow \cdots \Rightarrow A_n$, starting from a diagram A_0 such that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(A_0)$, and ending with a diagram A_n such that $\mathcal{L}(A_n) \subseteq \mathcal{L}(\varphi)$ can be shown algorithmically. The invariant $\mathcal{L}(A_i) \subseteq \mathcal{L}(A_{i+1})$ is maintained for all $0 \leq i < n$, so the existence of the sequence proves that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$.

Diagram A_0 can be a verification diagram for which $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(A_0)$ has been proved, or it can be constructed automatically from $\mathcal{S}$. In the latter case, A_0 consists of only one vertex v_0, labeled by $\mu(v_0) = \mathit{true}$ and $\theta(v_0) = \Theta$; its only edge (v_0, v_0) is labeled with the set $\mathcal{T}$ of all transitions. Starting from this diagram, the successive diagrams of the sequence are obtained by *diagram transformations*, described in Sections 5 and 6.

Guidance. After a partial sequence $A_0 \Rightarrow \cdots \Rightarrow A_k$ has been constructed, it is possible to compute the product $A_k \otimes T_{\neg\varphi}$, applying the algorithmic test described in Section 3. If $A_k \otimes T_{\neg\varphi}$ does not contain any fulfilling SCS, then $\mathcal{L}(A_k) \subseteq \mathcal{L}(\varphi)$ and the proof is concluded. Otherwise, each fulfilling SCS of $A_k \otimes T_{\neg\varphi}$ points to a possible counterexample to φ, and provides guidance for the extension of the sequence of diagrams, as described in [dAKM97].

Specifically, let X be the set of vertices of a fulfilling SCS of $A_k \otimes T_{\neg\varphi}$. Let Y be the set of vertices that appear in paths from initial vertex to X in $A_k \otimes T_{\neg\varphi}$. It is possible to project the sets X and Y back on A_k, obtaining the sets of vertices $\widehat{X}, \widehat{Y} \subseteq V_{A_k}$. The set $\widehat{X}$ corresponds to an SCS of A_k, and the set $\widehat{Y}$ corresponds to vertices that lie between initial vertices and $\widehat{X}$. The aim of the subsequent transformations is then to analyze the diagram to show that:

1. either $\widehat{X}$ cannot be reached through paths contained in $\widehat{Y}$,
2. or a trail, after reaching $\widehat{X}$ through $\widehat{Y}$, cannot stay in $\widehat{X}$ forever while visiting all vertices of $\widehat{X}$ infinitely often.

Alternatively, the components $\widehat{X}$ and $\widehat{Y}$ can be analyzed to determine the existence of counterexamples to φ, or to guide the simulation of the system along the problematic computations. We will give an example of the use of guidance in the proof of the *accessibility* property for BAKERY, presented in Section 6.1.

4.3 Deductive Model Checking

Deductive model checking provides an alternative way to gradually construct a proof. We now start with a diagram, called a *falsification diagram*, that represents the product of the *negated* specification and the transition system $\mathcal{S}$. The initial diagram A_0 is obtained from the tableau $T_{\neg\varphi}$ by taking as vertex labeling $\mu(v)$ the conjunction of the state formulas belonging to the corresponding atom. $\theta(v) = \Theta$ for vertices corresponding to initial atoms and $\theta(v) = \mathit{false}$ for all other vertices. The edges that are present in $T_{\neg\varphi}$ are labeled by $\mathcal{T}$, the full set of transitions; the others with the empty set. By construction, we have $\mathcal{L}(A_0) = \mathcal{L}(\neg\varphi) \cap \mathcal{L}(\mathcal{S})$.

The diagram A_0 is then studied by language-preserving transformations. They can modify the diagram by splitting vertices, strengthening vertex labels or dropping edge labels; other transformations exclude SCS's that are not accepting. The proof succeeds by obtaining a diagram whose language can be algorithmically shown to be empty. At any stage the falsification diagram embeds all computations that satisfy the negation of the specification, thus providing information regarding the possible counterexamples to the specification.

To check that a falsification diagram A has an empty language, it is sufficient to check that A, when interpreted as a tableau, does not contain any fulfilling SCS. This condition will later be extended to account for the fairness conditions derived from the fair transition system $\mathcal{S}$. Proof construction is thus guided by the SCS's of the falsification diagram: the aim is to analyze the diagram, either splitting the SCS's into smaller unfulfilling ones, or showing that they are unreachable or can be excluded by the acceptance conditions.

5 Diagram Transformations: Safety

The transformation rules on which our methods rely can be divided into the two classes of *safety rules* and *fairness rules*, depending on whether they are used to study the safety or progress properties of the behavior of the diagrams.

This section presents the safety rules for the three methodologies, which are closely related. The fairness rules are introduced in Section 6, after discussing how the fairness properties of diagrams are represented in each case.

5.1 Fairness Diagrams: the Simulation Rule

The main requirement for a transformation rule is to preserve language containment: if diagram A can be transformed into diagram B, $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ should hold. Simulation relations are a classical method to prove language containment

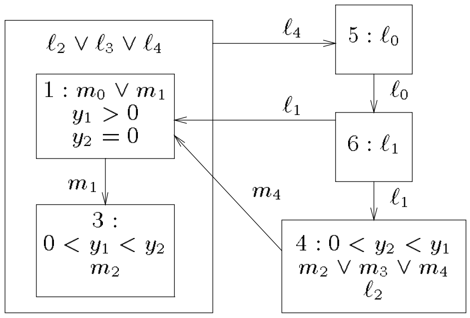

Fig. 3. Fairness diagram A_1, obtained from Figure 2.

of transition systems. The *simulation rule* for fairness diagrams is based on generalizing simulation relations to the case of first-order diagrams.

Since locations represent instantaneous configurations of the diagrams, simulation relations will be relations between locations. Given two diagrams A, B, a function $\gamma : V_A \mapsto 2^{V_B}$ induces a mapping that relates each location (u, s) of A with all the locations (v, s) of B such that $v \in \gamma(u)$. The simulation rule checks that the mapping induced on the locations is indeed a simulation relation and, in the affirmative case, allows us to transform A into B. To state the rule more concisely, given two vertices u, v of a diagram we introduce the abbreviation

$$\rho(u, v) \stackrel{\text{def}}{=} \mu(u) \wedge \mu'(v) \wedge \bigvee_{\tau \in \nu(u,v)} \rho_\tau$$

for all the possible state changes corresponding to edge (u, v).

Rule 1 (simulation). Let A and B be two diagrams sharing the same variables, and let $\gamma : V_A \mapsto 2^{V_B}$ be given. We can transform A into B provided the following conditions hold.

1. *Initiality.* For all $u \in V_A$, $\big(\theta_A(u) \wedge \mu_A(u)\big) \rightarrow \bigvee_{v \in \gamma(u)} \big(\theta_B(v) \wedge \mu_B(v)\big)$.
2. *Consecution.* For all $u, u' \in V_A$ and $v \in \gamma(u)$,

$$\big(\rho_A(u, u') \wedge \mu_B(v)\big) \rightarrow \bigvee_{v' \in \gamma(u')} \rho_B(v, v') \ .$$

□

Example. The simulation rule can transform the verification diagram A_0 of Figure 2 into diagram A_1 of Figure 3. The simulation is based on the mapping γ defined by $\gamma(0) = \gamma(2) = \{5, 6\}$, and $\gamma(i) = \{i\}$ for $i = 1, 3, 4$. The initial labeling of A_1 is $\theta(5) = \mathit{true}$ and $\theta(i) = \mathit{false}$ for $i = 1, 3, 4, 6$. □

5.2 Deductive Model Checking: Specialized Rules

While the simulation rule is very general, and can justify complex transformations in a single step, its drawback is having to specify each time the mapping γ between the vertices of the old and new diagram. Instead of the simulation rule, deductive model checking uses specialized versions, which carry out specific tasks in the process of proof construction. For conciseness, we present a minimal set of rules. Note that neither this set of rules nor the original one of [SUM96] is equivalent to the simulation rule, i.e. it is not possible to mimic all applications of the simulation rule by repeated applications of the specialized rules. Nonetheless, the set enables the proof of arbitrary safety properties, and is thus complete in this more relevant and practical respect.

The first rule, *vertex split*, splits a vertex into two vertices with mutually exclusive labels.

Rule 2 (vertex split). Given a diagram A, a vertex $v \in V$ and a formula $\varphi \in form(\mathcal{V})$, we can replace vertex v with two new vertices v_1, v_2, labeled with $\mu(v_1) = \mu(v) \wedge \varphi$, $\mu(v_2) = \mu(v) \wedge \neg\varphi$.

When a vertex v is split into v_1 and v_2, the edges (u, v) and (v, u) for all $u \neq v$ are replaced with new edges (u, v_1), (u, v_2), (v_1, u), (v_2, u) respectively. The edge (v, v) is likewise replaced by the four edges (v_i, v_j), $i, j \in \{1, 2\}$. The new edges have the same labels of the edges they replace, and the new vertices v_1, v_2 are labeled with the same atom as v. □

As special cases of this rule, vertices are often split according to the *weakest precondition* (resp. *strongest postcondition*) of a transition that labels an edge departing from (resp. arriving at) them.

The second rule, *vertex strengthen*, strengthens the labels of vertices, provided the strengthenings represent valid inductive invariants.

Rule 3 (vertex strengthen). Given a diagram A with vertices $v_1, \ldots, v_n$, and formulas $\varphi_1, \varphi_2, \ldots, \varphi_n$, assume that the implications $(\varphi_i \wedge \rho(v_i, v_j)) \rightarrow \varphi_j$ hold for all $i, j \in [1..n]$. Then we can simultaneously replace, for all $1 \leq i \leq n$, the label $\mu(v_i)$ of v_i with $\mu(v_i) \wedge \varphi_i$. □

The strengthening assertions φ_i can be obtained using the techniques presented in [BBM95]. These methods generate invariants by propagating pre- and post-conditions over *assertion graphs*, which are diagrams for general safety properties. Abstraction domains, which approximate the system, make automatic propagation possible.

The third rule drops a transition from the label of an edge, if it can be shown that the transition cannot be taken along the edge.

Rule 4 (remove transition label). For an edge (u, v) and $\tau \in \nu(u, v)$, if $(\rho_\tau \wedge \rho(u, v))$ is unsatisfiable then remove τ from $\nu(u, v)$. □

6 Fairness

The methodologies differ in the way they represent the fairness of the behavior of diagrams. Verification diagrams and deductive model checking rely on the transition labels on the edges to represent the fairness properties of the original system, and hence of the diagram. Fairness diagrams owe their name to the *fairness constraints* used to represent fairness and progress properties.

6.1 Fairness Diagrams

Fairness diagrams rely on *fairness constraints* to represent the fairness properties that have been proved. A fairness constraint is a triple (J, C, G), where the components J, C label each vertex $v \in V$ with formulas $J(v), C(v) \in \mathit{form}(\mathcal{V})$, and component G labels each edge (u, v) with a formula $G(u, v) \in \mathit{form}(\mathcal{V}, \mathcal{V}')$. Unlike in [dAM96], we do not require the validity of $J(v) \rightarrow \mu(v)$, $C(v) \rightarrow \mu(v)$, $G(u, v) \rightarrow \rho(u, v)$. Associated with each diagram is a set $\mathcal{F}$ of fairness constraints.

Given a trail $\sigma : (v_0, s_0), (v_1, s_1), \ldots$, we say that σ is *accepted* by a constraint (J, C, G) if the following condition holds:

> *If there is $n \geq 0$ such that $s_i \models J(v_i)$ for all $i \geq n$ and $s_i \models C(v_i)$ for infinitely many $i \geq 0$, then there are infinitely many $i \geq 0$ such that $(s_i, s_{i+1}) \models G(v_i, v_{i+1})$.*

The symbols J, C, G take their names from the concepts of *Justice*, *Compassion* and *Gratification* [MP91]. We say that a trail is accepted by $\mathcal{F}$ if it is accepted by all the constraints in $\mathcal{F}$: such a trail is called a *fair trail*. We define $\mathcal{L}(A)$ to be the set of state sequences corresponding to the fair trails of A.

Implicit constraints. To simplify the representation of diagrams, we adopt the following convention. For each fair transition $\tau \in \mathcal{J} \cup \mathcal{C}$, there is an *implicit fairness constraint* (J_τ, C_τ, G_τ), defined as follows. For $u \in V$, if $\tau \in \nu(u, v)$ for some $v \in V$ then

$$C_\tau(u) = \begin{cases} \mathit{enabled}(\tau) & \text{if } \exists v \in V \,.\, \tau \in \nu(u, v) \\ \mathit{false} & \text{otherwise} \end{cases} \qquad J_\tau(u) = \begin{cases} C_\tau & \text{if } \tau \in \mathcal{J} \\ \mathit{true} & \text{if } \tau \in \mathcal{C}. \end{cases}$$

For an edge (u, v), it is $G_\tau(u, v) = \rho_\tau$ if $\tau \in \nu(u, v)$, and $G_\tau(u, v) = \mathit{false}$ otherwise. It is easy to see that each constraint (J_τ, C_τ, G_τ) faithfully represents the fairness properties of $\tau \in \mathcal{T}$.

If A_0 is the diagram from which a proof of $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$ is constructed, we add to its fairness set all the implicit constraints corresponding to transitions in $\mathcal{T}$. In the following we will not distinguish between implicit and explicit fairness constraints, since they play the same role in the definition of diagram language.

Checking $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$. Once fairness constraints are added to the definition of a diagram, we can modify the test for $\mathit{Runs}(A) \subseteq \mathcal{L}(\varphi)$ provided by Theorem 2, obtaining a refined test for $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$ based on weaker conditions. Instead of requiring the absence of fulfilling SCS's, we require that for each fulfilling

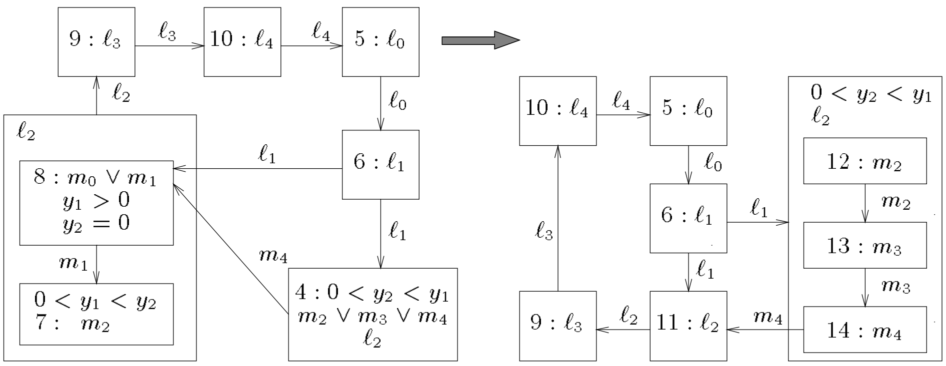

Fig. 4. Fairness diagrams A_2 and A_3, proving accessibility of BAKERY.

SCS X of $A \otimes T_{\neg\varphi}$ there is an *SCS-breaking constraint* $(J, C, G) \in \mathcal{F}_A$ such that the following implications and equivalences are substitution instance of valid propositional formulas (so that the test is again algorithmic):

1. For all $(u, v) \in X$, $\mu(u) \to J(u)$.
2. There is $(u, v) \in X$ such that $\mu(u) \to C(u)$.
3. For all $(u, v), (u', v') \in X$, $G(u, u') \equiv \mathit{false}$.

Fair simulation. To account for fairness constraints, we base the safety rule for fairness diagrams on *fair simulation*, instead of simple simulation. Rule 1 (Section 5) is extended by adding a third clause, which justifies each fairness constraint of the second diagram by a fairness constraint of the first:

3. *Fairness.* For each constraint $(J_B, C_B, G_B) \in \mathcal{F}_B$ there is a constraint $(J_A, C_A, G_A) \in \mathcal{F}_A$ such that, for all $u, u' \in V_A$ and $v \in \gamma(u)$:

$$\begin{aligned}
\mu_A(u) \wedge \mu_B(v) \wedge J_B(v) &\quad\to\quad J_A(u) \\
\mu_A(u) \wedge \mu_B(v) \wedge C_B(v) &\quad\to\quad C_A(u) \\
\rho_A(u, u') \wedge G_A(u, u') \wedge \mu_B(v) &\quad\to\quad \bigvee_{v' \in \gamma(u')} (\rho_B(v, v') \wedge G_B(v, v')) \ . \qquad \square
\end{aligned}$$

Example. Consider the *accessibility* property of program BAKERY expressed by $\varphi : \Box(\ell_1 \to \Diamond\, \ell_3)$. To prove that BAKERY satisfies φ, we construct a chain of diagram transformations $A_0 \Rightarrow \cdots \Rightarrow A_3$. Diagrams A_0 and A_1 are depicted in Figures 2 and 3, respectively; A_2 and A_3 are depicted in Figure 4.

To obtain a diagram that keeps track of the value of the location variable ℓ, we transform A_1 into A_2 using a simulation transformation based on the mapping $\gamma(1) = \{8, 9, 10\}$, $\gamma(3) = \{7, 9, 10\}$, and $\gamma(i) = i$ for $i = 4, 5, 6$. If the product

$A_2 \otimes T_{\neg\varphi}$ is computed, the resulting guidance points to the SCS $\{4\}$, for which there is no SCS-breaking constraint (recall that, by the graphical conventions used in this paper, vertices have implicit self-loops). To complete the proof, it is necessary to show that a fair trail cannot stay in vertex 4 forever.

To do so, we first transform A_2 into A_3 using a simulation transformation that splits vertex 4 into vertices 12, 13, 14, and merges vertices 7 and 8 into vertex 11. The split is used to analyze the structure of the SCS; the only purpose of the merge is to simplify the diagram. Since for every fulfilling SCS of $A_3 \otimes T_{\neg\varphi}$ there is an implicit SCS-breaking constraint, it is possible to show $\mathcal{L}(A_3) \subseteq \mathcal{L}(\varphi)$ algorithmically, and the proof is concluded. □

Adding fairness constraints. To discard all the fulfilling SCS's of $A \otimes T_{\neg\varphi}$, it is sometimes necessary to add fairness constraints to a diagram A. For this purpose, the *fairness rules* proposed in [dAM96] can be easily adapted to the notation of this paper. In order to add a constraint, the rules first check that the constraint accepts all the fair trails of the diagram: this ensures that the set of fair trails does not change with the addition of the new constraint. Together with the simulation rule, these rules enable the proof of arbitrary temporal logic properties of fair transition systems, provided that no temporal operator appears in the scope of a quantifier, as our definition of temporal logic ensures.

6.2 Deductive Model Checking: Fairness

A trail $(v_0, s_0), (v_1, s_1), \ldots$ in a falsification diagram is *accepting* if it is fulfilling and $s_0, s_1, \ldots$ satisfies the requirements of justice and compassion for $\mathcal{S}$, that is, it is a computation of $\mathcal{S}$. As usual, $\mathcal{L}(A)$ is the set of runs corresponding to accepting trails of A.

We say that a transition τ is *fully enabled* on a vertex v if $\mu(v) \rightarrow enabled(\tau)$ is valid. An SCS S is *just* if every just transition is either taken in S or not fully enabled at some vertex in S; it is *compassionate* if every compassionate transition is taken in S or not fully enabled on all vertices in S. An SCS is *fair* if it is both just and compassionate.

We say that an SCS is *terminating* if there is no accepting trail that visits every one of its vertices infinitely often, that is, if the full SCS cannot participate in a counterexample. It is easy to see that the set of vertices that appear infinitely often along a computation must form an SCS that is fair and not terminating, in addition to being reachable and fulfilling. Thus, to show that the language of a falsification diagram is empty, we need to show that every reachable, fulfilling SCS is not fair, or that it is terminating.

To keep track of SCS's that could possibly support a counterexample computation, deductive model checking maintains a list L of candidate SCS's. This list is updated at every application of a transformation. For the initial diagram, it contains all fulfilling MSCS's (maximal strongly connected subgraphs) of the initial graph. Unreachability can be shown using the safety transformations of Section 5. To show that an SCS is not fair with respect to some transition, we need to ensure that this transition is fully enabled on all (or some) vertices of the SCS, which leads to the following transformation rule:

Rule 5 (enabled split). Consider a just or compassionate transition τ, and an SCS containing a vertex v such that $\mu(v) \wedge \neg enabled(\tau)$ is satisfiable. Then split v into two new vertices v_1, v_2, labeled with $\mu(v_1) = \mu(v) \wedge enabled(\tau)$, $\mu(v_2) = \mu(v) \wedge \neg enabled(\tau)$. □

The following transformations remove SCS's from L because they are not fair or not terminating. Note that they only modify L and not the diagram itself.

Rule 6 (uncompassionate SCS). If an SCS S in L 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 vertices in S on which one of these transitions is fully enabled. □

Rule 7 (unjust SCS). If an SCS is not just, remove it from L. □

Ranking functions can be used to prove that a fair SCS cannot contain infinite computations.

Definition 3 (ranking functions). A *well-founded domain* is a set D together with an order relation $>$, such that there is no infinite descending chain $d_0 > d_1 > d_2 > \ldots$ of elements of D. A *ranking function* δ for a diagram A is a function mapping locations of A into elements of a well-founded domain D, and is represented by the family of terms $\{\delta(u)\}_{u \in V}$ over the state variables $\mathcal{V}$. □

Rule 8 (terminating SCS). Consider an SCS $S \in L$, where $S = \{v_1, \ldots, v_n\}$. Assume that there are a ranking function δ and assertions $\varphi_1, \ldots, \varphi_n$ labeling each v_j with a formula $\varphi_i(v_j)$ such that, for all $i, j, k \in [1 \ldots n]$:

$$\begin{array}{llcl} \varphi_i(v_i) = \mathit{false} & \rho(v_j, v_k) & \rightarrow & \delta(v_j) \succeq \delta'(v_k) \\ \mu(v_i) \rightarrow \bigvee_{j=1}^{n} \varphi_j(v_i) & \varphi_i(v_j) \wedge \neg\varphi_i'(v_k) \wedge \rho(v_j, v_k) & \rightarrow & \delta(v_j) \succ \delta'(v_k) \ . \end{array}$$

Then replace S in L with the MSCS's of $S_1, \ldots, S_n$, where $S_i = \{v \in S \mid \varphi_i(v) \neq \mathit{false}\}$. □

Note that the above rule reduces the size of the SCS's in L, since $S_i \subseteq S - \{v_i\}$. Intuitively, each assertion φ_i is a "contract" containing an obligation not to visit vertex v_i: each time a contract is voided, δ is decreased. With this rule, not in [SUM96], deductive model checking is complete for proving general temporal properties of fair transition systems (again, for formulas whose quantifiers appear only within assertions).

6.3 Verification Diagrams: Fairness

Similarly to falsification diagrams, a trail $(v_0, s_0), (v_1, s_1), \ldots$ in a verification diagram is *accepting* if $s_0, s_1, \ldots,$ satisfies all the fairness requirements of $\mathcal{S}$; $\mathcal{L}(A)$ is the corresponding set of computations.

In [BMS95], Streett acceptance conditions are added to the diagram to show that it satisfies φ. Alternatively, we can show that $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$ by extending the algorithmic test of Section 3. Instead of requiring the absence of fulfilling SCS's in $A \otimes T_{\neg\varphi}$, we can use Rules 6, 7 and 8 of Section 6.2 to show that every fulfilling SCS in $A \otimes T_{\neg\varphi}$ is either terminating or unfair.

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

References

[BBC+96] N. S. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 415–418. Springer-Verlag, July 1996.

[BBM95] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, volume 976 of *LNCS*, pages 589–623. Springer-Verlag, September 1995.

[BMS95] A. Browne, Z. Manna, and H.B. Sipma. Generalized verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498, 1995.

[BMS96] A. Browne, Z. Manna, and H.B. Sipma. Hierarchical verification using verification diagrams. In 2^{nd} *Asian Computing Science Conf.*, volume 1179 of *LNCS*, pages 276–286. Springer-Verlag, December 1996.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[dAKM97] L. de Alfaro, A. Kapur, and Z. Manna. Hybrid diagrams: A deductive-algorithmic approach to hybrid system verification. In *14th Symposium on Theoretical Aspects of Computer Science*, February 1997.

[dAM96] L. de Alfaro and Z. Manna. Temporal verification by diagram transformations. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 287–299, July 1996.

[dAMS97] L. de Alfaro, Z. Manna, and H.B. Sipma. Decomposing, transforming and composing diagrams: The joys of modular verification. Submitted, 1997.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274, 1987.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proc. 5^{th} Intl. Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 97–109. Springer-Verlag, 1993.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Intl. Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[SUM96] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[VW86] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comp. Sys. Sci.*, 32:183–221, 1986.