June 1994

STeP: the Stanford Temporal Prover

by

Zohar Manna, Anuchit Anuchitanukul, Nikolaj Bjørner,
Anca Browne, Edward Chang, Michael Colón,
Luca de Alfaro, Harish Devarajan, Henny Sipma, Tomás Uribe

Department of Computer Science

Stanford University
Stanford, California 94305

STeP: the Stanford Temporal Prover*

Zohar Manna, Anuchit Anuchitanukul, Nikolaj Bjørner,
Anca Browne, Edward Chang, Michael Colón,
Luca de Alfaro, Harish Devarajan, Henny Sipma, Tomás Uribe

Computer Science Department, Stanford University
Stanford, CA 94305

Abstract

We describe the *Stanford Temporal Prover* (STeP), a system being developed to support the computer-aided formal verification of concurrent and reactive systems based on temporal specifications. Unlike systems based on model-checking, STeP is not restricted to finite-state systems. It combines *model checking* and *deductive methods* to allow the verification of a broad class of systems, including programs with infinite data domains, N-process programs, and N-component circuit designs, for arbitrary N. In short, STeP has been designed with the objective of combining the expressiveness of deductive methods with the simplicity of model checking.

The verification process is for the most part automatic. User interaction occurs mostly at the highest, most intuitive level, primarily through a graphical proof language of verification diagrams. Efficient simplification methods, decision procedures, and invariant generation techniques are then invoked automatically to prove resulting first-order verification conditions with minimal assistance.

We describe the performance of the system when applied to several examples, including the N-process dining philosopher's program, Szymanski's N-process mutual exclusion algorithm, and a distributed N-way arbiter circuit.

*This research was supported in part by the National Science Foundation under grant CCR-92-23226, by the Defense Advanced Research Projects Agency under contract NAG2-892, and, by the United States Air Force Office of Scientific Research under contract F49620-93-1-0139.

Contents

1 Introduction **1**
- 1.1 Preliminaries 2
- 1.2 System Overview 3
- 1.3 Implementation 6

2 Overview: A Simple Example **6**
- 2.1 Verification Diagrams 7
- 2.2 Proving Invariance 8
- 2.3 Proving Precedence 10
- 2.4 Proving Response 12

3 Model Checking **16**

4 Invariant Generation **19**
- 4.1 Bottom Up: Local Invariants 21
- 4.2 Bottom Up: Linear Invariants 24
- 4.3 Top-down: Strengthening 25

5 Theorem-proving support **27**
- 5.1 Simplification 27
- 5.2 Decision Procedures 28
- 5.3 First-order Prover 29
- 5.4 Interactive Prover 29

6 Examples **30**
- 6.1 N-Process Dining Philosophers Program 30
- 6.2 Szymanski's N-Process Mutual Exclusion Algorithm 31
- 6.3 Distributed N-way Arbiter Circuit 35

7 Conclusions **40**

1 Introduction

The Stanford Temporal Prover, STeP, is being developed to support the computer-aided formal verification of concurrent and reactive systems based on temporal specifications. Unlike most systems for temporal verification, STeP is not restricted to finite-state systems, but combines model checking with deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains. STeP was briefly introduced in [Man94].

A verification system which combines model checking and deductive methods offers a number of advantages over purely model checking or purely deductive approaches. Such a system should:

- Reduce the complexity of the verification task by
  - Decomposition

  Each component may be verified by the most suitable verification method. For instance, this would allow a model checker to verify an individual component even if it could not verify, because of the state explosion problem, the entire system.

- Allow verification of a broader class of systems:
  - Parameterized programs
  - Parameterized circuits
  - Systems with infinite data domains
- Automate the verification task:
  - Automatic generation of invariants
  - Effective simplifications
  - Model checking
  - Decision procedures
  - Verification rules
- Allow visual interaction:
  - Verification diagrams
- Provide debugging tools:
  - Counter-examples
  - Debugging guidance

In short, STeP has been designed with the objective:

> To combine the expressiveness of deductive methods with the simplicity of model checking.

Our development efforts have been focused, in particular, on the following areas.

First, in addition to the textual language of temporal logic, the system supports a structured visual language of *verification diagrams* [MP94a] for guiding, organizing, and displaying proofs. Verification diagrams allow the user to construct proofs hierarchically, starting from a high-level, intuitive proof sketch and proceeding incrementally, as necessary, through layers of greater detail.

Second, the system implements powerful techniques for automatic *invariant generation*. Deductive verification in the temporal framework almost always relies on finding, for a given program and specification, suitably strong (inductive) invariants and intermediate assertions. The user can typically provide an intuitive, high-level invariant, from which the system derives stronger, more detailed, *top-down invariants*. Simultaneously, *bottom-up invariants* are generated automatically by analyzing the program text. By combining these two methods, the system can often deduce sufficiently detailed invariants to carry through the entire verification process.

Finally, the system provides an integrated suite of simplifications and decision procedures for automatically checking the validity of a large class of first-order and temporal formulas. This degree of automated deduction is sufficient to handle most of the verification conditions that arise during the course of deductive verification—and the few conditions that are not solved automatically typically correspond to the critical steps of manually constructed proofs, where the user is most able to provide guidance.

The remainder of this section provides a brief overview of the system and its components. Section 2 provides a concrete description of how the system can be used, by showing how several properties of Peterson's mutual exclusion algorithm are verified. Various aspects of the system are described in greater detail in the subsequent sections, including the model checker, verification rules and verification diagrams, automatic invariant generation, and theorem-proving support for establishing verification conditions. Finally, Section 6 presents some more substantial examples: the N-process dining philosopher's program, Szymanski's N-process mutual exclusion algorithm, and a distributed N-way arbiter circuit.

1.1 Preliminaries

A *reactive system (program)* is a system that maintains an ongoing interaction with its environment. Examples of reactive systems are concurrent and distributed programs, embedded systems, and communication networks. A reactive system must be specified by its behavior over time, represented as sequences of states, i.e., *computations*. The specification of a reactive system may be given as a formula of *linear-time first-order temporal logic*, a language which combines first-order formulas with temporal operators for describing state sequences. For instance, given a program $\mathcal{P}$,

$$\mathcal{P} \quad \vDash \quad x = 0 \;\Rightarrow \Diamond\, (y = 0)$$

states that, in every computation of $\mathcal{P}$, every state satisfying $x = 0$ is eventually followed by a state satisfying $y = 0$. A temporal formula φ is *$\mathcal{P}$-valid* if $\mathcal{P} \vDash \varphi$, i.e., φ holds over all computations of $\mathcal{P}$. A state (first-order) formula[1] φ is *$\mathcal{P}$-state valid* if $\mathcal{P} \vDash \Box\, \varphi$, i.e., φ holds in all states of all computations of $\mathcal{P}$. Our goal is to show the $\mathcal{P}$-validity of a given temporal specification φ for a reactive system $\mathcal{P}$.

Our computational model for reactive systems, based on [MP91b], is that of *(fair) transition systems*. A fair transition system consists of an *initial condition*, a set of *transitions*, i.e., next-state relations, and a fairness requirement. Fair transition systems can be used to define the semantics of a simple programming language SPL which includes constructs for concurrency, nondeterministic selection, and parameterized statements. For instance,

$$\overset{N}{\underset{i=1}{||}}\; S[i]$$

where the same process S is executed N times in parallel, is a typical parameterized statement, with parameter N. A program containing a parameterized statement is a *parameterized program*.

The remainder of this paper assumes that the reader is familiar with the fair transition model, SPL, and the language of temporal logic. For an in-depth treatment of these topics, see [MP91b].

1.2 System Overview

Figure 1 presents a high-level overview of the STeP system. A brief description of each component follows.

Input The basic input to STeP is an SPL program $\mathcal{P}$ and a temporal logic formula φ which expresses the property of $\mathcal{P}$ to be verified. The SPL program is modeled as a fair transition system S. Even though SPL can be used to describe both software and hardware systems, STeP is not restricted to SPL, and can be used to verify any system that can be modeled as a fair transition system.

Verification Diagrams The preferred approach to constructing a proof is through *verification diagrams*. Through a graphical user interface, the user can draw a diagram that represents the proof of a given formula φ (see Section 2.1). The corresponding verification conditions are generated automatically from the verification diagram and are checked by the automatic prover.

[1] We refer to first-order formulas as state formulas or assertions.

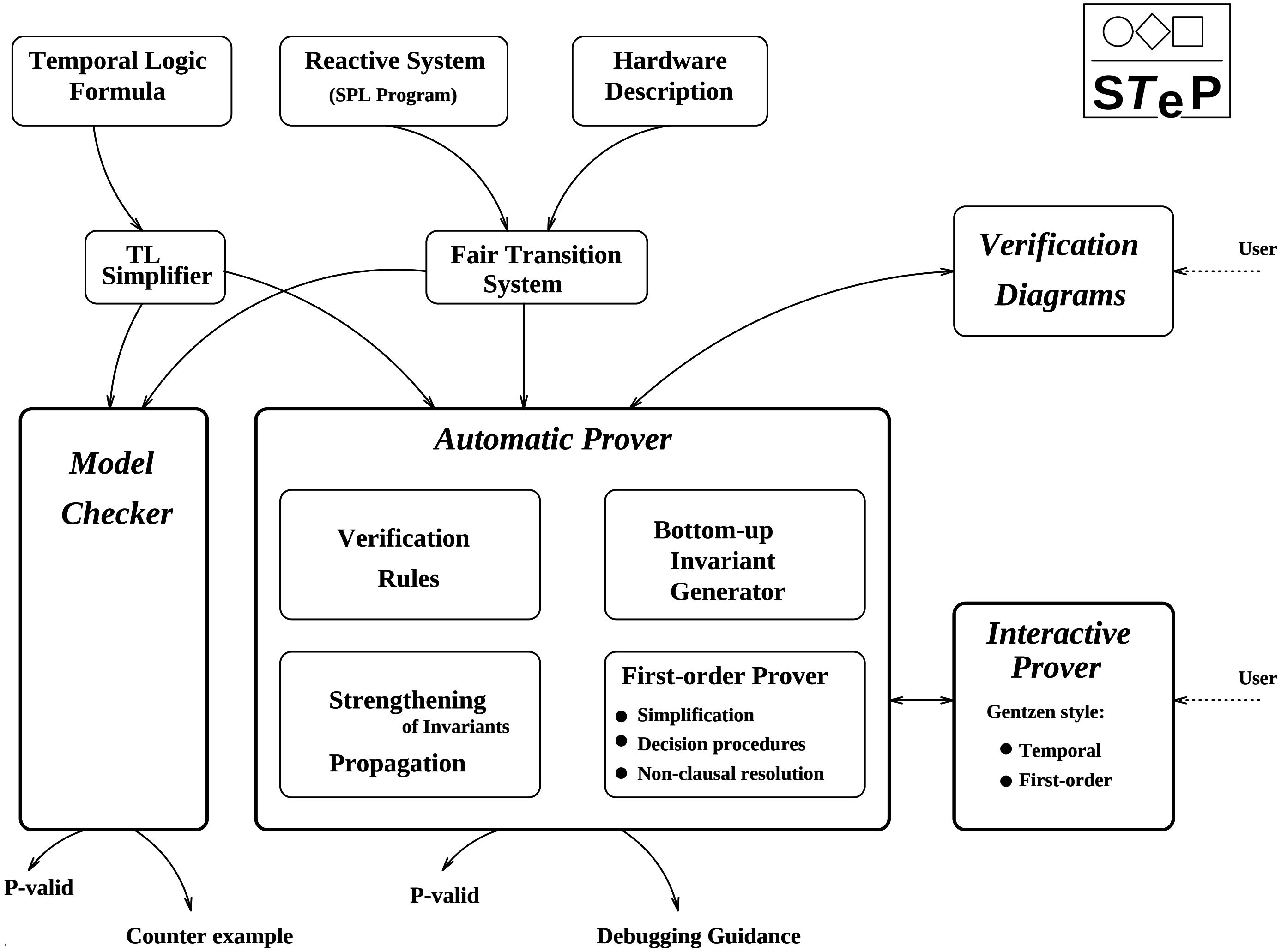

Figure 1: An overview of the STeP system

Model Checking The *model checker* takes as input the fair transition system S and the (simplified) formula φ. It tries to show that φ is valid for S by searching for a counterexample in the form of a computation satisfying $\neg\varphi$ (see Section 3). For finite-state systems, the algorithm guarantees termination (up to space/time limitations) with a positive answer or counterexample. The model-checker may also be applied to infinite-state systems; termination with a positive answer or counterexample is not guaranteed in this case.

Automatic Prover This is the main module of the deductive component of STeP, and comprises four distinct subcomponents that interact with each other in the course of a proof:

- *Verification rules* are used to reduce the proof of $\mathcal{P}$-validity of a temporal formula φ to the proof of validity of a set of first-order formulas, called *verification conditions*.
- *Bottom-up invariants*, generated by static analysis of the transition system and the program text, are used to simplify verification conditions.
- The first-order prover (subsections 5.1- 5.3) is responsible for simplifying verification conditions and proving their validity if possible. This is done with a combination of (contextual) rewriting techniques, decision procedures, and general theorem proving. This prover can also use previously proven invariants.
- A number of automatic techniques, including invariance strengthening and propagation, are available if the first-order prover is unable to prove all verification conditions. These techniques are primarily intended to strengthen invariants that are not inductive and to generate intermediate assertions.

Interactive Prover If the automatic prover is not able to prove a verification condition, the user can choose to give the simplified but unproven verification condition to the interactive prover, where, if it is indeed valid, it can be proved with some user guidance (see subsection 5.4).

If the formula is not valid, the user may be able to receive some suggestions on why it is not valid. This information can then be used to modify the program or strengthen an intermediate assertion or invariant. Note that the availability of the model checker allows the user to search for a counterexample while simultaneously attempting an interactive proof.

The interactive prover also features deduction rules for temporal logic that can be used to simplify and prove temporal formulas.

1.3 Implementation

STeP is written in Standard ML of New Jersey with the exception of the model checker, which is implemented in C.

A prototype X-windows version of the graphical user interface is being developed using the eXene library for Concurrent ML.

Currently, after six months of implementation, the size of the source code is approximately 40,000 lines.

2 Overview: A Simple Example

This section describes how STeP can be applied to the deductive verification of Peterson's mutual exclusion algorithm, as implemented by program PET of Figure 2. In fact, since program PET is finite-state, each of the properties proved below can also be verified automatically using the STeP model checker.

$$
\begin{array}{lll}
\textbf{local} & y_1, y_2 & : \textbf{boolean where } y_1 = \mathrm{F}, y_2 = \mathrm{F} \\
 & s & : \textbf{integer where } s = 1
\end{array}
$$

$$
\begin{array}{l}
P_1 :: \left[\begin{array}{l}
\ell_0\!: \ \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\!: \ \textbf{noncritical} \\
\ell_2\!: \ y_1 := \mathrm{T} \\
\ell_3\!: \ s := 1 \\
\ell_4\!: \ \textbf{await } \neg y_2 \ \vee \ s = 2 \\
\ell_5\!: \ \textbf{critical} \\
\ell_6\!: \ y_1 := \mathrm{F}
\end{array}\right]
\end{array}\right] \\
|| \\
P_2 :: \left[\begin{array}{l}
m_0\!: \ \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
m_1\!: \ \textbf{noncritical} \\
m_2\!: \ y_2 := \mathrm{T} \\
m_3\!: \ s := 2 \\
m_4\!: \ \textbf{await } \neg y_1 \ \vee \ s = 1 \\
m_5\!: \ \textbf{critical} \\
m_6\!: \ y_2 := \mathrm{F}
\end{array}\right]
\end{array}\right]
\end{array}
$$

Figure 2: Program PET (Peterson's algorithm for mutual exclusion).

In program PET, the basic mechanism protecting access to the critical sections (represented by statements ℓ_5 and m_5), is provided by the boolean variables y_1 and y_2. Each process P_i, for $i = 1, 2$, that is interested in entering its critical section sets its y_i variable to T. On exiting the critical section, the corresponding y_i is reset to

F.

The problem with this approach is that the two processes may arrive at their waiting positions, ℓ_4 and m_4 respectively, at about the same time, with both $y_1 = y_2 =$ T. If the only criterion for entry to the critical section was that the y_i of the competitor be F, this situation would result in a deadlock (tie).

The variable s is intended for breaking such ties. It may be viewed as a *signature*, in the sense that each process that sets its y_i variable to T also writes its identity number in s at the next step taken by the process. Then, if both processes are at the waiting position, the first to enter will be P_i such that $s \neq i$. For $i = 1, 2$, let $\bar{\imath}$ denote the index of the other process. The fact that $s \neq i$ implies that $s = \bar{\imath}$, which means that the competitor $P_{\bar{\imath}}$ was the *last* to assign a value to s. Therefore P_i should have priority.

We first introduce our graphical proof language of *verification diagrams*, and we then illustrate the deductive verification of a few properties of program PET. Details about our specification language can be found in [MP91b]. The deductive methods used are discussed in more detail in [MP91a] and [MP94b]. A more extensive explanation of verification diagrams is given in [MP94a].

2.1 Verification Diagrams

In proofs of properties of reactive systems, it is typically necessary to consider several assertions (state formulas) at the same time and to determine which transitions lead from one assertion to another. A *verification condition* $\{\varphi\}\tau\{\psi\}$ is an assertion stating that, whenever τ is taken from a state satisfying φ, the resulting state must satisfy ψ. It is convenient to visualize these conditions with a diagram that summarizes the assertions under consideration and the possible transitions between them.

A *verification diagram* [MP94a] is a directed labeled graph where:

- *Nodes* in the graph are labeled by assertions. We will often refer to the node by the assertion labeling it.

- *Edges* in the graph represent transitions between assertions. Each edge connects one assertion to another and is labeled by the name of a transition in the program. We refer to an edge labeled by τ as a *τ-edge*.

- One of the nodes may be designated as a *terminal node* ("goal" node). In the graphical representation, this node is distinguished by having a boldface boundary. No edges depart from a terminal node.

Verification diagrams provide a concise representation of sets of verification conditions as follows. For a nonterminal node (labeled by) φ and transition τ, let $\varphi_1, \ldots, \varphi_k$ be the nodes reached by τ-edges departing from φ. We say that $\varphi_1, \ldots, \varphi_k$ are the *τ-successors* of φ. The *verification condition associated with* φ and τ is given by:

$$\{\varphi\}\ \tau\ \{\varphi\ \vee\ \varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}.$$

In other words, there is an *implicit* τ-edge connecting φ to itself. Note that for the case $k = 0$, i.e., no τ-edges depart from φ, the verification condition associated with φ and τ is given by:

$$\{\varphi\}\ \tau\ \{\varphi\}.$$

No verification conditions are associated with terminal nodes.

Since a diagram provides a succinct representation of a large set of verification conditions, it can often present a useful and illuminating overview of a complex proof.

A diagram is *valid over program $\mathcal{P}$ ($\mathcal{P}$-valid)* if all the verification conditions associated with nodes of the diagram are $\mathcal{P}$-state valid.

2.2 Proving Invariance

The mutual exclusion property for program PET is expressed by the following *safety formula*:

$$\varphi_{ME}:\ \square\neg(at\_\ell_5\ \wedge\ at\_m_5).$$

where at_ℓ_5 and at_m_5 are predicates stating that control is at statements ℓ_5 and m_5, respectively.

Rule INV

Using deductive methods, the following verification rule, rule INV, can be used to prove that the state formula p is invariant in every computation of a program $\mathcal{P}$, where Θ is the initial condition and $\mathcal{T}$ is the set of transitions of the transition system corresponding to $\mathcal{P}$:

$$\texttt{INV}\qquad \begin{array}{l} \text{For strengthening assertion } \varphi: \\ \quad \text{S1.}\quad \Theta \rightarrow \varphi \\ \quad \text{S2.}\quad \{\varphi\}\ \mathcal{T}\ \{\varphi\} \\ \quad \text{S3.}\quad \varphi \rightarrow p \\ \hline \qquad\quad \square\, p \end{array}$$

The rule states that in order to establish the $\mathcal{P}$-validity of the temporal formula $\square\, p$, it suffices to find an assertion φ, strengthening p, such that premises S1–S3 are $\mathcal{P}$-state valid. Premise S1 states that the initial condition Θ implies φ. Premise S2 states that the verification condition $\{\varphi\}\ \tau\ \{\varphi\}$ holds for each transition $\tau \in \mathcal{T}$, i.e., if τ is taken from any state satisfying φ, the result is a state also satisfying φ. If premises S1 and S2 hold for φ, then φ is called an *inductive* assertion; by induction, φ holds in every state of a computation. By premise S3, it follows that p also holds in every state of a computation.

Note that all the premises of rule `INV` are state formulas, whereas the conclusion is a temporal formula. This is typical of the deductive methodology, which applies verification rules to reduce the proof of temporal formulas to the proof of first-order conditions.

PET: Mutual Exclusion

To prove mutual exclusion for program PET, p is taken to be:

$$p: \quad \neg(at\_\ell_5 \wedge at\_m_5).$$

In this example, as is often the case, verification requires identifying a suitable strengthening assertion φ. To assist in this task, STeP provides built-in mechanisms for automatically generating low-level invariants and automatically strengthening proposed invariants suggested by the user.

Low-level invariants (also called "bottom-up invariants") are guaranteed to be invariants by the way they are generated, so they can be used in establishing the premises of the above verification rule. The following automatically generated invariants are necessary for establishing mutual exclusion for program PET:

$$\chi_1: \quad at\_\ell_{3..6} \rightarrow y_1$$
$$\chi_2: \quad at\_m_{3..6} \rightarrow y_2$$

Strengthened invariants (also called "top-down invariants") are obtained by weakest precondition propagation. Consider, for instance, statement ℓ_4. If the corresponding transition τ_{ℓ_4} is never to violate mutual exclusion, it must be the case that $\neg y_2 \vee s = 2$ is false whenever control is at ℓ_4 and m_5. After simplifying with respect to χ_2, this yields the following strengthened invariant:

$$\varphi_1: \quad at\_\ell_4 \wedge at\_m_5 \quad \rightarrow \quad \neg(s = 2).$$

Similarly:

$$\varphi_2: \quad at\_\ell_5 \wedge at\_m_4 \quad \rightarrow \quad \neg(s = 1).$$

Thus, for this example, the proof of mutual exclusion is entirely automatic. First, STeP identifies the specification as a safety property and invokes rule `INV`. Since p is not inductive, the proof does not succeed. Therefore, bottom-up invariants, including χ_1 and χ_2, are generated. The system again attempts to establish the invariance of p, and in doing so, generates the strengthened invariant φ:

$$\varphi: \quad p \wedge \varphi_1 \wedge \varphi_2.$$

Finally, STeP is able to prove each of the premises of rule `INV`.

More typically, however, the user must provide direction to the system by suggesting a strengthening assertion φ. Even if φ is not immediately inductive, the system can apply invariant strengthening heuristics to complete the proof.

Invariant generation and strengthening methods are discussed more fully in Section 4.

2.3 Proving Precedence

The property of 1-bounded overtaking for process P_1 of program PET may be expressed by the following "nested waiting-for formula," where the wait-for ("weak until") operator $\mathcal{W}$ is right associative:

$$\varphi_B: \quad at\_\ell_4 \;\Rightarrow\; (\neg at\_m_5)\, \mathcal{W}\, at\_m_5\, \mathcal{W}\, (\neg at\_m_5)\, \mathcal{W}\, at\_\ell_5$$

In other words, once process P_1 has reached statement ℓ_4, process P_2 may enter its critical section m_5 at most once before P_1 enters its critical section.

Rule WAIT

The following verification rule, rule WAIT, may be used to establish nested waiting-for formulas for a program $\mathcal{P}$:

$$\begin{array}{ll} \texttt{WAIT} & \text{For intermediate assertions } \varphi_n, \ldots, \varphi_0 : \\ & \begin{array}{lll} \text{W1.} & p \rightarrow \bigvee_{j=0}^{n} \varphi_j & \\ \text{W2.} & \{\varphi_i\}\ \mathcal{T}\ \{\bigvee_{j=0}^{i} \varphi_j\} & \text{for } i = 1, \ldots, n \\ \text{W3.} & \varphi_i \rightarrow q_i & \text{for } i = 0, \ldots, n \\ \hline & p \;\Rightarrow\; q_n\, \mathcal{W}\, q_{n-1} \cdots q_1\, \mathcal{W}\, q_0 & \end{array} \end{array}$$

This rule states that to establish the $\mathcal{P}$-validity of the nested-for formula, it suffices to find intermediate assertions $\varphi_n, \ldots, \varphi_0$ such that premises W1–W3 are $\mathcal{P}$-state valid. Premise W1 states that every state satisfying p also satisfies some φ_i, for some intermediate assertion φ_i. By premise W2, every φ_i-state, for $i = 1, \ldots, n$, is followed by a φ_j-state, for $j = 0, \ldots, i$. It follows that

$$p \;\Rightarrow\; \varphi_n\, \mathcal{W}\, \varphi_{n-1} \cdots \varphi_1\, \mathcal{W}\, \varphi_0$$

holds for every computation of $\mathcal{P}$, and by monotonicity, premise W3 establishes the desired result.

Wait-for Diagram

We can visualize the proof with a verification diagram, in particular a wait-for diagram. A *wait-for diagram* is a weakly acyclic verification diagram with nodes $\varphi_n, \ldots, \varphi_0$, where φ_0 is a terminal node, satisfying the following requirement: whenever node φ_i is connected by an edge to node φ_j, then $i \geq j$. $\mathcal{P}$-valid wait-for diagrams can be used to establish the $\mathcal{P}$-validity of nested wait-for formulas, as stated by the following claim:

Claim 1 (WAIT-FOR) *A $\mathcal{P}$-valid wait-for diagram establishes that the formula*

$$\bigvee_{j=0}^{m} \varphi_j \;\Rightarrow\; \varphi_m \;\mathcal{W}\; \varphi_{m-1} \cdots \varphi_1 \;\mathcal{W}\; \varphi_0$$

is $\mathcal{P}$-valid.

If, in addition, we can establish the $\mathcal{P}$-state validity of the following implications:

$$p \rightarrow \bigvee_{j=0}^{m} \varphi_j \qquad \textit{and} \qquad \varphi_i \rightarrow q_i \quad \textit{for} \quad i = 0, \ldots, m$$

then we we can conclude the $\mathcal{P}$-validity of:

$$p \;\Rightarrow\; q_m \;\mathcal{W}\; q_{m-1} \cdots q_1 \;\mathcal{W}\; q_0$$

PET: 1-Bounded Overtaking

The following intermediate assertions can be used to establish 1-bounded overtaking for program PET:

$$\begin{array}{ll}
\varphi_3: & at_\ell_4 \wedge at_m_4 \wedge s = 1 \\
\varphi_2: & at_\ell_4 \wedge at_m_5 \\
\varphi_1: & at_\ell_4 \wedge (at_m_{0..3,6} \vee (at_m_4 \wedge s = 2)) \\
\varphi_0: & at_\ell_5
\end{array}$$

The wait-for diagram of φ_B for program PET is given in Figure 3. It presents useful information that is not found in the straightforward listing of φ_3, φ_2, φ_1, and φ_0 above. For instance, consider premise W2 with respect to φ_3 and transition τ_{m_4},

$$\{\varphi_3\}\ \tau_{m_4}\ \{\varphi_3 \vee \varphi_2 \vee \varphi_1 \vee \varphi_0\}$$

stating that:

> if τ_{m_4} is taken from a state satisfying φ_3, then the resulting state must satisfy $\varphi_3 \vee \varphi_2 \vee \varphi_1 \vee \varphi_0$.

However, in the verification diagram of Figure 3, there is a single arrow labeled m_4 departing from φ_3, indicating that

> if τ_{m_4} is taken from a state satisfying φ_3, then the resulting state must satisfy $\varphi_3 \vee \varphi_2$,

yielding the more precise verification condition:

$$\{\varphi_3\}\ \tau_{m_4}\ \{\varphi_3 \vee \varphi_2\}$$

As another example, premise W2 with respect to φ_3 and transition τ_{ℓ_4} yields the verification condition:

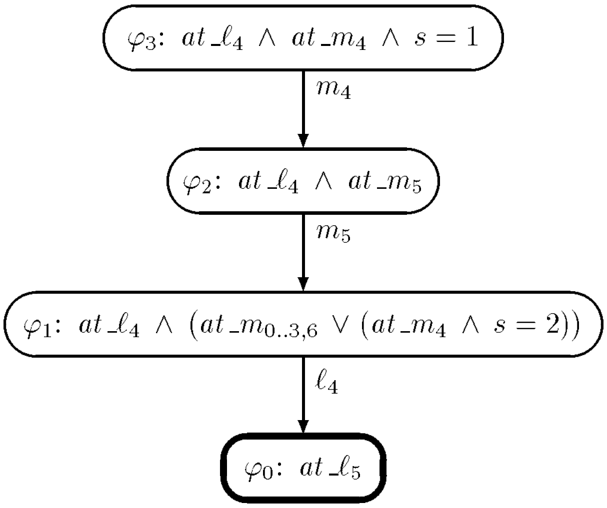

Figure 3: Verification diagram for 1-bounded overtaking.

$$\{\varphi_3\}\ \tau_{\ell_4}\ \{\varphi_3 \vee \varphi_2 \vee \varphi_1 \vee \varphi_0\}$$

whereas the verification diagram yields:

$$\{\varphi_3\}\ \tau_{\ell_4}\ \{\varphi_3\}$$

Both conditions can be established automatically, since φ_3 and the bottom-up invariant χ_2: $at\_m_{3..6} \rightarrow y_2$ imply that τ_{ℓ_4} cannot be taken from a φ_3-state, but the stronger condition can be verified more efficiently. For more complicated proofs, this efficiency is an important advantage. Furthermore, this gain is obtained at almost no cost, since it is in any case intuitive for the user to connect φ_3 to φ_2 by only the single arrow m_4.

In this case, for $n = 3$ and the number of transitions $|\mathcal{T}| = 16$, premise W2 yields 48 verification conditions. Once the user supplies the intermediate assertions $\varphi_0, \ldots, \varphi_3$, either textually or graphically, all 48 verification conditions are proved automatically, as well as premises W1 and W3. Again, as pointed out above, the automatically generated bottom-up invariants are used for these proofs.

2.4 Proving Response

The 1-bounded overtaking property for program PET does not state that P_1 is guaranteed eventual access to its critical section. The accessibility property is expressed as the following *response formula*:

$$\varphi_R: \quad at\_\ell_2 \Rightarrow \Diamond\ at\_\ell_5$$

Rule CHAIN

The following verification rule, rule CHAIN, can be used to prove simple response formulas like φ_R, i.e., formulas of the form

$$p \Rightarrow \diamondsuit q$$

where p and q are state formulas.

`CHAIN` For intermediate assertions $\varphi_n, \ldots, \varphi_1$ and helpful transitions $\tau_n, \ldots, \tau_1$:

$$\begin{array}{lll}
\text{R1.} & p \rightarrow q \lor \bigvee_{j=1}^{n} \varphi_j & \\
\text{R2.} & \{\varphi_i\}\ \mathcal{T}\ \{q \lor \bigvee_{j \le i} \varphi_j\} & \text{for } i = 1, \ldots, n \\
\text{R3.} & \{\varphi_i\}\ \tau_i\ \{q \lor \bigvee_{j < i} \varphi_j\} & \text{for } i = 1, \ldots, n \\
\text{R4.} & \varphi_i \rightarrow En(\tau_i) & \text{for } i = 1, \ldots, n \\
\hline
 & p \ \Rightarrow\ \diamondsuit q &
\end{array}$$

The rule states that to establish the $\mathcal{P}$-validity of response formulas of the above form, it suffices to identify a sequence of intermediate assertions $\varphi_n, \ldots, \varphi_1$, and a set of just transitions $\tau_n, \ldots, \tau_1$ such that the premises R1–R4 are $\mathcal{P}$-state valid. Premise R1 states that p implies q (in which case the proof is finished) or one of the intermediate assertions φ_i. Premise R2 requires that taking any transition from a φ_i-position results in a next position satisfying φ_j, for some $j \le i$. Premise R3 requires that taking the just ("helpful") transition τ_i from a φ_i-position results in a next position which satisfies φ_j for $j < i$. Premise R4 claims that the just transition τ_i is enabled at every φ_i-position.

Response Diagram

Like a proof of precedence properties, we can visualize the proof of such response properties with a verification diagram, in this case a response diagram. A *response diagram* is a verification diagram with nodes $\varphi_n, \ldots, \varphi_0$, and two kinds of edges (distinguished by single and double lines) that satisfies the following requirements:

- If a single edge connects node φ_i to node φ_j, then $i \ge j$.
- If a double edge connects node φ_i to node φ_j, then $i > j$.
- Every node φ_i, $i > 0$, has a double edge departing from it. This identifies the transition labeling such an edge as *helpful* for assertion φ_i. All helpful transitions must be just.

- No transition can label both a single and a double edge departing from the same node.
- φ_0 is a terminal node.

The first two requirements ensure that the diagram is weakly acyclic, i.e., whenever node φ_i is connected by an edge (single or double) to node φ_j, $j \leq i$. The stronger second requirement ensures that the subgraph based on the double edges is acyclic, forbidding self-connections by double edges. The third requirement demands that every nonterminal assertion (i.e., φ_i for $i > 0$) has at least one helpful transition associated with it.

The verification condition associated with φ and τ for the case that τ labels only single edges from φ is as defined in Section 2.1. If τ labels any double edges from φ, where $\varphi_1, \ldots, \varphi_k$, $k > 0$, are the τ-successors of φ, then the verification condition associated with φ and τ is as follows:

$$\{\varphi\}\ \tau\ \{\varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}$$

Transition τ, identified as helpful, is required to lead away from φ. This, with the requirement of acyclicity, implies that when this transition is taken from a φ-state, the computation gets closer to the goal φ_0.

Furthermore if τ labels a double edge departing from φ, we require:

$$\varphi\ \rightarrow\ En(\tau)$$

That is, a transition helpful for φ is enabled on all φ-states. We refer to this requirement as the *enabling requirement.*

A response diagram is said to be *valid over program $\mathcal{P}$ ($\mathcal{P}$-valid)* if all the verification conditions and enabling requirements are $\mathcal{P}$-state valid for every nonterminal node φ_i, $i > 0$, and every transition τ.

The consequences of having a $\mathcal{P}$-valid response diagram are stated in the following claim.

Claim 2 (RESPONSE) *A $\mathcal{P}$-valid response diagram establishes that the response formula*

$$\bigvee_{j=0}^{m} \varphi_j\ \Rightarrow\ \Diamond\, \varphi_0$$

is $\mathcal{P}$-valid.

If, in addition, we can establish the $\mathcal{P}$-state validity of the following implications:

$$p\ \rightarrow\ \bigvee_{j=0}^{m} \varphi_j \qquad \textit{and} \qquad \varphi_0\ \rightarrow\ q$$

then we can conclude the $\mathcal{P}$-validity of:

$$p\ \ \Rightarrow\ \ \Diamond\, q$$

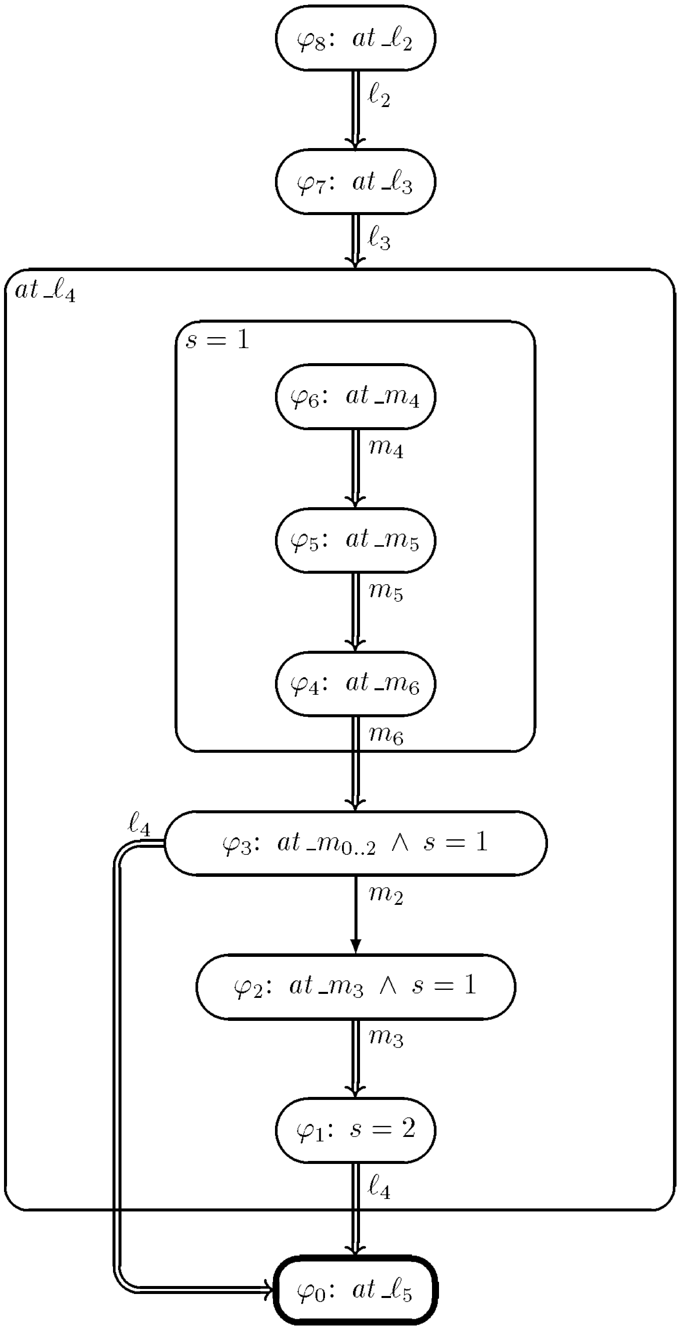

Figure 4: Verification diagram for accessibility.

PET: Accessibility

Figure 4 presents the response diagram establishing φ_R. The diagram is hierarchical. In particular, the nodes labeled φ_6, φ_5, and φ_4 are contained in the compound node labeled $s = 1$, which itself is contained with nodes φ_3, φ_2, and φ_1 in the compound node labeled at_ℓ_4. These encapsulations were inspired by Statecharts [Har87]. A hierarchical diagram may be interpreted as follows:

- The label of a compound node is implicitly a conjunct in the label of each of its subnodes.
- Each arrow from a compound node represents an arrow from each of its subnodes, with the same label and destination node.
- Each arrow to a compound node represents an arrow to each of its subnodes, with the same label and source node.

Thus, the diagram in Figure 4 may be presented explicitly by adding an arrow labeled ℓ_3 from φ_7 to each node $\varphi_6, \ldots, \varphi_1$ (deleting the original arrow from φ_7), adding $s = 1$ as a conjunct in the label of each node $\varphi_6, \ldots, \varphi_4$ (deleting the compound node labeled $s = 1$), and adding at_ℓ_4 as a conjunct in the label of each node $\varphi_6, \ldots, \varphi_1$ (deleting the compound node labeled at_ℓ_4).

The resulting diagram satisfies the requirements of the response diagram, i.e., it is acyclic, it has a goal node φ_0 (with no departing arrows), and there is a double arrow from each node, excluding the goal node, along a path to the goal node. Each double arrow represents a claim of single-step progress. For instance, the double arrow from φ_3 to φ_0 labeled ℓ_4 indicates that, if φ_3 holds "long enough," then eventually statement ℓ_4 will be executed and will lead to a φ_0-state. Note that, according to the diagram, it is also possible for m_2 to be taken from a state satisfying φ_3, leading to a φ_2-state.

Single-step progress is assured by requiring that, for each helpful transition τ labeling a double arrow from a node labeled φ, it must be the case that τ is *just*, i.e., has an associated weak fairness requirement, and that τ is enabled on every state satisfying φ. An "unhelpful" transition such as m_2 from φ_3 is indicated by a single arrow.

Given the diagram in Figure 4, the system is able to check all the associated verification conditions and establish the desired accessibility property for program PET.

3 Model Checking

Generally speaking, the model checking problem is to determine whether a given logical formula can be satisfied by some model by exploring the state space of the system. In STeP the logical formula is taken to be the program specification, expressed in (linear-time) temporal logic, and a model is some computation of the program.

STeP provides an efficient implementation of the model checking algorithm described in [MP94b] and originally proposed in [VW86]. We only sketch the algorithm here.

Given a program $\mathcal{P}$ and a linear-time temporal formula φ, the algorithm determines whether there exists a computation of $\mathcal{P}$ that satisfies $\neg\varphi$. The approach is based on automata: the program is represented as a transition graph, which is viewed as a generator $\mathcal{A}_{\mathcal{P}}$ of infinite words over the program's state space, and φ is viewed as an acceptor $\mathcal{A}_{\varphi}$ of infinite words.

There are several types of automata for infinite words. In our algorithm we use Streett automata [Str82]. A Streett automaton $\mathcal{A}$ consists of the following components:

- a finite set of nodes N,
- an initial node n_0,
- a finite set of edges E, and
- an acceptance list $\mathcal{L} = (R_1, P_1), \ldots, (R_m, P_m)$. $R_i \subseteq N$ are called *recurrent nodes* and $P_i \subseteq N$ are called *persistent nodes*.

An infinite sequence of automaton nodes, $n_0, n_1, \ldots$, is accepted by $\mathcal{A}$ if

- n_0 is the initial node of $\mathcal{A}$, and
- for every $i = 0, 1, \ldots$, there exists an edge $e \in E$ connecting n_i to n_{i+1}, and
- for the set of nodes, N_{inf}, that appear infinitely often, for each $i = 1, \ldots, m$, either $N_{inf} \cap R_i \neq \emptyset$, or $N_{inf} \subseteq P_i$.

To represent the fairness requirements of $\mathcal{P}$, *recurrent edges* are added to the Streett acceptance list [HSB93]. The acceptance list of this modified Streett automaton (also called Edge/Node Streett automaton) is thus a list of triplets, $\mathcal{L} = (R_1, P_1, E_1), \ldots, (R_m, P_m, E_m)$, where R_i and P_i are as before, and $E_i \subseteq E$ is a set of recurrent edges. The acceptance condition of an Edge/Node Streett automaton is the same as above except for the third condition, which becomes

- at least one of the following holds for each $i = 1, \ldots, m$:

  $$N_{inf} \cap R_i \neq \emptyset \quad \text{or} \quad N_{inf} \subseteq P_i, \quad \text{or} \quad E_{inf} \subseteq E_i,$$

  where N_{inf} is, as before, the set of nodes that appear infinitely often and E_{inf} is the set of edges that appear infinitely often.

When translating a fair transition system into an Edge/Node Streett automaton, each fair transition τ contributes one triplet (R_τ, P_τ, E_τ) to the Streett acceptance list. E_τ contains all edges labeled by τ for both compassionate and just transitions; for a just (weakly fair) transition, $P_\tau = \emptyset$ and R_τ contains all nodes labeled by an assertion on which τ is disabled, whereas for a compassionate (strongly fair) transition these are reversed: $R_\tau = \emptyset$ and P_τ contains all nodes labeled by an assertion on which τ is disabled.

In this representation, showing that $\mathcal{P}$ satisfies φ reduces to showing that

$$L(\mathcal{A}_\mathcal{P}) \subseteq L(\mathcal{A}_\varphi)$$

where $L(\mathcal{A}_\mathcal{P})$ is the language generated by $\mathcal{A}_\mathcal{P}$ (i.e., the set of all computations of $\mathcal{P}$), and $L(\mathcal{A}_\varphi)$ is the language accepted by $\mathcal{A}_\varphi$ (i.e., the set of all sequences that satisfy φ). The set inclusion given above can be rewritten as

$$L(\mathcal{A}_\mathcal{P}) \cap \overline{L(\mathcal{A}_\varphi)} = \emptyset$$

or alternatively:

$$L(\mathcal{A}_\mathcal{P}) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset$$

This can also be written as

$$L(\mathcal{B}_{\mathcal{P},\neg\varphi}) = \emptyset$$

where $\mathcal{B}_{\mathcal{P},\neg\varphi}$ represents the product automaton, also called the *behavior automaton*, of $\mathcal{A}_\mathcal{P}$ and $\mathcal{A}_{\neg\varphi}$. The nodes of $\mathcal{B}_{\mathcal{P},\neg\varphi}$ are labeled by pairs (s, n), where s is an element of the state space of $\mathcal{P}$ and n is a node of $\mathcal{A}_{\neg\varphi}$, and the edges are labeled by transitions of $\mathcal{P}$. The acceptance list of $\mathcal{B}_{\mathcal{P},\neg\varphi}$ is the union of the acceptance list of $\mathcal{A}_{\neg\varphi}$ and that of $\mathcal{A}_\mathcal{P}$.

In the context of fair transition systems, the automaton $\mathcal{B}_{\mathcal{P},\neg\varphi}$ is not empty iff it contains a *fulfilling subgraph*, i.e., a subgraph that satisfies the Streett acceptance criteria which result from the fulfillment requirements associated with formulas such as $\diamondsuit\, p$ and the fairness requirements of $\mathcal{P}$. A subgraph S satisfies the Streett acceptance criteria if (1) it is a strongly connected component, and (2) either $S \cap R_i \neq \emptyset$, $S \subseteq P_i$, or there exists $e \in E_i$ such that e connects two nodes in S, for every $i = 1, \ldots, m$.

Following this approach, the algorithm is given as follows. Given a (linear-time) temporal formula φ, the Streett automaton $\mathcal{A}_{\neg\varphi}$ is constructed using the algorithm presented in [KMMP93]. Starting from $\mathcal{A}_{\neg\varphi}$ and the transition graph of $\mathcal{P}$, $\mathcal{B}_{\mathcal{P},\neg\varphi}$ is incrementally constructed. The algorithm adds a maximal strongly connected component is found, and it then checks whether this component has a fulfilling subgraph. The algorithm terminates when it finds a fulfilling subgraph, or when it cannot add any new nodes. In the first case the corresponding computation is returned as a counter example. In the latter case the $\mathcal{P}$-validity of φ has been established.

To illustrate the algorithm we apply it to program INF and the $\mathcal{P}$-valid property:

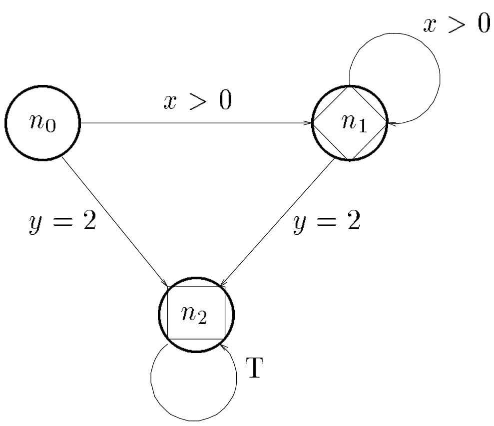

Figure 5: Automaton for $(x > 0)\ \mathcal{W}\ (y = 2)$.

$$\varphi:\ \neg((x > 0)\ \mathcal{W}\ (y = 2))$$

INF has the following transition relations:

$$\begin{array}{lccccc} \tau_1: & 0 \le x < 3 & \wedge & x' = x + 1 & \wedge & y' = y \\ \tau_2: & 0 < y < 3 & \wedge & y' = y + 1 & \wedge & x' = x \\ \tau_3: & x' = 0 & \wedge & y' = 1 & & \\ \tau_I: & x' = x & \wedge & y' = y & & \text{(idling transition)} \end{array}$$

INF's justice set is $\mathcal{J} = \{\tau_1, \tau_2, \tau_3\}$.

The automaton for $\neg\varphi$ is shown in Figure 5. Part of INF's (infinite) transition graph is shown in Figure 6; in this figure, $\langle a, b\rangle$ stands for the state where $x = a, y = b$. The algorithm constructs the behavior automaton shown in Figure 7, which has three strongly connected components: (s_0, n_1), (s_1, n_1), and (s_2, n_1). None of these are fulfilling: all of them fail to satisfy the acceptance triplet originating from transition τ_3 ($R_3 = \emptyset$, $P_3 = \emptyset$, $E_3 = \{\text{edge labeled by } \tau_3\}$). Intuitively, none of these subgraphs is fair with respect to τ_3: τ_3 is enabled infinitely often but never taken. Therefore no computation of INF satisfies $(x > 0)\ \mathcal{W}\ (y = 2)$, establishing the $\mathcal{P}$-validity of $\varphi:\ \neg((x > 0)\ \mathcal{W}\ (y = 2))$.

This example illustrates how the model checker is able to verify a property of an infinite-state program.

4 Invariant Generation

A large class of invariants can be generated automatically by STeP to simplify the verification process. Each of the invariant generation techniques can be loosely classified as *bottom-up* or *top-down*. In the bottom-up approach only the program is considered: inductive assertions are deduced from the program structure. The

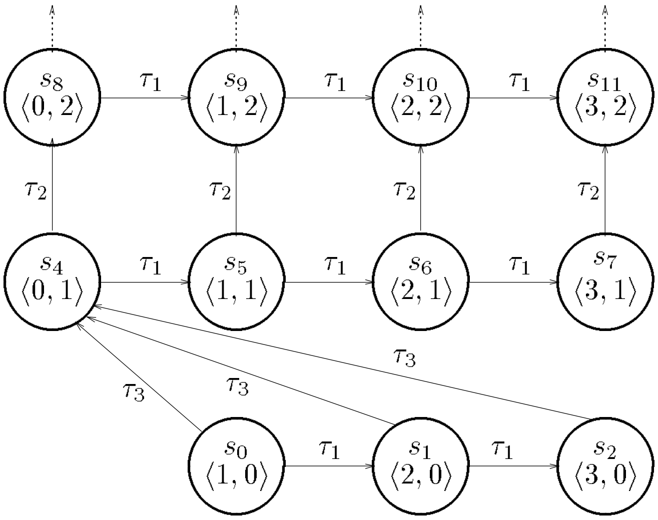

Figure 6: Part of the state transition graph.

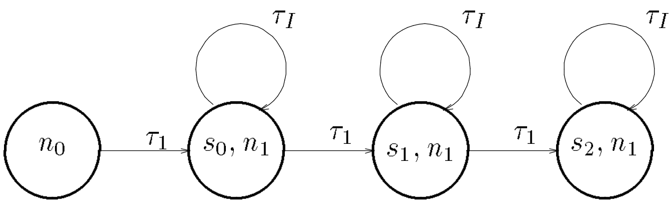

Figure 7: Behavior automaton.

top-down approach is goal-directed: it considers the property that has to be proven and strengthens some of its parts to produce an inductive assertion.

4.1 Bottom Up: Local Invariants

Local invariants are bottom-up invariants which relate program control predicates to assertions involving data variables. The system uses several heuristics for generating local invariants. An important concept in this context is ownership of variables: a variable y is *owned* by a statement ℓ if no transition parallel to ℓ modifies y.

Reaffirmed Invariants

The simplest type of bottom-up inductive assertions are those which are guaranteed to hold after execution of each transition that interferes with them, without any assumption about the state before the execution.

For example, a reaffirmed invariant can be deduced in the case where a transition sets a variable y to a constant expression c:

$$\ell_1: \quad y := c \quad \ell_2:$$

If y is owned by ℓ_2 we may conclude the inductiveness of

$$at\_\ell_2 \;\rightarrow\; y = c$$

i.e., when control is at ℓ_2 the value of y is c. Similarly, in the following example, if y is owned by ℓ_2, and c_1 and c_2 are constant expressions, then from

$$\ell_1: \textbf{ if } c \textbf{ then } y := c_1 \textbf{ else } y := c_2 \quad \ell_2:$$

we can conclude that

$$at\_\ell_2 \;\rightarrow\; y = c_1 \vee y = c_2$$

is an inductive invariant.

Another example of a reaffirmed invariant is if a location ℓ in the program is reachable only as a result of a test κ. In such a case we know that when the location is first entered the test is valid. If all variables appearing in the test are owned by ℓ we can conclude

$$at\_\ell \;\rightarrow\; \kappa.$$

For example, if all variables in c are owned by ℓ_1, then from

$$\ell_0: \textbf{ await } c \quad \ell_1:$$

we may directly infer the invariant:

$$at\_\ell_1 \;\rightarrow\; c$$

Similarly from

ℓ_0: [**while** c **do** S] $\quad \ell_1$:

we can infer

$$at\_\ell_1 \rightarrow \neg c$$

if all variables in c are owned by ℓ_1. Similar invariants can be generated for **when** statements and conditional statements.

If the possible values of a data variable are known for every program location. one can reverse the implications. For example, if it is known that

$$\begin{array}{rcl} at\_\ell_0 & \rightarrow & y = c_1 \\ at\_\ell_{1,2} & \rightarrow & y = c_2 \vee y = c_3 \\ at\_\ell_3 & \rightarrow & y = c_3 \end{array}$$

where ℓ_0, ℓ_1, ℓ_2, and ℓ_3 cover the range of possible program locations, then, if c_1, c_2 and c_3 are distinct, one may infer:

$$\begin{array}{lcl} y = c_1 & \rightarrow & at\_\ell_0 \\ y = c_2 & \rightarrow & at\_\ell_{1,2} \\ y = c_3 & \rightarrow & at\_\ell_{1..3} \end{array}$$

Range Invariants

Even if it is not possible to determine the exact value of a data variable at a given location, it is sometimes possible to determine the range from which the data variable takes its values, if that variable is modified only in a restricted and predictable way. Range invariants are of the form:

$$at\_\ell \rightarrow l \leq y \leq u$$

For instance, for the program RES-SEM, shown in Figure 8, STeP generates the range invariant

$$y \geq 0.$$

Invariants of Parameterized Programs

Parameterized programs often contain array variables x such that no single statement or process owns x. However if $x[i]$ is modified only by $P[i]$, invariants like those described above can still be generated. Consider, for example, program ORDER, shown in Figure 9. It grants each process access to its critical section in the order of its process sequence number. For this program STeP generates the following local invariants:

$$\chi_1: \quad \forall i : [1..N]. \; (at\_\ell_5[i] \longleftrightarrow y[i])$$

$$
\textbf{local } M, y : \textbf{integer where } y = 1
$$

$$
\begin{array}{ll}
\ell_0: & \overset{M}{\underset{i=1}{||}} P[i] :: \left[\begin{array}{l}\ell_1: \textbf{ loop forever do} \\ \quad \left[\begin{array}{l}\ell_2: \textbf{ noncritical} \\ \ell_3: \textbf{ request } y \\ \ell_4: \textbf{ critical} \\ \ell_5: \textbf{ release } y\end{array}\right] \\ \ell_6:\end{array}\right] \\
\ell_7: &
\end{array}
$$

Figure 8: Program RES-SEM (resource allocation by semaphores).

$$
\chi_2: \quad \forall i : [1..N]. \; (at\_\ell_3[i] \longrightarrow a[i] \geq i)
$$

$$
\chi_3: \quad \forall i : [1..N]. \; (at\_\ell_2[i] \longrightarrow y\,[a[i]])
$$

$$
\begin{array}{lll}
\textbf{in} & N & : \textbf{integer where } N > 0 \\
\textbf{local} & a & : \textbf{array } [1..N] \textbf{ of integer where } \forall i : [1..N].\; a[i] = 1 \\
 & y & : \textbf{array } [1..N] \textbf{ of boolean where } \forall i : [1..N].\; \neg y[i]
\end{array}
$$

$$
\overset{N}{\underset{i=1}{||}} P[i] :: \left[\begin{array}{l}\ell_0: \textbf{ while } a[i] < i \textbf{ do} \\ \quad \left[\begin{array}{l}\ell_1: \textbf{ await } y[a[i]] \\ \ell_2: \; a[i] := a[i] + 1\end{array}\right] \\ \ell_3: \textbf{ critical} \\ \ell_4: \; y[i] := \mathrm{T} \\ \ell_5:\end{array}\right]
$$

Figure 9: Program ORDER

The local invariants χ_2 and χ_3 are examples of reaffirmed invariants, and χ_1 is the conjunction of a reaffirmed invariant and a reverse implication. Using these invariants, the proof of mutual exclusion for program ORDER, expressed by

$$
\forall i, j : i < j : [1..N]. \; \Box \neg \left(at\_\ell_3[i] \wedge at\_\ell_3[j]\right)
$$

is automatic.

4.2 Bottom Up: Linear Invariants

A *linear invariant* is a linear arithmetic relation involving program variables and program control states. A typical linear invariant, for instance, is given by:

$$at\_\ell_{0..2} + y_1 = 1$$

where $at\_\ell_{0..2}$ stands for $at\_\ell_0 \vee at\_\ell_1 \vee at\_\ell_2$. Note that boolean expressions are converted to integers by taking T to be 1 and F to be 0.

Linear invariants can also be generated for parameterized programs, where each control predicate can be generalized to represent the number of processes at that control point, e.g., $N(at\_\ell_{0..2})$ rather than $at\_\ell_{0..2}$.

Let $\mathcal{P}$ be a program, represented as a transition system with set of transitions $\mathcal{T}$ and initial condition Θ. A set of variables $y_1, \ldots, y_m$ is *linear* if the effect of each transition $\tau \in \mathcal{T}$ can be expressed as

$$y_i' \;=\; c_i^\tau \;+\; \sum_{k=1}^{m} c_{ik}^\tau \cdot y_k$$

where c_i^τ and c_{ik}^τ are *constant expressions*, i.e., expressions whose variables are not modified by any transition of $\mathcal{P}$. Thus, each variable y_i is modified only by a linear combination of other linear variables and constants.

Given a set of linear variables $y_1, \ldots, y_m$ and control locations $\ell_1, \ldots, \ell_n$, a *linear invariant* is an equation of the form:

$$\chi\colon\quad \sum_{i=1}^{m} a_i \cdot y_i \;+\; \sum_{j=1}^{n} b_j \cdot N(at\_\ell_j) \;=\; K$$

where a_i and b_j are constant expressions and K is a constant. The values of a_i and b_j are determined by solving the system of linear equations that results from the requirements for an inductive invariant, i.e.,

- χ is implied by the initial condition Θ, which translates into

  $$\sum_{i=1}^{m} a_i \cdot y_i^0 \;+\; \sum_{j=1}^{n} b_j \cdot N(at\_\ell_j{}^0) \;=\; K$$

  where y_i^0 denotes the initial values of y_i and $N(at\_\ell_j{}^0)$ denotes the initial number of processes at ℓ_j, and

- χ is preserved by each transition $\tau \in \mathcal{T}$, which, for each $\tau \in \mathcal{T}$ translates into

  $$\sum_{i=1}^{m} a_i \cdot \Delta(\tau, y_i) \;+\; \sum_{j=1}^{n} b_j \cdot \Delta(\tau, N(at\_\ell_j)) \;=\; 0$$

  where $\Delta(\tau, y_i)$ is the increment in y_i due to τ and $\Delta(\tau, N(at\_\ell_j))$ denotes the increase or decrease in the number of processes at ℓ_j due to τ.

STeP constructs invariants based on a maximal set of linearly independent solutions (if the resulting system of linear equations is not independent, there is no unique solution). As an example, consider program RES-SEM, which was presented in Figure 8. The only linear variable is y, so linear invariants for RES-SEM are of the form:[2]

$$\chi_R\colon \quad a \cdot y \;+\; \sum_{j=0}^{7} b_j \cdot N(at\_\ell_j) \;=\; K$$

Imposing the invariance requirements results in the following system of equations:

$$\begin{array}{lcr}
\Theta & : & a + b_0 = K \\
\tau_0 & : & -b_0 + M \cdot b_1 = 0 \\
\tau_1^T & : & -b_1 + b_2 = 0 \\
\tau_1^F & : & -b_1 + b_6 = 0 \\
\tau_2 & : & -b_2 + b_3 = 0 \\
\tau_3 & : & -a - b_3 + b_4 = 0 \\
\tau_4 & : & -b_4 + b_5 = 0 \\
\tau_5 & : & a - b_5 + b_1 = 0 \\
\tau_6 & : & -M \cdot b_6 + b_7 = 0
\end{array}$$

from which STeP constructs, among others, the following invariant:

$$y + N(at\_\ell_4) + N(at\_\ell_5) = 1.$$

In conjunction with the local invariant $y \geq 0$, this is sufficient for establishing mutual exclusion for program RES-SEM.

4.3 Top-down: Strengthening

Top-down invariants, i.e., strengthened invariants, are generated in STeP by *invariant propagation.* Suppose STeP is given a proposed invariant ψ to be proven. The system first generates bottom-up invariants and checks whether ψ is inductive relative to the conjunction of all bottom-up invariants. If this is not the case, i.e., ψ cannot be proven, the next step is to strengthen ψ based on the verification conditions that could not be proven.

Suppose that ψ is a proposed invariant. Given a transition τ for which the verification condition

$$\{\psi\}\ \tau\ \{\psi\}$$

cannot be proven, the system automatically computes the weakest precondition $wpc(\psi, \tau)$ of ψ with respect to τ, i.e., the weakest assertion γ that guarantees φ is true when τ is taken from a state that satisfies γ. The strengthened invariant is then taken to be:

[2] Strictly speaking, M is also a linear variable, but since it is recognized to be a constant expression and, as such, does not contribute anything useful to a linear invariant, it is excluded.

$$\psi \wedge wpc(\psi, \tau)$$

Consider, for example, the proof of mutual exclusion, expressed by the invariant

$$\psi: \quad \neg(at\_\ell_5 \wedge at\_m_5)$$

for program PET, presented in Section 2. ψ is not inductive, since

$$\{\psi\}\ \tau\ \{\psi\}$$

is not valid for $\tau = \ell_4$. STeP automatically computes the weakest precondition of ℓ_4, yielding

$$wpc(\ell_4, \psi): \quad at\_\ell_4 \ \wedge\ (\neg y_2 \ \vee\ s = 2) \ \rightarrow\ \neg(\underbrace{at\_\ell_5{}'}_{\mathrm{T}} \wedge \underbrace{at\_m_5{}'}_{at\_m_5})$$

which simplifies to

$$\psi_1: \quad at\_\ell_4 \ \wedge\ at\_m_5 \ \rightarrow\ y_2 \ \wedge\ s \neq 2$$

Similarly for m_4:

$$\psi_2: \quad at\_\ell_5 \ \wedge\ at\_m_4 \ \rightarrow\ y_1 \ \wedge\ s \neq 1$$

The conjunction of the proposed invariant and the weakest preconditions,

$$\varphi: \quad \psi \ \wedge\ \psi_1 \ \wedge\ \psi_2$$

is inductive and all verification conditions are established automatically.

To summarize invariant generation, consider program PET once more. In order to prove mutual exclusion

$$\varphi_{ME}: \quad \neg(at\_\ell_5 \ \wedge\ at\_m_5)$$

STeP automatically generates the following invariants:

$$\begin{array}{ll}
\text{range} & 1 \leq s \leq 2 \\[1em]
\text{local} & \left\{ \begin{array}{l} y_1 \ \leftrightarrow\ at_\ell_{3..6} \\ y_2 \ \leftrightarrow\ at_m_{3..6} \end{array} \right. \\[1em]
\text{strengthening} & \left\{ \begin{array}{l} at_\ell_4 \ \wedge\ at_m_5 \ \rightarrow\ y_2 \ \wedge\ s \neq 2 \\ at_\ell_5 \ \wedge\ at_m_4 \ \rightarrow\ y_1 \ \wedge\ s \neq 1 \end{array} \right.
\end{array}$$

and, using these invariants, automatically establishes all verification conditions.

5 Theorem-proving support

Effective verification requires effective theorem-proving, in order to free the user from the many tedious low-level details of a formal proof. In STeP, most of the verification conditions that need to be proved for typical systems are trivial. However, automating the process of proving them requires the integration of a large variety of tools, which we now briefly describe.

5.1 Simplification

Most of the automated theorem-proving in STeP is done by a very general, but efficient, rewriting mechanism, which we call the *simplifier*. It can be best described as a form of *contextual rewriting* (a generalization of conditional rewriting, see [Zha93]) that incorporates a number of specialized features that we have found useful for dealing with the formulas that commonly occur in verification conditions. Thus, the contextual rewriting includes:

- A form of non-clausal propositional simplification that can, for instance, simplify a sentence of the form

  $$a \wedge b \wedge (d \vee c) \rightarrow (a \wedge d) \vee (c \wedge f)$$

  to

  $$a \wedge b \wedge c \rightarrow d \vee f$$

- Opportunistic reasoning about the interaction of equalities and quantification. For example,

  $$(\forall x)[x = 1 \ \wedge \ p(x) \ \rightarrow \ x = 2 \ \vee \ q(x)]$$

  simplifies to:

  $$p(1) \ \rightarrow \ q(1)$$

  via special strategies for quantifiers.

- Rewrite rules (conditional and unconditional) for interpreted function symbols. These are useful for simplifying terms involving lists and arrays; for instance, rewriting

  $$contents(assign(\texttt{Array1}, y, z), y)$$

  to z.

Furthermore, the simplifier relies heavily on *congruence closure* [NO80] for reasoning about equality and uninterpreted function symbols. Congruence closure is also tightly integrated with a decision procedure for inequalities over totally ordered domains. The combined decision procedure works in polynomial time in most practical cases and is an attractive alternative to the more general, but more expensive Sup-Inf procedure described below. As a result, for example,

$$f(x) = y \ \wedge \ y < z \ \wedge \ z \leq x \ \rightarrow \ f(x) < x$$

simplifies to *true*.

Integrating all of the above features into a single rewriting procedure results in an extremely effective tool. For instance, it will promptly rewrite

$$(f(x) \leq x) \ \wedge \ (g(y) > y) \ \wedge \ \begin{pmatrix} f(x) > g(y) \\ \vee \\ g(x) < f(y) \end{pmatrix} \ \rightarrow \ (x \neq y)$$

to *true*.

5.2 Decision Procedures

By *decision procedure* we mean an algorithm that can decide the validity or satisfiability of a class of formulas in a given theory, and always terminates with a positive or negative answer. Decision procedures for a given theory may vary depending on their degree of completeness (i.e., which formulas they can decide) and their complexity, which are traded off against each other.

Two decision procedures for Presburger arithmetic are available[3]. The first is based on the *Sup-Inf method* [Ble75] which efficiently decides a subset of the theory; the other is an implementation of Cooper's algorithm [Coo72], which is a decision procedure for the entire theory.

The Sup-Inf method is complete for rational quantifier-free Presburger arithmetic, and can be extended to handle uninterpreted function symbols [Sho79]. Although it is incomplete if variables are required to be integer-valued and its complexity is exponential, the Sup-Inf method often works well in practice. With it one can decide, for example, that the formula

$$x \geq (y + z) \ \wedge (x \leq z) \ \wedge (y = 0) \ \rightarrow \ f(x) = f(z)$$

should simplify to *true*. Cooper's algorithm can decide the full Presburger theory over the integers (without function symbols), but is of super-exponential complexity. It can establish the validity of sentences such as

$$\forall x \ \forall y \ \exists z \ ((x + z) > y).$$

Despite the fact that Sup-Inf is incomplete for the integer fragment of Presburger arithmetic, we have found that STeP has been able to prove most of the verification conditions that arise in practice using only Sup-Inf and the simplifier.

[3] Presburger formulas are first-order formulas over integers, integer variables, addition and $<$.

For deciding the validity of propositional formulas with small clausal forms, an efficient implementation of the classic Davis-Putnam procedure ([ZS94]) can be used. A decision procedure to check the validity of propositional temporal logic formulas is also provided [KMMP93].

We should note that while the problem of effectively and efficiently integrating different decision procedures has commanded much attention over the years (e.g., [NO79, BM88b]), we have not yet implemented the more general methods. We consider this to be a promising direction for future research and implementation.

5.3 First-order Prover

As pointed out in Section 5.1, the contextual rewriting mechanism can perform simple reasoning about quantifiers and equality. However, more complex reasoning involving unification is often needed to prove the validity of certain first-order formulas that arise in verification. Such theorems are seldom "deep," and can often be proved by applying a few mechanical inference rules with very little heuristic guidance.

A theorem prover based on non-clausal resolution and paramodulation [MW93] is available as a semi-decision procedure for the full first-order predicate calculus with equality, automated in a style similar to the SNARK [SWL$^+$94] and OTTER [McC94] provers: the search is agenda-based, term-indexing is used for efficient demodulation and subsumption, and paramodulation is restricted by a recursive path ordering on terms. This prover also uses the basic simplification procedures described above. Previously proven invariants can be used as lemmas by this prover.

5.4 Interactive Prover

Because of their worst-case complexity, the more powerful decision procedures need to be applied in a controlled fashion. Consequently, they are not included in the main simplifier, which is automatically invoked quite often, and must therefore be fast. Instead they are left for the user to invoke interactively.

In addition to controlling the application of decision procedures, the interaction also provides tools for proving the validity of formulas in the undecidable settings of classical and temporal first-order logic.

This interaction is managed through a Gentzen-style first-order prover (see e.g., [Gal87]), which is guided by the user. Subgoals in a proof can be established via simplification, decision procedures, automatic propositional temporal proof-search, or resolution. The overall proof search is directed by the user, who decides which inference rules and decision procedures are applied to any given goal.

We also support a Gentzen-style first-order temporal prover, which can verify propositional temporal logic formulas automatically; traditional Gentzen-style proof rules are supported, as well as temporal rules such as:

$$(\vdash \Box) \quad \frac{\Gamma \vdash \Delta, \varphi \qquad \varphi, \Gamma \vdash \Delta, \bigcirc \Box \varphi}{\Gamma \vdash \Delta, \Box \varphi} \qquad (\Box \vdash) \quad \frac{\Gamma, \varphi, \bigcirc \Box \varphi \vdash \Delta}{\Gamma, \Box \varphi \vdash \Delta}$$

Proof search proceeds in a bottom-up manner: from the goal below the line, the search proceeds to the new subgoals above the line.

6 Examples

6.1 N-Process Dining Philosophers Program

Dijkstra's dining philosophers problem describes N philosophers whose only activities in life are eating and thinking. The philosophers eat only rice, and for this purpose need two chopsticks each. Unfortunately, their round dining table is only equipped with N chopsticks. This excludes adjacent philosophers from eating simultaneously.

A solution to the dining philosophers problem is given in Figure 10. In program DINE, chopsticks are acquired via the binary semaphore variables $c[1], \ldots, c[N]$, and deadlock (the possibility that every philosopher picks up his left chopstick at the same time) is prevented by the semaphore variable r, having initial value $N - 1$. One may interpret r as a door between the library and the dining hall, only allowing at most $N - 1$ philosophers into the dining hall.

$$\begin{array}{lll}
\textbf{in} & N & : \textbf{integer where } N \geq 2 \\
\textbf{local} & c & : \textbf{array } [1..N] \textbf{ of integer where } \forall i : [1..N].\ c[i] = 1 \\
 & r & : \textbf{integer where } r = N - 1
\end{array}$$

$$\mathop{||}_{i=1}^{N} P[i] :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1\text{:} & \textbf{noncritical} \\ \ell_2\text{:} & \textbf{request } r \\ \ell_3\text{:} & \textbf{request } c[i] \\ \ell_4\text{:} & \textbf{request } c[(i \textbf{ mod } N) + 1] \\ \ell_5\text{:} & \textbf{critical} \\ \ell_6\text{:} & \textbf{release } c[i] \\ \ell_7\text{:} & \textbf{release } c[(i \textbf{ mod } N) + 1] \\ \ell_8\text{:} & \textbf{release } r \end{array} \right] \end{array} \right]$$

Figure 10: Program DINE (Dining Philosophers)

Mutual exclusion, stated as

$$\Box \neg (at\_\ell_5[i] \wedge at\_\ell_5[(i \textbf{ mod } N) + 1]),$$

follows from the invariants:

$$\chi_1: \quad c[i] \geq 0$$

$$\chi_2: \quad at\_\ell_{5..7}[i] \;+\; at\_\ell_{4..6}[(i \,\mathbf{mod}\, N)+1] \;+\; c[(i \,\mathbf{mod}\, N)+1] \;=\; 1$$

The invariant χ_1 is generated as a bottom-up invariant, while χ_2 is generated by the strengthening heuristics. Twelve verification conditions need to be proven to establish the inductiveness of χ_2, all of which are proven automatically.

6.2 Szymanski's N-Process Mutual Exclusion Algorithm

The system has also been applied to prove mutual exclusion for Szymanski's mutual exclusion algorithm [Szy88], which is a symmetric parameterized program that provides mutual exclusion for an arbitrary number of processes. In [MP90] and [MP91c], several temporal proof techniques were applied to prove some properties of this program. The safety property, mutual exclusion, was also formally verified in [NT91] using the Boyer-Moore prover [BM88a]. We discuss here a more recent version [SV94] of Szymanski's algorithm. We actually verified a slightly modified program from the one in the prepublished version of [SV94]. Our version is written in SPL and corrected to avoid deadlock.

Szymanski's mutual exclusion algorithm is available in two versions. The shortest, and most abstract, is the *atomic* version, which allows quantification over parameterized variables in test statements; these tests are treated as atomic constructs. The more refined *molecular* version replaces tests that involve quantified formulas with more primitive program constructs. The two versions are presented in Figures 11 and 12, respectively.

The atomic version

The atomic version of Szymanski's mutual exclusion algorithm is shown in Figure 11, which identifies three parts: the *doorway*, the *waiting room* and the *inner sanctum*. The variables a, s and w may be given the following interpretation: $a[i]$, $s[i]$ and $w[i]$ indicate whether process i has requested access to the critical section, has entered through the doorway and is not in the waiting room, or is in the waiting room, respectively. The quantified tests in ℓ_3, ℓ_5, ℓ_7, ℓ_{10} and ℓ_{11}, which are considered atomic, can be seen as gates between the different stages. Processes can only pass ℓ_3 if there are no processes in the doorway or in the inner sanctum. However, as long as processes are waiting at ℓ_3, all processes that enter are redirected to the waiting room, opening ℓ_3 again. The last process that passes through ℓ_3 locks ℓ_3 behind it and then bypasses the waiting room, thereby opening the gate ℓ_7 such that the waiting processes can come out of the waiting room. At this point ℓ_3 remains locked until all processes inside the doorway have passed the critical section. Gate ℓ_{10} is opened when all processes have left the waiting room. Gate ℓ_{11} allows the processes that are inside the doorway access to the critical section, one by one, and in order of process number.

$$
\begin{array}{lll}
\textbf{in} & N & : \textbf{integer where } N \geq 1 \\
\textbf{local} & a & : \textbf{array}\,[1..N] \textbf{ of boolean where } \forall i : [1..N].\neg a[i] \\
 & s & : \textbf{array}\,[1..N] \textbf{ of boolean where } \forall i : [1..N].\neg s[i] \\
 & w & : \textbf{array}\,[1..N] \textbf{ of boolean where } \forall i : [1..N].\neg w[i]
\end{array}
$$

$$
\overset{N}{\underset{i=1}{||}} P[i] :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } a[i] := \mathrm{T} \\
\ell_3\text{: } \textbf{await } \forall j : [1..N].\ \neg s[j] \\
\qquad \text{———— } \textit{doorway} \text{ ————} \\
\ell_4\text{: } (w[i], s[i]) := (\mathrm{T}, \mathrm{T}) \\
\qquad \text{——— } \textit{waiting room} \text{ ———} \\
\ell_5\text{: } \textbf{if } \exists j : [1..N].\ (a[j] \wedge \neg w[j]) \textbf{ then} \\
\quad \left[\begin{array}{l}
\ell_6\text{: } s[i] := \mathrm{F} \\
\ell_7\text{: } \textbf{await } \exists j : [1..N].\ (s[j] \wedge \neg w[j]) \\
\ell_8\text{: } s[i] := \mathrm{T}
\end{array}\right] \\
\qquad \text{——— } \textit{inner sanctum} \text{ ———} \\
\ell_9\text{: } w[i] := \mathrm{F} \\
\ell_{10}\text{: } \textbf{await } \forall j : [1..N].\ \neg w[j] \\
\ell_{11}\text{: } \textbf{await } \forall j : [1..(i-1)].\ \neg s[j] \\
\ell_{12}\text{: } \textbf{critical} \\
\ell_{13}\text{: } (s[i], a[i]) := (\mathrm{F}, \mathrm{F})
\end{array}\right]
\end{array}\right]
$$

Figure 11: Program SZY-A (Szymanski's algorithm: atomic version).

This procedure is reflected in the following four invariants,

$$
\begin{array}{lrcl}
A_0 : & at\_\ell_{8..13}[i] & \rightarrow & \neg at\_\ell_4[k] \\
A_1 : & at\_\ell_8[i] & \rightarrow & \exists k : [1..N].\ at\_\ell_{10}[k] \\
A_2 : & at\_\ell_{11..13}[i] & \rightarrow & \neg at\_\ell_{4..9}[k] \\
A_3 : & at\_\ell_{12,13}[i] \ \wedge \ k < i & \rightarrow & \neg at\_\ell_{4..13}[k]
\end{array}
$$

which establish mutual exclusion. These invariants may be interpreted as follows:

- A_0: once a process i has entered the inner sanctum, the doorway is locked, i.e., no process k may be at_ℓ_4.

- A_1: if a process is about to leave the waiting room, there is already a process k in the beginning of the inner sanctum.
- A_2: once a process is in the latter part of the inner sanctum, there is no process k in the waiting room or in the doorway.
- A_3: if a process is in the critical section, there is no other process with a smaller index in the doorway, waiting room or inner sanctum.

The inductive invariant A_3 is established using the conjunction of A_0, A_1, and A_2, where A_3 implies mutual exclusion:

$$\Box(at\_\ell_{12}[i] \wedge at\_\ell_{12}[j] \rightarrow i = j)$$

Bottom-up invariants play a crucial role in establishing the auxiliary invariants. For example, the system generates the local invariants

$$at\_\ell_{5,6,9..13}[i] \leftrightarrow s[i]$$

$$at\_\ell_{3..13}[i] \leftrightarrow a[i]$$

$$at\_\ell_{5..9}[i] \leftrightarrow w[i]$$

which are used to establish A_0, A_1, A_2 and A_3. Of the 69 required verification conditions, 54 were established automatically. The remainder required short sessions using our interactive prover.

The molecular version

Statements such as

$$\textbf{await}\ \exists j : [1..N].\ (s[j] \wedge \neg w[j])$$

involve quantifiers over every process and are not usually available as atomic primitives. Therefore, we must refine the quantifiers to available programming language constructs. Typically, statements like the one above can be refined into loops, e.g.:

$$\begin{array}{l} j := 1 \\ \textbf{while}\ \neg s[j] \vee w[j]\ \textbf{do} \\ \quad j := (j\ \textbf{mod}\ N) + 1 \end{array}$$

and similarly for universal quantifiers. The refined program is shown in Figure 12.

Along with the refinement of the program, we must also refine the invariants we expect to hold. The invariants A_0, A_1, A_2 and A_3 from the atomic case are thus refined into:

```
in     N  : integer where N ≥ 1
local  a  : array [1..N] of boolean where ∀i : [1..N]. ¬a[i]
       s  : array [1..N] of boolean where ∀i : [1..N]. ¬s[i]
       w  : array [1..N] of boolean where ∀i : [1..N]. ¬w[i]

 N
 ||  P[i] ::
i=1
[ℓ0: loop forever do
   [local j : integer
    ℓ1: noncritical
    ℓ2: (a[i], j) := (T, 1)
    ℓ3: while j ≤ N do
       ℓ4: when ¬s[j] do
          ℓ5: j := j + 1
          ——— doorway ———
    ℓ6: (w[i], s[i], j) := (T, T, 1)
       ——— waiting room ———
    ℓ7: while j ≤ N do
       [ℓ8: if a[j] ∧ ¬w[j] then
           [ℓ9: s[i] := F
            ℓ10: while ¬s[j] ∨ w[j] do
                  ℓ11: j := (j mod N) + 1
            ℓ12: (j, s[i]) := (N + 1, T)]
        else ℓ13: j := j + 1]
       ——— inner sanctum ———
    ℓ14: (w[i], j) := (F, 1)
    ℓ15: while j ≤ N do
       ℓ16: when ¬w[j] do
          ℓ17: j := j + 1
    ℓ18: j := 1
    ℓ19: while j < i do
       ℓ20: when ¬s[j] do
          ℓ21: j := j + 1
    ℓ22: critical
    ℓ23: (s[i], a[i]) := (F, F)]]
```

Figure 12: Program SZY-M (Szymanski's algorithm: molecular version).

$$M_0: \quad \left(\begin{array}{ll} & at\_\ell_{14..23}[i] \\ \vee & at\_\ell_{7,8}[i] \wedge j[i] > k \\ \vee & at\_\ell_{13}[i] \wedge j[i] \geq k \end{array} \right) \rightarrow$$

$$\exists r : [1..N]. \left(\begin{array}{ll} & (r = i \vee at\_\ell_{14..23}[r]) \\ \wedge & (at\_\ell_{3,4}[k] \rightarrow j[k] \leq r) \\ \wedge & (at\_\ell_5[k] \rightarrow j[k] < r) \\ \wedge & \neg at\_\ell_6[k] \end{array} \right)$$

$$M_1: \quad at\_\ell_{12}[i] \rightarrow \exists k : [1..N]. \left(\begin{array}{ll} & at\_\ell_{15,16}[k] \wedge j[k] \leq i \\ \vee & at\_\ell_{17}[k] \wedge j[k] < i \end{array} \right)$$

$$M_2: \quad \left(\begin{array}{ll} & at\_\ell_{18..23}[i] \\ \vee & at\_\ell_{15,16}[i] \wedge j[i] > k \\ \vee & at\_\ell_{16}[i] \wedge j[i] \geq k \end{array} \right) \rightarrow \neg at\_\ell_{7..14}[k]$$

$$M_3: \quad k < i \wedge \left(\begin{array}{ll} & at\_\ell_{22,23}[i] \\ \vee & at\_\ell_{19,20}[i] \wedge j[i] > k \\ \vee & at\_\ell_{21}[i] \wedge j[i] \geq k \end{array} \right) \rightarrow \neg at\_\ell_{7..23}[k]$$

The local variable j is represented as an array indexed over the parameterized processes. The invariant M_3, like A_3, implies mutual exclusion at the critical section.

Verification of mutual exclusion for the molecular version required proving 129 verification conditions, 99 of which were established automatically by the simplifier. The rest were established using the interactive prover.

The refinement of the invariants of the atomic algorithm into the invariants of the molecular algorithm was nontrivial. The most difficult part was refining A_0 into M_0. The interactive prover proved to be useful as a design tool in this case. When an incorrect invariant was presented to the interactive prover, the invalid verification conditions often gave valuable insight into how to correct the erroneous program assertion.

6.3 Distributed N-way Arbiter Circuit

As a final example, we consider the high-level specification of a distributed N-way arbiter circuit ARB, originally proposed by Martin [Mar85] and studied in [Dil88].

The proposed parametrized circuit manages mutual exclusion between N users having access to a shared resource. The circuit is composed of N arbiter cells

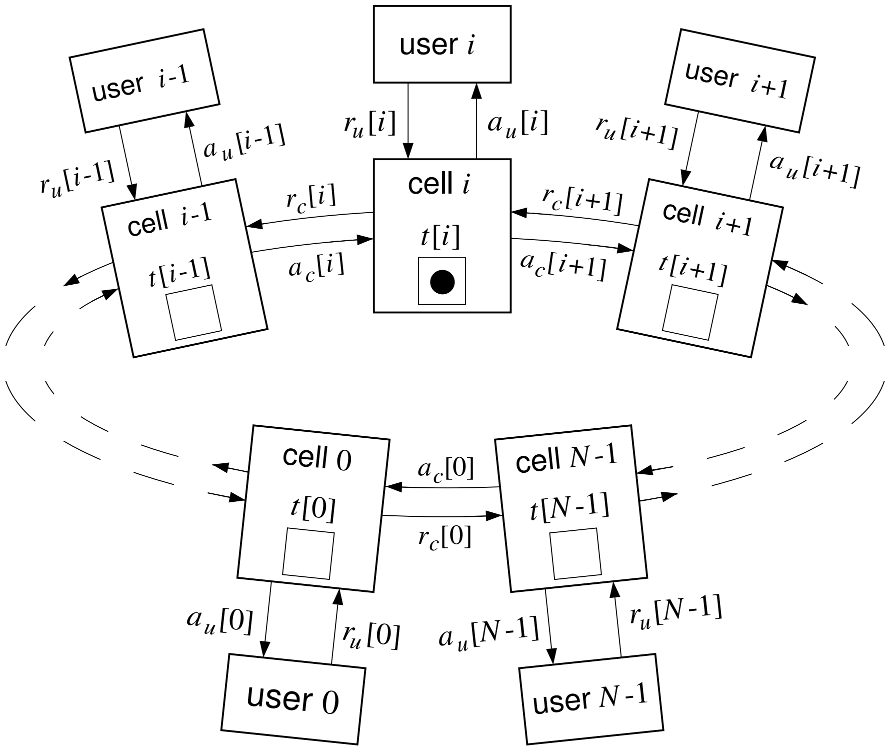

Figure 13: Distributed N-way arbiter circuit ARB.

connected in a circular pattern. Each user is connected to a cell of the arbiter, and there is a single token that circulates among the cells: whenever a cell has the token, the corresponding user can be granted access to the shared resource.

A cell can receive requests both from the user and from the cell to the right. If it has the token and receives a request from the user, the cell destroys the token and grants access to the user; the token reappears when the user releases the shared resource. If a cell has the token and receives a request from the cell to the right, it passes the token to the requesting cell. If both requests occur at the same time, the cell nondeterministically chooses which one to satisfy. If a cell receives a request but neither the cell nor its user has the token, the cell forwards the request to the cell to the left, and waits for the token.

The cells and the users communicate using a four-phase asynchronous handshake protocol based on request and acknowledge signals. The connections between the users and the cells are depicted in Figure 13. The signals r_c and a_c represent requests and acknowledges between cells, the signals r_u and a_u represent user requests and acknowledges, and t represents the token.

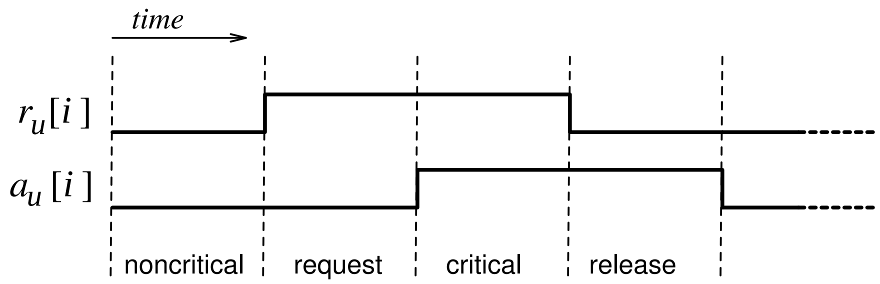

Figure 14: Four-phase handshake protocol between user i and cell i, $0 \leq i < N$.

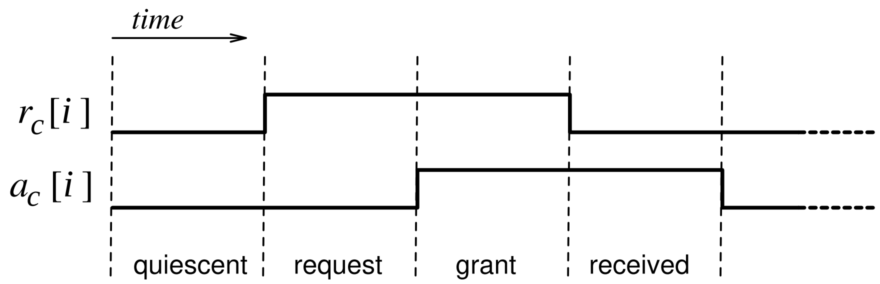

Figure 15: Four-phase handshake protocol between cell i and cell $(i-1)\,\mathbf{mod}\ N$, $0 \leq i < N$.

The protocol between user i and the corresponding cell i, $0 \leq i < N$, is shown in Figure 14. Initially, both $r_u[i]$ and $a_u[i]$ are F. When the user wishes to access the shared resource, it sets $r_u[i]$ to T. If the arbiter cell has the token, it responds to the request by setting $a_u[i]$ to T and destroying the token. When the user releases the shared resource, it sets $r_u[i]$ to F, and the arbiter cell acknowledges this by setting $a_u[i]$ to F and recreating the token.

The protocol between cell i and cell $(i-1)\,\mathbf{mod}\ N$, $0 \leq i < N$, is shown in Figure 15. Initially, both $r_c[i]$ and $a_c[i]$ are F. Cell i can request the token by setting $r_c[i]$ to T. If cell $(i-1)\,\mathbf{mod}\ N$ has the token, it can respond to the request by destroying the token and setting $a_c[i]$ to T. Cell i then acquires the token and acknowledges this by setting $r_c[i]$ to F. Finally, cell $(i-1)\,\mathbf{mod}\ N$ sets $a_c[i]$ to F.[4]

The high-level behavior of the circuit has been encoded in SPL as shown in Figure 16[5].

Mutual Exclusion

The mutual exclusion property for ARB can be stated as:

$$\Box\,\forall j,k:[0..N-1].\Big(a_u[j] \wedge a_u[k] \rightarrow j=k\Big).$$

This property is established with the help of the auxiliary invariant,

$$\begin{array}{c} \exists! j:[0..N-1].\Big(t[j] \vee a_u[j]\Big) \\ \wedge \\ \forall j:[0..N-1].\neg\Big(t[j] \wedge a_u[j]\Big) \end{array}$$

stating that at any given time there is exactly one cell that either has the token or is granting the user access to the resource. To prove this invariant, STeP automatically generates 12 verification conditions, which can be established with the usual combination of automatic and interactive theorem proving.

Absence of Unsolicited Requests

Another desirable property of the arbiter circuit is that a cell should not request the token, unless

1. it is receiving a request from the user or from the cell to the right, and

2. the cell does not have the token, nor it is granting access to the shared resource.

[4] In this model, the token simultaneously disappears from cell $(i-1)\,\mathbf{mod}\ N$ and reappears in cell i. This is consistent with the model presented in [Dil88].

[5] This program is slightly different from the model presented in [Dil88]: when an arbiter cell receives a request from its cell to the right it checks that its user is not accessing the resource before forwarding the request, while it does not in Dill's model.

$$
\begin{array}{lll}
\textbf{in} & N & : \textbf{integer where } N > 1 \\
\textbf{local} & r_c & : \textbf{array}\,[0..N-1] \textbf{ of boolean where } \forall i : [0..N-1].\neg r_c[i] \\
 & a_c & : \textbf{array}\,[0..N-1] \textbf{ of boolean where } \forall i : [0..N-1].\neg a_c[i] \\
 & r_u & : \textbf{array}\,[0..N-1] \textbf{ of boolean where } \forall i : [0..N-1].\neg r_u[i] \\
 & a_u & : \textbf{array}\,[0..N-1] \textbf{ of boolean where } \forall i : [0..N-1].\neg a_u[i] \\
 & t & : \textbf{array}\,[0..N-1] \textbf{ of boolean where } \forall i : [0..N-1].t[i] \leftrightarrow i = 0
\end{array}
$$

$$
\mathop{\Big\|}_{i=0}^{N-1}
\left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
\left[l_1 : \ \textbf{guard } \neg r_u[i] \wedge \neg a_u[i] \textbf{ do } r_u[i] := \mathrm{T}\right] \\
\textbf{or} \\
\left[l_2 : \ \textbf{guard } r_u[i] \wedge a_u[i] \textbf{ do } r_u[i] := \mathrm{F}\right] \\
\textbf{or} \\
\left[l_3 : \ \textbf{guard } r_u[i] \wedge \neg a_u[i] \wedge t[i] \textbf{ do } (t[i], a_u[i]) := (\mathrm{F}, \mathrm{T})\right] \\
\textbf{or} \\
\left[l_4 : \ \textbf{guard } \neg r_u[i] \wedge a_u[i] \textbf{ do } (t[i], a_u[i]) := (\mathrm{T}, \mathrm{F})\right] \\
\textbf{or} \\
\left[\begin{array}{ll}
l_5 : & \textbf{guard } \neg r_c[i] \wedge \neg a_c[i] \wedge \neg t[i] \wedge \neg a_c[i] \\
 & \quad \wedge \left(\begin{array}{c} r_u[i] \wedge \neg a_u[i] \\ \vee \\ r_c[(i+1)\textbf{mod } N] \ \wedge \ \neg a_c[(i+1)\textbf{mod } N] \end{array}\right) \\
 & \quad \textbf{do } r_c[i] := \mathrm{T}
\end{array}\right] \\
\textbf{or} \\
\left[\begin{array}{ll}
l_6 : & \textbf{guard } \neg r_c[(i+1)\textbf{mod } N] \ \wedge \ a_c[(i+1)\textbf{mod } N] \\
 & \quad \textbf{do } a_c[(i+1)\textbf{mod } N] := \mathrm{F}
\end{array}\right] \\
\textbf{or} \\
\left[\begin{array}{ll}
l_7 : & \textbf{guard } r_c[(i+1)\textbf{mod } N] \ \wedge \ \neg a_c[(i+1)\textbf{mod } N] \ \wedge \ t[i] \\
 & \quad \textbf{do } \left(\begin{array}{c} t[i] \\ t[(i+1)\textbf{mod } N] \\ a_c[(i+1)\textbf{mod } N] \end{array}\right) := \left(\begin{array}{c} \mathrm{F} \\ \mathrm{T} \\ \mathrm{T} \end{array}\right)
\end{array}\right] \\
\textbf{or} \\
\left[l_8 : \ \textbf{guard } r_c[i] \wedge a_c[i] \textbf{ do } r_c[i] := \mathrm{F}\right]
\end{array}\right]
\end{array}\right]
$$

Figure 16: High-level SPL encoding of ARB.

This property is not essential for mutual exclusion, but it contributes to the efficiency of the design. It is expressed by the temporal logic formula:

$$\Box\, \forall j : [0..N{-}1]\ .$$
$$\left[\begin{pmatrix} r_c[j] \\ \wedge \\ \neg a_c[j] \end{pmatrix} \rightarrow \begin{pmatrix} \neg t[j] \\ \wedge \\ \neg a_u[j] \end{pmatrix} \wedge \left[r_u[j] \vee \begin{pmatrix} r_c[(j+1)\,\mathbf{mod}\ N] \\ \wedge \\ \neg a_c[(j+1)\,\mathbf{mod}\ N] \end{pmatrix} \right] \right]$$

This invariant can also be proved by STeP.

7 Conclusions

Despite the fact that STeP is still at an early stage of development, it has already proved useful in understanding and debugging complex programs. For instance, the system helped identify an error in the mutual exclusion algorithm from a draft version of [SV94] that allowed the possibility of deadlock.

Although STeP is founded on the deductive methodology of Manna and Pnueli [MP94b], its development has been inspired by a large body of related work in formal verification, such as the PVS [SOR93] and SMV [BCMD90] systems, representing the deductive and model-checking approaches, respectively. Other recent approaches to combining model checking and deduction include [Hun93] and [KL93], where model checking is used to verify local properties of a system, which are then combined to prove global properties using deductive techniques.

The system presented in this paper reflects six months of implementation effort. Obviously there are many areas that need to be improved and completed. Major extensions that are being worked on include:

- Increased flexibility of verification diagrams;
- Inclusion of refinement verification rules [KMP94];
- Tighter integration of decision procedures, including more sophisticated constraint-solving techniques;
- Incorporation of decomposition, following the techniques described in [Cha93];
- Providing better debugging facilities;
- Connection of other systems to STeP (e.g., symbolic computation systems like Mathematica to support hybrid systems).
- Addition of the ability to handle real-time and hybrid systems.

Acknowledgements

We would like to thank Amir Pnueli and Arjun Kapur for their feedback and comments, and Xavier Leroy for his ML advice.

References

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, pages 428–439, June 1990.

[Ble75] W.W. Bledsoe. A new method for proving certain Presburger formulas. In *Proc. of the* 4^{th} *International Joint Conference on Artificial Intelligence*, pages 15–21, September 1975.

[BM88a] R.S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, MA, 1988.

[BM88b] R.S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[Cha93] E. Chang. *Compositional Verification of Reactive and Real-Time Systems.* PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1993.

[Coo72] D.C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99. American Elsevier, 1972.

[Dil88] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1988.

[Gal87] J.H. Gallier. *Logic for Computer Science—Foundations for Automatic Theorem Proving.* Wiley, New York, 1987.

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

[HSB93] R. Hojati, V. Singhal, and R.K. Brayton. Edge-Street/Edge-Rabin automata environment for formal verification using language containment. SRC report, University of California, Berkeley, 1993.

[Hun93] H. Hungar. Combining model checking and theorem proving to verify parallel processes. In *Proc.* 5^{th} *International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 154–165. Springer-Verlag, 1993.

[KL93] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proc. 5th Int. Conf. on Computer-Aided Verification*, number 697 in Lec. Notes in Comp. Sci, pages 166–179. Springer-Verlag, 1993.

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

[KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 273–346. Springer-Verlag, 1994.

[Man94] Z. Manna. Beyond model checking. In *Proc. 6^{th} International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 220–221. Springer-Verlag, 1994.

[Mar85] A.J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, editor, *Chapel Hill Conference on Very Large Scale Integration.* Computer Science Press, 1985.

[McC94] W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, Illinois, January 1994.

[MP90] Z. Manna and A. Pnueli. An exercise in the verification of multi-process programs. In W.H. J. Feijen, A.J.M van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business*, pages 289–301. Springer-Verlag, 1990.

[MP91a] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[MP91c] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. Rashid, editor, *Carnegie Mellon Computer Science: A 25-year Commemorative*, pages 121–156. ACM Press and Addison–Wesley, 1991.

[MP94a] Z. Manna and A. Pnueli. Temporal verification diagrams. In *Proc. of the 11^{th} Annual Symp. on Theoretical Aspects of Computer Science*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765. Springer-Verlag, 1994.

[MP94b] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1994. To appear.

[MW93] Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming.* Addison-Wesley, Reading, MA, 1993.

[NO79] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[NT91] M. Nagayama and C.L. Talcott. An NQTHM mechanization of "An exercise in the verification of multi-process programs". Technical Report STAN-CS-91-1370, Computer Science Department, Stanford University, Stanford, California, June 1991.

[Sho79] R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, April 1979.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (beta release). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, March 1993.

[Str82] R.S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54:121–141, 1982.

[SV94] B.K. Szymanski and J.M. Vidal. Automatic verification of a class of symmetric parallel programs. In *Proc. 13th IFIP World Computer Congress*, 1994. To appear.

[SWL+94] M.E. Stickel, R. Waldinger, M. Lowry, Th. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Proc. 12th Int. Conf. on Automated Deduction*, pages 341–355, 1994.

[Szy88] B.K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. 1988 International Conference on Supercomputing Systems*, pages 621–626, 1988.

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

[Zha93] H. Zhang. Contextual rewriting in automated reasoning. Technical Report Technical Report 93-07, Department of Computer Science, University of Iowa, August 1993.

[ZS94] H. Zhang and M.E. Stickel. Implementing the Davis-Putnam algorithm by tries. Draft manuscript, March 1994.