Liveness with Incomprehensible Ranking*

Yi Fang[1], Nir Piterman[2], Amir Pnueli[21], and Lenore Zuck[1]

[1] New York University, New York, {yifang,zuck}@cs.nyu.edu
[2] Weizmann Institute of Science, Rehovot, Israel
{nirp,amir}@wisdom.weizmann.ac.il

Abstract. The methods of Invisible Invariants and Invisible Ranking were developed originally in order to verify temporal properties of parameterized systems in a fully automatic manner. These methods are based on an instantiate-project-and-generalize heuristic for the automatic generation of auxiliary constructs and a *small model property* implying that it is sufficient to check validity of a deductive rule premises using these constructs on small instantiations of the system. The previous version of the method of Invisible Ranking was restricted to cases where the helpful assertions and ranking functions for a process depended only on the local state of this process and not on any neighboring process, which seriously restricted the applicability of the method, and often required the introduction of auxiliary variables.

In this paper we extend the method of Invisible Ranking to cases where the helpful assertions and ranking functions of a process may also refer to other processes. We first develop an enhanced version of the small model property, making it applicable to assertions that refer both to processes and their immediate neighbors. This enables us to apply the Invisible Ranking method to parameterized systems with ring topologies. For cases where the auxiliary assertions refer to all processes, we develop a novel proof rule which simplifies the selection of the next helpful transition, and enables the validation of the premises possible under the (old) small model theorem.

1 Introduction

Uniform verification of parameterized systems is one of the most challenging problems in verification today. Given a parameterized system $S(N) : P[1] \| \cdots \| P[N]$ and a property p, uniform verification attempts to verify $S(N) \models p$ for every $N > 1$. One of the most powerful approaches to verification which is not restricted to finite-state systems is *deductive verification*. This approach is based on a set of proof rules in which the user has to establish the validity of a list of premises in order to validate a given property of the system. The two tasks that the user has to perform are:

1. Identify some auxiliary constructs which appear in the premises of the rule.
2. Establish the logical validity of the premises, using the auxiliary constructs identified in step 1.

* This research was supported in part by the Minerva Center for Verification of Reactive Systems, the European Community IST project “Advance”, the Israel Science Foundation (grant no. 106/02-1), and NSF grant CCR-0205571.

When performing manual deductive verification, the first task is usually the more difficult, requiring ingenuity, expertise, and a good understanding of the behavior of the program and the techniques for formalizing these insights. The second task is often performed using theorem provers such as PVS [23] or STeP [4], which require user guidance and interaction, and place additional burden on the user. The difficulties in the execution of these two tasks is the main reason why deductive verification is not used more widely.

A representative case is the verification of invariance properties using the *invariance rule* of [18]. In order to prove that assertion r is an invariant of program P, the rule requires coming up with an auxiliary assertion φ which is *inductive* (i.e. is implied by the initial condition and is preserved under every computation step) and which strengthens (implies) r.

In [20, 2] we introduced the method of *invisible invariants*, which proposes a method for automatic generation of the auxiliary assertion φ for parameterized systems, as well as an efficient algorithm for checking the validity of the premises of the invariance rule. In [10] we extended the method of invisible invariants to *invisible ranking*, by applying the method for automatic generation of auxiliary assertions to general assertions (not necessarily invariant), and proposing a rule for proving liveness properties of the form $\square(p \rightarrow \diamondsuit q)$ (i.e, *progress properties*) that embeds the generated assertions in its premises, and efficiently checks for their validity.

The generation of invisible auxiliary constructs is based on the following idea: It is often the case that an auxiliary assertion φ for a parameterized system has the form $q(i)$, $\forall i.q(i)$ or, more generally, $\forall i \neq j.q(i,j)$. We construct an instance of the parameterized system taking a fixed value N_0 for the parameter N. For the finite-state instantiation $S(N_0)$, we compute, using BDD-techniques, some assertion ψ, which we wish to generalize to an assertion required form. Let r_1 be the projection of ψ on process index 1, obtained by discarding references to all variables which are local to all processes other than $P[1]$. We take $q(i)$ to be the generalization of r_1 obtained by replacing each reference to a local variable $P[1].x$ by a reference to $P[i].x$. The obtained $q(i)$ is our candidate for the body of the inductive assertion $\varphi : \forall i.q(i)$. We refer to this part of the process as *project&generalize*. For example, when computing invisible invariants, ψ is the set of reachable states of $S(N_0)$. The process can be easily generalized to generate assertions of the type $\forall i_1, \ldots, i_k.p(\vec{i})$.

Having obtained a candidate for the assertion φ, we still have to check the validity of the premises of the proof rule we wish to employ. Under the assumption that our assertional language is restricted to the predicates of equality and inequality between bounded range integer variables (which is adequate for many of the parameterized systems we considered), we proved a *small model* theorem, according to which, for a certain type of assertions, there exists a (small) bound N_0 such that such an assertion is valid for every N iff it is valid for all $N \leq N_0$. This enables using BDD techniques to check the validity of such an assertion. The assertions covered by the theorem are those that can be written in the form $\forall \vec{i} \exists \vec{j}.\psi(\vec{i}, \vec{j})$, where $\psi(\vec{i}, \vec{j})$ is a quantifier-free assertion which may refer only to the global variables and the local variables of $P[i]$ and $P[j]$.

Being able to validate the premises on $S[N_0]$ has the additional important advantage that the user never sees the automatically generated auxiliary assertion φ. This assertion is produced as part of the procedure and is immediately consumed in order to validate the premises of the rule. Being generated by symbolic BDD techniques, the representation of the auxiliary assertions is often extremely unreadable and non-intuitive, and will usually not contribute to a better understanding of the program or its proof. Because the user never gets to see it, we refer to this method as the "method of *invisible invariants*."

As shown in [20, 2], embedding a $\forall\vec{i}.q(\vec{i})$ candidate inductive invariant in the main proof rule used for safety properties results in premises that fall under the small model theorem. In [10], the proof rule used for proving progress properties requires that some auxiliary constructs have no quantifiers in order to result in $\forall\exists$-premises. In particular, it requires the "helpful assertions", describing when a transition is helpful (thus, leads to a lower ranked state), to be quantifier-free. This is the case for many simple protocols. In fact, many parameterized protocols that have been studied in the literature can be transformed into protocols that have unquantified helpful transitions by adding some auxiliary variables that allow, in each state, to determine the helpful assertions.

In this paper, we extend the method of invisible ranking and make it applicable to a much wider set of protocols in two directions:

- The first extension allows expression such as $i \pm 1$ to appear both in the transition relation as well as the auxiliary constructs. This extension is especially important for ring algorithms, where many of the assertion have a $p(i, i+1)$ or $p(i, i-1)$ component.
- The second extension, allows helpful assertions (and ranking functions) for, say process i, to be of the form $h(i) = \forall j.H(i, j)$, where $H(i, j)$ is a quantifier-free assertion. Such helpful assertions are common in "unstructured" systems where whether a transition of one process is helpful depends the states of all its neighbors. Substituted in the standard proof rules for progress properties, such helpful assertions lead to premises which do not conform to the required $\forall\exists$ form, and therefore cannot be validated using the small model theorem.

To handle the first extension, we establish a new small model theorem, to which we refer as the *modest model theorem* (introduced in Subsection 3.1). This theorem shows that, similarly to the small model theorem of [20] and [10], $\forall\exists$-premises, containing $i \pm 1$ sub-expressions, can be validated on relatively small models. The size of the models, however, is larger compared to the small model theorem of [20] and [10].

To handle the second extension, we introduce a novel proof rule. The main difficulty with helpful assertions of the form $h(i) = \forall j.H(i, j)$ is in premise D4 (of rule DISTRANK introduced in Subsection 2.5). This premise claims that every "pending" state has some helpful transitions enabled on it, and the major challenge is to select the particular transition which is helpful for each pending state. In the new rule PRE-RANK (introduced in Section 4) we implement a new mechanism for selecting the applicable helpful transition for each pending state. The selection mechanism is based on the installment of a *pre-order* relation among the helpful transitions in each state. The "helpful" transition is identified as any transition which is minimal according to this pre-order.

The paper is organized as follows: In Section 2, we present the general computational model of FDS, and the restrictions which enable the application of the invisible auxiliary constructs methods. We also review the small model property which enables automatic validation of the premises of the various proof rules. In addition, we outline a procedure that replaces compassion by justice requirements, describe the DISTRANK proof rule, and explain how we automatically generate ranking and helpful assertions for the parameterized case. In Section 3 we describe the modest model theorem which allows handling of $i \pm 1$ expressions within assertions, and demonstrate these techniques on the Dining Philosopher problem. In Section 4 we present the new DISTRANK proof rule that uses pre-order among transitions, discuss how to automatically obtain the pre-order, and demonstrate the techniques on the Bakery algorithm. All our examples have been run on TLV [22]. The interested reader may find the code, proof files, and output of all our examples in *cs.nyu.edu/acsys/Tlv/assertions*.

Related Work. The problem of uniform verification of parameterized systems is, in general, undecidable [1]. One approach to remedy this situation, pursued, e.g., in [8], is to look for restricted families of parameterized systems for which the problem becomes decidable. Unfortunately, the proposed restrictions are very severe and exclude many useful systems such as asynchronous systems where processes communicate by shared variables.

Another approach is to look for sound but incomplete methods. Representative works of this approach include methods based on: explicit induction ([9]), network invariants that can be viewed as implicit induction ([17]), abstraction and approximation of network invariants ([6]), and other methods based on abstraction ([11]). Other methods include those relying on "regular model-checking" (e.g., [13]) that overcome some of the complexity issues by employing *acceleration* procedures, methods based on symmetry reduction (e.g., [12]), or compositional methods (e.g., ([19]) that combine automatic abstraction with finite-instantiation due to symmetry. Some of these approaches (such as the "regular model checking" approach) are restricted to particular architectures and may, occasionally, fail to terminate. Others, require the user to provide auxiliary constructs and thus do not provide for fully automatic verification of parameterized systems.

Less related to our work is the work in [7] which presents methods for obtaining ranking functions for sequential programs.

In [10] we presented the invisible ranking methods and showed its application for cases where assertions contain only equality operators and helpful assertions are not quantified. The work in [21] studies the method of "counter-abstraction" to automatically prove liveness properties of parameterized systems. Counter-abstraction is an instance of data-abstraction [15] and has proven successful in instances of systems with a trivial (star or clique) topologies and a small state-space for each process. The work there is similar to counter abstraction is the work of the PAX group (see, e.g., [3]) which is based on the method of *predicate abstraction* [5]. While there are several differences between the two approaches, both are not "fully automatic" in the sense that the user has to provide the system with abstraction methodology.

In [24, 16] we used the method of *network invariants* [15] to prove liveness properties of parameterized systems. While extremely powerful, the main weakness of the method is that the user has to provide the candidate for the "network invariant" process.

2 Preliminaries

In this section we present our computation model, the small model theorem, and the proof rule we use for the verification of progress properties.

2.1 Fair Discrete Systems

As our computational model, we take a *fair discrete system* (FDS) $S = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

- V — A set of *system variables*. A *state* of the system S provides a type-consistent interpretation of the system variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of all states over V.
- Θ — The *initial condition*: An assertion (state formula) characterizing the initial states.
- $\rho(V, V')$ — The *transition relation*: An assertion, relating the values V of the variables in state $s \in \Sigma$ to the values V' in an S-successor state $s' \in \Sigma$.
- $\mathcal{J}$ — A set of *justice* (*weak fairness*) requirements: Each justice requirement is an assertion; A computation must include infinitely many states satisfying the requirement.
- $\mathcal{C}$ — A set of *compassion (strong fairness)* requirements: Each compassion requirement is a pair $\langle p, q \rangle$ of state assertions; A computation should include either only finitely many p-states, or infinitely many q-states.

A *computation* of an FDS S is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, ...$, the state $s_{\ell+1}$ is a S-successor of s_ℓ. That is, $\langle s_\ell, s_{\ell+1} \rangle \models \rho(V, V')$ where, for each $v \in V$, we interpret v as $s_\ell[v]$ and v' as $s_{\ell+1}[v]$.
- *Justice* — for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.
- *Compassion* – for every $\langle p, q \rangle \in \mathcal{C}$, either σ contains only finitely many occurrences of p-states, or σ contains infinitely many occurrences of q-states.

2.2 Bounded Fair Discrete Systems

To allow the application of the invisible constructs methods, we place further restrictions on the systems we study, leading to the model of *fair bounded discrete systems* (FBDS), that is essentially the model of bounded discrete systems of [2] augmented with fairness. For brevity, we describe here a simplified two-type model; the extension for the general multi-type case is straightforward.

Let $N \in \mathbb{N}^+$ be the *system's parameter*. We allow the following data types:

1. $\mathbf{type}_0$: the set of boolean and finite-range scalars (also denoted **bool**);
2. $\mathbf{type}_1$: a scalar data type that includes integers in the range $[1..N]$;

3. $\mathbf{type}_2$: a scalar data type that includes integers in the range $[0..N]$; and
4. arrays of the type $\mathbf{type}_1 \mapsto \mathbf{type}_i$ for $i = 0, 2$.

Atomic formulas may compare two variables of the same type. E.g., if y and y' are $\mathbf{type}_1$ variables, and z is a $\mathbf{type}_1 \mapsto \mathbf{type}_2$, then $y = y'$ and $z[y] < z[y']$ are both atomic formulas. For $z : \mathbf{type}_1 \mapsto \mathbf{type}_2$ and $y : \mathbf{type}_1$, we also allow the special atomic formula $z[y] > 0$. We refer to quantifier-free formulas obtained by boolean combinations of such atomic formulas as *restricted assertions*.

As the initial condition Θ, we allow assertions of the form $\forall i.u(i)$, where $u(i)$ is a restricted assertion.

As the transition relation ρ, as well as the justice requirements $\mathcal{J}$, we allow assertions of the form $\exists \vec{i} \forall \vec{j}. \psi(\vec{i}, \vec{j})$ for a restricted assertion $\psi(\vec{i}, \vec{j})$. For simplicity, we assume that all quantified and free variables are of type $\mathbf{type}_1$.

Example 1 (The Bakery Algorithm).
Consider program BAKERY in Fig. 1, which is a variant of Lamport's original Bakery Algorithm that offers a solution of the mutual exclusion problem for any N processes.

$$
\begin{array}{l}
\quad\quad\quad\quad \textbf{in} \quad N : \textbf{natural where } N > 1 \\
\quad\quad\quad\quad \textbf{local} \ \ y \ : \textbf{array } [1..N] \textbf{ of } [0..N] \textbf{ where } y = 0 \\
\overset{N}{\underset{i=1}{\Big|\Big|}} P[i] :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\quad \left[\begin{array}{l}
0 : \textbf{NonCritical} \\
1 : y := \text{maximal value to } y[i] \text{ while preserving order of elements} \\
2 : \textbf{await } \forall j \neq i : (y[j] = 0 \ \vee \ y[j] > y[i]) \\
3 : \textbf{Critical} \\
4 : y[i] := 0
\end{array} \right]
\end{array} \right]
\end{array}
$$

Fig. 1. Program BAKERY

In this version of the algorithm, location ℓ_0 constitutes the non-critical section which may non-deterministically exit to the trying section at location ℓ_1. Location ℓ_1 is the ticket assignment location. Location ℓ_2 is the waiting phase, where a process waits until it holds the minimal ticket. Location ℓ_3 is the critical section, and location ℓ_4 is the exit section. Note that y, the ticket array, is of type $\mathbf{type}_1 \mapsto \mathbf{type}_2$, and the program location array (which we denote by π) is of type $\mathbf{type}_1 \mapsto \mathbf{type}_0$. Note also that the ticket assignment statement at ℓ_1 is non-deterministic and may modify the values of all tickets. Fig. 2 describes the FBDS corresponding to program BAKERY.

Let α be an assertion over V, and R be an assertion over $V \cup V'$, which can be viewed as a transition relation. We denote by $\alpha \circ R$ the assertion characterizing all state which are R-successors of α-states. We denote by $\alpha \circ R^*$ the states reachable by an R-path of length zero or more from an α-state.

$$
\begin{array}{l}
V : \begin{cases} y : \textbf{array}[1..N] \textbf{ of } [0..N] \\ \pi : \textbf{array}[1..N] \textbf{ of } [0..4] \end{cases} \\
\Theta : \forall i : \pi[i] = 0 \ \wedge \ y[i] = 0 \\
\rho : \ \exists i : \forall j, k \neq i : (\pi'[j] = \pi[j]) \wedge \\
\qquad \begin{bmatrix} \quad \pi[i] = 0 \ \wedge \ \pi'[i] \in \{0,1\} \ \wedge \ y'[i] = y[i] \ \wedge \ y'[j] = y[j] \\ \vee \ \pi[i] = 1 \ \wedge \ \pi'[i] = 2 \ \wedge \ y'[j] < y'[i] \wedge \\ \qquad\qquad\qquad (y[j] = 0 \leftrightarrow y'[j] = 0) \ \wedge \ (y[j] < y[k] \leftrightarrow y'[j] < y'[k]) \\ \vee \ \pi[i] = 2 \ \wedge \ (y[j] = 0 \vee y[j] > y[i]) \ \wedge \ \pi'[i] = 3 \ \wedge \ y'[i] = y[i] \ \wedge \ y'[j] = y[j] \\ \vee \ \pi[i] = 3 \ \wedge \ \pi'[i] = 4 \ \wedge \ y'[i] = y[i] \ \wedge \ y'[j] = y[j] \\ \vee \ \pi[i] = 4 \ \wedge \ \pi'[i] = 0 \ \wedge \ y'[i] = 0 \ \wedge \ y'[j] = y[j] \end{bmatrix} \\
\mathcal{J} : \begin{pmatrix} \{J_1[i] : \pi[i] \neq 1 \mid i \in [1..N]\} \hfill \cup \\ \{J_2[i] : \neg(\pi[i] = 2 \ \wedge \ \forall j \neq i \, (y[j] = 0 \vee y[j] > y[i]) \mid i \in [1..N]\} \cup \\ \{J_3[i] : \pi[i] \neq 3 \mid i \in [1..N]\} \hfill \cup \\ \{J_4[i] : \pi[i] \neq 4 \mid i \in [1..N]\} \hfill \end{pmatrix} \\
\mathcal{C} : \ \emptyset
\end{array}
$$

Fig. 2. FBDS for Program BAKERY

2.3 The Small Model Theorem

Let $\varphi : \forall \vec{i} \exists \vec{j}. R(\vec{i}, \vec{j})$ be an AE-formula, where $R(\vec{i}, \vec{j})$ is a restricted assertion which refers to the state variables of a parameterized FBDS $S(N)$ in addition to the quantified (**type**$_1$) variables $\vec{i}$ and $\vec{j}$. Let N_0 be the number of universally quantified and free **type**$_1$ variables appearing in R. The following claim (stated first in [20] and extended in [2]) provides the basis for the automatic validation of the premises in the proof rules:

Theorem 1 (Small model property).
Formula φ is valid iff it is valid over all instances $S(N)$ for $N \leq N_0 + 2$.

The small model theorem allows us to check validity of AE-assertions on small model. In [20, 2] we obtain, using *project&generalize*, candidate inductive assertions for the set of reachable states that are A-formulae, checking their inductiveness required checking validity of AE-formulae, which can be accomplished, using BDD techniques. In [10] we obtain, using *project&generalize*, candidate assertions for various assertions (pending, helpful, ranking), all A- or E-formulae and, using these assertions, the premises of the progress proof rule are again AE-formulae, which can be checked using the theorem.

2.4 Removing Compassion

The proof rule we are employing to prove progress properties assumes a compassionless system. As was outlines in [14], every FDS S can be converted into a compassionless FDS $S_{\mathcal{J}} = \langle V_{\mathcal{J}}, \Theta_{\mathcal{J}}, \rho_{\mathcal{J}}, \mathcal{J}_{\mathcal{J}}, \emptyset \rangle$, where

$$
\begin{array}{ll}
V_{\mathcal{J}} : V \ \cup \ \{nvr_p : \ \textbf{boolean} \ \mid \ \langle p, q \rangle \in \mathcal{C}\} & \Theta_{\mathcal{J}} : \Theta \ \wedge \ \bigwedge_{\langle p,q \rangle \in \mathcal{C}} \neg nvr_p \\
\rho_{\mathcal{J}} : \rho \ \wedge \ \left(\bigwedge_{\langle p,q \rangle \in \mathcal{C}} nvr_p \rightarrow nvr'_p \right) & \mathcal{J}_{\mathcal{J}} : \mathcal{J} \ \cup \ \{nvr_p \ \vee \ q \ \mid \ \langle p, q \rangle \in \mathcal{C}\}
\end{array}
$$

This transformation adds to the system variables a new boolean variable nvr_p for each compassion requirement $\langle p, q\rangle \in \mathcal{C}$. The intended role of these variables is to identify, nondeterministically, a point in the computation, beyond which p will never be true again. The initial value of all these variables is 0 (*false*). The transition relation allows nondeterministically to change the value of any nvr_p variable from 0 to 1 but not vice versa. Finally, to the justice requirements we add a new justice requirement $nvr_p \;\vee\; q$ requiring that there are infinitely many states in which either nvr_p or q is true. Let Err denote the assertion $\bigvee_{\langle p,q\rangle\in\mathcal{C}} p \;\wedge\; nvr_p$, describing states where both p and nvr_p hold, which indicates that the prediction that p will never occur has been premature. For σ_J, a computations of S_J, we denote by $\sigma_J \Downarrow_V$ the sequence obtained from σ_J by projecting each state on the variables of S. The relation between S and its compassion-free version S_J can be stated as follows:

> Sequence σ is a computation of S iff there exists σ_J an err-free computation of S_J such that $\sigma_J \Downarrow_V = \sigma$.

It follows that

$$S \models q \Rightarrow \Diamond r \qquad \text{iff} \qquad S_J \models (q \wedge \neg Err) \Rightarrow \Diamond (r \vee Err)$$

Which allows us to assume that FBDSs we consider here have an empty compassion set.

2.5 The DistRank Proof Rule

In [10] we presented a proof rule for progress properties that exploits the structure of parameterized systems, by associating helpful assertions and ranking functions with each transition. The proof rule is presented in Fig. 3.

For a parameterized system with a transition domain $\mathcal{T} = \mathcal{T}(N)$
set of states $\Sigma(N)$,
justice requirements $\{J_\tau \mid \tau \in \mathcal{T}\}$,
invariant assertion φ,
assertions $q, r, pend$ and $\{h_\tau \mid \tau \in \mathcal{T}\}$,
and ranking functions $\{\delta_\tau : \Sigma \to \{0,1\} \mid \tau \in \mathcal{T}\}$

$$\begin{array}{lll}
\text{D1. } q \;\wedge\; \varphi & \rightarrow & r \;\vee\; pend \\
\text{D2. } pend \;\wedge\; \rho & \rightarrow & r' \;\vee\; pend' \\
\text{D3. } pend \;\wedge\; \rho & \rightarrow & r' \;\vee\; \bigwedge_{\tau\in\mathcal{T}} \delta_\tau \geq \delta'_\tau \\
\text{D4. } pend & \rightarrow & \bigvee_{\tau\in\mathcal{T}} h_\tau \\
\text{For every } \tau \in \mathcal{T} & & \\
\text{D5. } h_\tau \;\wedge\; \rho & \rightarrow & r' \;\vee\; h'_\tau \;\vee\; \delta_\tau > \delta'_\tau \\
\text{D6. } h_\tau & \rightarrow & \neg J_\tau \\
\hline
 & q \Rightarrow \Diamond r &
\end{array}$$

Fig. 3. The liveness rule DistRank

The rule is configured to deal directly with parameterized systems. Typically, the parameter domain provides a unique identification for each transition, and will have the form $\mathcal{T}(N) = [0..k] \times N$ for some fixed k. For example, in program BAKERY, $\mathcal{T}(N) =$

$[0..4] \times N$, where each justice transition can be identified as $J_m[i]$ for $m \in [0..4]$ (corresponding to the various locations in each process), and $i \in [1..N]$. In the rule, assertion φ is an invariant assertion characterizing all the reachable states. Assertion $pend$ characterizes the states which can be reached from a reachable q-state by a r-free path. For each transition τ, assertion h_τ characterizes the states at which this transition is *helpful*. That is, these are the states whose every J_τ-satisfying successor leads to a progress towards the goal, which is expressed by immediately reaching the goal or a decrease in the ranking function d_τ, as stated in premise D5. The ranking functions δ_τ are used in order to measure progress towards the goal. See [10] for justification of the rule.

Thus, in order to prove a progress property we need to identify the assertions φ, $pend$, and δ_τ, h_τ for every $\tau \in \mathcal{T}$. For a parameterized system, the progress properties we are considering are of the form $\forall z.q(z) \Rightarrow \Diamond\, r(z)$. We instantiate the system to a small number of processes, fix some process z, use *project&generalize* to obtain candidates for $pend$ and δ_τ, h_τ, and use the small model theorem to check the premises D1–D6, as well as the inductiveness of φ. However, in order for this to succeed, the generated assertions should adhere to some strict syntactic form. Most notably, h_τ can either be a restricted or an E-assertion in order to prove the validity of D4, since when h_τ has an A-fragment, D4 is no longer an AE-assertion.

Unfortunately, the success of this approach depends on the helpful assertions referring only the the process they "belong" to, without mention of any other process. In many cases, this cannot be the case – helpful transitions need to refer to neighboring processes. We study two such main cases: One in which processes are arranged in a ring. and a process can access some variables of its immediate neighbors, and the other where a process can access variables of all other processes.

3 Protocols with $p(i, i+1)$ Assertions

In many algorithms, particularly those based on ring architectures, the auxiliary assertions depend only on a process and its immediate neighbors. Consider such an algorithm for a ring of size N. For every $j = 1, .., N$, define $j \oplus 1 = (j \bmod N) + 1$ and $j \ominus 1 = ((j-2) \bmod N) + 1$. We are interested in assertions of the type $p(i, i \oplus 1)$ and $p(i, i \ominus 1)$. Having the ± 1 operator, these assertions do not fall into our small model theorem that restricts the operators to comparisons (and, expressing ± 1 using comparisons requires additional quantification.) However, as we show here, there is a small model theorem that allows proving validity of $\forall\exists p(i, i \pm 1)$ assertions. The size of the model, however, is larger than the previous one, which is why we refer to it as "modest".

3.1 Modest Model Theorem and Incomprehensible Assertions

Theorem 2 (Modest Model Theorem). *Consider a parameterized* FBDS S *with no* **type**$_2$ *variables*[3]. *Let* φ: $\forall \vec{i} \exists \vec{j}.R(\vec{i}, \vec{j})$ *be such that* $\vec{i}$ *and* $\vec{j}$ *are of type*$_1$, *and* $R(\vec{i}, \vec{j})$ *is a restricted assertion augmented by operators* $\oplus 1$ *and* $\ominus 1$. *Let* K *be the number of*

[3] This assumption is here for simplicity's sake and can be removed at the cost of increasing the bound.

*universally quantified and free variables in φ. Assume there are L **type**$_1 \mapsto$ **bool** arrays in S. Define $N_0 = K(2^L + 1)$. Then:*

$$\varphi \text{ is valid over } S(N) \text{ for every } N \geq 2 \quad \text{iff} \quad \varphi \text{ is valid over } S(N) \text{ for every } N \leq N_0$$

Proof Outline: Let $\psi = \neg\varphi$, i.e,. $\psi = \exists\vec{i}\forall\vec{j}.\neg R(\vec{i},\vec{j})$. It suffices to show that if ψ is satisfiable, then it is satisfiable in an instantiation $S(N)$ for some $N \leq N_0$.

Assume that ψ is satisfiable in some state s of $S(N_1)$ and that $N_1 > N_0$. Let $u_1 < u_2 < \ldots < u_k$ be the sequence of values of $type_1$-variables which appear existentially quantified or free in ψ. Since there are at most K such values, $k \leq K$. Since $N_1 > N_0$, either $u_1 > 2^L$ or there exist some u_i and u_{i+1} such that $u_{i+1} - u_i > 2^L$. We restrict here to the latter case, and construct a state s', in an instantiation $N_1' < N_1$, such that $s' \models \psi$. This process is repeated, until we obtain an instantiation where the u_j's are at most 2^L apart from one another. (Both u_1 and u_k may require separate, but similar, handling.)

Since $u_{i+1} - u_i > 2^L$, there exist two indices, m and n, such that $u_i \leq m < n < u_{i+1}$ and $a[n] = a[m]$ for every **type**$_1 \rightarrow$ **type**$_0$ array a. Intuitively, removing the processes whose indices are $m+1, \ldots, n$ does not impact any of the other processes u_j's, since the **type**$_1 \mapsto$ **type**$_0$ values of their immediate neighbors remain the same. After the removal, the remaining processes are renumbered, to reflect the removal.

Thus, we construct from s a new state s', leaving the **type**$_1$ variables in the range $1..m$ intact, and reducing the **type**$_1$ indices larger than n by $n - m$, maintaining the assignments of their **type**$_1 \mapsto$ **type**$_0$ variables. Obviously, s' is a state of $S(N_1-(n-m))$ that satisfies ψ. $\square$

The generation of all assertions is completely invisible; so is the checking of the premises on the instantiated model. However, the instantiation of the modest model requires feeding the assertions into the larger model. This can be done completely automatically, or with some user intervention. Whichever it is, while the user may see the assertions, there is no need for the user to comprehend them. In fact, being generated using ADD techniques, they are often incomprehensible.

3.2 Example: Dining Philosophers

We demonstrate the use of the modest model theory on validating DISTRANK on a classical solution to the dining philosophers problem.

Consider program DINE that offers a solution to the dining philosophers problem for any N philosophers. The program uses semaphores for forks. In this program, $N-1$ philosophers, $P[1], \ldots, P[N-1]$, reach first for their left forks and then for their right forks, while $P[N]$ reaches first for its right fork and only then for its left fork.

The semaphore instructions "**request** x" and "**release** x" appearing in the program stand, respectively, for "$\langle$**when** $x = 1$ **do** $x := 0\rangle$" and "$x := 1$". Consequently, we have a compassion requirement for each "**request** x", indicating that if a process is requesting a semaphore that is available infinitely often, it obtains it infinitely many times.

As outlined in Section 2.4, we transform the FBDS into a compassion-free FBDS by adding two new boolean arrays, *nvr*$_1$ and *nvr*$_2$, each *nvr*$_\ell[i]$ corresponding to the request

$$
\begin{array}{l}
\textbf{in} \quad N : \textbf{natural where } N > 1 \\
\textbf{local } y \ : \textbf{array } [1..N] \textbf{ of bool where } y = 1
\end{array}
$$

$$
\prod_{i=1}^{N-1} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_0 : \textbf{NonCritical} \\ \ell_1 : \textbf{request } y[i] \\ \ell_2 : \textbf{request } y[i \oplus 1] \\ \ell_3 : \textbf{Critical} \\ \ell_4 : \textbf{release } y[i], y[i \oplus 1] \end{array} \right] \end{array} \right] \quad \| \quad P[N] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_0 : \textbf{NonCritical} \\ \ell_1 : \textbf{request } y[1] \\ \ell_2 : \textbf{request } y[N] \\ \ell_3 : \textbf{Critical} \\ \ell_4 : \textbf{release } y[N], y[1] \end{array} \right] \end{array} \right]
$$

Fig. 4. Program DINE: Solution to the Dining Philosophers Problem

of process i at location ℓ. Fig. 5 describes the variables, initial conditions, and justice requirements of the FBDS we associate with Program DINE.

$$
\begin{array}{l}
V : \left\{ \begin{array}{ll} y, nvr_1, nvr_2 : & \textbf{array } [1..N] \textbf{ of bool} \\ \pi : & \textbf{array } [1..N] \textbf{ of } [0..4] \end{array} \right\} \\
\Theta : \forall i. \, (\pi[i] = 0 \ \wedge \ y[i] \ \wedge \ \neg nvr_1[i] \ \wedge \ \neg nvr_2[i]) \\
\mathcal{J} : \left(\begin{array}{ll} \{J_1[i] : nvr_1[i] \ \vee \ \pi[i] \neq 1 \mid i \in [1..N]\} & \cup \\ \{J_2[i] : nvr_2[i] \ \vee \ \pi[i] \neq 2 \mid i \in [1..N]\} & \cup \\ \{J_3[i] : \pi[i] \neq 3 \mid i \in [1..N]\} & \cup \\ \{J_4[i] : \pi[i] \neq 4 \mid i \in [1..N]\} & \end{array} \right)
\end{array}
$$

Fig. 5. FBDS for Program DINE

The progress property of the original system is $(\pi[z] = 1) \Rightarrow \Diamond(\pi[z] = 3)$, which be proved in two steps, the first establishing that $(\pi[z] = 1) \Rightarrow \Diamond(\pi[z] = 2)$ and the second establishing that $(\pi[z] = 2) \Rightarrow \Diamond(\pi[z] = 3)$. For simplicity of presentation, we restrict discussion to the latter progress property.

Since $P[N]$ differs from $P[1], \ldots, P[N-1]$, and since it accesses $y[1]$, which is also accessed by $P[1]$, and $y[N]$, which is also accessed by $P[N-1]$, we choose some z in the range $2, \ldots, N-2$ and prove progress of $P[z]$. The progress property of the other three processes can be established separately (and similarly.) Taking into account the translation into a compassion-less system, the property we attempt to prove is

$$
(\pi[z] = 2) \ \Rightarrow \ \Diamond(\pi[z] = 3 \ \vee \ \mathit{Err}) \qquad (2 \leq z \leq N-2)
$$

where

$$
\begin{array}{rcl}
\mathit{Err} & = & \bigvee_{i=1}^{N-1} \ (\pi[i] = 1 \ \wedge \ y[i] \ \wedge \ nvr_1[i]) \ \ \vee (\pi[i] = 2 \ \wedge \ y[i+1] \ \wedge \ nvr_2[i]) \\
& & \vee \quad (\pi[N] = 1 \ \wedge \ y[1] \ \wedge \ nvr_1[N]) \vee (\pi[N] = 2 \ \wedge \ y[N] \ \wedge \ nvr_2[N])
\end{array}
$$

3.3 Automatic Generation of Symbolic Assertions

Following the guidelines in [10], we instantiate DINE according to the small model theorem, compute the auxiliary *concrete* constructs for the instantiation, and abstract them. Here, we chose an instantiation of $N_0 = 6$ (obviously, we need $N_0 \geq 4$; it seems safer to allow at least a chain of three that does not depend on the "special" three, hence we obtained 6.) For the progress property, we chose $z = 3$, and attempt to prove $(\pi[3] = 2) \Rightarrow \Diamond(\pi[3] = 3)$. Due to the structure of Program DINE, process $P[i]$ depends only on it neighbors, thus, we expect the auxiliary constructs to include only assertions that refer to two neighboring process at the time. We chose to focus on pairs of the form $(i, i \ominus 1)$.

We first compute $\varphi^a(i, i \ominus 1)$, which is the abstraction of the set of reachable states. We distinguish between three cases, $i = 1$, $i = N$, and $i = 2, \ldots, N-1$. For the first, we project the concrete φ on 1 and 6 (and generalize to 1 and N), for the second, we project the concrete φ on 6 and 5 (and generalize to N and $N-1$), and for the third we project the concrete φ on 3 and 2 (and generalize to i and $i-1$). Thus, for the general $i \notin \{1, N\}$ case we obtain:

$$\varphi^a(i, i-1) \;=\; \begin{pmatrix} & (y[i-1] \rightarrow \pi[i-1] < 2) \;\wedge\; (\pi[i-1] > 2 \rightarrow \pi[i] < 2) \\ \wedge & (y[i] \leftrightarrow \pi[i-1] < 3) \;\wedge\; (\pi[i] < 2) & \end{pmatrix}$$

We then take :

$$\varphi^a \;=\; \varphi^a(1, N) \;\wedge\; \varphi^a(N, N-1) \;\wedge\; \forall i \neq 1, N. \varphi^a(i, i-1)$$

and define $pend^a \;=\; reach^a \;\wedge\; \neg Err \;\wedge\; \pi[3] = 2$.

For the helpful sets, and the δ's, we obtain, as expected, assertions of the type $p(i, i \ominus 1)$. E.g., for every $j = z + 1, \ldots, N-1$, we get

$$\begin{array}{l} h_2^a[j] : \pi[j-1] = 2 \;\wedge\; nvr_2[j-1] \;\wedge\; \pi[j] = 2 \;\wedge\; \neg nvr_2[j] \\ \delta_2[j] : \neg nvr_2[j] \;\wedge\; (\pi[j-1] = 2 \;\wedge\; nvr_2[j-1] \rightarrow \pi[j] < 3) \end{array}$$

Thus, the proof of inductiveness of φ, as well as all premises of DISTRANK are now of the form covered by the modest model theorem.

To compute the size of the instantiation needed, note that the product of ranges of $\mathbf{type}_1 \mapsto \mathbf{type}_0$ variables is 40 (5 locations, and 2 each for the fork and two *nvr*'s). There are three free variables in the system, 1 and N, and $N-1$. (The reason we include $N-1$ is, e.g., its explicit mention in φ^a). Following the remarks on the modest model theorem, since the three variables are consecutive, and since all constructs have we only have $i \ominus 1$, the size of the (modest) model we need to take is $40(u + 1) + 1$, where u is the number of universally quantified variables. Since $u \leq 2$ for each of D1–D6 (it is 0 for D4, 1 for D1, and 2 for D2, D3, and D5), we chose an instantiation of 121.

The following tables present the number of BDD nodes computed for each auxiliary construct, and the time it took to validate each of the inductiveness of φ and the premises D1–D6 on the instantiation.

4 Imposing Ordering on Transitions

In the previous section we showed how to deal with helpful assertions that are, strictly speaking, universal, but the "cause" for the universal quantifier is a ∓ 1 operation, that, once admitted into the set of restricted assertions, results in the helpful assertions being unquantified. In this section we study helpful assertions that are "truly" universal.

Construct	BDD nodes
φ	1,779
$pend$	3,024
ρ	10,778
h_p's	< 10
δ	≤ 10

Premise	Time to Validate
φ (inductiveness)	0.39 seconds
D1	$<$ 0.01 seconds
D2	0.42 seconds
D3	163.74 seconds
D4	0.01 seconds
D5	138.59 seconds
D6	0.02 seconds

Universal helpful assertions appear quite frequently. As a matter of fact most helpful assertions seem to be of the type $h(i) : \forall j.p(i, j)$ where i is the index of the process who can take a helpful step, and all other processes (j) satisfy some supporting conditions. However, such a helpful assertion presents a problem when trying to verify premise D4 of rule DISTRANK, since we obtain an EA-disjunct in the premise. In this section we show a new proof rule for progress, that allows us to order the helpful assertions in terms of the precedence of their helpfulness. "The helpful" assertion is then the minimal in the ordering, so that we can avoid the disjunction in the r-h-s of Premise D4.

4.1 Pre-Ordering Transitions

A binary relation $\preceq$ is a pre-order over domain $\mathcal{D}$ if it is reflexive, transitive, and total.

Let S be a FBDS. Let the set of transitions be $\mathcal{T}(N) = [0..4] \times N$ (as in Subsection 2.5). For every state in $S(N)$, define a pre-order $\preceq$ over $\mathcal{T}$. From the totality of $\preceq$, every $S(N)$-state has some $\tau_\ell[i] \in \mathcal{T}$ which is minimal according to $\preceq$. We replace premise D4 in DISTRANK with a premise stating that that for every pending state s, the transition that is minimal in s is also helpful at s. We call the new rule PRERANK and, to avoid confusion, name its premises R1–R6. Thus, PRERANK is exactly like DISTRANK, with the addition of a pre-order $\preceq : \Sigma \rightarrow 2^{\mathcal{T}\times\mathcal{T}}$, and replacing D4 by:

$$\textbf{R4.}\ \text{For every } \tau_1 \in \mathcal{T}, \quad pend \wedge \left(\bigwedge_{\tau_2 \in \mathcal{T}} \tau_1 \preceq \tau_2 \right) \quad \longrightarrow \quad h_{\tau_1}$$

In order to automate the application of PRERANK, we need to be able to automatically generate the pre-order relation $\preceq$. As usual, we first instantiate $S(N_0)$, compute concrete $\preceq$, and then use the method *project&generalize* to compute an abstract $\preceq^a$. The main problem is the computation of the concrete $\preceq$. We define $s \models \tau_1 \preceq \tau_2$ if:

$$s \models ((\neg h_{\tau_2} \wedge pend)\, \mathcal{W}\, (h_{\tau_1} \wedge pend)) \vee \neg((\neg h_{\tau_1} \wedge pend)\, \mathcal{W}\, (h_{\tau_2} \wedge pend)) \quad (1)$$

where $\mathcal{W}$ is the *weak-until* or *unless* operator.

The intuition behind the first disjunct is that for a state s, h_{τ_1} is "helpful earlier" than h_{τ_2} if every path leading from s that reaches h_{τ_1} doesn't reach $h\tau_2$ before. The role of the second disjunct is to guarantee the totality of $\preceq$, so that when h_{τ_1} precedes h_{τ_2} in some of computations, and h_{τ_2} precedes h_{τ_1} in others, we obtain both $\tau_1 \preceq \tau_2$ and $\tau_2 \preceq \tau_1$. To abstract a formula $\varphi(\tau_{\ell_1}[i])\, \mathcal{W}\, \varphi(\tau_{\ell_2}[j])$, we use *project&generalize*,

projecting onto processes i and j. To abstract the negation of such a formula, we first abstract the formula, and then negate the result. Therefore, to abstract Formula (1), we abstract each disjunct separately, and then take the disjunction of the abstract disjuncts.

4.2 Case Study: Bakery

Consider again program BAKERY of Example 1, and suppose we want to verify the liveness property $(\pi[z] = 1) \Rightarrow \diamondsuit(\pi[z] = 3)$. We instantiate the system to $N_0 = 3$, and obtained the auxiliary assertions φ, $pend$, and the h's and δ's[4]. After applying the *project&generalize* method, we obtain for $h_\ell[i]$, two type of assertions. One is for the case that $i = z$, and then, as expected, $h_2[z]$ is the most interesting one, having an A-construct claiming the z's ticket is the minimal among ticket holders. The other case is for $j \neq z$, and there we have a similar A-construct (for j's tickets minimality) for $\ell = 2, 3, 4$. For the pre-order, one must consider every $\tau_{\ell_1}[i] \preceq \tau_{\ell_2}[j]$ for every $\ell_1, \ell_2 = 1, ..., 4$ and $i = z \neq j, i = j \neq z, i, j \neq z$ for $(\ell_1, i) \neq (\ell_2, j)$. We list here the results for $\tau_{\ell_1}[i] \preceq \tau_{\ell_2}[j]$ for $i \neq z$ that are not trivially T. We deonte:

$$\alpha(i,j) : \pi[i] = 2 \rightarrow y[j] < y[i]; \qquad \beta(i,j) : \pi[i] = 2 \wedge y[i] < y[j];$$
$$\gamma(i,L) : \pi[i] \in L \rightarrow y[z] < y[i]$$

	$\tau_1[i]$	$\tau_2[2]$	$\tau_3[i]$	$\tau_4[i]$
$\tau_1[j]$	$i = j$ $\vee\ j \neq z$ $\vee\ \pi[z] = 2$	$j \neq z \wedge \pi[z] = 2 \wedge \alpha(j,z)$ $\vee\ i = j = z \wedge \pi[z] = 1$	$j = z$ $\vee\ \pi[z] = 2 \wedge \alpha(j,z) \wedge \pi[j] \neq 3$	$j = z$ $\vee\ \pi[z] = 2 \wedge \alpha(j,z)$ $\wedge \pi[j] < 3$
$\tau_2[j]$	$j \neq z$ $\vee\ \pi[z] = 2$	$i = j$ $\vee\ \beta(i,j)$ $\vee\ \pi[j] \neq 2$ $\vee\ j \neq z \wedge y[z] < y[i]$	$j = z$ $\vee\ i = j \wedge \pi[j] \neq 3$ $\vee\ \pi[z] = 1$ $\vee\ i \neq j \wedge (\pi[j] \notin \{2,3\} \vee$ $\beta(i,j) \vee y[z] < y[j])$	$j = z$ $\vee\ i = j \wedge \pi[j] < 3$ $\vee\ \pi[z] = 1$ $\vee\ i \neq j \wedge (\pi[< 2 \vee \beta(i,j)$ $\vee\ y[z] < y[j])$
$\tau_3[j]$	$j \neq z$ $\vee\ \pi[z] = 2$	$\neg(i = j = z) \wedge (\pi[z] = 1$ $\vee \beta(i,j) \vee \pi[i] = 3$ $\vee \gamma(j, [2,3])$	$i = j \vee j = z$ $\vee\ \beta(i,j) \vee \pi[i] = 3$ $\vee\ \gamma(j, [2,3])$	$i = j \vee j = z$ $\vee\ \beta(i,j) \vee \pi[i] = 3$ $\vee\ \gamma(j, [0,1])$ $\vee\ \pi[z] = 1$
$\tau_4[j]$	$j \neq z$ $\vee\ \pi[z] = 2$	$i, j \neq z \wedge (\pi[z] = 1 \vee$ $\beta(i,j) \vee \pi[i] \vert 2 \vee$ $\gamma(j,z)$	$j = z \vee \beta(i,j)$ $\vee\ i \neq j \wedge \pi[i] > 2$ $\vee\ \gamma(j, [2,3])$	$i = j \vee j = z$ $\vee\ \beta(i,j) \vee \pi[i] > 2$ $\vee\ \gamma(j, [2..4])$

Using the above pre-order, we succeeded in validating Premises R1–R6 of PRE-RANK, thus establishing the liveness property of program BAKERY.

References

1. K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986.

[4] *cs.nyu.edu/acsys/Tlv/assertions* contains full list of assertions and pre-order definitions

2. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In CAV'01, pages 221–234. LNCS 2102, 2001.
3. K. Baukus, Y. Lakhnesche, and K. Stahl. Verification of parameterized protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001.
4. N. Bjørner, I. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, 1995.
5. N. Bjørner, I. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, pages 589–623. LNCS 976, 1995.
6. E. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *(*CONCUR'95), pages 395–407. LNCS 962, 1995.
7. M. Colon and H. Sipma. Practical methods for proving program termination. In CAV'02, pages 442–454. LNCS 2404, 2002.
8. E. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *(*CADE'00), pages 236–255, 2000.
9. E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In POPL'95, 1995.
10. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In G. Levi and B. Steffen, editors, *Proc. of the* $5^t h$ *workshop on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2004. To appear.
11. E. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In TACAS'98, pages 424–438. LNCS 1384, 1998.
12. V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In CAV'97. LNCS 1254, 1997.
13. B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In TACAS'00. LNCS 1785, 2000.
14. Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. In *Proc.* 15^{th} *Intl. Conference on Computer Aided Verification (CAV'03), volume 2775 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 381–393, 2003.
15. Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 4(2):328–342, 2000.
16. Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In CONCUR'02, pages 101–105. LNCS 2421, 2002.
17. D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In POPL'97, 1997.
18. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
19. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In CAV'98, pages 110–121. LNCS 1427, 1998.
20. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In TACAS'01, pages 82–97. LNCS 2031, 2001.
21. A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In CAV'02, pages 107–122. LNCS 2404, 2002.
22. E. Shahar. *The TLV Manual*, 2000. http://www.wisdom.weizmann.ac.il/˜verify/tlv.
23. N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.
24. L. Zuck, A. Pnueli, and Y. Kesten. Automatic verification of free choice. In *Proc. of the* $3^r d$ *workshop on Verification, Model Checking, and Abstract Interpretation*, pages 208–224. LNCS 2294, 2002.