International Journal on Software Tools for Technology (2006) 8(3): 261–279
DOI 10.1007/s10009-005-0193-x

SPECIAL SECTION ON TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS

Yi Fang · Nir Piterman · Amir Pnueli · Lenore Zuck

Liveness with invisible ranking

Published online: 17 March 2006

Abstract The method of invisible invariants was developed originally in order to verify safety properties of parameterized systems in a fully automatic manner. The method is based on (1) a *project&generalize* heuristic to generate auxiliary constructs for parameterized systems and (2) a *small-model theorem*, implying that it is sufficient to check the validity of logical assertions of a certain syntactic form on small instantiations of a parameterized system. The approach can be generalized to any deductive proof rule that (1) requires auxiliary constructs that can be generated by *project&generalize*, and (2) the premises resulting when using the constructs are of the form covered by the small-model theorem.

The method of *invisible ranking*, presented here, generalizes the approach to liveness properties of parameterized systems. Starting with a proof rule and cases where the method can be applied almost "as is," the paper progresses to develop deductive proof rules for liveness and extend the small-model theorem to cover many intricate families of parameterized systems.

Keywords Automatic verification · Parametrized systems · Liveness · Deductive verification · BDD techniques

Y. Fang
New York University, New York, NY, USA
E-mail: yifang@cs.nyu.edu

N. Piterman
Weizmann Institute, Rehovot, Israel
E-mail: nir.piterman@weizmann.ac.il

A. Pnueli (✉)
New York University, New York, NY, USA;
Weizmann Institute, Rehovot, Israel
E-mail: amir@cs.nyu.edu

L. Zuck
University of Illinois at Chicago, IL, USA
E-mail: lenore@cs.uic.edu

1 Introduction

Uniform verification of parameterized systems is one of the most challenging problems in verification. Given a parameterized system $S(N) : P[1] \| \cdots \| P[N]$ and a property p, uniform verification attempts to verify that $S(N)$ satisfies p for every $N > 1$. One of the most powerful approaches to verification that 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 temporal property of the system. The two tasks that the user has to perform are:

1. Provide some auxiliary constructs that appear in the premises of the rule;
2. Use the auxiliary constructs to establish the logical validity of the premises.

When performing manual deductive verification, the first task is usually the more difficult one, 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 [19] or STeP [3], which require user guidance and interaction and place additional burden on the user. The difficulties in the execution of these two tasks are the main reasons why deductive verification is not used more widely.

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

In [2, 20], we introduced the method of *invisible invariants*, which offers 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 INV.

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 $S(N)$ has the form

$\forall i : [1 \cdots N] q(i)$ or, more generally, $\forall i \neq j \dot{q}(i, j)$. 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 s, some assertion ψ that we wish to generalize to an assertion in the required form. Let r_1 be the projection of ψ on process $P[1]$, obtained by discarding references to variables that 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 generalization procedure as *project&generalize*. For example, when computing invisible invariants, ψ is the set of reachable states of $S(N_0)$. The procedure can be easily generalized to generate assertions of the type $\forall i_1, \ldots, i_k.p(\vec{i})$.

Having obtained a candidate for 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 certain type of assertions, there exists a (small) bound N_0 such that 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 cases covered by the theorem are those whose premises 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 that may refer only to the global variables and the local variables of $P[i]$ and $P[j]$ ($\forall\exists$-*assertions* for short).

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 nonintuitive, and it usually does 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 [2, 20], embedding a $\forall \vec{i}.q(\vec{i})$ candidate inductive invariant in INV results in premises that fall under the small-model theorem. In this paper, we extend the method of invisible invariants to apply to proofs of the second most important class of properties – the class of *response properties*. Response properties are liveness properties that can be specified by the temporal formula $\diamondsuit(q \rightarrow \diamondsuit r)$ (also written as $q \Rightarrow \diamondsuit r$) and guarantee that every q-state is eventually followed by an r-state. To handle response properties, we consider a certain variant of rule WELL [17], which establishes the validity of response properties under the assumption of *justice* (weak fairness). As is well known to users of this and similar rules, such a proof requires the generation of two kinds of auxiliary constructs: *helpful assertions* h_i that characterize, for transition τ_i, the states from which the transition is helpful in promoting progress toward the goal (r), and *ranking functions*, which measure progress toward the goal.

To apply *project&generalize* to the automatic generation of the ranking functions, we propose a variant of rule WELL. In this variant rule, called DISTRANK, we associate, with each potentially helpful transition τ_i, an individual ranking function $\delta_i : \Sigma \mapsto [0 \ldots c]$, mapping states to integers in a small range $[0 \ldots c]$ for some fixed small constant c. The global ranking function can be obtained by forming the multiset $\{\delta_i\}$. In most of the examples we consider, it suffices to take $c = 1$, which allows us to view each δ_i as an assertion and generate it automatically using *project&generalize*.

If, when applying rule DISTRANK, the auxiliary constructs h_i and δ_i have no quantifiers, all the resulting premises are $\forall\exists$ premises, and the small-model theorem can be used. One of the constructs required to be quantifier free are the helpful assertions that characterize the set of states from which a given transition is helpful. Many simple protocols have helpful assertions that are quantifier free (or, with the addition of some auxiliary variables, can be transformed into protocols that have quantifier-free helpful assertions). Some protocols, however, cannot be proven with such restricted assertions. To deal with such protocols, we extend the method of invisible ranking in two directions:

- Allowing expressions such as $i \pm 1$ to appear both in the transition relation as well as the auxiliary constructs; this is especially useful for ring algorithms, where many of the assertions have a $p(i, i+1)$ or $p(i, i-1)$ component.
- Allowing helpful assertions (and ranking functions) belonging to transitions of 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 on the states of all its neighbors. Substituted in the standard proof rules for progress properties, these assertions lead to premises that 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 prove, in Sect. 6.1, a *modest-model theorem*. The modest-model theorem establishes that $\forall\exists$ premises containing $i \pm 1$ subexpressions can be validated on relatively small models. The size of the models, however, is larger when compared to the small-model theorem of [20].

To handle the second extension, we introduce a novel proof rule, PRERANK: the main difficulty with helpful assertions of the form $h(i) = \forall j.H(i, j)$ is in the premise that claims that every "pending" state has some helpful transition enabled on it (D3 of rule DISTRANK in Sect. 2). Identifying such a helpful transition is the hardest step when applying the rule. The new rule, PRERANK (introduced in Sect. 7), implements a new mechanism for selecting a helpful transition based on the establishment of a *preorder* among transitions in each state. The "helpful" transitions are identified as the transitions that are minimal according to this preorder.

We emphasize that the two extensions are part of the same method, so that we can handle systems that both use ± 1 and require universal helpful assertions. For simplicity of exposition, we separate the extensions here.

1.1 Overview of paper

In Sect. 2 we present the general computational model of FTS and the restrictions that enable the application of the invisible auxiliary construct methods. We also review the small-model theorem, which enables automatic validation of the premises of the various proof rules. In addition, we outline a procedure that replaces compassion requirements by justice requirements, which justifies our focus on proof rules that assume justice only. Section 3 introduces the new DISTRANK proof rule and explains how we automatically generate ranking and helpful assertions for the parameterized case. We refer to the new method as the method of *invisible ranking*. We use a version of the token ring protocol for an ongoing example in this section. Section 4 shows how to enhance the *project&generalize* method to enable the generation of invariants in the form of Boolean combinations of universal assertions. This is demonstrated on a (different) version of the token ring protocol. In Sect. 5 we study a version of the Bakery algorithm, which seems beyond the scope of the invisible ranking method, and show how enhancing a protocol with some auxiliary variables can make it a suitable candidate for the method.

The method studied in Sects. 3–5 is adequate for cases where the set of reachable states can be satisfactorily over-approximated by Boolean combinations of $\forall$ assertions, and the helpful assertions as well as individual ranking functions δ_i can be represented by quantifier-free assertions. Not all examples can be handled by assertions that depend on a single parameter. In Sect. 6 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 Sect. 7 we present the PRERANK proof rule that uses preorder among transitions, discuss how to automatically obtain the preorder, and demonstrate the technique on the Bakery algorithm. Finally, we discuss the advantages of combining several preorder relations and demonstrate it on Szymanski's protocol for mutual exclusion [23].

All our examples have been run on TLV [22]. The interested reader may find the code, proof files, and output of all our examples at: *http://cs.nyu.edu/acsys/Tlv/assertions*.

1.2 Related work

This is the full version of [9, 10]. See [25] for a survey on the method of invisible constructs and an earlier version of invisible ranking.

The problem of uniform verification of parameterized systems is undecidable [1]. One approach to remedy this situation, pursued, e.g., in [7], 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 [8], network invariants that can be viewed as implicit induction [15], abstraction and approximation of network invariants [4], and other methods based on abstraction [12]. 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., [11]), or compositional methods (e.g., ([16]), combining 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.

Most of the mentioned methods deal only with safety properties. Among the methods dealing with liveness properties, we mention [6], which handles termination of sequential programs, network invariants [15], and *counter abstraction* [21].

2 Preliminaries

In this section we present our computational model, the small-model theorem, and the procedure that allows one to remove compassion (strong fairness). We assume that the reader is familiar with LTL, CTL, first-order logic, and fixpoint operators.

2.1 Fair transition systems

As our computational model, we take a *fair transition system* (FTS) [18] $S = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, with:

- $V = \{u_1, \ldots, u_n\}$ – a finite set of typed *system variables*. A *state* s of the system provides a type-consistent interpretation of the system variables V, assigning to each variable $v \in V$ a value $s[v]$ in its domain. Let Σ denote the set of all states over V. An *assertion* over V is a first-order formula over V. A state s satisfies an assertion φ, denoted $s \models \varphi$, if φ evaluates to T by assigning $s[v]$ to every variable v appearing in φ. We say that s is a φ-state if $s \models \varphi$.
- Θ – the *initial condition*: an assertion characterizing the initial states. A state is called *initial* if it is a Θ-state.
- $\mathcal{T}$ – a finite set of transitions. Every transition $\tau \in \mathcal{T}$ is an assertion $\tau(V, V')$ relating the values V of the variables in state $s \in \Sigma$ to the values V' in an S-successor state $s' \in \Sigma$. Given a state $s \in \Sigma$, we say that $s' \in \Sigma$ is a *τ-successor* of s if $\langle s, s' \rangle \models \tau(V, V')$ where, for each $v \in V$, we interpret v as $s[v]$ and v' as $s'[v]$. We say

that transition τ is *enabled* in state s if it has some τ-successor; otherwise, we say that τ is *disabled* in s. Let $En(\tau)$ denote the assertion $\exists V'.\tau(V, V')$ characterizing the set of states in which τ is enabled, and let ρ denote the disjunction of all transitions, i.e., $\rho = \bigvee_{\tau \in \mathcal{T}} \tau$. The assertion ρ represents the *total transition* relation of S.

- $\mathcal{J} \subseteq \mathcal{T}$ – a set of *just* transitions (also called *weakly fair* transitions). Informally, $\tau \in \mathcal{J}$ rules out computations where τ is continuously enabled but taken only finitely many times.
- $\mathcal{C} \subseteq \mathcal{T}$ – a set of *compassionate* transitions (also called *strongly fair* transitions). Informally, $\tau \in \mathcal{C}$ rules out computations where τ is enabled infinitely many times but taken only finitely many times.

For technical reasons, and with no loss of generality, we assume that $\mathcal{T}$ always contains the *idling transition* $\tau_0 : V' = V$, which preserves the values of all system variables. Taking such a transition is often described as a *stuttering step*. We also require that the idling transition be taken to be a just transition.

Let $\sigma : s_0, s_1, s_2, \ldots$ be an infinite sequence of states. We say that transition $\tau \in \mathcal{T}$ is *enabled at position k* of σ if τ is enabled on s_k. We say that τ is *taken at position k* if s_{k+1} is a τ-successor of s_k. Note that several different transitions can be considered as taken at the same position.

We say that σ is a *computation* of an FTS S if it satisfies the following requirements:

- *Initiality* – s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* – for each $\ell = 0, 1, \ldots$, state $s_{\ell+1}$ is a ρ-successor of s_ℓ.
- *Justice* – for every $\tau \in \mathcal{J}$, it is not the case that τ is continuously enabled beyond some point j in σ (i.e., τ is enabled at every position $k \geq j$) but not taken beyond j.
- *Compassion* – for every $\tau \in \mathcal{C}$, it is not the case that τ is enabled at infinitely many positions in σ but taken at only finitely many positions.

Note the fact that the idling transition is just implies that every computation contains infinitely many stuttering steps.

2.2 Bounded fair transition systems

To allow the application of the invisible constructs methods, we further restrict the systems we study, leading to the model of *bounded fair transition systems* (BFTS), which 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 multitype case is straightforward.

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

1. **bool**: the set of Boolean and finite-range scalars;
2. **index**: a scalar data type that includes integers in the range $[1 \,..\, N]$;
3. **data**: a scalar data type that includes integers in the range $[0 \,..\, N]$;

```
in N : natural where N > 1
   tloc : [1..N]
             ┌ loop forever do                          ┐
  N          │  ┌ 0 : if tloc = i then tloc := i ⊕_N 1 ┐ │
 ||  P[i] :: │  │     go to {0,1}                       │ │
i=1          │  │ 1 : await tloc = i                    │ │
             └  └ 2 : Critical                          ┘ ┘
```

Fig. 1 Program TOKEN-RING

4. Any number of arrays of the type **index** $\mapsto$ **bool**. We refer to these arrays as *Boolean arrays*; and
5. At most one array of the type b : **index** $\mapsto$ **data**. We refer to this array as the *data array*.

Atomic formulas may compare two variables of the same type. For example, if y and y' are **index** variables, and z is an **index** $\mapsto$ **data** array, then $y = y'$ and $z[y] < z[y']$ are both atomic formulas. For z : **index** $\mapsto$ **data** and y : **index**, 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 $\forall \vec{i}.u(\vec{i})$, where $u(\vec{i})$ is a restricted assertion.

As the transitions $\tau \in \mathcal{T}$, we allow assertions of the form $\tau(\vec{i}) : \forall \vec{j}.\psi(\vec{i}, \vec{j})$ for a restricted assertion $\psi(\vec{i}, \vec{j})$. This results in total transition relatin $\rho : \exists \vec{i}.\forall \vec{j}.\psi(\vec{i}, \vec{j})$. For simplicity, we assume that all quantified and free variables are of type **index**.

Example 1 (The token ring algorithm) Consider the program TOKEN-RING in Fig. 1, which is a mutual exclusion algorithm for any N processes.

In this version of the algorithm, the global variable *tloc* represents the index of the process currently holding the token. Location 0 constitutes the noncritical section that may nondeterministically exit to the trying section at location 1. While being in the noncritical section, a process guarantees to move the token to its right neighbor whenever it receives it. This is done by incrementing *tloc* by 1, modulo N. At the trying section, a process $P[i]$ waits until it receives the token, which is signaled by the condition $tloc = i$.

Figure 2 describes the BFTS corresponding to the program TOKEN-RING, where for a variable $v \in V$, $pres(v)$ denotes $v' = v$ and for a set $U \subseteq V$, $pres(U)$ denotes $\bigwedge_{v \in U} pres(v)$. When there is no danger of confusion, we use $pres(a_1, \ldots, a_k)$ instead of $pres(\{a_1, \ldots, a_k\})$. Note that *tloc* is an **index** variable, while the program counter π is an **index** $\mapsto$ **bool** array. Actually, π is of type **index** $\mapsto$ $[0 \ldots 2]$, but it can be encoded by two Boolean arrays, hence we are justified in referring to it here and in future examples as an **index** $\mapsto$ **bool** array.

Strictly speaking, the transition relation as presented above does not conform to the definition of a Boolean assertion since it contains the atomic formula $tloc' = i \oplus_N 1$. However, this can be rectified by a two-stage reduction. First, we replace $tloc' = i \oplus_N 1$ by $(i < N \;\wedge\; tloc' =$

$$V : \begin{cases} tloc : [1..N] \\ \pi : \quad \textbf{array}[1..N] \textbf{ of } [0..2] \end{cases}$$
$$\Theta : \forall i.\pi[i] = 0$$
$$\mathcal{T} : \begin{cases} \tau_0^1(i) : \forall j \neq i.\ \pi[i] = 0 \wedge tloc = i \wedge tloc' = i \oplus_N 1 \\ \qquad\qquad \wedge \pi'[i] \in \{0,1\} \wedge pres(\pi[j]) \\ \tau_0^2(i) : \forall j \neq i.\ \pi[i] = 0 \wedge tloc \neq i \wedge \pi'[i] = 1 \wedge \\ \qquad\qquad pres(\pi[j], tloc) \\ \tau_1(i) : \forall j \neq i.\ \pi[i] = 1 \wedge tloc = i \wedge \pi'[i] = 2 \wedge \\ \qquad\qquad pres(\pi[j], tloc) \\ \tau_2(i) : \forall j \neq i.\ \pi[i] = 2 \wedge \pi'[i] = 0 \wedge pres(\pi[j], tloc) \\ \tau_{id} : \quad \forall j. \quad pres(\pi[j], tloc) \end{cases}$$
$$\mathcal{J} : \{\tau_0^1(i), \tau_1(i), \tau_2(i), \tau_{id} \mid i \in [1..N]\}$$

Fig. 2 BFTS for program TOKEN-RING

$i+1) \vee (i = N \wedge tloc' = 1)$. Then, we replace the formula $\tau(i) : \forall j \neq i.(\ldots tloc' = i+1\ldots)$ by $\tau(i, i_1) : \forall j \neq i, j_1.(j_1 \leq i \vee i_1 \leq j_1) \wedge (\ldots tloc' = i_1 \ldots)$, which guarantees that $i_1 = i+1$.

Note that transition $\tau_0^2(i)$ is not listed as a just transition. This allows a process to remain forever in its noncritical location (0) as long as it diligently transfers any incoming token to its right neighbor. Also note that this system has an empty set of compassion transitions, which we omitted from the presentation in Fig. 2.

Example 2 (The bakery algorithm) Consider the program BAKERY in Fig. 3, which is a variant of Lamport's original bakery algorithm that offers a solution to the mutual exclusion problem for any N processes.

In this version of the algorithm, location 0 constitutes the noncritical section which a process may nondeterministically exit to the trying section at location 1. Location 1 is the ticket assignment location. Location 2 is the waiting phase, where a process waits until it holds the minimal ticket. Location 3 is the critical section, and location 4 is the exit section. Note that y, the ticket array, is of type **index** $\mapsto$ **data**, and the program location array (which we denote by π) is of type **index** $\mapsto$ **bool**. Note also that the ticket assignment statement at 1 is nondeterministic and may modify the values of all tickets. Figure A.1 in Appendix A.1 describes the BFTS corresponding to the 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 states that 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. In a symmetric way, we denote by $R \circ \alpha$ the assertion characterizing all the states that are R-predecessors of α-states.

```
in    N : natural where N > 1
local y : array [1..N] of [0..N]
          where y = 0

 N          [ loop forever do
 ||  P[i] ::  [ 0 : NonCritical
i=1             1 : y := maximal value to y[i] while
                      preserving order of elements
                2 : await ∀j ≠ i. ( y[j] = 0 ∨
                                    y[j] > y[i] )
                3 : Critical
                4 : y[i] := 0 ] ]
```

Fig. 3 Program BAKERY

2.3 The small-model theorem

Let $\varphi : \forall\vec{i}\exists\vec{j}.R(\vec{i}, \vec{j})$ be an $\forall\exists$-formula, where $R(\vec{i}, \vec{j})$ is a restricted assertion that refers to the state variables of a parameterized BFTS $S(N)$ in addition to the quantified (**index**) variables $\vec{i}$ and $\vec{j}$. We show that if there exists some model that does not satisfy this assertion, then there exists a model smaller than a certain bound that does not satisfy it. It follows that, in order to check the validity of this formula, it is enough to check all models up to the given bound. The proof follows by contracting a model that does not satisfy φ in to a smaller model that does not satisfy ϕ. In order to decrease the size of the model, we consider the existentially quantified variables in the negation of φ. These variables refer to processes in the model that not satisfy ϕ. We keep the processes referred to by these variables and throw away the rest.

For simplicity, we assume that the only data variable/constant that may appear in R is the data constant 0. Let N_0 be the number of universally quantified variables, free **index** variables, and **index** constants appearing in R. The following theorem, 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) *Let φ be an $\forall\exists$-formula as above. Then φ is valid over $S(N)$ for every $N \geq 2$ iff φ is valid over $S(N)$ for every $N \leq N_0$.*

For completeness of presentation we include the proof.

Proof We denote by ψ the formula $\exists\vec{i}\forall\vec{j}.\neg R(\vec{i}, \vec{j})$, which is the negation of φ. Assume ψ is satisfiable in state s of a system $S(N_1)$ for $N_1 > N_0$. We show that it is satisfiable in a state s' of a system $S(N)$ for some $N \leq N_0$.

Let $\mathcal{V}_\exists$ be the set of **index** variables that appear existentially quantified in ψ. Let F be the set of **index** constants (including 1) and variables that appear free in ψ. Note that state s provides an interpretation for all the variables in F and all the arrays that appear in s. Similarly, let $\mathcal{V}_\forall$ be the set of **index** variables that appear universally quantified in ψ, i.e., the $\vec{j}$ variables.

The fact that $\psi : \exists\vec{i}\forall\vec{j}.\neg R(\vec{i}, \vec{j})$ is satisfiable in s means that there exists an assignment α that interprets all variables of $\mathcal{V}_\exists$ by values in the domain $[1\ldots N_1]$ such that $(s, \alpha) \models \chi$, where $\chi : \forall\vec{j}.\neg R(\vec{i}, \vec{j})$ and (s, α) is the joint interpretation that interprets all system variables according to state s and all $\mathcal{V}_\exists$-variables according to the assignment α.

Let $U = \{u_1 < u_2 < \cdots < u_k\}$ be a sorted list of values assigned to the $\mathcal{V}_\exists \cup F$-variables by α and s. Obviously,

$k \leq N_0$. Let $f : U \rightarrow [1 \ldots k]$ be the bijection such that $f(u) = i$ iff $u = u_i$.

Similarly, let $D = \{0 = d_0 < d_1 < d_2 < \cdots < d_r\}$ be a sorted list of all the values assigned by s to the elements $b[u_i]$ for the data array b and $i \in [1 \ldots k]$. We always include 0 in D, even if it is not obtained as the value of some $b[u_i]$. Obviously, $r \leq k$. Let $g : D \rightarrow [1 \ldots r]$ be the bijection such that $g(d) = j$ iff $d = d_j$.

We construct a state s' of system $S(k)$ and an assignment $\alpha' : \mathcal{V}_\exists \mapsto [1 \ldots k]$ such that $(s', a') \models \chi$. The state s' is an interpretation defined as follows: for each variable $v \in F$, s' interprets v as $s'[v] = f(s[v])$. That is, $s[v] = u_i$ iff $s'[v] = i$. For every Boolean array a : **index** $\mapsto$ **bool**, we have $s'[a[i]] = s[a[u_i]]$, i.e., the value of $a[i]$ in state s' equals the value of $a[u_i]$ in state s. For the data array b : **index** $\mapsto$ **data**, we take $s'[b[i]] = g(s[b[u_i]])$, for each $i \in [1 \ldots k]$. That is, $s'[b[i]] = j$ iff $s[b[u_i]] = d_j$. Next, we define the interpretation α' as follows: for each variable $v \in \mathcal{V}_\exists$, α' interprets v as $\alpha'[v] = f(\alpha[v])$. That is, $\alpha[v] = u_i$ iff $\alpha'[v] = i$.

We proceed to show that $(s', \alpha') \models \chi$. To do so, consider an arbitrary assignment β' assigning to each variable $v \in \vec{j}$ a value $\beta'[v] \in [1 \ldots k]$. We will show that $(s', \alpha', \beta') \models \neg R(\vec{i}, \vec{j})$. As we show this for an arbitrary assignment β', it follows that $(s', \alpha') \models \forall \vec{j}. \neg R(\vec{i}, \vec{j})$. That is, $(s', \alpha') \models \chi$.

Consider the assignment β interpreting each $v \in \vec{j}$ as u_i iff $\beta'[v] = i$. It follows that β interprets each variable $v \in \vec{j}$ by a value in $[1 \ldots N_1]$. Since $(s, \alpha) \models \chi$, it follows that $(s, \alpha, \beta) \models \neg R(\vec{i}, \vec{j})$. By induction on the structure of the formula $\neg R(\vec{i}, \vec{j})$, we can show that every subformula $\gamma \in \neg R(\vec{i}, \vec{j})$ evaluates to T under the joint interpretation (s, α, β) iff γ evaluates to T under the interpretation (s', α', β').

We conclude that $(s', \alpha') \models \chi$, which leads to the result that ψ is satisfied in the state s' of system $S(k)$. □

The small-model theorem allows one to check the validity of ∀∃-assertions on small models. In [2, 20] we obtain, using *project&generalize*, candidate inductive assertions for the set of reachable states that are ∀-formulas, checking their inductiveness required checking validity of ∀∃-formulae, which can be accomplished using BDD techniques.

2.4 Removing compassion

The proof rule we are employing to prove progress properties assumes an noncompassionate system (system with no compassionate transitions). As outlined in [14],[1] every FTS S can be converted into an noncompassionate FTS $S_J = \langle V_J, \Theta_J, \mathcal{T}_J, \mathcal{J}_J, \emptyset \rangle$, where

$$V_J : V \cup \{nvr_\tau : \textbf{boolean} \mid \tau \in \mathcal{C}\},$$
$$\Theta_J : \Theta,$$
$$\mathcal{T}_J : \bigcup_{\tau \in \mathcal{T} \setminus \mathcal{C}} f_1(\tau) \cup \bigcup_{\tau \in \mathcal{C}} f_2(\tau),$$
$$\mathcal{J}_J : \bigcup_{\tau \in \mathcal{J} \setminus \mathcal{C}} f_1(\tau) \cup \bigcup_{\tau \in \mathcal{C}} f_2(\tau),$$

where $f_1, f_2 : \tau \rightarrow \tau_{c_J}$ are defined by:

$$f_1(\tau) = \tau \wedge pres(Nvr),$$
$$f_2(\tau) = \left[\begin{array}{l} \tau \wedge pres(Nvr) \vee , \\ \neg nvr_\tau \wedge nvr'_\tau \wedge pres(V_J \setminus \{nvr_\tau\}) \end{array} \right],$$
$$Nvr = \{nvr_\tau \mid \tau \in \mathcal{C}\}.$$

This transformation adds to the system variables, for each compassionate transition τ, a new Boolean variable nvr_τ. The intended role of nvr_τ is, nondeterministically, to identify a point in the computation beyond which τ is never enabled. The new transition relation includes two types of transitions: for each original noncompassionate transition τ, a transition $f_1(\tau)$ that behaves like τ while preserving the values of all nvr_τ variables. For each original compassionate transition $\tau \in \mathcal{C}$, $\mathcal{T}_J$ contains a transition $f_2(\tau)$ that either takes τ and preserves all nvr_τ variables or changes nvr_τ from F to T and preserves all other variables. Intuitively, as long as $nvr_\tau = $ F, $f_2(\tau)$ is enabled and, to comply with the justice requirement associated with $f_2(\tau)$, either τ is taken infinitely often or nvr_τ eventually set to T. Once nvr_τ is set to T, τ is not expected to be enabled (and therefore taken) ever again.

Let *Err* denote the assertion $\bigvee_{\tau \in \mathcal{C}} (En(\tau) \wedge nvr_\tau)$, describing states where both τ is enabled and nvr_τ holds, which indicates that the prediction that τ will never be enabled is premature. For a computation σ_J of S_J, denote by $\sigma_J \Downarrow_V$ the sequence obtained from σ_J by projecting away the *nvr* variables. The relation between S and its compassion-free version S_J is stated by the following claim.

Claim Let σ be an infinite sequence of S-states. Then σ is an S-computation iff there exists an *Err*-free computation σ_J of S_J such that $\sigma_J \Downarrow_V = \sigma$.

Proof In one direction, let $\sigma = s_0, s_1, \ldots$ be a computation of S. We will show how to define the values of nvr_τ at each position of the computation, such the resulting sequence of S_J-states $\tilde{\sigma} = \tilde{s}_0, \tilde{s}_1, \ldots$ is an *Err*-free computation of S_J.

The intention is to guarantee that transition $\tau \in \mathcal{C}$ is continuously disabled beyond some position j of σ iff nvr_τ is set to T at some position beyond j. For simplicity, assume that the compassionate transitions are $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$ and that we may refer to nvr_{τ_i} simply as nvr_i.

The initial values are determined as follows: for each $i = 1, \ldots, k$, the initial value of nvr_i is taken to be T iff τ_i is disabled at all positions of σ.

Next, we consider a step from position j to position $j+1$. If $s_j[V] \neq s_{j+1}[V]$, then we let $\tilde{s}_{j+1}[Nvr] = \tilde{s}_j[Nvr]$.

[1] The proof in [14] is an adaptation of the proofs in [5, 24] to the case of transition systems.

That is, if at least one system variable of system S is modified in step j, then all the *Nvr* variables preserve their values.

On the other hand, if step j is a stuttering step, i.e., $s_j[V] = s_{j+1}[V]$, we search for a transition $\tau_i \in \mathcal{C}$ such that $\tilde{s}_j[nvr_i] = \mathrm{F}$ but τ_i is disabled at all positions beyond j. If there exists such a transition, let m be such a transition with the minimal index. We set $\tilde{s}_{j+1}[nvr_m] = \mathrm{T}$ and $\tilde{s}_{j+1}[nvr_\ell] = \tilde{s}_j[nvr_\ell]$, for all $\ell \neq m$.

If there does not exist a τ_i such as described above, we let again $\tilde{s}_{j+1}[Nvr] = \tilde{s}_j[Nvr]$.

Since, as was previously observed, all computations contain infinitely many stuttering steps, the above definition guarantees that nvr_i eventually turns T iff τ_i eventually becomes continuously disabled. Furthermore, we never have a state in which τ_i is enabled while $nvr_i = \mathrm{T}$.

In the other direction, consider an *Err*-free computation σ_J of S_J. We claim that $\sigma = \sigma_J \Downarrow_V$ is a computation of S. Suppose, by contradiction, that some $\tau \in \mathcal{C}$ is enabled infinitely often but taken only finitely often in σ. Then it must be the case that $f_2(\tau)$ is enabled infinitely often in σ_J. As τ is taken finitely often in σ, it must be the case that nvr_τ is set in σ_J so as not to violate $\mathcal{J}_J$. Since τ is enabled infinitely often, it is enabled after nvr_τ is increased and σ_J is not *Err* free. □

We can therefore conclude that for every q and r,

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

which allows us to assume that all BFTSs we consider here have an empty compassion set.

3 The method of invisible ranking

In this section we present a new proof rule that allows, in some cases, to obtain an automatic verification of liveness properties for a BFTS of any size. We first describe the new proof rule and then present methods for the automatic generation of the auxiliary constructs required by the rule using TOKEN-RING as an ongoing example.

3.1 A distributed ranking proof rule

In Fig. 4 we present proof rule DISTRANK (short for DISTributed RANKing) for verifying response properties for BFTSs whose only fair transitions are just. The rule is configured to deal directly with parameterized systems. As in other rules for verifying response properties (e.g., [17]), progress is accomplished by the actions of *helpful transitions* in the system. In a parameterized system, the set of transitions has the structure $\mathcal{T}(N) = \{\tau_\ell[i] | \ell \in [0..m] \text{ and } i \in [1..N]\}$ for some fixed m. Typically, $[0..m]$ enumerates the locations within each process. For example, in the program TOKEN-RING, $\mathcal{T}(N) = \{\tau_\ell[i] | \ell \in [0..2] \text{ and } i \in [1..N]\}$, where each transition $\tau_\ell[i]$ is associated with location $\ell \in [0..2]$ within process $i \in [1..N]$. Requiring that

For a parameterized system with transitions $\mathcal{T}(N)$ where $\rho = \bigvee_{\tau\in\mathcal{T}(N)} \tau$,
set of states $\Sigma(N)$,
just transitions $\mathcal{J} \subseteq \mathcal{T}(N)$,
invariant assertion φ,
assertions $q, r, pend$ and $\{h_\tau \mid \tau \in \mathcal{J}\}$, and
ranking functions $\{\delta_\tau : \Sigma \to \{0,1\} \mid \tau \in \mathcal{J}\}$

D1.	$q \wedge \varphi$	$\rightarrow$	$r \vee pend$
D2.	$pend \wedge \rho$	$\rightarrow$	$r' \vee pend'$
D3.	$pend$	$\rightarrow$	$\bigvee_{\tau\in\mathcal{J}} h_\tau$
D4.	$pend \wedge \rho$	$\rightarrow$	$r' \vee \bigwedge_{\tau\in\mathcal{J}} \delta_\tau \geq \delta'_\tau$
For every $\tau \in \mathcal{J}$			
D5.	$h_\tau \wedge \rho$	$\rightarrow$	$r' \vee h'_\tau \vee \delta_\tau > \delta'_\tau$
D6.	$h_\tau \wedge \tau$	$\rightarrow$	$r' \vee \delta_\tau > \delta'_\tau$
D7.	h_τ	$\rightarrow$	$En(\tau)$

$$q \Rightarrow \Diamond r$$

Fig. 4 The liveness rule DISTRANK

$\tau_\ell[i]$ be just guarantees that it is taken or disabled infinitely often, thus that $\tau_\ell[i]$ is not continuously enabled and never taken beyond some point.

Assertion φ is an invariant assertion characterizing all the reachable states. Assertion *pend* characterizes the states that can be reached from a reachable q-state by an r-free path. For each transition τ, assertion h_τ characterizes the states at which τ is *helpful*. These are the states s that have a τ-successor s', and the transition from s to s' leads to a progress toward the goal. This progress is observed by immediately reaching the goal or a decrease in the ranking function δ_τ, as stated in premises D5 and D6. The ranking functions δ_τ measure progress toward the goal. The disabling of τ is often caused by τ being taken (D6), but it may also be caused by some condition turning false (D5). We require a decrease in ranking in both cases.

Premise D1 guarantees that any reachable q-state satisfies r or *pend*. Premise D2 guarantees that any successor of a *pend*-state also satisfy r or *pend*. Premise D3 guarantees that any *pend*-state has at least one transition that is helpful in this state. Premise D4 guarantees that ranking never increases on transitions between two *pend*-states. Note that, due to D2, every ρ-successor of a *pend*-state that has not reached the goal is also a *pend*-state. Premise D5 guarantees that taking a step from an h_τ-state leads into a state that either already satisfies the goal r or causes the rank δ_τ to decrease, or is again an h_τ-state. Premise D6 guarantees that taking a τ-transition from an h_τ-state either reaches the goal r or decreases the rank δ_τ. Premise D7 guarantees that in all h_τ-states τ is enabled. Together, premises D5–D7 imply that the computation cannot stay in h_τ forever, otherwise justice w.r.t τ is violated. Therefore, the computation must eventually decrease δ_τ. Since there are only finitely many δ_τ, and until the goal is reached they monotonically decrease, we can conclude that eventually an r-state is reached.

3.2 Automatic generation of the auxiliary constructs

We now proceed to show how the auxiliary constructs necessary for the application of rule DISTRANK can be automatically generated. Recall that we have to construct a *symbolic* version of each construct so that the rule can be applied to a generic N. We consider each auxiliary construct, provide a method for its generation, and illustrate it on the case of the program TOKEN-RING.

In TOKEN-RING, the progress property we wish to check is:

$$\pi[z] = 1 \Longrightarrow \Diamond\pi[z] = 2\,.$$

For simplicity, as all processes are symmetric we choose $z = 1$; thus we check

$$\pi[1] = 1 \Longrightarrow \Diamond\pi[1] = 2\,.$$

This property claims that every state in which process $P[1]$ is at location 1 is eventually followed by a state in which process $P[1]$ is at location 2.

The construction uses the instantiation $S(N_0)$ for the cutoff value N_0 required in Theorem 1. For TOKEN-RING, as explained in Sect. 3.3, $N_0 = 6$. We denote by Θ_C and ρ_C the initial condition and transition relation for $S(N_0)$. The construction begins by computing the *concrete* auxiliary constructs for $S(N_0)$, denoted by φ_C, $pend_C$. We then compute the concrete $h_k^C[j]$'s and $\delta_k^C[j]$'s. Next, we apply *project&generalize* to derive the symbolic (*abstract*) versions of these constructs: φ_A, $pend_A$, $h_k^A[j]$'s, and $\delta_k^A[j]$'s.

Since we focus on process 1, we would expect the constructs to have the symbolic forms $\varphi : \forall i.\varphi_A(i)$ and $pend : pend^A_{=1} \wedge \forall i{\neq}1.pend^A_{\neq 1}(i)$. For each $k \in [0..m]$, we need to compute $h_k^A[1]$, $\delta_k^A[1]$, and the generic $h_k^A[i]$, $\delta_k^A[i]$, which should be symbolic in i and apply for all i, $1 < i \leq N$. All generic constructs are allowed to refer to the global variables and to the variables local to $P[1]$ and $P[i]$.

3.2.1 Computing concrete and abstract φ

All concrete assertions are computed on $S(N_0)$. We set φ_C to be $reach_C = \Theta_C \circ \rho_C^*$, the assertion characterizing all states reachable within $S(N_0)$. Compute $\varphi_A(i) = reach_C[3 \mapsto i]$ by projecting $reach_C$ on index 3 and then generalizing 3 to i, that is, maintaining only variables pertaining to process 3 and then replacing every reference to index 3 by a reference to index i.

For example, in TOKEN-RING(6),

$$\varphi_C = \bigwedge_{j=1}^{6} (at\_\ell_{0,1}[j] \vee tloc = j)\,,$$

where $at\_\ell_{0,1}[j]$ is an abbreviation for $\pi[j] \in \{0, 1\}$.

The projection of φ_C on $j = 3$ yields

$$(at\_\ell_{0,1}[3] \vee tloc = 3)\,.$$

The generalization of 3 to i yields

$$\varphi_A(i) : at\_\ell_{0,1}[i] \vee tloc = i\,.$$

The assertion φ_A is $\forall i.\varphi_A(i)$.

Note that when we generalize, we should generalize not only the values of the variables local to $P[3]$ but also the case that the global variable, such as *tloc*, has the value 3. The choice of 3 as the generic value is arbitrary. Any other value would do as well, but we prefer indices different from 1 and N.

In this part we computed $\varphi_A(i)$ as the generalization of 3 into i in φ_C, which is denoted by $\varphi_A(i) = \varphi_C[3 \mapsto i]$. In later parts we may need to generalize two indices, such as $\alpha_A = \alpha_C[2 \mapsto i, 4 \mapsto j]$, where α_C and α_A are concrete and abstract versions of some assertion α. The way we compute such abstractions over the state variables *tloc* and π of system TOKEN-RING is given by

$$\alpha_A(tloc, \pi) = i < j \;\wedge\; \exists tloc', \pi'.\begin{pmatrix} \alpha_C(tloc', \pi') \;\wedge \\ map(2, i, 4, j) \end{pmatrix},$$

where

$$map(2, i, 4, j) = \begin{bmatrix} \pi[i] = \pi'[2] \;\wedge\; \pi[j] = \pi'[4] & \wedge \\ tloc = i \iff tloc' = 2 & \wedge \\ tloc = j \iff tloc' = 4 & \wedge \\ tloc < i \iff tloc' < 2 & \wedge \\ tloc < j \iff tloc' < 4 & \end{bmatrix}.$$

Note that this computation is very similar to the symbolic computation of the predecessor of an assertion, where $map(2, i, 4, j)$ serves as a transition relation. Indeed, we use the same module used by a symbolic model checker for carrying out this computation.

3.2.2 Computing concrete and abstract pend:

Compute the assertion

$$pend_C = (\varphi_C \wedge q \wedge \neg r) \circ (\rho_C \wedge \neg r')^*$$

characterizing all the states that can be reached from a reachable $(q \wedge \neg r)$-state by an r-free path. Then we take $pend^A_{=1} = pend_C[1 \mapsto 1]$ and $pend^A_{\neq 1}(i) = pend_C[1 \mapsto 1, 3 \mapsto i]$.

Thus, for TOKEN-RING(6),

$$pend_C = \varphi_C \;\wedge\; at\_\ell_1[1]\,.$$

We therefore take

$$pend^A_{=1} : at\_\ell_1[1]$$

and

$$pend^A_{\neq 1}(i) : at\_\ell_1[1] \;\wedge\; (at\_\ell_{0,1}[i] \vee tloc = i)\,.$$

Finally, $pend_A = pend^A_{=1} \wedge \forall i{\neq}1.pend^A_{\neq 1}(i)$, yielding

$$pend_A = at\_\ell_1[1] \;\wedge\; \forall i{\neq}1.(at\_\ell_{0,1}[i] \vee tloc = i).$$

3.2.3 Computing concrete and abstract $h_k[i]$'s

We compute the concrete helpful assertions $h^C_k[i]$. This is based on the following analysis. Assume that *set* is an assertion characterizing a set of states, and let τ be some just transition. We wish to identify the subset of states ϕ within *set* for which the transition τ is an *escape* transition. That is, any application of this transition to a ϕ-state takes us out of *set*. Consider the fix-point equation:

$$\phi \;=\; set \wedge En(\tau) \wedge AX(\phi \vee \neg set) \wedge AX_\tau(\neg set)\,. \qquad (1)$$

The equation states that every ϕ-state must satisfy $set \wedge En(\tau)$, every ρ-successor of a ϕ-state is either a ϕ-state or lies outside of *set*, and every τ-successor of a ϕ-state lies outside of *set*. Note that the expressions $AX\psi$ and $AX_\tau\psi$ can be computed by $\neg(\rho \circ (\neg\psi))$ and $\neg(\tau \circ (\neg\psi))$, respectively.

By taking the maximal solution of the fix-point equation (1), denoted $\nu\phi(set \wedge En(\tau) \wedge AX(\phi \vee \neg set) \wedge AX_\tau(\neg set))$, we compute the subset of states within *set* for which τ is helpful.

Following is an algorithm that computes the concrete helpful assertions $\{h^C_k[i]\}$ corresponding to the just transitions $\{\tau_k[i]\}$ of system $S(N_0)$. For simplicity, we will use $\tau \in \mathcal{T}(N_0)$ as a single parameter. Let

$$\mathbf{maxfix}(set, \tau) : \nu\phi \begin{bmatrix} set \wedge En(\tau) & \wedge \\ AX(\phi \vee \neg set) & \wedge \\ AX_\tau(\neg set) & \end{bmatrix}$$

for each $\tau \in \mathcal{T}(N_0)$ **do** $h_\tau := 0$
set $:= pend_C$
for all $\tau \in \mathcal{T}(N_0)$ **s.t. maxfix**$(set, \tau) \neq 0$ **do**

$$\begin{bmatrix} h_\tau := h_\tau \;\vee\; \mathbf{maxfix}(set, \tau) \\ set := set \;\wedge\; \neg h_\tau \end{bmatrix}$$

The "**for all** $\tau \in \mathcal{T}(N_0)$" iteration terminates when it is no longer possible to find a $\tau \in \mathcal{T}(N_0)$ that satisfies the non-emptiness requirement. The iteration may choose the same τ more than once. When the iteration terminates, *set* is 0, i.e., for each of the states covered under $pend_C$ there exists a helpful justice requirement that causes it to progress.

Having found the concrete $h^C_k[i]$, we compute the abstract $h^A_k[i]$ by using *project&generalize* as follows: for each $k \in [0 \ldots m]$, we let $h^A_k[1] = h^C_k[1][1 \mapsto 1]$ and $h^A_k[i] = h^C_k[3][1 \mapsto 1, 3 \mapsto i]$.

Applying this procedure to TOKEN-RING(6), we obtain the symbolic helpful assertions described in Appendix A.2.

3.2.4 Computing concrete and abstract $\delta_k[i]$'s

As before, we begin by computing the concrete ranking functions $\delta^C_k[i]$. We observe that $\delta^C_k[i]$ should equal 1 on every state for which $\tau_k[i]$ is helpful and should decrease from 1 to 0 on any transition that causes a helpful $\tau_k[i]$ to become unhelpful. Furthermore, $\delta^C_k[i]$ can never increase. It follows that $\delta^C_k[i]$ should equal 1 on every pending state from which there exists a pending path to a pending state satisfying $h^C_k[i]$. Thus, we compute $\delta^C_k[i] = pend_C \wedge ((\neg r)\, E\mathcal{U}\, h^C_k[i])$, where $E\mathcal{U}$ is the "existential-until" CTL operator. This formula identifies all states from which there exists an r-free path to an $(h^C_k[i])$-state.

Having found the concrete $\delta^C_k[i]$, we obtain the abstract $\delta^A_k[i]$ by using *project&generalize* as follows: for each $k \in [0 \ldots m]$, we let $\delta^A_k[1] = \delta^C_k[1][1 \mapsto 1]$ and $\delta^A_k[i] = \delta^C_k[3][1 \mapsto 1, 3 \mapsto i]$.

The abstract ranking function obtained by applying this procedure to TOKEN-RING(6) are described in Appendix A.2.

3.3 Validating the premises

Having computed internally the necessary auxiliary constructs, and checking the invariance of φ, it only remains to check that the six premises of rule DISTRANK are all valid for any value of N. Here we use the small-model theorem stated in Theorem 1, which allows us to check their validity for all values of $N \leq N_0$ for the cutoff value of N_0 specified in the theorem. First, we have to ascertain that all premises have the required $\forall\exists$ form. For auxiliary constructs of the form we have stipulated in this section, this is straightforward. Next, we consider the value of N_0 required in each of the premises and take the maximum. Note that once φ is known to be inductive, we can freely add it to the left-hand side of each premise, which we do for the case of premises D5–D7, which, unlike others, do not include any inductive component.

Usually, the most complicated premise is D2, and this is the one that determines the value of N_0. For the program TOKEN-RING, this premise has the form (where we renamed the quantified variables to remove any naming conflicts):

$$\begin{Bmatrix} (\forall a.pend(a)) \wedge \\ (\exists i, i_1 \forall j, j_1.\psi(i, i_1, j, j_1)) \end{Bmatrix} \;\rightarrow\; r' \;\vee\; (\forall c.pend(c)),$$

which is logically equivalent to

$$\forall i, i_1, c\; \exists a, j, j_1. \left(\begin{pmatrix} pend(a) \wedge \\ \psi(i, i_1, j, j_1) \end{pmatrix} \;\rightarrow\; r' \vee pend(c) \right).$$

The **index** variables, which are universally quantified or appear free in the formula above, are $\{i, i_1, c, tloc, 1, N\}$ whose count is 6. It is therefore sufficient to take $N_0 = 6$. Having determined the size of N_0, it is straightforward to compute the premises of $S(N)$ for all $N \leq N_0$ and check that they are valid, using BDD symbolic methods.

We cannot use the same form of auxiliary constructs to automatically verify algorithm BAKERY(N), for every N. Indeed, it is straightforward to see that in order to conclude that $\tau_2[2]$ is helpful, one has to consider helpful assertions of the form $\forall j.\psi(i, j)$. In Sect. 7 we show how to obtain helpful assertions that relate to all processes and how to change the proof rule for such a case. We can still use the simple proof

```
              in  N   : natural where N > 1
              chan : array[1..N] of boolean
                     where chan[i] = (i = 2)
  N         ⎡loop forever do                                  ⎤
 ||  P[i] ::⎢ ⎡0 : if chan[i] then                          ⎤ ⎥
i=1         ⎢ ⎢      (chan[i], chan[i ⊕_N 1]) := (0, 1)     ⎥ ⎥
            ⎢ ⎢    go to {0, 1}                             ⎥ ⎥
            ⎢ ⎢1 : await chan[i]                            ⎥ ⎥
            ⎣ ⎣2 : Critical                                 ⎦ ⎦
```

Fig. 5 Program CHANNEL-RING

rule in order to automatically verify algorithm BAKERY(N). However, this requires the introduction of an auxiliary variable **minid** into the system, which is the index of the process that holds the ticket with minimal value. This is explained in detail in Sect. 5.

We emphasize that the generation of all assertions is *completely invisible*, as is the checking of the premises on the instantiated model. While the user may see the assertions, there is no need for the user to comprehend them. In fact, being generated using BDD techniques, they are often incomprehensible.

4 Cases requiring an existential invariant

In some cases, $\forall$-assertions, i.e., assertions of the form $\forall i.u(i)$, are insufficient for capturing all the relevant features of the constructs φ_A and $pend_A$, and we need to consider assertions of the form $\forall i.u(i) \wedge \exists j.e(j)$. In this section we describe how to obtain constructs that are Boolean combinations of $\forall$-assertions, illustrating the procedure and its applications on the program CHANNEL-RING, presented in Fig. 5.

In this program the location of the token is identified by the index i such that $chan[i] = 1$. Computing the universal invariant according to the previous methods we obtain $\varphi_A : \forall i.(at\_\ell_{0,1} \vee chan[i])$, which is inductive but insufficient to establish the existence of a helpful transition for every pending state.

4.1 Generalizing *project&generalize*

We provide a sketch of the extension that enables computation of a ($\forall \wedge \exists$) construct by obtaining a $\forall i.u(i) \wedge \exists j.e(j)$ invisible invariant. As before, we pick a value N_0, instantiate $S(N_0)$, and use the *project&generalize* procedure to derive an inductive $\forall$-assertion $\varphi : \forall i.u(i)$. As a byproduct of *project&generalize*, we compute $reach_C$ – the set of states reachable in $S(N_0)$. Being inductive and implied by the initial condition, the assertion φ is an overapproximation of $reach_C$. In order to isolate the (anticipated) assertion $e(j)$, we first compute the difference between the concrete reachable set and φ, denoted here by α_1. Obviously, we proceed only if α_1 is nonempty. Then, we *project&generalize* α_1 by replacing index 1 by k (α_2 below). Finally, we negate the result to get the proposed existential invariant (α_3 below).

Algorithm
$\alpha_1 := \bigwedge_{i=1}^{N_0} u(i) \wedge \neg reach_C$
$\alpha_2 := \alpha_1[1 \mapsto k]$
$\alpha_3 := \neg\alpha_2$

We use $\exists k.\alpha_3(k)$ as the candidate for an existential invariant. Below we list the results of these computations for the case that $reach_C$ equals precisely the conjunction $\bigwedge_{i=1}^{N_0} w(i) \wedge \bigvee_{j=1}^{N_0} e(j)$ and the application of *project&generalize* to $reach_C$ yields precisely $u(i) = reach_C[1 \mapsto i] = w(i)$.

Results when $reach_C = \bigwedge_i w(i) \wedge \bigvee_j e(j)$
$\alpha_1 = \bigwedge_i w(i) \wedge \bigwedge_j \neg e(j)$
$\alpha_2 = w(k) \wedge \neg e(k)$
$\alpha_3 = w(k) \rightarrow e(k)$

Note that, while we did not succeed in precisely isolating $e(k)$, we computed instead the implication $w(k) \rightarrow e(k)$. However, the conjunction $\forall i.w(i) \wedge \exists k.(w(k) \rightarrow e(k))$ is logically equivalent to the conjunction $\forall i.w(i) \wedge \exists k.e(k)$.

This technique of obtaining an existential conjunct to an auxiliary assertion can be used for other auxiliary constructs.

4.2 Verifying progress of CHANNEL-RING

Applying the extended *project&generalize* to CHANNEL-RING we obtain, for the set of reachable states, the auxiliary construct

$$\varphi_A : \left[\begin{array}{l} \forall i{\neq}k. \left[\begin{array}{l} (at\_\ell_{0,1} \vee chan[i]) \;\wedge \\ \neg(chan[i] \wedge chan[k]) \end{array}\right] \wedge \\ \exists j.chan[j] \end{array}\right].$$

Using this extended form of an invariant for both φ_A and $pend_A$, we can complete the proof of the program CHANNEL-RING using the methods of Sect. 3.

Applying the method of invisible ranking, with the new addition, to the program CHANNEL-RING and the response property $at\_\ell_1[1] \Rightarrow \Diamond at\_\ell_2[1]$, we obtain, for example, $pend_A : at\_\ell_1[1] \wedge \varphi_A$, and for $i > 1$, $h_m^A[i] : at\_\ell_1[1] \wedge at\_\ell_m[i] \wedge chan[j]$. Thus, premise D3 becomes:

$$\left[\begin{array}{l} at\_\ell_1[1] \\ \quad\wedge \\ \forall i{\neq}k.(at\_\ell_{0,1} \vee chan[i]) \wedge \neg(chan[i] \wedge chan[k]) \\ \quad\wedge \\ \exists j.chan[j] \end{array}\right] \rightarrow$$

$$at\_\ell_1[1] \wedge \exists j.chan[j] ,$$

which is obviously valid and has the $\forall\exists$ form.

```
                in     N     : natural where N > 1
                local  y     : array [1..N] of [0..N]
                               where y = 0
                       minid : natural
                               where minid = 1
                 ⎡ loop forever do                                    ⎤
                 ⎢  ⎡ 0 : NonCritical                              ⎤  ⎥
                 ⎢  ⎢ 1 : y := maximal value to y[i]               ⎥  ⎥
  N              ⎢  ⎢        while preserving order                ⎥  ⎥
  ||  P[i] ::    ⎢  ⎢ 2 : await ∀j : ( y[j] = 0 ∨                  ⎥  ⎥
 i=1             ⎢  ⎢                  y[j] > y[i] )               ⎥  ⎥
                 ⎢  ⎢ 3 : Critical                                 ⎥  ⎥
                 ⎢  ⎣ 4 : y[i] := 0                                ⎦  ⎥
                 ⎢ maintain ∀j : (        y[j] = 0 ∨            )     ⎥
                 ⎣               ( 0 < y[minid] ≤ y[j]           )     ⎦
```

Fig. 6 Program BAKERY with auxiliary variable **minid**

5 The bakery algorithm

As another example of the application of the invisible ranking method, we consider the modified version of the program BAKERY, presented in Fig. 6.

As previously explained, in order to be able to use the rule in its current form, we introduce the variable **minid**. The variable **minid** is expected to hold the index of a process whose y value is minimal among all the positive y-values. The **maintain** construct implies that this variable is updated, if necessary, whenever some y variables change their values. Already in [20] we pointed out that in some cases, it is necessary to add auxiliary variables in order to find inductive assertions with fewer indices. This version of BAKERY illustrates the case that such auxiliary variables may also be needed in the case of the invisible ranking method.

The property we wish to verify for this parameterized system is $at\_\ell_1[z] \implies \Diamond\, at\_\ell_3[z]$, which implies accessibility for an arbitrary process $P[z]$.

Having the auxiliary variable **minid** as part of the system variables, we can proceed with the computation of the auxiliary constructs as explained in Sect. 3: after some simplifications, we can present the automatically derived constructs as detailed in Appendix A.1. Using these derived auxiliary constructs we can verify the validity of the premises of rule DISTRANK over $S(5)$ and conclude that for every value of N the property of accessibility holds.

6 Protocols with $p(i, i+1)$ assertions

In algorithms for ring architectures, the auxiliary assertions for a process often depend, in addition to the process itself, on its immediate neighbors. Assume a ring of size N. For every $j = 1, \ldots, N$, denote $j \oplus 1 = (j \bmod N) + 1$ and $j \ominus 1 = ((j-2) \bmod N) + 1$. Assertions of the type $p(i, i \oplus 1)$ and $p(i, i \ominus 1)$ can be replaced by equivalent $\pm$-less $\forall\exists$-assertions.[2] Unfortunately, this often results in formulae not covered by our small-model theorem. We bypass the problem by establishing a new small-model theorem that allows us to prove validity of $\forall\exists p(i, i \pm 1)$ assertions. The size of the model in the new theorem is larger than the one indicated by the small-model theorem, which is why we refer to it as "modest." We state the modest-model theorem and prove it in Sect. 6.1, describe how to fine-tune the bounds in Sect. 6.2, and demonstrate its application in Sect. 6.3.

6.1 Modest-model theorem

Consider a parameterized BFTS $S(N)$ with no **data** variables or arrays.[3] Let the formula $\varphi\colon \forall\vec{i}\exists\vec{j}.R(\vec{i}, \vec{j})$ be an $\forall\exists$-formula, where $R(\vec{i}, \vec{j})$ is a restricted assertion (augmented by operators $\oplus 1$ and $\ominus 1$) that refers to quantified **index** variables $\vec{i}$ and $\vec{j}$. We show that if there exists some model that does not satisfy this assertion, then there exists a model smaller than a certain bound that does not satisfy it. The proof follows by contracting a model that does not satisfy φ to a smaller model that does not satisfy φ. In order to decrease the size of the model, again, we count the number of existentially quantified variables in the negation of φ. This time, as R may contain $\oplus 1$ and $\ominus 1$, we ensure that in the smaller model each of these variables refers to a different process and, in addition, also pay attention to the way we handle the chain of processes between every two "existentially quantified processes."

Let K be the number of universally quantified **index** variables, index constants (including 1 and N), and free **index** variables appearing in R. Assume there are ℓ **index** $\mapsto$ **bool** arrays in S, and let $L = 2^\ell$, i.e., L is the number of different values that can be assigned to all variables indexed by a single process. Define $N_0 = (K-1)(L^2+1)+K$.

Theorem 2 (Modest-model theorem) *Let φ be an $\forall\exists$-formula as above. Then φ is valid over $S(N)$ for every $N \geq 2$ iff φ is valid over $S(N)$ for every $N \leq N_0$.*

Proof We denote by ψ the formula $\exists\vec{i}\forall\vec{j}.\neg R(\vec{i}, \vec{j})$, which is the negation of φ. Assume ψ is satisfiable in state s of system $S(N_1)$ for $N_1 > N_0$. We show that ψ is also satisfiable in a state s' of a system $S(N)$ for some $N \leq N_0$.

Let $\mathcal{V}_\exists$ be the set of **index** variables that appear existentially quantified in ψ. Let F be the set of **index** constants (including 1 and N) and variables that appear free in ψ. Note that state s provides an interpretation for all the variables in F. Observe that $|\mathcal{V}_\exists \cup F| = K$. Similarly, let $\mathcal{V}_\forall$ be the set of **index** variables that appear universally quantified in ψ, i.e., the $\vec{j}$ variables.

[2] This is, in fact, the way assertions containing $+1$ and $\oplus 1$ are handled in [2]. A simple conversion of this type is given in Example 1.

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

The fact that $\psi : \exists\vec{i}\forall\vec{j}.\neg R(\vec{i}, \vec{j})$ is satisfiable in s means that there exists an assignment α that interprets all variables of $\mathcal{V}_\exists$ by values in the domain $[1 \ldots N_1]$ such that $(s, \alpha) \models \chi$, where $\chi : \forall\vec{j}.\neg R(\vec{i}, \vec{j})$, and (s, α) is the joint interpretation that interprets all system variables according to state s and all $\mathcal{V}_\exists$-variables according to the assignment α.

Let $U = \{1 = u_1 < u_2 < .s < u_k = N_1\}$ be a sorted list of values assigned to the $\mathcal{V}_\exists \cup F$-variables by the joint interpretation (s, α).

Since $N_1 > N_0$, there exist some $i < k$ such that $u_{i+1} - u_i > L^2 + 1$. We construct a state s', in an instantiation $S(N')$, $N' < N_1$, such that $s' \models \psi$. The process is repeated until we obtain an instantiation that satisfies ψ where the us are at most L^2+1 apart from one another.

Since $u_{i+1} - u_i > L^2 + 1$, there exist two pairs of adjacent indices between u_i and u_{i+1} that agree on their local array values, i.e., there exist some m and n such that $u_i < m < n < n + 1 < u_{i+1}$ and, for every Boolean array a: **index** $\mapsto$ **bool**, we have $a[m] = a[n]$ and $a[m + 1] = a[n + 1]$. Intuitively, removing all processes $m + 1, \ldots, n$ does not impact any of the other processes whose indices are in U since the array values of their immediate neighbors remain the same. In particular, since $m+1$ and $n+1$ are identical, processes m and $n+1$ maintain the same neighbors after the removal. Once the processes are removed, the remaining processes are renumbered.

Formally, let $N' = N_1 - (n - m)$, and define the function $g: [1 \,.\,.\, N_1] \to [1 \,.\,.\, N']$ such that $g(i) = i$ for $i \leq m$, and $g(i) = i-(n-m)$ for $i \geq n+1$. It is easy to see that g is injective and onto, hence g^{-1} is well defined. Consider the state s' of system $S(N')$ such that for every array a : **index** $\mapsto$ **bool** we have $s'[a[i]] = s[a[g^{-1}(i)]]$, i.e., the value of a in state s' at index i is the value of a in state s at index $g^{-1}(i)$.

We proceed to show that $(s', \alpha') \models \chi$. To do so, consider an arbitrary assignment β' assigning to each variable $v \in \vec{j}$ a value $\beta'[v] \in [1 \,.\,.\, N']$. We will show that $(s', \alpha', \beta') \models \neg R(\vec{i}, \vec{j})$. If this can be shown for every arbitrary assignment β', it follows that $(s', \alpha') \models \forall\vec{j}.\neg R(\vec{i}, \vec{j})$. That is, $(s', \alpha') \models \chi$.

Consider the assignment β interpreting each $v \in \vec{j}$ as r, $r \in [1 \,.\,.\, N_1]$ iff $\beta'[v] = g(r)$. Since $(s, \alpha) \models \chi$, it follows that $(s, \alpha, \beta) \models \neg R(\vec{i}, \vec{j})$. By induction on the structure of the formula $\neg R(\vec{i}, \vec{j})$, we can show that every subformula $\gamma \in \neg R(\vec{i}, \vec{j})$ evaluates to T under the joint interpretation (s, α, β) iff γ evaluates to T under the interpretation (s', α', β').

We conclude that $(s', \alpha') \models \chi$, which leads to the result that ψ is satisfied in the state s' of system $S(N')$.

Thus s' is obtained from s by leaving the values of the **index** variables in the range $1, \ldots, m$ intact, reducing the **index** variables larger than n by $n-m$ while maintaining the assignments of their **index** $\mapsto$ **bool** variables. Obviously, s' is a state of $S(N_1 - (n - m))$ that satisfies ψ. □

6.2 Calibrating N_0

The bound computed in Theorem 2 may be quite large. In some cases it can be reduced significantly, as we explain below.

General **bool***'s:* if there are **index** $\mapsto$ **bool** arrays for arbitrary (finite) **bool**, L in the bound should be replaced by the product of the sizes of ranges of all **index** $\mapsto$ **bool** variables.

Primed occurrences: when a variable appears both unprimed and primed in $R(.)$, both occurrences add to the count (unless equal). This is in general the case with the transition relation ρ (that appears on the l-h-s of several implicants in our proof rules). While it may seem that each additional variable that can be modified doubles the count, only a single step is to be considered at a time, which is further restricted by *reach* (*reach* appears explicitly in all the implicants; moreover, it can always be added since it is shown to be an invariant). Hence, in practice, the bound can often be reduced so as to be manageable.

Restricted use of $\pm$*:* Assume that for each $\mathcal{V}_\forall$ variable under a $\pm$ operator, all occurrences of the operator are of the same kind (only $\oplus$ or $\ominus$ for each variable). Then, when reducing a large model into a smaller one, instead of finding two processes at the endpoint of a chain that agree on values of both their neighbors, it suffices to find a pair that agrees on one neighbor, which implies a chain of length L. Consequently, in this case the cutoff value is $N_0 = (K - 1)L+K$. Further analysis reveals that if only one operator ($\oplus$ or $\ominus$) is applied to $\mathcal{V}_\exists$ variables, then the bound can be further reduced to $N_0 = (K - 1)(L - 1) + K$.

Restricting to "observable" states: suppose that a process only has a "partial" view of its neighbor, i.e., can access some, but not all, of its neighbor **index** $\mapsto$ **bool** array entries. Then, it suffices to find processes that agree on the part of the state observable by their neighbors, and not the complete state.

Chains of consecutive free variables: if, in addition to 1 and N, there are longer, or other, chains of consecutive values, the bound is reduced accordingly, since there are fewer "gaps" to collapse. For example, when there is a $N-1, N, 1$ combination, the $(K - 1)$ in the bound can be replaced by $(K - 2)$.

6.3 Example: dining philosophers

We demonstrate the use of the modest-model theorem by validating accessibility for a classical solution to the dining philosophers problem, using rule DistRank.

Consider the program Dine which offers a solution to the dining philosophers problem for any N philosophers. The program uses semaphores for forks. In this the program, $N - 1$ philosophers (processes $P[1], \ldots, P[N - 1]$) reach first for their left forks and then for their right forks, while

```
                in      N : natural where N > 1
                local y : array [1..N] of bool where y = 1
 N−1         [loop forever do                    ]        [loop forever do                  ]
  ||  P[i] :: [  [0 : NonCritical          ]     ] || P[N] :: [ [0 : NonCritical        ] ]
 i=1         [  [1 : request y[i]          ]     ]        [ [1 : request y[1]       ] ]
             [  [2 : request y[i+1]        ]     ]        [ [2 : request y[N]       ] ]
             [  [3 : Critical              ]     ]        [ [3 : Critical           ] ]
             [  [4 : release y[i], y[i+1]  ]     ]        [ [4 : release y[1], y[N] ] ]
```

Fig. 7 Program DINE: solution to the dining philosophers problem

$P[N]$ reaches first for its right fork and only then for its left fork. Program DINE is presented in Fig. 7.

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, the transition associated with "**request** x" is compassionate, indicating that if a process is requesting a semaphore that is available infinitely often, it obtains it infinitely many times.

As outlined in Sect. 2.4, we transform the BFTS into a compassion-free BFTS by adding two new Boolean arrays, nvr_1 and nvr_2, each $nvr_\ell[i]$ corresponding to the request of process i at location ℓ. Appendix A.3 describes the BFTS we associate with the program DINE.

The progress property of the original system is

$$(\pi[z] = 1) \Rightarrow \Diamond(\pi[z] = 3)\,,$$

which is 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 the 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-free system, the property we attempt to prove is

$$(\pi[z] = 2) \;\Rightarrow\; \Diamond(\pi[z] = 3 \;\vee\; Err) \qquad (2 \le z \le N-2),$$

where

$$Err = \left[\begin{array}{ll} \bigvee_{i=1}^{N-1}(\pi[i] = 1 \;\wedge\; y[i] \;\wedge\; nvr_1[i]) & \vee \\ \bigvee_{i=2}^{N}(\pi[i-1] = 2 \;\wedge\; y[i] \;\wedge\; nvr_2[i-1]) & \vee \\ (\pi[N] = 1 \;\wedge\; y[1] \;\wedge\; nvr_1[N]) & \vee \\ (\pi[N] = 2 \;\wedge\; y[N] \;\wedge\; nvr_2[N]) & \end{array}\right].$$

6.4 Automatic generation of symbolic assertions

Following the guidelines in Sect. 3, we instantiate the program 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 $N_0 \ge 4$; 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 choose $z = 3$ and attempt to prove $(\pi[3] = 2) \Rightarrow \Diamond(\pi[3] = 3)$. Due to the structure of the the program DINE, process $P[i]$ depends only on its neighbors; thus we expect the auxiliary constructs to include only assertions that refer to two neighboring process at the same time. We choose 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 case, we take $\varphi^A_{=1} = reach_C[1 \mapsto 1, 6 \mapsto N]$ (i.e., project the concrete $reach_C$ on 1 and 6 and generalize to 1 and N); for the second case, we take $\varphi^A_{=N} = reach_C[6 \mapsto N, 5 \mapsto N-1]$ (i.e., project the concrete $reach_C$ on 6 and 5 and generalize to N and $N-1$); and for the third case, we take $\varphi^A_{\neq 1,N} = reach_C[3 \mapsto i, 2 \mapsto i-1]$ (i.e., project the concrete $reach_C$ on 3 and 2 and generalize to i and $i-1$). The abstract pending sets we obtain are in Appendix A.3. We then define:

$$\varphi_A = \varphi^A_{=1} \;\wedge\; \varphi^A_{=N} \;\wedge\; \forall i \not\in \{1, N\} : \varphi^A_{\neq 1,N}(i, i-1)$$

and define $pend_A = \varphi_A \;\wedge\; \neg Err \;\wedge\; \pi[3] = 2$.

For the helpful sets, and the δs, we obtain, as expected, assertions of the type $p(i, i \ominus 1)$. The assertions are described in Appendix A.3.

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.

We now compute the size of the instantiation needed. Premises D1, D3, and D7 relate only to unprimed copies of the variables. Other premises relate to both unprimed and primed copies of the variables. When we use the modest-model theorem "as is," the resulting figures are $L = 40^2 = 1600$ (five possible locations, one fork, two nvr variables, all counted as current and next), $L^2+1 \sim 2.5 \times 10^6$, which results in a bound of about 10^7 processes. To get a reasonable figure, we use the following reductions.

- We syntactically analyze all the resulting assertions and find that only variables in $\mathcal{V}_\exists$ are referenced by both $\oplus 1$ and $\ominus 1$. Variables in $\mathcal{V}_\forall$ are referenced only by $\ominus 1$. Thus we have to search only for two identical processes and not for two pairs of adjacent processes.
- The transition ρ is on the left-hand side of the implication in all the premises that include primed variables (D2,

Table 1 Run time and space results for DINE

Construct	BDD nodes	Premise	Time to validate
φ	1,779	φ (inductiveness)	0.39 s
pend	3,024	D1	< 0.01 s
ρ	10,778	D2	0.42 s
h_ℓ's	< 10	D3	0.01 s
δ_ℓ's	≤ 10	D4	163.74 s
		D5+D6	138.59 s
		D7	0.02 s

D4, D5, and D6). This implies that all possible counterexamples to these premises satisfy ρ. According to ρ, all primed variables for every $j \notin \{i, i\oplus 1\}$ are equal to their unprimed versions. Thus, if we treat $i, i\oplus 1$ as another two-element long chain of universally quantified variables, we do not have to consider different values of the primed variables. It follows that we can use $L = 40$ for our search for duplicate entries.

As a result, the value L above (the maximal length of chain with no "equivalent" processes) is 40. There are three free variables in the system, 1, 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 with all universally quantified variables we use only $i \ominus 1$, the size of the (modest) model we need to take is $40(u+1)+u+4$, where u is the number of universally quantified variables. Since $u \leq 2$ for each of premises D1–D7 (it is 0 for D4, 1 for D1, and 2 for D2, D3, and D5), it is sufficient to choose an instantiation of 128.[4]

In Table 1, we present the number of BDD nodes computed for each auxiliary construct and the time it took to validate the inductiveness of φ and each of the premises (D1–D7) on the largest instantiation (128 philosophers). Checking all instantiations (2–128) took less than 8 h.

7 Imposing ordering on transitions

Sections 3 and 4 dealt with helpful transitions $h_k[i]$ (and ranking functions) that depended only on the single index i. In the previous section we showed how to extend this approach to the case in which $h_k[i]$ may also depend on indices $i \ominus 1$ and $i \oplus 1$. In this section we study helpful assertions that depend on all $j \neq i$. Such multiple-index 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 that 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 $\exists\forall$-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] By modifying *project&generalize* to include only part of the variables of a process and not all variables, this can be further reduced to 83 processes.

7.1 Preordering transitions

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

Consider a BFTS S with set of transitions $\mathcal{T}(N) = [0..m] \times N$ (as in Sect. 3.1). For every state in $S(N)$, define a preorder $\preceq$ over $\mathcal{T}$. From the totality of $\preceq$, every $S(N)$-state has some $\tau_\ell[i] \in \mathcal{T}$ that is minimal according to $\preceq$. We replace premise D4 in DISTRANK with a premise stating 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–R7. Thus, PRERANK is exactly like DISTRANK, with the addition of a preorder $\preceq : \Sigma \rightarrow 2^{\mathcal{T}\times\mathcal{T}}$, premises ascertaining that the relation $\preceq$ is a preorder (R8–R10), and replacement of D4 by R4 (Fig. 8).

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

$$\Phi(\tau_1, \tau_2) : \begin{bmatrix} \mathbf{A}((\neg h_{\tau_2} \wedge pend)\ \mathcal{W}\ h_{\tau_1}) \vee \\ \neg\mathbf{A}((\neg h_{\tau_1} \wedge pend)\ \mathcal{W}\ h_{\tau_2}) \end{bmatrix}, \qquad (2)$$

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

The intuition behind the first disjunct is that for a state s, transition τ_1 is "helpful earlier" than τ_2 if none of the pathes departing from s reaches h_{τ_2} before it reaches h_{τ_1}. The role of the second disjunct is to guarantee the totality of $\preceq$, so that when τ_1 becomes helpful earlier

For	a parameterized system with a transition $\mathcal{T} = \mathcal{T}(N)$ set of states $\Sigma(N)$, just transitions $\mathcal{J} \subseteq \mathcal{T}(N)$, invariant assertion φ, assertions $q, r, pend$ and $\{h_\tau \mid \tau \in \mathcal{J}\}$, ranking functions $\{\delta_\tau : \Sigma \rightarrow \{0,1\} \mid \tau \in \mathcal{J}\}$, and a pre-order $\preceq : \Sigma \mapsto 2^{\mathcal{T}\times\mathcal{T}}$		
R1.	$q \wedge \varphi$	$\rightarrow$	$r \vee pend$
R2.	$pend \wedge \rho$	$\rightarrow$	$r' \vee pend'$
R3.	$pend \wedge \rho$	$\rightarrow$	$r' \vee \bigwedge_{\tau\in\mathcal{J}} \delta_\tau \geq \delta'_\tau$
For every $\tau \in \mathcal{J}$			
R4.	$pend \wedge \left(\bigwedge_{\tau_1\in\mathcal{J}} \tau \preceq \tau_1\right)$	$\rightarrow$	h_τ
R5.	$h_\tau \wedge \rho$	$\rightarrow$	$r' \vee h'_\tau \vee \delta_\tau > \delta'_\tau$
R6.	$h_\tau \wedge \tau$	$\rightarrow$	$r' \vee \delta_\tau > \delta'_\tau$
R7.	h_τ	$\rightarrow$	$En(\tau)$
R8.	$pend$	$\rightarrow$	$\tau \preceq \tau$
For every $\tau_1, \tau_2 \in \mathcal{J}$			
R9.	$pend \wedge \tau \preceq \tau_1 \wedge \tau_1 \preceq \tau_2$	$\rightarrow$	$\tau \preceq \tau_2$
R10.	$pend$	$\rightarrow$	$\tau \preceq \tau_1 \vee \tau_1 \preceq \tau$

$$q \Rightarrow \Diamond r$$

Fig. 8 The liveness rule PRERANK

than τ_2 in some computations, and τ_2 precedes τ_1 in others, we obtain both $\tau_1 \preceq \tau_2$ and $\tau_2 \preceq \tau_1$. To abstract a formula $\mathbf{A}(\varphi(h^C_k[i])\,\mathcal{W}\,\psi(h^C_m[j]))$, we compute the assertion $\mathbf{A}(\varphi(h^C_k[2])\,\mathcal{W}\,\psi(h^C_m[3]))$ over $S(N_0)$ (2 and 3 being chosen arbitrarily to represent two generic indices), and then generalize 2 to i and 3 to j. To abstract the negation of such a formula, we first abstract the formula and then negate the result. Therefore, to abstract Eq. (2), we abstract each $\mathbf{A}\,\mathcal{W}$-formula separately and then take the disjunction of the first abstract assertion with the negation of the second abstract assertion.

7.2 Case study: BAKERY

Consider the the program BAKERY of Example 2 (Fig. 3). Suppose we want to verify $(\pi[z] = 1) \Rightarrow \Diamond(\pi[z] = 3)$. We instantiate the system to $N_0 = 3$ and obtain the auxiliary assertions φ and *pend* and the hs and δs. After applying *project&generalize*, we obtain for $h_\ell[i]$ two types of assertions. One is for the case that $i = z$, and then, as expected, $h_2[z]$ is the most interesting one, having an $\forall$-construct claiming that z's ticket is the minimal among ticket holders. The other case is for $j \neq z$, and there we have a similar $\forall$-construct (for j's ticket minimality) for $\ell = 2, 3, 4$. For the preorder, one must consider $\tau_{\ell_1}[i] \preceq \tau_{\ell_2}[j]$ for every $\ell_1, \ell_2 = 1, \ldots, 4$ and $i = z \neq j, i = j \neq z, i, j \neq z$ for $(\ell_1, i) \neq (\ell_2, j)$. The results for $\tau_{\ell_1}[i] \preceq \tau_{\ell_2}[j]$ for $i \neq z$ that are not trivially T are described in Appendix A.1

Using the above preorder, we succeeded in validating premises R1–R9 of PRERANK, thus establishing the liveness property of the program BAKERY.

8 Multiple preorder relations

In the previous section we described how to compute the preorder relation. Eq. (2) is one alternative for computing the preorder. We can view rule DISTRANK as a special case of rule PRERANK, with a trivial preorder defined by $s \models \tau_1 \preceq \tau_2$ if $s \models \Psi(\tau_1, \tau_2)$, where

$$\Psi(\tau_1, \tau_2) : h_{\tau_1} \;\vee\; \neg h_{\tau_2}\,. \tag{3}$$

Obviously, other definitions are also possible. In fact, by allowing different schemes of computing preorder on different states, the rule PRERANK can be applied to a wider range of protocols. In this section we demonstrate this idea on a version of SZYMANSKI's mutual exclusion protocol described in Fig. 9.

The progress property we would ideally like to verify is $(\pi[z] = 1 \Rightarrow \Diamond(\pi[z] = 7)$. This property, however, is beyond the scope of the methods and rules described here since it requires some just transition to be helpful twice. It is not difficult, but rather tedious, to extend our technique for generating ranking so as to deal with cases where transitions may be helpful up to k times, for any bounded k. We bypass

$$
\begin{array}{l}
\quad\quad\quad \textbf{in } N : \textbf{natural where } N > 1 \\
\underset{i=1}{\overset{N}{\Big\|}} P[i] :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
0 : \textbf{NonCritical} \\
1 : \textbf{await } \forall j.at\_\ell_{0,1,2,4}[j] \\
2 : \textbf{skip} \\
3 : \textbf{If } \exists j : at\_\ell_{1,2}[j] \\
\quad\quad \textbf{then go-to } l_4 \\
\quad\quad \textbf{else go-to } l_5 \\
4 : \textbf{await } \exists j : at\_\ell_{5,6,7}[j] \\
5 : \textbf{await } \forall j : \neg at\_\ell_{3,4}[j] \\
6 : \textbf{await } \forall j : j < i : at\_\ell_{0,1,2}[j] \\
7 : \textbf{Critical}
\end{array}\right]
\end{array}\right]
\end{array}
$$

Fig. 9 Program SZYMANSKI

this difficulty here by restricting ourselves to a "smaller" progress property to which the proof applies, namely, to the progress property

$$(\pi[z] = 1 \;\wedge\; \forall i : \pi[i] \leq 4) \Rightarrow \Diamond(\pi[z] = 7)\,. \tag{4}$$

An inspection of the protocol reveals that $\tau_6[i]$ is the only transition whose enabling condition is of the form $\forall j.p(i, j)$, which is an obvious candidate for preordering of the type we used in Sect. 7. The other transitions all have enabling conditions of the form $p(i) \;\wedge\; \forall j.q(j)$ (or simpler) that can be easily handled by the trivial preorder that we implicitly use when applying DISTRANK. Consequently, we partition the concrete pending states into $pend_1 = \exists i. \bigvee_{\ell \notin \{0,6\}} En(\tau_\ell[i])$ and $pend_2 = pend \;\wedge\; \neg pend_1$. The (concrete) preorder is now defined for $pend_1$-states by

$$\tau_\ell[i] \preceq \tau_{\ell'}[i'] = \begin{cases} \Psi(\tau_\ell[i], \tau_{\ell'}[i']) & \text{if } \ell, \ell' \neq 6 \\ \text{T} & \text{if } \ell' = 6 \\ \text{F} & \text{otherwise} \end{cases}$$

and for $pend_2$-states by

$$\tau_\ell[i] \preceq \tau_{\ell'}[i'] = \begin{cases} \Phi(\tau_\ell[i], \tau_{\ell'}[i']) & \text{if } \ell = \ell' = 6 \\ \text{T} & \text{if } \ell' \neq 6 \\ \text{F} & \text{otherwise}\,, \end{cases}$$

where Ψ is defined in Eq. (3) and Φ is defined in Eq. (2).

These definitions allow us to use *project&generalize* on the concrete preorder (as described in Sect. 7) and successfully prove Eq. (4) for the program SZYMANSKI.

9 Discussion

We have presented a method for automatically verifying liveness properties of parameterized systems. The method is based on automatic computation of the assertions needed by a deductive rule according to the analysis of a small instance of the problem. Then, using a small-model theorem, the verification conditions of the deductive rule are discharged using BDD techniques on a (sometimes not so) small instance of the parameterized system. Being able to discharge the verification conditions on a finite model has the additional advantage that the user never gets to see the assertions, which is why we termed the method "invisible constructs."

Deductive proofs for liveness require the identification of helpful transitions and, in addition, a ranking function that measures the progress toward the goal. The deductive proof rule we are using is similar. To facilitate the generation of the ranking function, we partition it and include one ranking function per helpful transition. The range of these ranking functions is usually {0, 1}. There are cases where a single transition must be helpful more than once before other helpful transitions can be taken. In such cases the restricted range of {0, 1} (i.e., the helpful transition was or was not taken) is not sufficient. We would have to consider ranking functions with a larger-range set. In general, we believe that it is best to use the smallest range possible for the ranking functions. The main burden in using our method is in devising the method to compute the explicit ranking functions and in deciding how to generalize these explicit assertions. Thus having a larger range for the ranking functions would make the method harder to use and is inadvisable.

A key feature of our method is generalizing a concrete set of states into a universal assertion. In the paper, we explain briefly how to obtain existential auxiliary assertions in the case that our approximation of the concrete set is too abstract. This process can be iterated as follows. When the assertion we have is too abstract, we can add an existential conjunct that tightens the abstraction. When the assertion does not capture the entire concrete set, we can generalize the difference and add a universal disjunct. Thus, we can get assertions of the general form $((\cdots(\forall\wedge\exists)\vee\forall)\wedge\exists)\cdots)$. Note that the quantifiers are not nested, hence using these assertions we can still employ the small-model theorem. We have studied examples where this iterative computation of the generalization is necessary in order to get assertions that fulfill the requirements of the deductive rule. This, however, is beyond the scope of this paper.

Finally, we recall that the problem of uniform verification of parameterized systems is undecidable; thus we cannot hope that our method, or other methods, will always succeed. When the method does not work immediately, it may help to obtain tighter abstractions. It may help to increase the size of the small model on which we compute the concrete assertions. The cornerstone of our method is a deductive rule, so manual intervention of the user may help push a proof forward. Sometimes, unfortunately, it would be best to try something completely different.

Acknowledgements This research was supported in part by NSF Grant CCR-0205571, ONR Grant N000140310916, the John von Neumann Minerva Center for Verification of Reactive Systems, the European Community IST project "Advance," and the Israel Science Foundation Grant 106/02-1.

Appendix 1

A BFTS and auxiliary constructs

A.1 Program BAKERY

BFTS*:* See Fig. A.1.

Auxiliary constructs: The auxiliary constructs for the program BAKERYwith **minid** are:

$$\varphi_A : \forall i. \begin{bmatrix} (at\_\ell_{0,1}[i] \leftrightarrow y[i]=0)\ \wedge \\ (at\_\ell_{3,4}[i] \rightarrow \textbf{minid}=i) \end{bmatrix} \wedge$$
$$\forall i{\neq}j. \begin{bmatrix} (\textbf{minid} \neq i \vee\ y[j] > y[i] \wedge y[i] \neq 0 \vee \\ y[j]=0) \wedge (y[i]=y[j] \rightarrow\ y[i]=0) \end{bmatrix}$$

$$pend_A : \quad \varphi_A\ \wedge\ at\_\ell_{1,2}[z]$$

$$(h_\ell[j])_A : \begin{pmatrix} \ell & \text{For } j=z & \text{For } j\neq z \\ \hline 1 & at\_\ell_1[z] & 0 \\ 2 & at\_\ell_2[z]\ \wedge & at\_\ell_2[z]\ \wedge\ at\_\ell_2[j] \\ & \textbf{minid}=z & \wedge\ \textbf{minid}=j \\ 3 & 0 & at\_\ell_2[z]\ \wedge\ at\_\ell_3[j] \\ 4 & 0 & at\_\ell_2[z]\ \wedge\ at\_\ell_4[j] \end{pmatrix}$$

$$V : \begin{cases} y : \textbf{array}[1..N]\ \textbf{of}\ [0..N] \\ \pi : \textbf{array}[1..N]\ \textbf{of}\ [0..4] \end{cases}$$
$$\Theta : \forall i : \pi[i]=0\ \wedge\ y[i]=0$$
$$\mathcal{T} : \begin{cases} \tau_0(i) : \forall j \neq i\ . & \pi[i]=0 \wedge \pi'[i] \in \{0,1\} \wedge \\ & pres(\pi[j], y[i], y[j]) \\ \tau_1(i) : \forall j,k \neq i\ . & \pi[i]=1 \wedge \pi'[i]=2 \wedge y'[j] < y'[i] \\ & \wedge \begin{pmatrix} y[j]=0 \leftrightarrow y'[j]=0 \wedge \\ y[j] < y[k] \leftrightarrow y'[j] < y'[k] \end{pmatrix} \\ & \wedge pres(\pi[j]) \\ \tau_2(i) : \forall j \neq i\ . & \pi[i]=2 \wedge (y[j]=0 \vee y[j] > y[i]) \\ & \wedge \pi'[i]=3 \wedge pres(\pi[j], y[i], y[j]) \\ \tau_3(i) : \forall j \neq i\ . & \pi[i]=3 \wedge \pi'[i]=4 \wedge \\ & pres(\pi[j], y[i], y[j]) \\ \tau_4(i) : \forall j \neq i\ . & \pi[i]=4 \wedge \pi'[i]=0 \wedge y'[i]=0 \wedge \\ & pres(\pi[j], y[j]) \\ \tau_{id} :\quad \forall j\ . & pres(\pi[j], y[j]) \end{cases}$$

$$\mathcal{J} : \{\tau_1(i), \tau_2(i), \tau_3(i), \tau_4(i), \tau_{id}\ \mid\ i \in [1..N]\}$$

$$\mathcal{C} :\ \emptyset$$

Fig. A.1 BFTS for program BAKERY

$$(\delta_\ell[j])_A : \left(\begin{array}{c|l|l} \ell & \text{For } j = z & \text{For } j \neq z \\ \hline 1 & at\_\ell_1[z] & 0 \\ 2 & at\_\ell_{1,2}[z] & \zeta(z, j, \{2\}) \\ 3 & 0 & \zeta(z, j, \{2,3\}) \\ 4 & 0 & \zeta(z, j, \{2,3,4\}) \end{array}\right)$$

where $\zeta(z, j, A) = at\_\ell_1[z] \vee at\_\ell_2[z] \wedge y[z] > y[j] \wedge at\_\ell_A[j]$.

Preorder relation for non-minid-version: Let $\alpha\colon \pi[j] = 2 \rightarrow y[z] < y[j]$, $\beta\colon \pi[i] = 2 \wedge y[i] < y[j]$, and $\gamma(L)\colon \pi[j] \in L \rightarrow y[z] < y[j]$. The preorder is described in Fig. A.3.

$$V : \left\{\begin{array}{lll} y, nvr_1, nvr_2 : & \textbf{array } [1..N] & \textbf{of bool} \\ \pi : & \textbf{array } [1..N] & \textbf{of } [0..4] \end{array}\right\}$$

$$\Theta : \forall i.\ (\pi[i] = 0 \ \wedge\ y[i]$$

$$\mathcal{T} : \left\{\begin{array}{l}
\tau_0(i) : \quad \forall j \neq i : \\
\quad \left(\begin{array}{ll} \pi[i] = 0 \wedge \pi'[i] \in \{0,1\} & \wedge \\ pres(y[i], nvr_1[i], nvr_2[i]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_1(i) : \quad \forall j \notin \{i, i \oplus 1\} : \\
\quad \left(\begin{array}{ll} \pi[i] = 1 \wedge \pi'[i] = 2 \wedge pres(nvr_1[i], nvr_2[i]) & \wedge \\ (i < N \rightarrow (y[i] \wedge \neg y'[i] \wedge pres(y[i+1]))) & \wedge \\ (i = N \rightarrow (y[1] \wedge \neg y'[1] \wedge pres(y[N]))) & \wedge \\ pres(\pi[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j])) & \end{array}\right) \\
\qquad \vee \\
\quad \left(\begin{array}{ll} \neg nvr_1[i] \wedge nvr_1'[i] \wedge pres(\pi[i], y[i], nvr_2[i]) & \wedge \\ pres(\pi[i \oplus 1], y[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_2(i) : \quad \forall j \notin \{i, i \oplus 1\} : \\
\quad \left(\begin{array}{ll} \pi[i] = 2 \wedge \pi'[i] = 3 \wedge pres(nvr_1[i], nvr_2[i]) & \wedge \\ (i < N \rightarrow (y[i+1] \wedge \neg y'[i+1] \wedge pres(y[i]))) & \wedge \\ (i = N \rightarrow (y[N] \wedge \neg y'[N] \wedge pres(y[1])) & \wedge \\ pres(\pi[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j])) & \end{array}\right) \\
\qquad \vee \\
\quad \left(\begin{array}{ll} \neg nvr_2[i] \wedge nvr_2'[i] \wedge pres(\pi[i], y[i], nvr_1[i]) & \wedge \\ pres(\pi[i \oplus 1], y[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_3(i) : \quad \forall j \neq i : \\
\quad \left(\begin{array}{ll} \pi[i] = 3 \wedge \pi'[i] = 4 & \wedge \\ pres(y[i], nvr_1[i], nvr_2[i]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_4(i) : \quad \forall j \notin \{i, i \oplus 1\} : \\
\quad \left(\begin{array}{ll} \pi[i] = 4 \wedge \pi'[i] = 0 \wedge pres(nvr_1[i], nvr_2[i]) & \wedge \\ y'[i] \wedge y'[i \oplus 1] & \wedge \\ pres(\pi[i \oplus 1], y[i \oplus 1], nvr_1[i \oplus 1], nvr_2[i \oplus 1]) & \wedge \\ pres(\pi[j], y[j], nvr_1[j], nvr_2[j]) & \end{array}\right) \\
\tau_{id} : \quad \forall j : pres(\pi[j], y[j], nvr_1[j], nvr_2[j])
\end{array}\right.$$

$$\mathcal{J} : \{\tau_1(i), \tau_2(i), \tau_3(i), \tau_4(i), \tau_{id} \ \mid\ i \in [1..N]\}$$

Fig. A.2 BFTS for program DINE

A.2 Program TOKEN-RING

Symbolic assertions:

	$k = 0$	$k = 1$	$k = 2$
$h_k^A[1]$	0	$at\_\ell_1[1] \ \wedge\ tloc = 1$	0
$h_k^A[i], i > 1$	$at\_\ell_1[1] \ \wedge\ at\_\ell_k[i] \ \wedge\ tloc = i$		

Symbolic ranking:

$$\delta_0^A[1] : 0$$
$$\delta_1^A[1] : at\_\ell_1[1]$$
$$\delta_2^A[1] : 0$$

$$\left.\begin{array}{l}
\delta_0^A[i] : at_\ell_1[1] \wedge \\
\qquad (1 < tloc < i \ \wedge\ at_\ell_{0,1}[i] \ \vee\ tloc = i) \\
\delta_1^A[i] : at_\ell_1[1] \wedge \\
\qquad \left[\begin{array}{c} 1 < tloc < i \ \wedge\ at_\ell_{0,1}[i] \vee \\ tloc = i \ \wedge\ at_\ell_1[i]) \end{array}\right] \\
\delta_2^A[i] : at_\ell_1[1] \wedge \\
\qquad \left[\begin{array}{c} 1 < tloc < i \ \wedge\ at_\ell_{0,1}[i] \vee \\ tloc = i \ \wedge\ at_\ell_{1,2}[i] \end{array}\right]
\end{array}\right\} \text{ for } i > 1$$

A.3 Program DINE

BFTS*:* See Fig. A.2

Abstract pending sets:

$$\varphi_{=1}^A = \left(\begin{array}{l} (y[N] \rightarrow \pi[N] < 2) \\ \wedge\ (\pi[1] > 1 \rightarrow \pi[N] < 2) \\ \wedge\ \left(y[1] \leftrightarrow \left(\begin{array}{l} \pi[N] < 2 \ \wedge \\ \pi[1] < 2 \end{array}\right)\right) \end{array}\right)$$

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

Fig. A.3 Preorder for Program BAKERY

$$\varphi^{A}_{\neq 1,N}(i,i{-}1) = \left(\begin{array}{l} \quad (y[i{-}1] \rightarrow \pi[i{-}1] < 2) \\ \wedge\ (\pi[i{-}1] > 2 \rightarrow \pi[i] < 2) \\ \wedge\ \left(y[i] \leftrightarrow \left(\begin{array}{l}\pi[i{-}1] < 3\ \wedge \\ \pi[i] < 2\end{array}\right)\right)\end{array}\right)$$

$$\varphi^{A}_{=N} = \left(\begin{array}{l} \quad y[N-1] \rightarrow \pi[N-1] < 2 \\ \wedge\ \pi[N-1] > 2 \rightarrow \pi[N] < 3 \\ \wedge\ \left(y[N] \leftrightarrow \left(\begin{array}{l}\pi[N-1] < 3\ \wedge \\ \pi[N] < 3\end{array}\right)\right)\end{array}\right)$$

Symbolic ranking and helpful sets: For every $j = z + 1, \ldots, N{-}1$:

$h^A_1[j]$: F
$h^A_2[j] : \pi[j{-}1] = 2\ \wedge\ nvr_2[j{-}1] \wedge$
$\quad \pi[j] = 2\ \wedge\ \neg nvr_2[j]$
$h^A_3[j] : \pi[j{-}1] = 2\ \wedge\ nvr_2[j{-}1] \wedge$
$\quad \pi[j] = 3\ \wedge\ \neg y[i]$
$h^A_4[j] : \pi[j{-}1] = 2\ \wedge\ nvr_2[j{-}1] \wedge$
$\quad \pi[j] = 4\ \wedge\ \neg y[i]$
$\delta^A_1[j]$: T
$\delta^A_2[j] : \neg nvr_2[j] \wedge$
$\quad (\pi[j{-}1] = 2\ \wedge\ nvr_2[j{-}1] \rightarrow \pi[j] < 3)$
$\delta^A_3[j] : \neg nvr_2[j] \wedge$
$\quad (\pi[j{-}1] = 2\ \wedge\ nvr_2[j{-}1] \rightarrow \pi[j] < 4)$
$\delta^A_4[j]$: T

References

1. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Proc. Lett. **22**(6), 307–309 (1986)
2. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of the 13th International Conference on Computer-Aided Verification (CAV'01). Lecture Notes in Computer Science, vol. 2102, pp. 221–234. Springer, Berlin Heidelberg New York (2001)
3. Bjørner, N., Browne, I.A., Chang, E., Colóon, M., Kapur, A., Manna, Z., Sipma, H.B., Uribe, T.E.: STeP: The Stanford temporal prover, user's manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, Stanford, CA (1995)
4. Clarke, E.M., Grumberg, O., Jha, S.: Verifying parametrized networks using abstraction and regular languages. In: Proceedings of the 6th International Conference on Concurrency Theory (CONCUR92), Philadelphia. Lecture Notes in Computer Science, vol. 962, pp. 395–407. Springer, Berlin Heidelberg New York (1995)
5. Choueka, Y.: Theories of automata on ω-tapes: a simplified approach. J. Comput. Syst. Sci. **8**, 117–141 (1974)
6. Colon, M., Sipma, H.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) Proceedings of the 14th International Conference on Computer-Aided Verification (CAV'02). Lecture Notes in Computer Science, vol. 2404, pp. 442–454. Springer, Berlin Heidelberg New York (2002)
7. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: Proceedings of the 17th International Conference on Automated Deduction (CADE-17), pp. 236–255 (2000)
8. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: Procedings of the 22nd ACM Conference on Principles of Programming Languages (POPL'95), San Francisco (1995)
9. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with incomprehensible ranking. In: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04). Lecture Notes in Computer Science, vol. 2988, pp. 482–496. Springer, Berlin Heidelberg New York (2004)
10. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with invisible ranking. In: Proceedings of the 5th conference on Verification, Model Checking, and Abstract Interpretation, Venice, Italy. Lecture Notes in Computer Science, vol. 2937, pp. 223–238. Springer, Berlin Heidelberg New York (2004)
11. Gyuris, V., Sistla, A.P.: On-the-fly model checking under fairness that exploits symmetry. In: Grumberg, O. (ed.) Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97). Lecture Notes in Computer Science, vol. 1254. Springer, Berlin Heidelberg New York (1997)
12. Gribomont, E.P., Zenner, G.: Automated verification of szymanski's algorithm. In: Steffen, B. (ed.) Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98). Lecture Notes in Computer Science, vol. 1384, pp. 424–438. Springer, Berlin Heidelberg New York (1998)
13. Jonsson, B., Nilsson, M.: Transitive closures of regular relations for verifying infinite-state systems. In: Graf, S., Schwartzbach, M. (eds.) Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00). Lecture Notes in Computer Science, vol. 1785. Springer, Berlin Heidelberg New York (2000)
14. Kesten, Y., Piterman, N., Pnueli, A.: Bridging the gap between fair simulation and trace inclusion. In: Hunt Jr, W., Somenzi, F. (eds.) Proceedings of the 15th International Conference on Computer-Aided Verification (CAV'03), pp. 381–392. Boulder, CO (2003)
15. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parameterized linear networks of processes. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris (1997)
16. McMillan, K.L.: Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Hu, A.J., Vardi, M.Y., Hu, A.J., Vardi, M.Y. (eds.) Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98). Lecture Notes in Computer Science, vol. 1427, pp. 110–121. Springer, Berlin Heidelberg New York (1998)
17. Manna, Z., Pnueli, A.: Completing the temporal picture. Theor. Comput. Sci. **83**(1), 97–130 (1991)
18. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, Berlin Heidelberg New York (1995)
19. Owre, S., Shankar, N., Rushby, J.M.: User guide for the PVS specification and verification system (draft). Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA (1993)
20. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01). Lecture Notes in Computer Science, vol. 2031, pp. 82–97. Springer, Berlin Heidelberg New York (2001)
21. Pnueli, A., Xu, J., Zuck, L.: Liveness with (0, 1, ∞)-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02). Lecture Notes in Computer Science, vol. 2404, pp. 107–122. Springer, Berlin Heidelberg New York (2002)

22. Shahar, E.: The TLV manual (2000). `http://www.wisdom.weizmann.ac.il/~verify/tlv`
23. Szymanski, B.K.: A simple solution to Lamport's concurrent programming problem with linear wait. In: Proceedings of the International Conference on Supercomputing Systems, pp. 621–626, St. Malo, France (1988)
24. Vardi, M.Y.: Verification of concurrent programs – the automata-theoretic framework. Ann. Pure Appl. Logic **51**, 79–98 (1991)
25. Zuck, L., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems. Comput. Lang. Syst. Struct. **30** (3/4), 139–169 (2004)