Liveness by Invisible Invariants[⋆]

Yi Fang[1], Kenneth L. McMillan[2], Amir Pnueli[3], and Lenore D. Zuck[4]

[1] Microsoft, Redmond, Washington
yfang@microsoft.com
[2] Cadence Design Systems, Berkeley, California
mcmillan@cadence.com
[3] New York University, New York, New York
amir@cs.nyu.edu
[4] University of Illinois at Chicago
lenore@cs.uic.edu

Abstract. The method of Invisible Invariants was developed in order to verify safety properties of parametrized systems in a fully automatic manner. In this paper, we apply the method of invisible invariant to "bounded response" properties, i.e., liveness properties of the type $p \implies \diamondsuit q$ that are bounded – once a p-state is reached, it takes a bounded number of rounds (where a round is a sequence of steps in which each process has been given a chance to proceed) to reach a q-state – thus, they are essentially safety properties.

With a "liveness monitor" that observes certain behavior of a system, establishing "bounded response" properties over the system is reduced to the verification of invariant properties.

It is often the case that the inductive invariants for systems with "liveness monitors" contain assertions of a certain form that the original method of invisible invariant is not able to generate, nor to check inductiveness. To accommodate invariants of such forms, we extend the techniques used for invariant generation, as well as the small model theorem for validity check.

1 Introduction

Uniform verification of parameterized systems is one of the most challenging problems in verification. Given a parameterized system $S(N) : P[1] \parallel \cdots \parallel P[N]$ 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, requiring ingenuity, expertise, and a good understanding of the behavior of the

[⋆] This research was supported in part by NSF grant CCR-0205571 and ONR grant N00014-99-1-0131.

program and the techniques for formalizing these insights. The second task is often performed using theorem provers such as PVS [OSR93] or STeP [BBC+95], 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 reason why deductive verification is not used more widely.

A representative case is the verification of invariance properties using the proof rule INV of [MP95]: in order to prove that assertion r is an invariant of program P, the rule requires coming up with an auxiliary assertion φ that is *inductive* (i.e. is implied by the initial condition and is preserved under every computation step) and that strengthens (implies) r. The rule is described in Fig. 1, where Θ is the initial condition of program P.

$$\begin{array}{l}
\text{I1. } \Theta \rightarrow \varphi \\
\text{I2. } \varphi \wedge \rho \rightarrow \varphi' \\
\text{I3. } \varphi \rightarrow r \\
\hline
\quad \square\, r
\end{array}$$

Fig. 1. The proof rule INV

In [PRZ01, APR+01], we introduced the method of *invisible invariants*, that 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 $\forall i : [1..N].q(i)$ or, more generally, $\forall i \neq j.q(i,j)$. We construct an instance of the parameterized system taking a fixed value N_0 for the parameter N. For the finite-state instantiation $S(N_0)$, we compute, using BDDs, 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 the assertion φ, we still have to check the validity of the premises of the proof rule we wish to employ. Under the assumption that our assertional language is restricted to the predicates of equality and inequality between bounded-range integer variables (which is adequate for many of the parameterized systems we considered), we proved a *small-model* theorem, according to which, for a certain type of assertions, there exists a (small) bound N_0 such that such an assertion is valid for every N iff it is valid for all $N \leq N_0$. This enables using BDD-techniques to check the validity of such an assertion. The 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 non-intuitive, 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 [PRZ01, APR+01], 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 apply the method of invisible invariants to the second-most important properties of concurrent systems, namely, "response" properties. Response properties are properties of the type $q \Rightarrow \Diamond\, r$ (i.e., $\Box(q \rightarrow \Diamond\, r)$), and they are the most common liveness properties. The most frequent form of response properties of parameterized systems is $\forall\vec{i}.(q(\vec{i}) \Rightarrow \Diamond\, r(\vec{i}))$, where $q(\vec{i})$ and $r(\vec{i})$ are quantifier-free. Since the systems we are dealing with are finite-state, that is, for every value N, $S[N]$ is finite-state, every valid response property is *bounded* by some of the parameters of the system.

The ability to prove only *bounded* progress may seem like a limitation. However, note that we are dealing here only with finite-state systems. That is, for every N, $S[N]$ is finite-state. If a finite-state system satisfies a progress property, then it satisfies a corresponding bounded progress property, for a suitable bound. In the case of a simple transition system without fairness assumptions, the bound can be given in terms of the maximum number of transitions required to satisfy the progress condition. In the case of "justice" assumptions (of the form $\Box\Diamond\, p(i)$), the bound can be given in terms of the number of "rounds" in which every justice condition $p(i)$ is satisfied. Of course, the bound may be a rapidly increasing function of N. The main limitation of the present method is that it handles only the case when the bound increases linearly with N. We will show, however, that this condition is satisfied for several typical examples of parameterized protocols.

Roughly speaking, the bound determines "how fast" progress is achieved. In the case that the bound depends on the transition relation, the proof of progress can be replaced by a proof of a simpler safety property, *bounded progress*, that establishes that once $q(\vec{i})$ holds and enough transitions (where "enough" is determined by the bound) occur, $r(\vec{i})$ obtains. Since we are dealing with parameterized systems, the bound depends on the parameter N. For simplicity of notation, assume that $\vec{(i)}$ is of size 1, i.e., the progress property at hand is $\forall i.q(i) \Rightarrow \Diamond\, r(i)$. Let z be some process. It suffices to show that $q(z) \Rightarrow \Diamond\, r(z)$. Let K range over *rounds*, in each of which each process is to take at least one step. Since we want to rule out stuttering rounds, we allow a process to take a stuttering step only if it has no non-stuttering step available to it. We show how to obtain K and how to automatically construct a non-interfering "liveness monitor" such that, once (synchronously) composed with the original system, the method of invisible invariants can be used to show a (K-dependent) bounded progress (safety) property that establishes the liveness property of the parameterized system.

Often it is the case that the safety property obtained is too large for the model checker. Our experience has shown that splitting such individual proofs into two parts, livelock freedom and bounded overtaking, often helps to avoid those two obstacles. "Livelock freedom" establishes $\exists i.q(i) \Rightarrow \exists i. \diamond r(i)$, and "bounded overtaking" establishes that once $\exists i.q(i)$, there is a bound b such that for every $j \neq i$, the (regular) sequence $q(j)\Sigma^* r(j)\Sigma^* \neg r(j)$ can occur at most b times before $r(i)$ becomes true. Bounded overtaking is a safety property, and, as we show, can be proved using the method of invisible invariants. Put together, livelock freedom and bounded overtaking establish individual liveness.

It is often the case that the invisible invariants we obtain contain $\exists\forall$-formulae, which are not covered under the small model theorem previously proven. We extend the small model theorem to deal with invariants that have $\exists\forall$-subformulae.

The paper is organized as follows: In Section 2, we give an informal overview of our method. In Section 3, we present the general computational model of FTS and the restrictions that enable the application of the invisible auxiliary constructs methods. We also review the small model theorem, which enables automatic validation of the premises of the various proof rules. In Section 4, we describe how to construct liveness monitors. In some cases, the inductive invariant requires $\exists\forall$-components, to which the invisible invariant method no longer applies. Section 5 shows an extended small model theorem that allows handling such invariants, as well as an enhanced *project & generalize* method that generates invariants with $\exists\forall$-components. In Section 6, we illustrate the method on an example of a BAKERY protocol.

Related Work. Proving "bounded liveness" properties by safety techniques was proposed in [BAS02]. There, the justice requirements are incorporated into the safety model. It is not clear whether the method can be extended to parameterized systems. Incorporating the justice of such systems into the safety model seems to be prohibitively costly.

A survey on the method of invisible invariants is in [ZP04]. A tool that allows automatic generation of invariants using the method is described in [BFPZ05].

The problem of uniform verification of parameterized systems is undecidable [AK86]. One approach to remedy this situation, pursued, e.g., in [EK00], 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 [EN95], network invariants that can be viewed as implicit induction [LHR97], abstraction and approximation of network invariants [CGJ95], and other methods based on abstraction [GZ98]. Other methods include those relying on "regular model-checking" (e.g., [JN00]) that overcome some of the complexity issues by employing *acceleration* procedures, methods based on symmetry reduction (e.g., [GS97]), or compositional methods (e.g., ([McM99]), 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 only deal with safety properties. Among the methods dealing with liveness properties, we mention [CS02], which handles termination of sequential programs, network invariants [LHR97], and *counter abstraction* [PXZ02].

Most relevant to the work here are [FPPZ04b, FPPZ04a] that extend the method of invisible invariants to *invisible ranking*, by applying the method for automatic generation of auxiliary assertions to general assertions (not necessarily invariant), and proposing a rule for proving progress properties that embed the generated assertions in the rule's premises, and efficiently checks for their validity. As is well known to users of such rules, such a proof requires the generation of two kinds of auxiliary constructs: *helpful assertions* and *ranking functions*. To automatically generate ranking functions, we associate, with each potentially helpful transition an individual ranking function mapping states to integers in a small range. If the auxiliary constructs have no quantifiers, all the resulting premises are $\forall\exists$-premises and the small-model theorem can be used.

For protocols that cannot be proven with such restricted assertions, [FPPZ04a] extends the method of invisible ranking by allowing helpful assertions (and ranking functions) belonging to transitions to be of the form $\forall j. H(i,j)$, where $H(i,j)$ is a quantifier-free assertion. (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 such premises the proof rule is extended by implementing a new mechanism for selecting a helpful transition based on the establishment of a *pre-order* among transitions in each state.

Similarly to the method of invisible ranking, the method proposed here is applicable to the same type of "bounded progress" properties. However, the invisible ranking method requires numerous auxiliary construct, some (especially the pre-order) are at times hard to compute. The method proposed here is much simpler. The bound is derived from a small instantiation of the system, and, once the bound is computed, the only auxiliary construct needed is the strengthening invariant, which is well studied.

2 From Bounded Progress into Safety

This section contains a somewhat intuitive overview of the method that will be formalized and detailed in the following sections.

Consider a parameterized system S and a progress property $\phi\colon\ q \Rightarrow \Diamond\, r$. The property ϕ is bounded, if there is a bound K, independent of N, such that once a q-state is reached, after at most K rounds in which each process takes at least one step, a goal r-state is reached.

Consider a "liveness monitor" M_ϕ that observes S. Once a q-state is reached, M_ϕ resets a counter of rounds. Once each process takes (at least) one step, M_ϕ increases the round counter. When there are no pending states – states on a r-free path that originates at a q-state – the monitor keeps the round count at zero and does not keep track of the processes. If ϕ is bounded by K, then in the monitored systems the round counter never exceeds K. Thus, proving ϕ is equivalent to proving that the $S|||M_\phi \models \Box(rnd < K)$ where rnd is the round counter.

Of course, one has to choose K. One can either try to compute it (e.g., by instantiating $S(N)$ for a small number of processes, say N_0, and considering the pending paths on the instantiation) or one may choose some small instantiation, try increasing values of K until one succeeds, and then try the resulting K on larger (yet small) instantiations.

Once K is chosen, the method of invisible invariants can be used to show that for every N, $S|||M_\phi \models \square(rnd < K)$. In fact, since the monitor needs to be finite-state, we construct it with the knowledge of (the assumed) K and bound the round counter by K.

The method may fail for the following reasons:

1. For some N, $S(N) \not\models \phi$ or ϕ is not bounded;
2. The bound K is too small;
3. The heuristics used for the generation of invisible invariants are not sufficient for the given system;

We cannot deal with the first case. As to the second case, a larger instantiation usually solves the problem. Hence, it makes sense to try K on several instantiations before attempting to prove the property.

To deal with the third case, we present a new heuristic to generate candidate invariants with $\exists\forall$-assertions, and extend the small model theorem to accommodate invariants in such forms.

3 Preliminaries

As a computational model for parameterized bounded-data systems we use *bounded just transition systems*, that are a compassion-less variant of the model of *bounded fair transition system* of [FPPZ04a].

3.1 Just Transition Systems

We present a variant of the *just transition system* of [MP95]. A JTS is described by $S = \langle V, \Theta, \mathcal{T} \rangle$, 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. In the system we consider, every transition is disabled immediately after it is taken. Let $En(\tau)$ denote the assertion $\exists V'.\tau(V, V')$ characterizing the set of states in which τ is enabled.

Let $\sigma : s_0, s_1, s_2, \ldots$, be an infinite sequence of states. We say that transition $\tau \in \mathcal{T}^a$ 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 transitions can be considered as taken at the same position.

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

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, ...,$ state $s_{\ell+1}$ is a τ-successor of s_ℓ for some $\tau \in \mathcal{T}$.
- *Justice* — for every $\tau \in \mathcal{T}$, there are infinitely many positions $k \geq 0$, such that τ is disabled or taken at position k. Since we assume that transition are disbled immediately after they are taken, this is equivalent to requiring that τ is disbaled infinitely many times.

Composition of Just transition Systems. Assume two JTS's $S_1 \colon \langle V_1, \Theta_1, \mathcal{T}_1 \rangle$ and $S_2 \colon \langle V_2, \Theta_2, \mathcal{T}_2 \rangle$. The *synchronous parallel composition* of S_1 and S_2, denoted by $S_1 |||S_2$, is the JTS

$$\big(V_1 \cup V_2, \Theta_1 \wedge \Theta_2, \bigvee_{\tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2} \tau_1 \wedge \tau_2\big)$$

The *asynchronous parallel composition* of S_1 and S_2, denoted by $S_1 \| S_2$, is the JTS

$$\big(V_1 \cup V_2, \Theta_1 \wedge \Theta_2, \mathcal{T}_1^+ \cup \mathcal{T}_2^+\big)$$

where for every $i = 1, 2$, $\mathcal{T}_i^+$ includes, for every transition $\tau \in \mathcal{T}_i$, the transition τ with a conjunct requiring that all non-V_i variable are presevered. Formally, for a set of variables U, let *pres*(U) denote the assertion $\bigcup_{u \in U}(u' = u)$ stating that no U-variables is modified. Then $\mathcal{T}_i^+ = \{\tau \wedge \mathit{pres}(V_1 \cup V_2 \setminus V_i) : \tau \in \mathcal{T}_i\}$.

3.2 Bounded Just Transition Systems

To allow the application of the invisible invariants method, we further restrict the systems we study, leading to the model of *bounded just transition systems* (BJTS). For brevity, we describe here a simplified two-type model; the extension for the general multi-type 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 $[0..N]$; and
4. Any number of arrays of the type **index** $\mapsto$ **bool**. We refer to these arrays as *boolean arrays*.
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. E.g., 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 $\tau(i) : \forall j.\psi(i,j)$ for a restricted assertion $\psi(i,j)$.

Example 1 (A Simple Mutual Exclusion Algorithm).
Consider program SIMPLE in Fig. 2, which is a simple mutual exclusion algorithm that guarantees deadlock-freedom access to critical section for any N processes.

```
            in     N : natural where N > 1
            local  t : bool where t = 1
  N                ┌ loop forever do                       ┐
 ||                │     ┌ 0 : NonCritical              ┐  │
 ||   P[i] ::      │     │ 1 : when t = 1 do t := 0     │  │
i=1                │     │ 2 : Critical                 │  │
                   └     └ 3 : t := 1                   ┘  ┘
```

Fig. 2. Program SIMPLE

In this version of the algorithm, location 0 constitutes the non-critical section which a process may non-deterministically exit to the trying section at location 1. Location 1 is the waiting location where a process waits until the token (t) is available and then takes it. Location 2 is the critical section, and location 3 is the exit section where the process returns the token. As we show, the program guarantees that if some processes are waiting to enter the critical section, eventually some process will succeed. Fig. 3 describes the BJTS corresponding to program SIMPLE.

$$
\begin{array}{l}
V : \begin{cases} \pi : \textbf{array}[1..N] \textbf{ of } [0..3] \\ t : \textbf{ bool}; \end{cases} \\
\Theta : \forall i : \pi[i] = 0 \ \wedge \ t = 1 \\
\mathcal{T} : \begin{cases} \tau_0(i) : \forall j \neq i : \pi[i] = 0 \wedge \pi'[i] \in \{0,1\} \wedge \mathit{pres}(\{\pi[j], t\}) \\ \tau_1(i) : \forall j \neq i : \pi[i] = 1 \wedge t = 1 \wedge \pi'[i] = 2 \wedge t' = 0 \wedge \mathit{pres}(\{\pi[j]\}) \\ \tau_2(i) : \forall j \neq i : \pi[i] = 2 \wedge \pi'[i] = 3 \wedge \mathit{pres}(\{\pi[j], t\}) \\ \tau_3(i) : \forall j \neq i : \pi[i] = 3 \wedge \pi'[i] = 0 \wedge t' = 1 \wedge \mathit{pres}(\{\pi[j]\}) \end{cases}
\end{array}
$$

Fig. 3. BJTS for Program SIMPLE

As seen in the example of Fig. 3, the transition relation of process $P[i]$ is a disjunction of individual transitions of the form $\tau_0[i] \vee \tau_1[i] \vee \cdots \tau_k[i]$. We denote this disjunction by $\rho[i]$ and refer to it as the *process transition relation*. We denote by $\mathit{dis}[i] = \neg En(\rho[i])$ the assertion stating that the process transition is disabled, and by $\mathit{dis\_or\_taken}[i]$ the disjunction $\mathit{dis}[i] \vee \rho[i]$ claiming that process $P[i]$ is currently disabled.

3.3 The Small Model Theorem

Let $\varphi : \forall \vec{i} \exists \vec{j}.R(\vec{i}, \vec{j})$ be an AE-formula, where $R(\vec{i}, \vec{j})$ is a restricted assertion that refers to the state variables of a parameterized system $S(N)$ and to the quantified (**index**) variables $\vec{i}$ and $\vec{j}$. Let N_0 be the number of universally quantified, free **index** variables and **index** constants appearing in R. The claim below (stated in [PRZ01] and extended in [APR$^+$01]) provides the basis for automatic validation of the premises in the proof rules:

Theorem 1 (Small model property).
An AE-formula φ is valid iff it is valid over all instances $S(N)$ for $N \leq N_0$.

The small model theorem allows to check validity of AE-assertions on small models and to derive from that their validity on arbitrary large instantiations. This can be accomplished using BDD techniques. The method of invisible invariants applies *project & generalize* to produce candidate inductive assertions for the set of reachable states that are A-formulae. Checking their inductiveness requires checking validity of AE-formulae. The method of invisible ranking applies *project & generalize* to produce candidate assertions for various assertions (pending, helpful, ranking), all A- or E-formulae and, the premises obtained using these assertions are again all AE-formulae. Thus, the theorem implies they can be verified on small instantiations.

4 Monitoring Liveness with Safety

Assume a progress property ϕ: $q \Rightarrow \Diamond\, r$. It is often the case that such a progress property ϕ is "bounded", that is, there exists some bound K, such that after K *rounds* where each process is given at least one chance to progress, a goal state is guaranteed to be reached. If ϕ is a bounded progress property with bound K, then, instead of showing that $S \models \phi$, we can construct a non-interfering *monitor* $M_\phi(K)$ which we synchronously compose with S, and show that the simple invariance property $\Box(rnd < K)$ holds over the new system $S|||M_\phi(K)$.

Thus, for the case of bounded progress, liveness can be reduced to safety. The process can be done automatically, since one can derive K from the reachability graph of S.

Assume a BJTS S: $\langle V, \Theta, \tau\rangle$ and a progress property ϕ: $q \Rightarrow \Diamond\, r$. The monitor $M_\phi(K)$ is a BJTS M_ϕ: $\langle V_M, \Theta_M, \{\tau_M\}\rangle$, where:

V_M – consists of V and three new variables: a boolean *pend*, a variable *rnd* in the range $[0..K]$, and *moved* is an array [1..N] of booleans. The variable *pend* is set when the system is in a state that follows a q-state on a r-less path. The variable *rnd* counts the number of rounds. The variable $moved[i]$ is set when process i is disabled.

Θ_M – $pend = (q \wedge \neg r) \wedge rnd = 0 \wedge \bigwedge_{i=1}^{N} \neg moved[i]$, i.e., initially the round is 0 and every *moved* is F.

τ_M – τ_M consists of three conjuncts, one for each of the variables (the *moved*-part is further composed of N conjuncts). The transition τ_M consists of the following parts:

1. $pend' = \neg r' \wedge (pend \vee q')$. This conjunct states that *pend* becomes true when it was false and $q \wedge \neg r$ is true, and that *pend* becomes false when r is realized. In all other cases *pend* retains its previous value;

2. $\bigwedge_{i=1}^{N} moved'[i] \;=\; \left(\begin{array}{l} \textbf{if} \quad \neg pend' \vee \bigwedge_{j=1}^{N} moved[j] \;\textbf{then}\; \text{F} \\ \textbf{else}\; dis\_or\_taken[i] \vee moved[i] \end{array}\right)$
   This conjunct states that for every i, $moved[i]$ is true in pending states that are reached from $moved[i]$-states or when process i is disabled, but only if the round is not over (since then all the $moved[i]$'s need be reset).
3. $rnd' \;=\; \left(\begin{array}{lll} \textbf{if} & \neg pend' & \textbf{then}\; 0 \\ \textbf{else if} & rnd < K \wedge \bigwedge_{j=1}^{N} moved[j] & \textbf{then}\; rnd + 1 \\ & & \textbf{else}\;\; rnd \end{array}\right)$
   This conjunct states that a new round starts from pending states once all processes are were found disabled and a r-state was not reached. Similarly, rnd becomes 0 when an r-state is reached. In all other cases it remains intact.

Note that none of the conjuncts update the variables in V, justifying our description of M_ϕ as "non constraining."

Thus, as long as S is not in a pending state, $pend$, rnd, and all the $moved[i]$'s are F. Once S is in a pending state, $pend$ is set. From thereon, whenever every process is found disabled, rnd is incremented (as long as it is less than K). Obviously, if rnd ever reaches K, than it means that the goal q was not reached after K rounds, thus refuting the assumption that ϕ is a bounded progress property with bound K. However, if $\square(rnd < K)$, we can be assured that ϕ holds over S. This is captured by following claim:

Lemma 1 (Soundness).

$$(S|||M_\phi(K)) \models \square(rnd < K) \quad \Longrightarrow \quad S \models \phi$$

Proof. Assume that $S \not\models \phi$. Thus, there exists an S-computation σ of the form $\Sigma^k q\,(\Sigma - \{r\})^\omega$. Consider the behavior of $M_\phi(K)|||S$ when run on σ. Obviously, $\sigma \models \diamondsuit\,\square\, pend$. Since every process is guaranteed to be disabled infinitely many times, we have that $\sigma \models \square(\neg moved[i] \rightarrow \diamondsuit\, moved[i])$. We can therefore conclude that $\sigma \models \diamondsuit(rnd = K)$, thus $(S|||M_\phi(K)) \not\models \square(rnd < K)$. □

.

Example 2 (Liveness Monitor for Program SIMPLE*).*
Consider the program of Example 1, and suppose we want to establish the progress property ϕ: $(\exists i.\pi[i] = 1) \Rightarrow \diamondsuit(\exists i.\pi[i] = 2)$. We guess $K = 2$, and run the program for instantiations of $N = 3, 4, 5$ to confirm that this is a reasonable bound. We then construct the progress monitor $M_\phi(2)$ as above, where q: $\exists i.\pi[i] = 1$ and r: $\exists i.\pi[i] = 2$. We instantiated Program SIMPLE to 4 processes and run it composed with M_ϕ. We obtained the invariant

$$\begin{array}{ll} \forall i \neq j. & rnd < 2 \;\wedge\; (\neg pend \vee t = 1 \vee rnd = 0) \;\wedge \\ & (\neg pend \rightarrow rnd = 0 \;\wedge\; \neg moved[i]) \;\wedge\; (pend \rightarrow \pi[i] \neq 2) \;\wedge \\ & (rnd = 1 \rightarrow \pi[i] = 1 \;\wedge\; \neg moved[i]) \;\wedge \\ & (\pi[i] = 0 \;\wedge\; t = 0 \vee \pi[i] = 3 \rightarrow \neg moved[i]) \;\wedge \\ & (\pi[i] \geq 2 \rightarrow t = 0 \wedge \pi[j] < 2) \;\wedge\; (\pi[i] = 1 \wedge (t = 1 \vee \pi[j] = 3) \rightarrow pend) \wedge \\ & (\pi[i] = 0 \wedge t = 1 \wedge moved[i] = 1 \rightarrow \pi[j] = 1 \vee \neg moved[j]) \;\wedge \\ \exists i. & (rnd = 0 \wedge t = 0) \rightarrow (\pi[i] = 0 \wedge moved[i] \vee \pi[i] \geq 2) \end{array}$$

which is inductive and implies $rnd < 2$ over $(simple(4)|||M_\phi(2))$. It follows Theorem 1 that $\square(rnd < 2)$ is valid over the composed program with every N, and, according to Lemma 1, this implies that ϕ is valid over every instantiation of Program SIMPLE.

5 Cases Requiring an EA-Invariant

The method of invisible invariants obtains auxiliary assertions that are boolean combination of $\forall$-formulae. Used in the proof rule INV, the premises to be proved are then (at most) $\forall\exists$- formulae, whose validity, as the small model theorem establishes, can be shown on small instantiations.

In some cases, however, the auxiliary assertions obtained can have $\exists\forall$-components (thus the proof rule has to establish validity of such formulae), to which the theorem no longer applies. For example, when attempting to establish the livelock freedom property of Program BAKERY in Section 6, we need an invariant that contains a clause:

$$\exists i : (at\_l_2[i] \ \wedge \ moved[i] = 0 \ \wedge \ \forall j \neq i : (y[j] = 0 \ \vee \ y[i] < y[j]))$$

claiming that (at the last round) some process has the lowest ticket and has not yet taken a step. This is an $\exists\forall$-assertion, the likes of which are quite common when establishing progress properties.

In this section we present a new small model theorem that applies to some cases where $\exists\forall$-premises need to be validated. To automatically obtain an $\exists\forall$-assertion as a component in invariant assertions, we divide the reachable states into N symmetric subsets $D[1], \ldots, D[N]$, where each $D[i]$ can be over-approximated by an assertion of the type $D_\alpha(i) : \forall j.q(i,j)$, so that the disjunction of $D_\alpha(i)$'s is our desired $\exists\forall$-assertion. The body of the $\exists\forall$-assertion $q(i,j)$ is computed by the procedure *project & generalize*.

5.1 An Extended Small Model Theorem

Consider a parameterized BJTS $S(N)$ and a formula of the type $\forall\exists \ \vee \ \exists\forall$ that we want to show valid over all instantiations of S. The Small Model Theorem establishes that, when only the first disjunct exists, it suffices to show validity of the formula only on small instantiations whose size is bounded by the number of free and universally quantified variables. We extend it here for the case that the second disjunct exists, however, its scope is limited.

Theorem 2 (Extended Small Model Theorem). *Consider the formula*

$$\phi: \forall\vec{i}\exists\vec{j}.R(\vec{i},\vec{j}) \ \vee \ \exists i\forall j.Q(i,j)$$

where R and Q are restricted assertions, and, in addition, we have:

$$\forall i,j,k.(\neg Q(i,j) \ \wedge \ \neg Q(j,k) \ \rightarrow \ (\neg Q(i,k) \ \vee \ \neg Q(j,j))$$

Let N_0 be the number of universally quantified, free **index** *variables and* **index** *constants appearing in R. Then ϕ is valid over $S(N)$ for every $N \geq 2$ iff ϕ is valid over $S(N)$ for every $N \leq 2N_0$.*

Proof Outline: We show that if $\neg\phi$ is satisfiable over a model of size $N_1 > 2N_0$, then it is satisfiable over a model of size $N_2 \leq 2N_0$. The formula $\neg\phi$ is equivalent to:

$$\psi\colon \exists\vec{i}\forall\vec{j}.\neg R(\vec{i},\vec{j}) \;\wedge\; \forall i.\exists j.\neg Q(i,j)$$

and assume $s \models \psi$ for some state s of $S(N_1)$ where $N_1 > 2N_0$. Following the proof of the original theorem ([APR$^+$01]), we take the (no more than N_0) values assigned to the constants, free, and existentially quantified **index** variables in the first conjuncts that s assigns, say to $u_1,\ldots,u_L$ (where $L \leq N_0$). Obviously, if we project s onto $U = \{u_1,\ldots,u_L\}$ (i.e., remove references to any **index** variables outside U), the resulting state satisfies the first conjunct, while adding back all the variables that refer to some particular **index** variable that is not in this set, will not change that.

We next add to U at most L other **index** variable that will guarantee the satisfiability of the second conjunct. Starting with $V_0 = \emptyset$, we iterate L steps. At the ℓ^{th} step, we start with a set $V_{\ell-1}$ and a state $s_{\ell-1}$, such that $s_{\ell-1}$ is the projection of s onto $V_{\ell-1}$, and $s_{\ell-1} \models \forall i \in V_{\ell-1}.\exists j \in V_{\ell-1}.\neg Q(i,j)$. We then add to $V_{\ell-1}$ the element u_ℓ, and, possibly, another element, to obtain V_ℓ.

Assume $1 \leq \ell < L$ and consider u_ℓ. If $s \models \neg Q(u_\ell, v)$ for some $v \in V_\ell \cup \{u_\ell\}$, then $V_\ell = V_{\ell-1} \cup \{u_\ell\}$. Assume therefore that for all $v \in V_{\ell-1}$, $s \not\models \neg Q(u_\ell, v)$ and that $s \not\models \neg Q(u_\ell, u_\ell)$. Since $s \models \psi$, it follows that for some $j_1 \in [1..N_1]$, $s \models \neg Q(u_\ell, j_1)$. We continue along a $\neg Q$-chain in $[1..N_1]$ of the form $u_\ell = j_0, j_1, \ldots, j_m$ that the j_i's are mutually distinct, for every $i = 0,\ldots,m$, $s \models Q(j_i, j_i)$, and for every $i = 1,\ldots,m$, $s \models \neg Q(j_{i-1}, j_i)$. (The finiteness of $[1..N_1]$ guaratees that the chain is finite.) It thus follows that $s \models \neg Q(j_m, j_m) \wedge \neg Q(u_\ell, j_m)$. We then let $U_\ell = U_{\ell-1} \cup \{u_\ell, j_m\}$.

Note that the process adds at most L new elements to U, thus the state attained is of size at most $2N_0$. Suppose $U_L = \{v_1,\ldots,v_{2L}\}$ where $v_1 < \ldots v_{2L}$. We can now contract the state to $1..2L$ and obtain a state s' of $S(2L)$ such that $s' \models \psi$. $\square$

6 Example: BAKERY

Consider program BAKERY in Fig. 4, 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 non-critical section which a process may non-deterministically exit to the trying section at location 1. Location 1 is the ticket assignment location – to guarantee the finiteness of the state-space, the ticket values are $[1..N]$; when a process i takes a ticket, the tickets help by the other processes may be changed preserving their relative order, and process i gets a ticket whose value is higher than the others. 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**. In fact, π is of type **index** $\mapsto$ $[0..4]$, but it can be encoded by three boolean arrays. Note also that the ticket assignment statement at 1 is non-deterministic and may modify the values of all tickets.

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

Fig. 4. Program BAKERY

The livelock freedom property of the program is:

$$\phi\colon (\exists z : at\_l_1[z]) \Rightarrow \Diamond(\exists z : at\_l_3[z])$$

The bound obtained for the property is $K = 2$.

Following are the results of our verification experiments applied to the BAKERY protocol.

1. We chose (arbitrarily) to instantiate the system to $N = 4$. We applied the enhanced *project & generalize* method [FPPZ04a] to BAKERY(4), generating candidate invariants in the forms of a boolean combinations of universal assertions. The best candidate obtained was φ_1 of the form

$$\phi_1\colon \forall i, j.\alpha_1(i,j) \ \wedge \ \exists i, j.\alpha_2(i,j) \ \vee \ \forall i, j.\alpha_3(i,j)^1$$

   The assertion ϕ_1 failed to be inductive.
2. We used our invisible invariant generator to generate an $\exists\forall$-assertion $\phi_2\colon \exists i \forall j \beta(i,j)$ over BAKERY(4). We then define $\phi\colon\ \phi_1 \wedge \phi_2$, which is both inductive and implies the safety property $\Box(rnd < 2)$ over BAKERY(4).
3. We next checked whether $\neg\beta$ is reflexive or transitive. Since the test requires checking a universal assertion, we can apply Theorem 1 and derive that it suffices to check the reflexivity/transitivity of $\neg\beta$ over BAKERY(N_0) for $N_0 \leq 4$ to derive that it is reflexive/transitive over BAKERY(N) for every N.
4. By applying Theorem 2, we derived $N = 8$ as the size of the small model to establish the validity of the premises in INV using ϕ as the auxiliary invariant. The candidate invariant $\varphi_1 \wedge \varphi_2$ was reconstructed over BAKERY(8), and proved to be inductive and to imply the safety property $\Box(rnd < 2)$. We can therefore conclude that the protocol satisfy the livelock freedom property for any instantiation.

The code for the programs can be found in *http://eeyore.cs.nyu.edu/acsys/forte06/*.

[1] We can "guide" our automated invisible invariant generator as to the form of the assertion to be produced; however, being invisible and produced by BDD techniques, the generated assertions cannot be neatly displayed.

We would like to point out that the proof obtained by the method proposed here is considerably simpler than the proof presented in [FPPZ04a] which calls for auxiliary constructs other than invariants, thus requires considerably more interaction with the user.

7 Discussion and Future Work

The paper presents a method for automatic verification of progress properties of parameterized systems based on the method of invisible invariants. The method is based on the observation that such progress properties are usually "bounded," and can thus be converted into safety properties. The heuristic proposed attempts to find a bound for the progress property, and use the method of invisible invariants to prove the resulting safety property.

There are several cases where the proposed method is bound to fail:

Super-linear bounds: As it is now, the method can only be successful when the bound is linear in the number of processes. Some protocols (e.g., Peterson's N-process mutual exclusion protocol) have bounds that are non-linear in the number of processes. We are currently working on extending the method to apply to cases where the bound is quadratic in the number of processes.

Fairness-dependent bounds: The method cannot be applied to cases where the bound depends on non-justice assumptions. Such non-justice fairness assumptions occur, for example, when using semaphores, the bound depends on the number of compassion (strong fairness) assumptions. However, compassion can be translated into justice, at the cost of adding some new variables to the system, hence our method can indirectly deal with such cases.

Probability-dependent progress: When protocols involve probabilistic choices among transitions, progress often depends on probabilistic arguments. As shown in [APZ03], one can often transform such protocols to non-probabilistic protocols by a "planner" that occasionally determines the results of some probabilistic choices, leaving the others non-deterministic. In fact, the projection used in the method of invisible invariants can be applied to obtain the planner automatically, and then the progress property can be bounded. Consequently, the method proposed here, in conjunction with the automatically obtained planner. can be applied to probabilistic protocols as well.

Failure of invisible invariants: The method of invisible invariant is heuristic in nature, and may sometimes fail. As we showed here, sometimes a $\forall\exists$ invariant is called for, which we can obtain only in certain cases. In some cases, there is no strengthening invariant of the type we can generate. For these cases, the method presented here is bound to fail.

As in the case of all BDD-based techniques, it is always possible that the invariant generated is too large for the model checker to handle. In fact, this may happen much faster than when checking "regular" safety properties, since those required here include the round counter.

References

[AK86] K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Info. Proc. Lett.*, 22(6), 1986.

[APR$^+$01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *G. Berry, H. Comon, and A. Finkel, editors,* Proc. 13^{th} Intl. Conference on Computer Aided Verification (CAV'01)*, volume 2102 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 221–234, 2001.

[APZ03] T. Arons, A. Pnueli, and L. Zuck. Parameterized verification by probabilistic abstraction. In *6th International Conference on Foundations of Software Science and Computational Structures*, volume 2620 of *Lect. Notes in Comp. Sci.*, pages 87–102, Warsaw, Poland, April 2003. Springer-Verlag.

[BAS02] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In Rance Cleaveland and Hubert Garavel, editors, *Electronic Notes in Theoretical Computer Science*, volume 66. Elsevier, 2002.

[BBC$^+$95] N. Bjørner, I.A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[BFPZ05] I. Balaban, Y. Fang, A. Pnueli, and L.D. Zuck. An invisible invariant verifier. In Proc. 17^{th} Intl. Conference on Computer Aided Verification (CAV'05), Springer-Verlage LNCS 3576, pp. 291–295, 2005.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR92)*, volume 962 of *Lect. Notes in Comp. Sci.*, pages 395–407, Philadelphia, PA, August 1995. Springer-Verlag.

[CLP84] S. Cohen, D. Lehmann, and A. Pnueli. Symmetric and economical solutions to the mutual exclusion problem in a distributed system. *Theor. Comp. Sci.*, 34:215–225, 1984.

[CS02] M. Colon and H. Sipma. Practical methods for proving program termination. In *E. Brinksma and K. G.Larsen, editors,* Proc. 14^{th} Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 442–454, 2002.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE-17)*, pages 236–255, 2000.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22nd ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, 1995.

[FPPZ04a] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with incomprehensible ranking. In *Proc.* 10^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), volume 2988 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 482–496, April 2004.

[FPPZ04b] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *Proc. of the* 5^{th} *conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lect. Notes in Comp. Sci.*, pages 223–238, Venice, Italy, January 2004. Springer-Verlag.

[GS97] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In *O. Grumberg, editor, Proc.* Proc. 9^{th} Intl. Conference on Computer Aided Verification, (CAV'97)*, volume 1254 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, 1997.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In *B. Steffen, editor, Proc.* 4^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 424–438, 1998.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *S. Graf and M. Schwartzbach, editors, Proc.* 6^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), volume 1785 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 2000.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, 1997.

[McM99] K.L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Proc. Charme 1999*, volume 1703 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pages 219–234, 1999.

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

[OSR93] S. Owre, N. Shankar, and J.M. Rushby. User guide for the PVS specification and verification system (draft). Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc.* 7^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 82–97, 2001.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction, 2002.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986.

[ZP04] L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems. *Computer Languages, Systems, and Structures*, Volume 30(3–4), pp. 139–169 2004.