Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Computer Languages, Systems & Structures 30 (2004) 139–169
COMPUTER LANGUAGES, SYSTEMS & STRUCTURES
www.elsevier.com/locate/cl

Model checking and abstraction to the aid of parameterized systems (a survey)☆

Lenore Zuck[a,*], Amir Pnueli[a,b]

[a] *Department of Computer Science, New York University, Courant Institute, 251 Mercer Street, New York, NY 10012, USA*
[b] *Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel*

Received 18 February 2004; accepted 18 February 2004

Abstract

Parameterized systems are systems that involve numerous instantiations of the same finite-state module, and depend on a parameter which defines their size. Examples of parameterized systems include sensor systems, telecommunication protocols, bus protocols, cache coherence protocols, and many other protocols that underly current state-of-the-art systems. Formal verification of parameterized systems is known to be undecidable (Inform. Process. Lett. 22 (6)) and thus cannot be automated. Recent research has shown that it is often the case that a combination of methodologies allows to reduce the problem of verification of a parameterized system into the problem of verification of a finite-state system, that can be automatically verified.

This paper describes several recent methodologies, based on model checking and abstraction. We start with the method of *invisible auxiliary assertions* that combines a small-model theorem with heuristics to automatically generate auxiliary constructs used in proofs of correctness of parameterized systems. We also describe the method of *counter abstraction* that offers simple liveness proofs for many parameterized systems, and discuss novel methodologies of using counter abstraction to automatically verify that *probabilistic* parameterized system satisfy their temporal specifications with probability 1.

Keywords: Parameterized systems; Invisible invariants; Invisible ranking; Counter abstraction; Probabilistic verification; Safety; Liveness; Progress

☆ This research was supported in part by NSF Grant CCR-0205571, the Israel Science Foundation (Grant No. 106/02-1), and the John von Neumann Minerva Center for Verification of Reactive Systems.
* Corresponding author. Tel.: +1-212-998-3002; fax: +1-212-995-4121.
E-mail addresses: zuck@cs.nyu.edu (L. Zuck), amir@cs.nyu.edu (A. Pnueli).

1477-8424/$ - see front matter © 2004 Published by Elsevier Ltd.
doi:10.1016/j.cl.2004.02.006

1. Introduction

The main challenge of verification is undoubtedly automatic verification of very large systems. *Parameterized systems* are systems that consist of potentially numerous instantiations of "simple" modules, where the size of each module is the parameter of the system. Examples include sensor systems, cache coherence protocols, bus protocols (e.g., PCI), and telecommunication protocols. Typically of such systems, the programs of the individual processes (modules) are given by the process id. With the growing emphasis on embedded systems, we expect a growing interest in parameterized systems.

Verifying that a parameterized system satisfies some specification entails proving that *every* instantiation of the system satisfies the specification. In particular, it is necessary to show that no matter how many individual processes participate in the system, it satisfies its specification.

In this work we focus on specifications that are expressible by linear time temporal logic (LTL), to which we refer as "temporal properties". There are two families of properties one usually wishes to verify, *safety* and *liveness*. Roughly speaking, safety properties state that a bad state never happens. For example, the safety property of mutual exclusion algorithms is that no two processes are ever at their critical section at the same time. Liveness properties guarantee that something good eventually happens. For example, the liveness property of mutual exclusion algorithms is that every process that is trying to access its critical section, eventually succeeds. Usually, liveness properties can only be proven under appropriate *fairness* requirements. Roughly speaking, fairness requirements capture a reasonable behavior of processes related to their eventual progress. For example, fairness will guarantee that a process in the critical section eventually leaves it.

When considering a parameterized system, both safety and liveness have to be established for every instantiation of the system. Thus, one must establish that, for every set of processes, the system that consists of the composition of these processes satisfies both safety and liveness properties. A highly desirable objective is to develop methods for the *uniform verification* of parameterized systems, that is, to produce a single proof that *all* instances of the system satisfy their specifications. There are two main approaches for verification of temporal properties. The first is using *deductive methods* where proofs are constructed incrementally and manually until the desired property is proved. While proofs that are obtained deductively can be checked by theorem provers, they usually require considerable amount of time and expertise to construct. The second approach for verification of temporal properties is *enumeration*, or *model checking* [2,3]. There, the prover models the system to be verified as a finite-state transition system and proves that the system satisfies its specification. While model checking is fully automatic and has had numerous successes, a system to be model-checked has to be finite-state (and, practically speaking, rather small).

Obviously, for the purpose of uniform verification of parameterized systems (naive application of) model checking is ruled out since it can verify only a single instance at a time. However, to be applicable, methods of uniform verification should have a high degree of automation, which rules out extensively interactive deductive verification.

In this paper, we explore two families of methods that allow fully automated uniform verification of parameterized systems. One family, called the *method of auxiliary constructs*, automates the user-supplied interactive steps in deductive verification. The other, *counter abstraction*, abstracts the parameterized system and its specifications, so that its verification is reduced to simple model checking. The premise underlying both approaches is that the "infiniteness" of such systems is

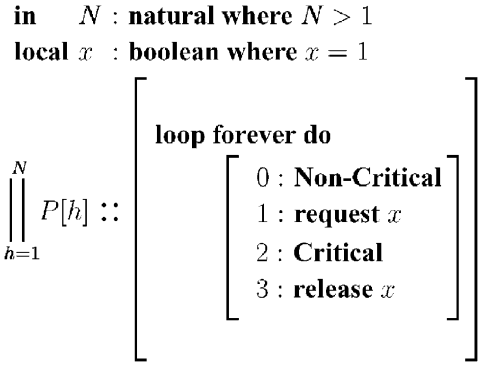

Fig. 1. Program MUXSEM.

restricted to the *number* of processes, since each process is usually finite-state. Manual verification of such systems is usually accomplished using few proof rules. The method of auxiliary constructs exploits the inherent symmetry of a system to derive inductive assertions about it. Counter abstraction uses the relative small size of the individual modules to construct a finite-state abstraction. We also demonstrate how the methods can be extended to deal with *probabilistic parameterized systems*.

In the rest of the introduction we include (in Section 1.1) an example of a simple parameterized system which will be used throughout the paper. Section 1.2 is an overview of related work on parameterized systems. Section 1.3 describes the methodologies we introduce and their relations to existing work. Section 1.4 describes the organization of the paper.

1.1. A parameterized system

For an example of a parameterized system consider the program MUXSEM in Fig. 1, which is a mutual exclusion protocol using semaphores. The processes are indexed $1,\ldots,N$, where N is the system's parameter. Thus, for every $N > 1$, an instantiation of MUXSEM consists of processes $P[1],\ldots,P[N]$.

The semaphore instructions "**request** x" and "**release** x" appearing in the program stand, respectively, for

$$\langle \textbf{when } x = 1 \textbf{ do } x := 0 \rangle \quad \text{and} \quad x := 1.$$

A process in location 0 (the "idle" state) may stay there indefinitely or move on to location 1 (the "trying" state), where it remains until it gains access to the (single) semaphore x. It then enters location 2 (the "critical" section), upon exit from it, releases the semaphore in location 3 (the "exit" state), and goes back to its idle state.

The (safety) mutual exclusion property of MUXSEM is that no two processes are ever simultaneously at their critical sections. Expressed in LTL, this property is

$$\forall N > 1 : \forall i, j : 1 \leqslant i \neq j \leqslant N \rightarrow \Box\neg(at\_\ell_2[i] \wedge at\_\ell_2[j]),$$

where "$at\_\ell_\ell[j]$" stands for "the program counter of process j is ℓ," and $\Box$ is the temporal operator "always".

The liveness property of MUXSEM states that every process that wishes to gain access to its critical section eventually succeeds; it is expressed by the LTL formula

$$\forall N > 1 : \forall i : 1 \leqslant i \leqslant N \rightarrow \Box(at\_\ell_1[i] \rightarrow \Diamond at\_\ell_2[i]),$$

where $\Diamond$ is the temporal operator "eventually". The fairness properties of MUXSEM are that no process stays forever in location 1 if the semaphore is available infinitely many times, and that no process stays forever in locations 2 or 3. (Note, however, that there is no fairness property associated with location 0: processes may stay idle forever.)

1.2. Related work

The problem of uniform verification of parameterized systems is, in general, undecidable [1]. There are two possible remedies to the undecidability: either we should look for restricted families of parameterized systems for which the problem becomes decidable, or devise methods which are sound but, necessarily incomplete, and hope that the system of interest will yield to one of these methods.

Among the representatives of the first approach we can count the work of German and Sistla [4] which assumes a parameterized system where processes communicate synchronously, and shows how to verify single-index properties. Similarly, Emerson and Namjoshi [5] proved a PSPACE complete algorithm for verification of synchronously communicating processes. Many of these methods fail when we move to asynchronous systems where processes communicate by shared variables. Perhaps the most advanced of this approach is the paper [6], which considers a general parameterized system allowing several different classes of processes. However, this work provides separate algorithms for the cases that the guards are either all disjunctive or all conjunctive. A protocol such as the Cache Coherence protocol we handle in [7], which contains some disjunctive and some conjunctive guards, cannot be handled by the methods of [6].

The sound but incomplete methods include methods based on explicit induction [8], network invariants, which can be viewed as implicit induction [9–12], methods that can be viewed as abstraction and approximation of network invariants [13–17], and other methods that can be viewed as based on abstraction [18]. Most of these methods require the user to provide auxiliary constructs, such as a network invariant or an abstraction mapping. Other attempts to verify parameterized protocols such as Burn's protocol [19] and Szymanski's algorithm [20–22] relied on abstraction functions or lemmas provided by the user. The work in [23] deals with the verification of safety properties of parameterized networks by abstracting the behavior of the system. PVS [24] is used to discharge the generated verification conditions.

Among the automatic incomplete approaches, we should mention the methods relying on "regular model-checking" [25–28], where a class of systems which include our bounded-data systems as a special case is analyzed representing linear configurations of processes as a word in a regular language. Unfortunately, many of the systems analyzed by this method cause the analysis

procedure to diverge and special *acceleration* procedures have to be applied which, again, requires user ingenuity and intervention.

The works in [29–32] study symmetry reduction in order to deal with state explosion. The work in [18] detects symmetries by inspection of the system description.

1.3. Bird's eye view of the presented methods

1.3.1. The method of invisible auxiliary constructs

When applying deductive proof rules to verification, the main burden on the user is to supply auxiliary assertions, that are often shown to be inductive. For example, verification of safety properties is often accomplished by means of a *strengthening invariant*, which is then shown to be inductive over the system (i.e., to hold in every reachable state) and to imply the desired safety property. The method of *invisible auxiliary constructs* automatically generates such auxiliary assertions for parameterized systems, so that the verification conditions, once the generated assertions are "plugged" into them, can be model checked.

The method consists of two parts: The first exploits the symmetry of the parameterized system and, given a set of reachable states (e.g., "all reachable states", or "all states where process number 1 is trying to enter the critical section"), "guesses" a universally quantified assertion that over-approximates the given set. The second part is a "small model" theorem, that establishes when assertions over parameterized systems can be model checked on small instantiations (whose size depends on the parameterized system itself and on the assertion generated) to derive their validity over any instantiation.

Put together, for some deductive proof rules, the method of invisible auxiliary constructs automatically generates assertions that traditionally are supplied by the user, and the resulting verification conditions can be model checked, so that the proof rules are applied without user intervention. As a matter of fact, the method has "invisible" in its name since the user need never see the generated auxiliary assertions.

We describe the method and demonstrate its utility in applying deductive proof rules that establish both safety and liveness properties.

1.3.1.1. Related work: The method of *invisible auxiliary constructs* was introduced in [7] (under the name *invisible invariants*) for the automatic proofs of safety properties of parameterized systems, developed further in [33] for a larger family of systems, and in [34,35] for automatic proofs of liveness properties. The method has been successfully applied to verification of safety properties of numerous protocols, including the Illinois Cache Coherence protocol, 3-stage Pipeline, and a Cache Coherence protocol developed for the ASCII Blue, as well as to the verification of liveness properties of numerous mutual exclusion algorithms. The idea of the invisible invariant method came from McMillan's work on compositional model-checking (e.g., [36]), which combines automatic abstraction with finite-instantiation due to symmetry.

1.3.2. Counter abstraction

While the method of auxiliary constructs can be used to prove both safety and liveness properties of parameterized systems, its application is at times far from trivial. In addition, the systems that are

covered by the method are "bounded data", which means that the data structures used are assumed to be of a certain type. The method of counter abstraction is somewhat simpler, and, when applicable, trivial to apply. Its shortcomings is that it is only suitable for system in a unstructured, or clique, architecture, and that the number of individual local states of each process should be rather small. The latter restriction can be overcome by some manual (deductive) intervention in the method, while the former is inherent to the method. For the types of systems counter abstraction can handle, it offers extremely elegant and simple uniform verification.

Counter abstraction is an application of *finitary abstraction*, which calls for an abstraction of the parameterized system into a finite-state system, and an abstraction of the property to be verified so that it refers to the abstract system. The abstraction is such that if the abstract system satisfies the abstract property, one can safely conclude that the concrete system satisfies the concrete property. A general theory of finitary abstraction which abstracts a system *together* with the property to be proven is presented in [16], and can be applied to the verification of arbitrary LTL properties, including liveness properties.

Roughly speaking, a concrete state is *counter abstracted* by counting the number of processes in each local state, truncating the count at 2. The idea of using this abstraction is not new. It had been studied by numerous works whose goal was to prove safety properties of parameterized systems, see, e.g., [37]. In fact, checking a safety property of a counter abstracted system reduces to reachability of a finite graph.

Proving liveness properties of counter abstracted system is, however, harder, since one has to take into account the fairness properties, that rule out some infinitary pathologically bad behaviors of a system. Thus, while a liveness property may not hold over the system if fairness properties are ignored (which is often the case since we allow the idle transition from every state), it may hold under fairness assumptions. Applying the algorithm of [16] for deriving *abstract* fairness properties of parameterized systems results often in the trivial T (*true*), which does not allow for verification of liveness properties. This is because the fairness properties often refer to a single process and counter abstraction gives no tools to track down a single process. To overcome this difficulty, we compiled, for many types of common concrete fairness properties, (counter) abstract fairness properties that can be derived from them. Armed with these abstract fairness properties, we can prove concrete liveness properties whose abstraction is non-trivial.

The same problems we face with abstracting fairness properties haunt us when we try to abstract individual liveness properties. To overcome these difficulties, we "counter abstract but one"—that is, counter abstract a system with all but the process whose liveness we want to establish. Proving individual liveness then reduces to proving that the composition of the abstract system (of all but the concrete process) with the concrete process satisfies the individual liveness property of the concrete system. This all can of course be model checked.

The method can be applied with no extensions to parameterized systems in which every process has only finitely many local states and the global variables range over finite domains. It can also be extended to deal with systems in which individual processes may have infinitely many local states. The necessary extension is that we identify some state assertions and then allocate (bounded) counters that count the number of processes whose local states satisfy these assertions.

We describe how parameterized systems and their properties are counter abstracted, and show how various liveness properties can be proven over the abstract systems.

1.3.2.1. Related work: Counter abstraction was first studied in [38]. The work which obtains results close to ours is [39] and previous articles describing the parameterized systems abstracted and explored (PAX) system. There, the problem of verification of parameterized systems is addressed, with emphasis on liveness properties. One of the differences between the two approaches is that [39] is based on the method of *predicate abstraction* [40]. As a result, the structure of the abstraction varies from case to case and the computation of the added fairness requirements depends on a marking algorithm which has to be re-applied for each case separately, searching for appropriate well-founded ranking. In comparison, counter abstraction offer a *uniform* abstraction approach, which provides a standard recipe for the additional compassion (strong fairness) and justice (weak fairness) requirements, that, according to [39], cannot be automatically lifted from the concrete to the abstract systems.

Another relative advantage of our method is its extended ability to deal with parameterized systems whose individual processes are not necessarily finite-state.

1.3.3. Probabilistic protocols

Probabilistic elements have been introduced into concurrent systems in the early 1980s to provide solutions (with high probability) to problems that do not have deterministic solutions. Among the pioneers of probabilistic protocols were [41,42]. One of the most challenging problems in the study of probabilistic protocols has been their formal verification. While methodologies for proving safety (invariance) properties still hold for probabilistic protocols, formal verification of their liveness properties has been, and still is, a challenge. The main difficulty stems from the two types of non-determinism that occur in such programs: Their asynchronous execution, that assumes a potentially adversarial (though somewhat fair) scheduler, and the non-determinism associated with the probabilistic actions, that assumes an even-handed coin-tosser.

Since many of the probabilistic protocols that have been proposed and studied (e.g., [41–43]) are parameterized, a naturally arising question is whether we can combine automatic verification tools of parameterized systems with those of probabilistic ones. In [44,17,45] we studied automatic verification of probabilistic parameterized systems. The approach we took in [44,17] is based on user-supplied network invariants. The approach in [45], which we describe here, combines the previously described abstraction methods with some treatment of the probabilistic elements of the system.

1.3.3.1. Related work: To our knowledge, with the exception of [45], there are no works that have attempted a uniform approach to the automatic verification of probabilistic parameterized systems. The PRISM probabilistic model checker [46], based on Markov chains and processes, allows the user to verify that a property holds with arbitrary probability, and not just probability 1. However, PRISM does not support the uniform verification of parameterized systems, but rather the verification of individual system instantiations.

1.4. Overview

In Section 2 we describe our formal model. We introduce *fair bounded discrete systems*—fair transition systems with bounded data types and somewhat restricted transition relation. We also

elaborate on the types of properties that are of interest to us. The topic of Section 3 is the method of auxiliary constructs. We start by describing ProGen, a heuristics to generate auxiliary assertions. We then describe the method of *invisible invariants*, that uses ProGen to obtain automatic verification of safety properties of parameterized systems, and the method of *invisible ranking*, that uses ProGen to obtain automatic verification of liveness properties of parameterized systems. The topic of Section 4 is the method of counter abstraction. We first describe how a parameterized system is abstracted into a finite-state on, ignoring fairness requirements. We then show how to counter abstract the justice requirements, and how to prove group liveness (live-lock freedom) and individual liveness properties of the original parameterized systems using the counter abstracted systems. In Section 5, we turn our attention to parameterized probabilistic systems. We describe an almost automatic method to prove liveness properties of such systems, based on transforming the system into a computationally equivalent system in which probability is replaced by non-determinism. The transformed system can then be verified using either invisible ranking or counter abstraction. We conclude in Section 6.

2. The model

For a programming language to describe protocols we use SPL, the simple programming language of [47,21], an example of which is the code for MuxSem in Section 1. Here we introduce the formal model of *fair bounded discrete systems* (FBDS), that underlies the SPL code. The FBDS model is essentially the model of bounded discrete systems of [33] augmented with fairness.

2.1. Data types and signatures

Let $\mathbf{type}_0$ denote the set of boolean and finite-range scalars which we sometimes denote by **bool**. Let $N \in \mathbb{N}^+$, and let $\mathbf{type}_1$ denote the scalar data type that includes integers in the range $[1..N]$. The systems we study include variables that are either $\mathbf{type}_i$ scalars, for $i = 0, 1$, or arrays of the type $\mathbf{type}_1 \mapsto \mathbf{type}_0$. We refer to N as the *system's parameter*.

In MuxSem, there is a single $\mathbf{type}_0$ variable, namely the boolean semaphore x. There are no $\mathbf{type}_1$ variables ($\mathbf{type}_1$ variables store process indices; see Section 3.5 for an example). There is a single type $\mathbf{type}_1 \mapsto \mathbf{type}_0$ variable, which is the array of program counters $\pi : [1..N] \mapsto [0..3]$. Obviously, the range of π is not boolean, however, we allow $\mathbf{type}_0$ to be any finite scalar types, $\mathbf{type}_0$ to be defined as 0..3. (Alternatively, we can represent π by $\pi_0, \ldots, \pi_3$, so that $\pi[i] = j$ can be represented as $\pi_j[i] \wedge \bigwedge_{j' \neq j} \neg \pi_{j'}[i]$.)

Atomic formulae may compare two variables of the same type. Thus, for $\mathbf{type}_i$ variables y and y', $i = 0, 1$, $y \leqslant y'$ is an atomic formula, and so is $z[y] = x$ where y is of $\mathbf{type}_1$, z is a $\mathbf{type}_1 \mapsto \mathbf{type}_0$, and x of $\mathbf{type}_0$.

Formulae, used in the transition relation and the initial condition, are obtained from the atomic formulae by closing them under negation, disjunction, and existential quantification over $\mathbf{type}_1$ variables. We refer to a quantifier-free formula as an *assertion*.

2.2. Fair bounded discrete systems

A *fair bounded-data discrete system* (FBDS) $S = \langle V, \Theta, \rho, \mathscr{J}, \mathscr{C} \rangle$ consists of:

- V—A set of *system variables*, as described above. A *state* of the system S provides a type-consistent interpretation of the system variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of states over V.
- $\Theta(V)$—The *initial condition*: A formula characterizing the initial states. We assume that the initial condition can be written as

$$\exists h_1, \ldots, h_H \in \mathbf{type}_1 : \Theta_E(\vec{h}) \wedge \forall t_1, \ldots, t_T \in \mathbf{type}_1 : \Theta_A(\vec{t}). \tag{1}$$

- $\rho(V, V')$—The *transition relation*: A formula, relating the values V of the variables in state $s \in \Sigma$ to the values V' in an $\mathscr{S}$-successor state $s' \in \Sigma$. For all the systems we consider here, we assume that the transition relation can be written as

$$\exists h_1, \ldots, h_H \in \mathbf{type}_1 : \forall t_1, \ldots, t_T \in \mathbf{type}_1 : R(\vec{h}, \vec{t}), \tag{2}$$

  where R is a well-typed quantifier-free formula.
- $\mathscr{J}$—A set of *justice* (*weak fairness*) requirements: Each justice requirement is an assertion; A computation must include infinitely many states satisfying the requirement.
- $\mathscr{C}$—A set of *compassion* (*strong fairness*) requirements: Each compassion requirement is a pair $\langle p, q \rangle$ of assertions; A computation should include either only finitely many p-states, or infinitely many q-states.

Note that Θ and ρ are restricted to "formulae" defined in the previous section. A *computation* of S is an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$, satisfying the requirements:

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

We denote the set of all reachable states of a system S by *reach*(S). A (linear) temporal formula φ is *valid over* S if every computation of S satisfies φ.

Example 1. Consider again MUXSEM. The set of variables consists of the semaphore x, and the program counters (π). The initial condition is

$$\forall i : \pi[i] = 0 \wedge x = 1.$$

The transition relation of the system can be expressed as $\exists h : \forall t : R(h,t)$, where R is given by

$$h \neq t \rightarrow \pi'[t] = \pi[t] \wedge \begin{pmatrix} \pi'[h] = \pi[h] \wedge x' = x & \vee \\ \pi[h] = 0 \wedge \pi'[h] = 1 \wedge x' = x & \vee \\ \pi[h] = 1 \wedge x \wedge \pi'[h] = 2 \wedge \neg x' & \vee \\ \pi[h] = 2 \wedge \pi'[h] = 3 \wedge x' = x & \vee \\ \pi[h] = 3 \wedge \pi'[h] = 0 \wedge x' & \end{pmatrix},$$

where we use the primed notation to describe variables after the transition.

The justice requirements of MuxSem are

$$\{\neg(\pi[i] = 1 \wedge x), \pi[i] \neq 2, \pi[i] \neq 3 \mid i = 1, \ldots, N\}$$

and the compassion requirements are

$$\{\langle \pi[i] = 1 \wedge x, \pi[i] = 2 \rangle \mid i = 1, \ldots, N\},$$

indicating that if a process can obtain the semaphore infinitely many times, then eventually it will.

Given a state formula φ and a transition relation ρ, we denote by $\varphi \diamond \rho$ the formula which characterizes the set of all ρ-successors of φ-states. Symmetrically, we denote by $\rho \diamond \varphi$ the formula characterizing all the ρ-predecessors of φ-states. We can generalize these one-step successor and predecessor transformers to ρ-paths taking zero or more ρ-steps. Thus, we denote by $\varphi \diamond \rho^*$ the formula characterizing the states reachable form a φ-state by zero or more ρ-steps. Similarly, $\rho^* \diamond \varphi$ denotes the states from which we can reach a φ-state in zero or more ρ-steps.

2.3. Properties

We consider linear time temporal logic (LTL) [48] properties. The safety properties we are interested in are of the form

$$\forall i_1, \ldots, i_k : \Box p(i_1, \ldots, i_k),$$

where $i_1, \ldots, i_k$ are all mutually distinct (auxiliary) $\mathbf{type}_1$ variables and $p(i_1, \ldots, i_k)$ is an assertion. (The temporal operator $\Box$, read "always" or "box", means "from this point onward". Thus, the formula above means that $p(\vec{i})$ holds in all reachable states of the system.)

For example, the safety property of MuxSem is $\forall i, j : \Box\neg(\pi[i] = 2 \wedge \pi[j] = 2)$.

The liveness properties we are interested in are either *individual liveness* properties, or simply *liveness* properties, of the form

$$\forall i_1, \ldots, i_k : p(i_1, \ldots, i_k) \Rightarrow \diamond q(i_1, \ldots, i_k),$$

where q and p are assertions. (The temporal operator $\diamond$, read "eventually" or "diamond", is the dual of $\Box$. The connective $\Rightarrow$ stands for "entails". Thus, $p \Rightarrow \diamond q$ is an abbreviation for the formula $\Box(p \rightarrow \diamond q)$, i.e., "it is always the case the p implies eventually q.") We sometimes study *livelock freedom* (group liveness) properties of the form

$$\exists i_1, \ldots, i_k : p(i_1, \ldots, i_k) \Rightarrow \diamond \exists i_1, \ldots, i_k : q(i_1, \ldots, i_k).$$

For example, the liveness property of MUXSEM is $\forall i : (\pi[i] = 1 \Rightarrow \Diamond(\pi[i] = 2))$, and the livelock property of MUXSEM is $(\exists i : \pi[i] = 1) \Rightarrow \Diamond(\exists i : \pi[i] = 2)$.

2.4. Extending the FBDS model

The model described above is a *single parameter* model, with a rather simple structure. There are systems that have several parameters, or systems that allow for larger variety of array types, e.g., $[1..N] \mapsto [1..N]$ arrays. To accommodate such systems, we can extend the model to allow for types $\mathbf{type}_1, \ldots, \mathbf{type}_m$ to be a set of scalar data types, where each $\mathbf{type}_i$ includes integers in the range $[1..N_i]$ for some $N_i \in \mathbb{N}^+$. The systems then include variables that are either $\mathbf{type}_i$ scalars, for some $i \in \{0, \ldots, m\}$, arrays of the type $\mathbf{type}_i \mapsto \mathbf{type}_j$ for $1 \leqslant i < j \leqslant m$, or arrays of the type $\mathbf{type}_j \mapsto \mathbf{bool}$. For a system that includes types $\mathbf{type}_1, \ldots, \mathbf{type}_k$, we refer to $N_1, \ldots, N_k$ as the *system's parameters*. Systems are distinguished by their *signatures*, which determine the types of variables allowed, as well as the assertions allowed in the transition relation and the initial condition. Whenever the signature of a system includes the type $\mathbf{type}_i \mapsto \mathbf{type}_j$, we assume by default that it also includes the types $\mathbf{type}_i$ and $\mathbf{type}_j$.

As before, *atomic formulae* may compare two variables of the same type, and *Formulae*, used in the transition relation and the initial condition, are obtained from the atomic formulae by closing them under negation, disjunction, and existential quantification, for appropriately typed quantified variables.

Many of the results reported here can be extended to this generalized model. However, since the goal of this paper is to give an overview of the methods, we prefer to restrict to the simpler model detailed above. The method of invisible invariants for the generalized model is discussed in [33].

3. The method of invisible auxiliary constructs

As mentioned in the Introduction, a naive application of model checking is not suitable for uniform verification of parameterized systems, which leaves us with deductive tools. The most commonly used deductive proof rules for both safety and liveness properties require the user to (1) come up with some auxiliary assertions, and (2) verify that some premises, which use the auxiliary assertions, are met. The method of invisible auxiliary constructs has two parts. The first automatically generates candidate auxiliary assertions. The second is a small model theorem, that establishes that, under certain restrictions, it suffices to prove the premises of step (2) for small instantiations. The two parts together allow generation of the auxiliary constructs and automatic validation of the premises using BDD techniques.

For the cases when the constructs obtained result in premises that can be validated, the method offers a sound (but, necessarily, incomplete) uniform verification methodology of parameterized systems. The success of the method depends on the heuristic used in the generation of the auxiliary constructs, and on whether the premises that need to be validated can be cast in a syntactic form which is amenable to the small model method. The term "invisible" is to emphasize that the auxiliary constructs generated need not be visible to the user—they are automatically embedded in the premises and checked. In fact, being generated by symbolic BDD techniques, the constructs' representation is

often extremely unwieldy and non-intuitive, and will usually not contribute to a better understanding of the system or its correctness proof.

3.1. The small model theorem

Consider a (single parameter) FBDS S, and let

$$\varphi : \forall i_1, \ldots, i_\ell \exists j_1, \ldots, j_k : Q(\vec{i}, \vec{j}) \tag{3}$$

be a premise whose validity we wish to check for all instants of S, where Q is an assertion.

The small model theorem computes a *cutoff* value, N_0, and shows that Eq. (3) is valid over all instantiations if it is valid over all instantiations of size N_0 or less. The cutoff number N_0 is computed by

$$\ell + \{\textit{number of } \mathbf{type}_1 \textit{ variables in } Q\},$$

where, in $\mathbf{type}_1$ variables, we also include constants such as 1 or N.

Example 2. Consider MUXSEM. Let Ψ be the formula $\forall i, j : \psi(i, j)$, where $\psi(i, j) : i \neq j \wedge at\_\ell_2[i] \rightarrow \neg at\_\ell_2[j] \wedge x = 0$, asserting that when a process is at the critical (or exit) section, no other process is in the critical section and the semaphore is not available. Suppose we want to show that Ψ is inductive over MUXSEM, i.e., that $\chi : \Psi \wedge \rho \rightarrow \Psi'$ is valid. Putting χ in prenex normal form, leaving only disjunctions, conjunctions, and pushing negations inside, results in the formula

$$\forall h, k, l : \exists i, j, t : \psi(i, j) \wedge R(h, t) \rightarrow \psi'(k, l),$$

which is a $\forall\exists$ formula with three universally quantified variables. The small model property then claims that in order to validate χ it suffices to do so for instantiations of MUXSEM of size three or less.

The theorem was first stated and proved in [7]. It was extended in [33] to the multi-parameter case. In both, the statement of the theorem is somewhat more restrictive than the one here, the proofs, however, apply to this version as well.

3.2. PROGEN: automatic generation of auxiliary invariants

With the small model theorem at hand, it remains to automatically generate the auxiliary constructs that will be used in the premises (whose form should conform to the small model theorem). The technique PROGEN is a heuristics to automatically produce $\forall$-formulae (again, assuming prenex normal form) for parameterized systems. In this section we describe the technique. In the next sections, we discuss how the technique can be used to produce formulae that are useful in safety and liveness proofs.

It is often the case that the auxiliary construct for a parameterized system is intended to capture the set of ϕ-states for some formula ϕ, which can be computed precisely for any concrete value of the parameter $N = N_0$. A case in point is the set of reachable states, which can be computed precisely for any specific N. It is also often the case that it suffices to use an auxiliary construct of the form $\forall i_1, \ldots, i_\ell : q(\vec{i})$ where the i_j's are mutually distinct. PROGEN attempts to construct a

$\forall$-formula that (possibly over-) approximates the set of ϕ-states for any value of N. Its outlines are:

- *instantiate* the system for parameter $N_0 = \ell + H + b$ where H is the number of existentially quantified variables in the transition relation ρ, and b is the number of $\mathbf{type}_1$ variables in the system;
- *compute* (using symbolic BDD techniques) the set of ϕ-states in the instantiation;
- *project* the set of states obtained on processes $P[1], \ldots, P[\ell]$ by discarding references to all other processes (and variables which are local to them). As part of the projection, we retain in $\mathbf{type}_1$-variables their precise value if it equals to one of $1, \ldots, \ell$. Otherwise, we just record that their value is different from all of $1, \ldots, \ell$;
- *generalize* by replacing each reference to a local variable $P[j].x, 1 \leqslant j \leqslant \ell$, by a reference to $P[i_j].x$. Also, for each $\mathbf{type}_1$-variable v, we generalize $v = j$ (or $v \neq j$) into $v = i_j$ (or $v \neq i_j$). This generalization yields $q(i_1, \ldots, i_\ell)$;
- return $\forall i_1, \ldots, i_\ell : q(\vec{i})$.

When $\ell = 1$, the projection and generalization steps are:

- let ψ_1 be the assertion obtained from ϕ by projecting away all references to $type_1$ values other than 1;
- let $\psi(i)$ be the assertion obtained from ψ_1 by *generalization*, which involves the following transformations: Replace every reference to a $z[1]$ by a reference to $z[i]$, and for every $\mathbf{type}_1$ variable y and $\sim \in \{=, \neq, >\}$, replace any sub-formula of the form "$y \sim 1$", by the formula "$y \sim i$".

Thus, for $\ell = 1$, ProGen returns

$$\forall i : \phi_1[i/1]$$

(where, for a formula q, ϕ_1 denotes the projection of ϕ over process 1), and in the general case ProGen returns

$$\forall i_1, \ldots, i_\ell : \phi_{c_1, \ldots, c_\ell}[i_1, \ldots, i_\ell / c_1, \ldots, c_\ell],$$

where the i_j's are mutually distinct, and the c_j's are mutually distinct constants in the range $[1..N_0]$.

The choice of ℓ can be determined by the user, or, alternatively, the system can start with $\ell = 1$, and successively build auxiliary constructs, until either one that satisfies the requirements (thus, allows the premises to be validated) is obtained, or N_0 becomes too large.

3.3. Proving safety: the method of invisible invariants

The most commonly used proof deductive proof rule for a safety property $\Box p$ is the rule INV presented in Fig. 2. Premise I1 claims that the initial state of the system satisfies φ. Premise I2 claims that φ remains invariant under ρ. An assertion φ satisfying premises I1 and I2 is called *inductive*. In rare cases, the property p is already inductive. In all other cases, the deductive verifier has to invent the auxiliary assertion φ and establish the validity of premises I1–I3.

In most cases, the auxiliary assertion φ is a $\forall$-assertion. Checking I1–I3 on such a φ requires validating $\forall$-properties (for I1 and I3) and an $\forall\exists$-property (for I2). Hence, we can use ProGen

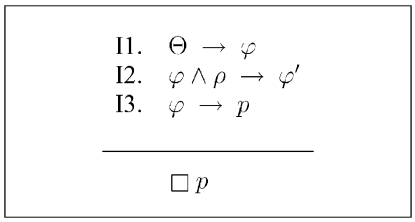

Fig. 2. The invariance Rule INV.

(starting with the set of reachable states) to generate φ, and the small model theorem to establish the premises.

Example 3. We show the results of the procedure for MUXSEM. We start with $\ell = 1$, searching for an auxiliary assertion of the form $\forall i : \psi(i)$. We thus instantiate the system for $N_0 = 2$ and obtain:

(1) The set of reachable states of MUXSEM (2) is

$$reach^2 : \begin{pmatrix} x = 1 \wedge at\_\ell_{0,1}[1] \wedge \neg at\_\ell_{0,1}[2] & \vee \\ x = 0 \wedge at\_\ell_{0,1}[1] \leftrightarrow \neg at\_\ell_{0,1}[2] & \end{pmatrix},$$

(2) $reach_1^2[i/1] : (x = 1) \rightarrow at\_\ell_{0,1}[i]$.

Unfortunately, $\forall i : reach_1^2[i/1]$ is not inductive (intuitively, this is because it provides no information about the location of i when $x = 0$). We then search for an auxiliary assertion of the type $\forall i \neq j : \psi(i, j)$. We instantiate the system to $N_0 = 3$ and obtain:

(1) $reach^3 : (at\_\ell_{2,3} + at\_\ell_{2,3}[2] + \pi[3] \in \{2, 3\} + x) = 1$.
In this presentation, expressions such as $at\_\ell_{2,3}[i]$ should be interpreted arithmetically, yielding 1 if true and 0 otherwise. This assertion compactly states that either $x = 1$ and then no process is in $\{2, 3\}$, or $x = 0$ and then precisely one process is in $\{2, 3\}$.

(2) $reach_{1,2}^3[i, j/1, 2] : \begin{bmatrix} at\_\ell_{2,3}[i] \rightarrow x = 0 \wedge \neg at\_\ell_{2,3}[j] & \wedge \\ at\_\ell_{2,3}[j] \rightarrow x = 0 \wedge \neg at\_\ell_{2,3}[i] & \end{bmatrix}$.

(3) $\forall i \neq j : reach_{1,2}^3[i, j/1, 2]$ does satisfy I1–I3 for MUXSEM (3). Thus, the formula $\forall i \neq j$: $at\_\ell_{2,3}[i] \rightarrow (x = 0 \wedge at\_\ell_{0,1}[j])$ is an inductive invariant for the program which implies the desired safety property.

We can therefore conclude that MUXSEM satisfies its safety properties for any instantiation.

The method of invisible invariants was successfully applied to a variety of protocols, including a 3-stage pipeline [49,50], German's cache [51,7], the Illinois' Cache Algorithm [52,53], Szymanski's mutual exclusion algorithm [54], the Bakery algorithm, and Peterson's N-process mutual exclusion algorithm (which requires some extension of the FBDS model since the arrays are not stratified).

For a system with a set of states Σ
justice requirements $J_1, \ldots, J_m$
invarinat assertion φ,
assertions $p, q, pend, h_1, \ldots, h_m$,
and ranking functions $\delta_1, \ldots, \delta_m : \Sigma \to \{0, 1\}$

D1.	$p \wedge \varphi$	$\longrightarrow$	$q \vee pend$
D2.	$pend \wedge \rho$	$\longrightarrow$	$q' \vee pend$
D3.	$pend$	$\longrightarrow$	$\bigvee_{i=1}^{m} h_i$
D4.	$pend \wedge \rho$	$\longrightarrow$	$q' \vee \bigwedge_{j=1}^{m} \delta_j \geq \delta'_j$
For every $i = 1, \ldots, m$			
D5.	$pend \wedge h_i \wedge \rho$	$\longrightarrow$	$q' \vee h'_i \vee \delta_i > \delta'_i$
D6.	h_i	$\longrightarrow$	$\neg J_i$

$$p \Longrightarrow \Diamond q$$

Fig. 3. The liveness rule dis-rank.

3.4. The method of invisible ranking

Deductive rules for proving liveness properties of the form $p \Rightarrow \Diamond q$ usually call for the user to supply a well-founded domain W and a ranking function from pending states (that are on p-to-q paths) to elements of W. The premises then establish that every pending state leads either to the goal q or to a state whose rank is no higher, and that, from fairness considerations, no computation can indefinitely remain in non-q states of the same rank.

We present such a proof rule, dis-rank (for *distributed ranking*) in Fig. 3. This rule assumes that the system to be verified has only justice requirements (in particular, it has no compassion requirements). Since we are dealing with parameterized systems, we assume that the justice requirements are parameterized, so that each justice requirement is associated with some transition in some process. Suppose there are m justice requirements (usually, $m = N \times c$, where N is the system's parameter and c is a constant that is bounded by the number of locations of each process). The rule dis-rank uses the well-founded domain $(\{0,1\}^m, \succcurlyeq)$, where for $r_1, r_2 \in \{0,1\}^m$, $r_1 \succcurlyeq r_2$ if the number of 1's in r_1 is greater or equal to the number of 1's in r_2. (Note that this choice relieves the user from the need to define the well-founded domain.) The user has to come up with m assertions, $h_1, \ldots, h_m$ as well as m ranking functions $\delta_1, \ldots, \delta_m$.

Each h_i includes a set of states that violate justice requirement J_i. They are intended to identify the states under which J_i is "helpful", i.e. the rank δ_i is expected to decrease when J_i is next satisfied. Premise (D1) states that every reachable p-state is either also a q-state, or it is in a *pend*-state. Premise (D2) states that every *pend*-state leads to either a q-state or to a *pend*-state. Thus, (D1) and (D2) show that *pend* is inductive over the set of pending states, i.e., that every pending state satisfies *pend*. Premise (D3) states that every *pend*-state (and, therefore, every pending state) belongs to some helpful set. Premise (D4) states that every pending state leads to either a q-state or to a state that has no higher δ_j ranking. Premise (D5) shows that each pending h_i-state leads either to a q state, or to a h_i state, or to a state whose δ_i is lower (i.e., δ_i of the source state is 1, and δ_i of

the target state is 0). Finally, Premise (D6) states that every h_i-state is J_i-violating. The soundness of dis-rank follows from the observations that (1) a computation cannot remain indefinitely in an h_i-state (since they are J_i-violating); (2) each time a computation leaves a h_i-state, it either enters a q-state or one of the ranks decreases, and (3) ranks can only decrease m times. Hence, every computation that starts with a *pend*-state must eventually enter a q-state.

Suppose we have a finite-state (non-parameterized) system, or a given instantiation of a parameterized system, and a liveness property $p \Rightarrow \Diamond q$ whose validity over the system we wish to establish by an automatic application of dis-rank. Let *reach* be the set of reachable states of the system, i.e., $reach = \Theta \Diamond \rho^*$. Obviously, we can add *reach* to the left hand side of every premise of dis-rank. The set *pend* defined by

$$pend = (p \wedge \neg q \wedge reach) \Diamond (\rho \wedge \neg q')^*.$$

Thus, *pend* includes all the states which can be reached from a reachable $(p \wedge \neg q)$-state by a q-free path.

To compute the h_j's, we start with *pend*, remove strongly connected components (SCCs) that violate J_j and include them in h_j, until all pending states are processed: The actual computation of the h_j's is based on the following analysis: assume that ζ is an assertion characterizing a set of states, and let J be some justice requirement. We wish to identify the subset of states ϕ within ζ from which any transition that leads to a J-state leads outside of ζ. Consider the fix-point equation

$$\phi = \zeta \wedge \neg J \wedge AX(\phi \vee \neg\zeta). \tag{4}$$

The equation states that every ϕ-state must satisfy $\zeta \wedge \neg J$, and that every successor of a ϕ-state is either a ϕ-state or lies outside of ζ. In particular, all J-satisfying successors of a ϕ-state do not belong to ζ. By taking the maximal solution of this fix-point equation, denoted $\nu\phi.(\zeta \wedge \neg J \wedge AX(\phi \vee \neg\zeta))$, we compute the subset of states which are helpful for J within ζ. Starting with $\zeta = pend$, the algorithm iteratively computes the fix-point equation for every justice set, includes the computed ϕ in the corresponding h_j, and removes it from *pend*. It terminates when *pend* is empty. The order in which SCCs are removed is not necessarily deterministic. In the worst case, every possible ordering can be attempted until the h_j's include all of *pend*. See [34,35] for details.

As for the δ_j's, note that δ_j should equal 1 on every h_j-state and should decrease from on any transition that takes from an h_j state to an $\neg h_j$-state. Furthermore, δ_j can never increase. It follows that δ_j should equal 1 on every pending state from which there exists a pending path to a pending state satisfying h_j. Thus, we compute $\delta_j = pend \wedge ((\neg q) E \mathcal{W} h_j)$, where $E\mathcal{W}$ is the "existential-until" CTL operator. This formula identifies all states from which there exists an q-free path to an h_j-state.

It is easy to see that if the procedure that defines the h_i's terminates successfully (processing all pending states), the constructed h_i and δ_i's meet the premises D1–D6.

Suppose now that we have a parameterized system S for which we want to establish a liveness property $\forall i : p(i) \Rightarrow \Diamond q(i)$. Assuming the system is fully symmetric, we show the claim for $i = 1$ and from this conclude it holds for every i. Assume that each process $P[i]$ has k justice requirements $J_1[i], \ldots, J_k[i]$. Then, to apply dis-rank, $m = k \times N$. Fix some instantiation $S(N_0)$ of the system and generate the concrete $reach, pend, h_1, \ldots, h_m$, and $\delta_1, \ldots, \delta_m$ as per the procedure above. We next

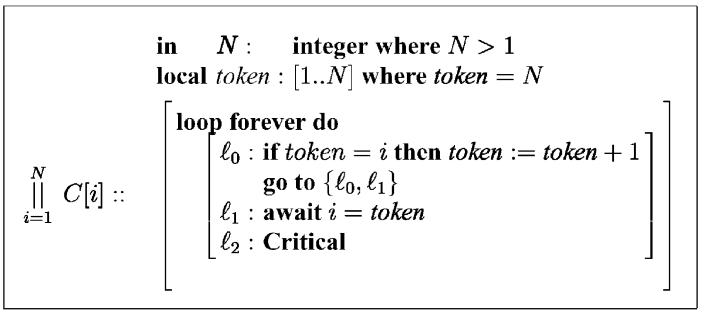

Fig. 4. Parameterized mutual exclusion Algorithm TOKENRING.

compute:

(1) φ: the invisible invariant described in Section 3.3. Assume that φ is universally quantified with $i_1, \ldots, i_j$.
(2) $pend^a$, the set of pending states, is to be computed by taking *pend*, and projecting and generalizing it separately for 1 and for all processes $j \neq 1$. Thus, $pend^a = pend^a_{=1} \wedge pend^a_{i \neq 1}$, where $pend^a_{=1} = pend[1 \mapsto 1]$, and $pend^a_{\neq 1}(i) = pend[1 \mapsto 1, i_0 \mapsto i]$ for some $i_0 \in [2..N_0]$.
(3) For $\ell = 1, \ldots, k$, let $h_\ell[i]$ denote the helpful set obtained by the procedure above for the ℓth justice property associated with process i. Again, we use PROGEN, and compute $h^a[i]$ separately for $i = 1$ and $\neq 1$.
(4) Similarly, for $\ell = 1, \ldots, k$, we use PROGEN to compute $\delta_\ell[i]$ separately for $i = 1$ and $\neq 1$.

Note that φ and $pend^a$ are both $\forall$-formulae, the transition relation is a $\exists\forall$-formula, and all other sets are represented by unquantified expressions. Thus, all premises of dis-rank fall now under the scope of the small model theorem and can be checked by BDD techniques.

3.5. Example: token ring

Since MUXSEM contains compassion requirements, we cannot, without further ado, use it for an example of the method. We therefore take another system, TOKENRING, whose purpose is also to provide a mutual exclusion protocol. The processes, however, are arranged in a ring, and numbered, in order, 1 to N. A single $\mathbf{type}_1$ variable, *token*, is passed around the ring. An idle process that has the token moves it on (by incrementing the token, where N is "incremented" to 1). A trying process that has the token (i.e., the token has the id of the process) enters the critical section, and processes from the critical section return to be idle. The code is presented in Fig. 4.

The statement in location ℓ_0, is for the processes in their non-critical (idle) section, who may decide to stay at there or move on to location ℓ_1 (trying). In parallel, if an idle process receives the token (as signaled by $token = i$), it passes it on to right neighbor. A process may exit its non-critical section only if it currently does *not* hold the token. The justice requirements associated with the system are $\{J_0[i], J_1[i], J_2[i] \mid i = 1, \ldots, N\}$, where $J_0[i] : \neg(at\_\ell_0 \wedge token = i)$, $J_1[i] : \neg(at\_\ell_1[i] \wedge token = i)$, and $J_2[i] : \neg at\_\ell_2[i]$.

The liveness property we want to establish is $\forall i : at\_\ell_1[i] \Rightarrow \Diamond at\_\ell_2[i]$. We follow our guidelines and focus on $C[1]$, that is, we attempt to show that $at\_\ell_1[1] \Rightarrow \Diamond at\_\ell_2[1]$.

To prove the liveness property, we instantiated the system to $N_0 = 6$ which is the bound called for by the small model theorem. The set of reachable states is given by $reach = \bigwedge_{j=1}^{6}(at\_\ell_{0,1}[j] \vee tloc = j)$. Applying the method of invisible invariants (taking $i_0 = 3$ for the projection), we obtain $\varphi^a(i) : at\_\ell_{0,1}[i] \vee tloc = i$. Similarly, $pend = (reach \wedge at\_\ell_1[1])$. To compute the abstract $pend^a$, we obtain $pend^a_{=1} = pend[1 \mapsto 1] = at\_\ell_1[1]$ and $pend^a_{\neq 1}(i) : at\_\ell_1[1] \wedge (at\_\ell_{0,1}[i] \vee tloc = i)$.

The concrete helpful assertions are $h_1[1] = pend \wedge token = 1$; $h_0[1] = h_2[1] = \text{F}$, and for every $i = 2, \ldots, 6$ and $\ell = 0, 1, 2, h_\ell[i] : (pend \wedge [i] = \ell \wedge token = i)$. For the abstract helpful assertions, we obtain:

	$h^a_\ell[1]$	$h^a_\ell[i], i > 1$
$\ell = 0$	0	$at\_\ell_1[1] \wedge at\_\ell_0[i] \wedge tloc = i$
$\ell = 1$	$at\_\ell_1[1] \wedge tloc = 1$	$at\_\ell_1[1] \wedge at\_\ell_1[i] \wedge tloc = i$
$\ell = 2$	0	$at\_\ell_1[1] \wedge at\_\ell_2[i] \wedge tloc = i$

The concrete $\delta_\ell[j]$'s are similar to the abstract ones, which are described by

$$\left.\begin{array}{l} \delta^a_0[1] = \delta^a_2[1] : 0, \\ \delta^a_1[1] : at\_\ell_1[1], \\ \delta^a_0[i] : at\_\ell_1[1] \wedge (1 < tloc < i \wedge at\_\ell_{0,1}[i] \vee tloc = i) \\ \delta^a_1[i] : at\_\ell_1[1] \wedge (1 < tloc < i \wedge at\_\ell_{0,1}[i] \vee tloc = i \wedge at\_\ell_1[i]) \\ \delta^a_2[i] : at\_\ell_1[1] \wedge (1 < tloc < i \wedge at\_\ell_{0,1}[i] \vee tloc = i \wedge at\_\ell_{1,2}[i]) \end{array}\right\} \quad \text{for } i > 1.$$

Having computed internally the auxiliary constructs, and checking the invariance of φ, it only remains to check that the six premises of rule dis-rank are all valid for any value of N. Here we use the small model theorem which allows us to check their validity for all values of $N \leqslant N_0$ for the cutoff value of N_0 which is specified in the theorem. First, we have to ascertain that all premises have the required AE 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 and D6 that, unlike others, do not include any inductive component.

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

$$((\forall a.\, pend(a)) \wedge (\exists i, i_1 \forall j, j_1.\psi(i, i_1, j, j_1))) \rightarrow q' \vee (\forall c.\, pend(c)),$$

which is logically equivalent to

$$\forall i, i_1, c : \exists a, j, j_1 : (pend(a) \wedge \psi(i, i_1, j, j_1) \rightarrow q' \vee pend(c)).$$

The $\mathbf{type}_1$ variables which are universally quantified or appear free in this formula 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 $N \leqslant N_0$ and check that they are valid, using BDD symbolic methods.

4. Automatic verification by counter abstraction

Assume a parameterized system $\mathscr{S} = (V, \Theta, \rho, \mathscr{J}, \mathscr{C})$. Following the method of *finitary abstraction* of [16], let V_A be a set of *abstract variables* and let $\mathscr{E}^\alpha$ be a set of expressions. The equation $V_A = \mathscr{E}^\alpha(V)$ defines an *abstraction mapping* α which maps each concrete state to a corresponding abstract state. For an assertion p, we define $\alpha^+(p)$ to be the set of abstract states that have *some* p-states abstracted into them, that is, the abstract state S^A belongs to $\alpha^+(p)$ if *some* p-state is mapped into S^A. Similarly, $\alpha^-(p)$ is the set of abstract states such that *all* states abstracted into them are p-states, that is, S^A belongs to $\alpha^-(p)$ if *all* concrete states mapped into S^A satisfy p. Both α^- and α^+ can be generalized to temporal formulae. For a transition relation R, define $\alpha^{++}(R) = \exists V, V' : V_A = \mathscr{E}^\alpha(V) \wedge V'_A = \mathscr{E}^\alpha(V') \wedge R(V, V')$.

Then $\mathscr{S}^\alpha$, the *abstraction of* $\mathscr{S}$ *with respect to* α, is the system

$$\mathscr{S}^\alpha = (V_A, \alpha^+(\Theta), \alpha^{++}(\rho), \{\alpha^+(J) \mid J \in \mathscr{J}\}, \{\langle \alpha^-(p), \alpha^+(q)\rangle \mid \langle p, q\rangle \in \mathscr{C}\}).$$

It is shown in [16] that if $\mathscr{S}^\alpha \models \alpha^-(\psi)$ then $\mathscr{S} \models \psi$ for every temporal formula ψ. Thus, in order to verify that $\mathscr{S} \models \psi$, it suffices to show that $\mathscr{S}^\alpha \models \alpha^-(\psi)$. When $\mathscr{S}$ is a parameterized system and $\mathscr{S}^\alpha$ is a finite-state system, this finitary abstraction offers an efficient and attractive tool for verification. Of course, it still leaves the user with the burden of choosing the appropriate α.

As mentioned in the introduction, *counter-abstraction* has been one of the oldest abstractions for parameterized systems: Suppose that the control variables of $\mathscr{S}$'s processes ($\pi[i]$) range over $0, \ldots, L$, and that the (finite-domain) shared variables are $x_1, \ldots, x_b$. For simplicity, we assume first that the processes do not have any local variables. Thus, the concrete state variables are N, $\pi[1..N]$, and $x_1, \ldots, x_b$. Define a *counter abstraction* α, where the abstract variables are $\kappa_0, \ldots, \kappa_L \in \{0, 1, 2\}$, $X_1, \ldots, X_b$ and the abstraction mapping $\mathscr{E}^\alpha$ is given by

$$\kappa_\ell = \left\{ \begin{array}{ll} 0 & \text{if } \forall i : [1..N] : \pi[i] \neq \ell \\ 1 & \text{if } \exists i_1 : [1..N] : \pi[i_1] = \ell \wedge \forall i_2 \neq i_1 : \pi[i_2] \neq \ell \\ 2 & \text{otherwise} \end{array} \right\} \quad \text{for } \ell \in [0..L]$$

$$X_j = x_j \quad \text{for } j = 1, \ldots, b.$$

That is, κ_ℓ is 0 when there are no processes at location ℓ, it is 1 if there is exactly one process at location ℓ, and it is 2 if there are two or more processes at location ℓ.[1]

Since the systems we are dealing with are symmetric, all the processes have initially the same control value. Without loss of generality, assume it is location 0. Thus we have that Θ^α is $\kappa_0 = 2 \wedge \kappa_1 = \cdots = \kappa_L = 0 \wedge X_1 = x_1^0 \wedge \cdots \wedge X_b = x_b^0$, where $x_1^0, \ldots, x_b^0$ denote the initial values of the concrete shared variables $x_1, \ldots, x_b$.

Next, we obtain $\rho^\alpha = \alpha^{++}(\rho)$. This can be computed automatically using a system supporting the logic WS1S, such as TLV[P] [25] or MONA [55]. Alternately, we can follow the more efficient

[1] The upper bound of 2 can be substituted with other positive integers. In particular 1 can be used for most locations.

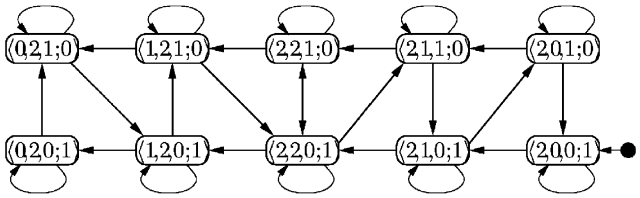

Fig. 5. Counter Abstraction of MUXSEM ($N \geqslant 5$).

procedure sketched below. For every location ℓ in the SPL program that describes a process, assume the instruction at location ℓ is of the form:

$$\ell : \textbf{if } c \textbf{ then } [\vec{x} := f_1(\vec{x}); \textbf{goto } \ell_1] \textbf{ else } [\vec{x} := f_2(\vec{x}); \textbf{goto } \ell_2].$$

Suppose c^α is defined (i.e., that the set of states satisfying c is abstracted into c^α) and that $\ell \neq \ell_1, \ell_2$. For $i = 1, 2$, let τ_i be the formula

$$\kappa_\ell > 0 \wedge \kappa'_\ell = \kappa_\ell \ominus 1 \wedge \kappa'_{\ell_i} = \kappa_{\ell_i} \oplus 1 \wedge \forall j \notin \{\ell, \ell_i\} : \kappa'_j = \kappa_j \wedge x' = f_i(x)$$

where $v' = v \oplus 1$ is an abbreviation for: $v' = \min\{v + 1, 2\}$ and $v' = v \ominus 1$ is an abbreviation for: $v' = \max\{0, v - 1\}$. Then, we include in ρ^α the disjunction $c^\alpha \wedge \tau_1 \vee \neg c^\alpha \wedge \tau_2$.

Example 4. Consider program MUXSEM of Example 1, where locations 2 and 3 are combined (to allow for simpler graphical representation). Fig. 5 presents the FBDS obtained by counter abstracting MUXSEM, where each state displays the values assigned to the abstract variables $\kappa_0, \kappa_1, \kappa_2; X$.

Following the guidelines before, we obtain the following abstract transition relation:

$$\begin{aligned}
&\kappa_0 > 0 \wedge \kappa'_0 = \kappa_0 \wedge \forall j \neq 0 : \kappa_j = \kappa'_j \wedge X' = X && \vee \\
&\kappa_0 > 0 \wedge \kappa'_0 = \kappa_0 \ominus 1 \wedge \kappa'_1 = \kappa_1 \oplus 1 \wedge \kappa'_2 = \kappa_2 \wedge X' = X && \vee \\
&\kappa_1 > 0 \wedge X > 0 \wedge \kappa'_1 = \kappa_1 \ominus 1 \wedge \kappa'_2 = \kappa_2 \oplus 1 \wedge \kappa'_0 = \kappa_0 \wedge X' = 0 && \vee \\
&\kappa_2 > 0 \wedge \kappa'_2 = \kappa_2 \ominus 1 \wedge \kappa'_0 = \kappa_0 \oplus 1 \wedge \kappa'_1 = \kappa_1 \wedge X' = 1 && \vee \\
&\forall j \in \{0, 1, 2\} : \kappa'_j = \kappa_j \wedge X' = X.
\end{aligned}$$

The first two disjuncts capture the transition in location 0-processes may choose to stay there, or progress to location 1. The next two disjuncts capture the transitions from locations 1 and 2, respectively. The last disjunct captures the idle transition of the concrete system, which results in an idle transition of the abstract system. Note that the abstract idle transition implies that the first disjunct (corresponding to a process remaining in location 0) is superfluous.

To make the figure more compact, we removed states and transitions that occur only for small values of N (namely, for $N < 5$). For example, all states where every κ_i is less than two (implying $N < 4$) is removed; so is the transition that leads from $\langle 2, 1, 1; 0\rangle$ into $\langle 1, 2, 1; 0\rangle$ (since it implies that $N = 4$). The initial state is identified by an entry edge.

Establishing the safety property is now trivial, since the counter abstraction of the safety property $\varphi : \forall i \neq j : \Box\neg(\pi[i] = 2 \wedge \pi[j] = 2)$ is $\varphi^\alpha = \Box(\kappa_2 \leqslant 1)$, which trivially holds for MUXSEM$^\alpha$.

4.1. Deriving abstract justice requirements

Consider a single justice requirement $J : (\pi[i] \neq \ell)$, claiming that process i never stays indefinitely at location ℓ. Computing $\alpha^+(J)$ yields $\bigvee_{\ell' \neq \ell} \kappa_{\ell'} > 0$. Thus, if we take $\alpha^+(J)$ to be an abstract justice property, it merely indicates that infinitely many times some processes should be in some location other than ℓ. Such a justice requirement is meaningless since it is often satisfied by every (abstract) state. For example, the only ℓ for which $\alpha^+(\pi[i] \neq \ell)$ is not satisfied by every abstract state in MuxSem$^\alpha$ is 0, for which we have no such justice requirement. Thus, J^α does not introduce any meaningful constraint on computations, and the counter abstraction of a system with the requirement J will be equivalent to the system without this requirement. It follows that any liveness property that is not valid for system MuxSem without the justice requirement J, cannot be proven by an abstraction which abstracts J into $\alpha^+(J)$.

Consider again the justice requirement $J : (\pi[i] \neq \ell)$:

- From the definition of $\kappa, \kappa_\ell = 1$ iff $\exists i : \pi[i] = \ell \wedge \forall j \neq i : \pi[j] \neq \ell$.
- Consider any concrete states s_1 and s_2 for which $\kappa_\ell = 1$. Thus, there exist i_1 and i_2 such that $s_1 \models \pi[i_1] = \ell \wedge \forall j \neq i_1 : \pi[j] \neq \ell$ and $s_2 \models \pi[i_2] = \ell \wedge \forall j \neq i_2 : \pi[j] \neq \ell$. Since our model is an interleaving model, at most one process can change its location in a single transition. Thus, if $\rho(s_1, s_2)$ then it must be the case that $i_1 = i_2$.
- It therefore follows that if $\kappa_\ell = 1$ holds continuously from a certain point, then for some process i, $\pi[i] = \ell$ from that point, which violates the justice property.
- We can therefore conclude that $\kappa_\ell \neq 1$ is an *abstract justice* property.

Generalizing the above reasoning, we can now develop a method by which stronger abstract justice requirements can be derived. Let φ be an abstract assertion, i.e., an assertion over the abstract state variables V_A. We say that *φ suppresses the concrete justice requirement J* if, for every two concrete states s_1 and s_2 such that s_2 is a ρ-successor of s_1, and both $\alpha(s_1)$ and $\alpha(s_2)$ satisfy $\varphi, s_1 \models \neg J$ implies $s_2 \models \neg J$. For example, the abstract assertion $\varphi : \kappa_2 = 1$ suppresses the concrete justice requirement $J : \pi[i] \neq 2$.

The abstract assertion φ is defined to be *justice suppressing* if, for every concrete state s such that $\alpha(s) \models \varphi$, there exists a concrete justice requirement J such that $s \models \neg J$ and φ suppresses J. For example, the assertion $\kappa_2 = 1$ is justice suppressing, because every concrete state s whose counter-abstraction satisfies $\kappa_2 = 1$ must have a single process, say $P[i]$, executing at location 2. In that case, s violates the justice requirement $J : \pi[i] \neq 2$ which is suppressed by φ.

Theorem 1. *Let $\mathcal{S}$ be a concrete system and α be an abstraction applied to $\mathcal{S}$. Assume that $\varphi_1, \ldots, \varphi_k$ is a list of justice suppressing abstract assertions. Let $\mathcal{S}^\alpha$ be the abstract system obtained by following the abstraction recipe above with $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ added to the set of abstract justice requirements. If $\mathcal{S}^\alpha \models \psi^\alpha$ then $\mathcal{S} \models \psi$.*

Thus, we can safely add $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ to the set of justice requirement, while preserving the soundness of the method.

The proof of the theorem is based on the observation that every abstraction of a concrete computation must contain infinitely many $\neg\varphi$-states for every justice suppressing assertion φ. Therefore,

the abstract computations removed from the abstract system by the additional justice requirements can never correspond to abstractions of concrete computations, and it is safe to remove them.

Theorem 1 is very general (not even restricted to counter abstraction) but does not provide us with guidelines for the choice of the justice suppressing assertions. For the case of counter-abstraction, we can provide some practical guidelines, as follows:

G1. If the concrete system contains the justice requirements $\neg(\pi[i]=\ell)$, then the assertion $\kappa_\ell = 1$ is justice suppressing.
G2. If the concrete system contains the justice requirements $\neg(\pi[i]=\ell \wedge c)$, where c is a condition on the shared variables (that are kept intact by the counter-abstraction), then the assertion $\kappa_\ell = 1 \wedge c$ is justice suppressing.
G3. If the concrete system contains the justice requirements $\neg(\pi[i]=\ell)$ and the only possible move from location ℓ is to location $\ell+1$, then the two assertions $\kappa_\ell > 0 \wedge \kappa_{\ell+1} = 0$ and $\kappa_\ell > 0 \wedge \kappa_{\ell+1} = 1$ are justice suppressing.
G4. If the concrete system contains the justice requirements $\neg(\pi[i]=\ell \wedge c)$, where c is a condition on the shared variables, and the only possible move from location ℓ is to location $\ell+1$, then the assertions $\kappa_\ell > 0 \wedge \kappa_{\ell+1} = 0 \wedge c$ and $\kappa_\ell > 0 \wedge \kappa_{\ell+1} = 1 \wedge c$ are justice suppressing.

G1–G4 describe counter abstraction for the most common justice properties. Justice properties that are not covered by G1–G4 may need to be "manually" counter abstracted.

Example 5. We state justice properties of MUXSEM$^\alpha$ according to the above guidelines. Since for MUXSEM we have the justice $\neg(\pi[i]=2)$, and since every move from location 2 leads to location 0, then the assertions $\kappa_2 = 1$, $\kappa_2 > 0 \wedge \kappa_0 = 0$, and $\kappa_2 > 0 \wedge \kappa_0 = 1$ are all justice suppressing and their negation can be added to $\mathscr{J}^\alpha$. Similarly, the concrete compassion requirement $\langle \pi[i]=1 \wedge y, \pi[i]=2\rangle$ implies that the concrete assertion $\neg(\pi[i]=1 \wedge X)$ is a justice requirement for system MUXSEM. We can therefore add $\neg(\kappa_1=1 \wedge X)$ to the abstract justice requirement. Since every move from location 1 leads to location 2, we can also add $\neg(\kappa_1 > 0 \wedge \kappa_2 = 0 \wedge X)$ to the abstract justice requirements.

Note: All justice requirements of MUXSEM$^\alpha$ have been generated automatically by a general procedure implementing all the rules specified by the guidelines above.

4.2. Proving liveness

The liveness property one usually associates with parameterized systems is *individual accessibility* of the form $\forall i : \Box(\pi[i]=\ell_1 \rightarrow \Diamond(\pi[i]=\ell_2))$. Unfortunately, counter-abstraction does not allow us to observe the behavior of an individual process. Therefore, the property of individual accessibility cannot be expressed (and, consequently, verified) in a counter-abstracted system. In Section 4.3, we show how to extend counter-abstraction to handle individual accessibility properties.

There are, however, liveness properties that *are* expressible and verifiable by counter abstraction. Such is the *livelock freedom* property of MUXSEM $\psi : (\exists i : \pi[i]=\ell_1) \Rightarrow \Diamond(\exists i : \pi[i]=\ell_2)$, stating that if *some* process is at location ℓ_1, then eventually *some* process (not necessarily the same) will enter ℓ_2. The counter-abstraction of such a property is $\psi^\alpha : \kappa_{\ell_1} > 0 \Rightarrow \Diamond(\kappa_{\ell_2} > 0)$. Model checking that ψ^α holds over S^α can be accomplished by standard model checking techniques of response properties,

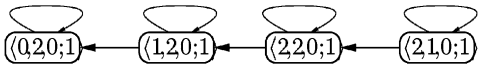

Fig. 6. Reachability Graph for χ.

e.g., the procedure in [56] suggests extracting from the state-transition graph the subgraph of *pending states* and showing that it contains no infinite fair path. A pending state for a property $p \rightarrow \Diamond q$ is any state which is reachable from a p-state by a q-free path.

Example 6. Consider the system MUXSEM$^{\alpha}$ of Fig. 5 and the abstract livelock freedom property $\chi : \kappa_1 > 0 \Rightarrow \Diamond(\kappa_2 > 0)$. In Fig. 6, we present the subgraph of pending states for the property χ over the system MUXSEM.

The way we show that this graph contains no infinite fair path is to decompose the graph into maximal strongly connected components and show that each of them is *unjust*. A strongly connected subgraph (SCC) S is unjust if there exists a justice requirement which is violated by all states within the subgraph. In the case of the graph in Fig. 6 there are four maximal SCCs. Each of these subgraphs is unjust towards the abstract justice requirement $\neg(\kappa_1 > 0 \wedge \kappa_2 = 0 \wedge X)$ derived in Example 5.

We conclude that the abstract property $\Box(\kappa_1 > 0 \rightarrow \Diamond(\kappa_2 > 0))$ is valid over system MUXSEM$^{\alpha}$ and, therefore, the property $\Box(\exists i : \pi[i] = 1 \rightarrow \Diamond(\exists i : \pi[i] = 2))$ is valid over MUXSEM.

4.3. Proving individual accessibility

As indicated before, individual accessibility cannot be directly verified by standard counter abstraction which cannot observe individual processes. To prove individual accessibility for the generic process $P[t]$, we abstract the system by counter abstracting all the processes except for $P[t]$, whom we leave intact. We then prove that the abstracted system satisfies the liveness property (the abstraction of which leaves it unchanged, since it refers to $\pi[t]$ that is kept intact by the abstraction), from which we derive that the concrete system satisfies it as well.

The new abstraction, called "counter abstraction save one", is denoted by γ. As before, we assume for simplicity that the processes possess no local variables except for their program counter. The abstract variables for γ are given by $\kappa_0, \ldots, \kappa_L : [0..2], \Pi : [0..L]; X_1, \ldots, X_b$ and for the abstraction mapping $\mathscr{E}^{\gamma}$ we have $\Pi = \pi[t]$, $X_k = x_k$ for $k = 1, \ldots, b$, and

$$\kappa\ell = \left\{ \begin{array}{ll} 0 & \text{if } \forall r \neq t : \pi[r] \neq \ell \\ 1 & \text{if } \exists r \neq t : \pi[r] = \ell \wedge \forall j \notin \{r,t\} : \pi[t] \neq \ell \\ 2 & \text{otherwise} \end{array} \right\} \quad \text{for } \ell \in [0..L].$$

We obtain ρ^{γ} in the obvious way. For $\mathscr{J}^{\gamma}$, we include all the justice requirements obtained by the guidelines of Section 4.1, with all requirements in $\mathscr{J}$ that relate to $P[t]$. For $\mathscr{C}^{\gamma}$ we take all the requirements in $\mathscr{C}$ that relate only to $P[t]$. To prove $\varphi : \Pi = 1 \Rightarrow \Diamond(\Pi = 2)$, consider the subgraph of $\mathscr{S}^{\gamma}$ that consists of all the abstract states that are reachable from a $(\Pi = 1)$-state by a $(\Pi = 2)$-free path, and show, as before, that this subgraph contains no infinite fair path.

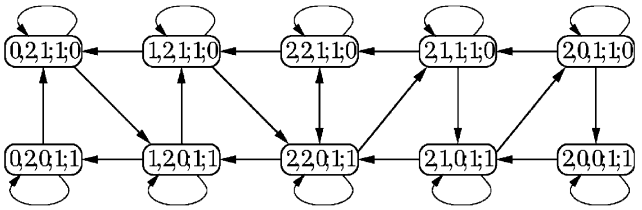

Fig. 7. The states of MUXSEM$^{\gamma}$ which are pending with respect to φ.

Example 7. Consider the system MUXSEM$^{\gamma}$ and the liveness property φ^{γ} given by $\Pi=1 \Rightarrow \Diamond(\Pi=2)$. The subgraph of pending states is presented in Fig. 7. Each state in this graph is labeled by a tuple which specifies the values assigned by the state variables $\kappa_0, \kappa_1, \kappa_2$; Π; X.

Unlike the previous case, this system has the compassion requirement $\langle \Pi=1 \wedge X, \Pi=2\rangle$ associated with $P[t]$. After removing from the graph all states satisfying $\Pi = 1 \wedge X$, it is easy to see that no infinite fair paths are left. We can thus conclude that the abstract property $\Box((\Pi = 1) \rightarrow \Diamond(\Pi = 2))$ is valid over MUXSEM$^{\gamma}$ and, therefore, $\Box((\pi[t] = 1) \rightarrow \Diamond(\pi[t] = 2))$ is valid over MUXSEM.

5. Probabilistic parameterized systems

In this section, we outline some of the recent research in uniform verification of probabilistic parameterized systems. Probabilistic systems are systems that allow explicit coin flipping to determine the result of non-deterministic transitions. Thus, transitions involve probabilistic choices.

We focus on systems where the probability assigned to each choice is fixed and positive. Following common probabilistic arguments (see [57]), it follows that we may restrict our discussion to the case of $k = 2$. Since we are only interested in properties that hold with probability 0/1, we can ignore the exact probabilities that appear in each transition (or, alternatively, assume that all probabilities are $\frac{1}{2}$).

5.1. A model for probabilistic parameterized system

A *probabilistic discrete system* (PDS) $\mathscr{S} : \langle V, \Theta, \rho, \mathscr{P}, \mathscr{J}, \mathscr{C}\rangle$ is defined by letting $V, \Theta, \rho, \mathscr{J}$, and $\mathscr{C}$ be just like in the non-probabilistic case, and letting $\mathscr{P}$ be a finite set of *probabilistic requirements*, each is a triplet (r, t_1, t_2) where r is an assertion and t_1 and t_2 are bi-assertions (i.e., assertions over (V, V')). The bi-assertions t_1 and t_2 can be viewed as partial transition relations. A probabilistic requirement (r, t_1, t_2) describes that there is a probabilistic transition originating at r-state s and resulting in a state s' which is either a t_1-successor or a t_2-successor of s.

In the sequel, we define some properties over computations that depend on the probabilistic transitions made in them. We thus need a mechanism that, given a state s and a successor s', determines whether the transition used in the transition from s to s' is probabilistic, i.e., is consistent with some probabilistic requirement (and, if so, which one), or whether it is non-probabilistic. Consequently, we require that for each such s and s', there is at most one probabilistic requirement that is consistent with a transition from s into s'. More formally, we require that for every s and s' such that s' is a

$$
\begin{array}{l}
\textbf{in} \quad N : \textbf{natural where } N > 1 \\
\textbf{local } y \ : [0..4] \textbf{ where } y = 2 \\
\Big\|_{h=1}^{N} P[h] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} 0 : y := \{\frac{1}{2} : \min(y+1,4), & \frac{1}{2} : \max(y-1,0)\} \\ 1 : \textbf{go to } 0 \end{array}\right] \end{array}\right]
\end{array}
$$

Fig. 8. Up–down: a simple probabilistic parameterized system.

ρ-successor of s, there is at most one probabilistic requirement $(r, t_1, t_2) \in \mathscr{P}$ and one $i \in \{1,2\}$ such that s is an r-state and s' is a t_i-successor of s.

Example 8. Consider the program *up–down* in Fig. 8 in which N processes increment and decrement $y \in [0..4]$.

The meaning of the instruction at location 0 is that, with probability 0.5, y is incremented (up to 4), and with probability 0.5 it is decremented (down to 0). The probabilistic requirements associated with the program are

$$\bigcup_{i=1}^{N} \{(\pi[i] = 0, \pi'[i] = 1 \wedge y' = \min(y+1,4), \pi'[i] = 1 \wedge y' = \max(y-1,0))\}.$$

We define a *computation tree* of $\mathscr{S}$ to be an infinite tree whose nodes are labeled by Σ defined as follows:

- *Initiality*: The root of the tree is labeled by an initial state, i.e., by a Θ-state.
- *Consecution*: Let n be a tree node labeled by $s \in \Sigma$. Then one of the following holds:

(1) n has two children, n_1 and n_2, labeled by s_1 and s_2 respectively, such that for some $(r, t_1, t_2) \in \mathscr{P}$, s is an r-state, and s_1 and s_2 are a t_1- and a t_2-successors of s, respectively.

(2) n has a single child n' labeled by s', such that s' is a ρ-successor of s and for no $(r, t_1, t_2) \in \mathscr{P}$ is it the case that s is an r-state and s' is a t_i-successor of s for some $i \in \{1,2\}$.

Consider an infinite path $\pi : s_0, s_1, \ldots$ in a computation tree. The path π is called *just* if it contains infinitely many occurrences of J-states for each $J \in \mathscr{J}$. It is called *compassionate* if, for each $\langle p, q \rangle \in \mathscr{C}, \pi$ contains only finitely many occurrences of p-states or π contains infinitely many occurrences of q-states. The path π is called *fair* if it is both just and compassionate.

A computation tree induces a probability measure over all the infinite paths that can be traced in the tree, the edges leading from a node with two children have probability 0.5 each, and the others have probability 1. See [58,59] for the exact definition of this measure space.We say that a computation tree is *admissible* if the measure of fair paths in it is 1. Following [58], we say that a temporal property φ is *P-valid* over a computation tree if the measure of paths in the tree that satisfy φ is 1. Similarly, φ is *P-valid over the* PDS $\mathscr{S}$ if it is P-valid over every admissible computation tree of $\mathscr{S}$.

Note that when $\mathscr{S}$ is non-probabilistic, that is, when $\mathscr{P}$ is empty, then the notion of P-validity over $\mathscr{S}$ coincides with the usual notion of validity over $\mathscr{S}$.

5.2. Planner to the aid of P-validity

In order to uniformly verify a probabilistic system, we need to either construct model checkers that can check for P-validity, or to somehow transform the system into a non-probabilistic system that preserves the properties we wish to verify.

There are explicit model checkers for checking P-satisfiability of temporal formulae over finite state probabilistic program [58,59]. For symbolic probabilistic model checking, together with Tamarah Arons, we have recently developed SYMPMC [45] that checks for P-satisfiability of "simple" temporal properties, whose only temporal operators are $\Diamond$ and $\Box$. (The reason for the restriction is that the algorithms for checking P-validity for general temporal formulae requires using a version of LTL that includes the past operators; this is avoided by restricting to the simpler version of LTL.) However, model checkers can only deal with finite-state systems, which parameterized systems are not.

One approach to the problem is to abstract the system into a *probabilistic finite-state system*, and use SYMPMC in order to verify the probabilistic finite-state abstraction. Another approach, which we describe here, is to first transform the system into a *non-probabilistic* parameterized system, and then to verify the non-probabilistic system using coutner abstraction or invisible ranking. The transformation of a parameterized probabilistic system into a non-probabilistic one should be such that if the property is valid over the transformed system, it is P-valid over the original system.

The transformation is accomplished by means of a *Planner*: to transform a probabilistic program Q into a non-probabilistic program Q_T, a *Planner* fixes some $k > 0$, and *occasionally* pre-determines the results of k consecutive "random" choices, allowing the others to be performed in a completely non-deterministic manner. The transformation guarantees that every temporal formula (over Q's variables) that is valid over Q_T, is P-valid over Q. Thus, the planner transformation converts a PDS into an FBDS,[2] and reduces P-validity into validity. The soundness of the approach is established in [45].

The transformation from Q to Q_T can be described as follows: non-probabilistic instructions in Q remain as they are in Q_T. A probabilistic instruction at location ℓ of the form "$\ell : y := \{0.5 : e_1, 0.5 : e_2\}$" in Q is transformed into

$$\textbf{if } consult_\ell > 0 \textbf{ then } \begin{bmatrix} consult_\ell := consult_\ell - 1 \\ y := \textbf{if } planner_\ell \textbf{ then } e_1 \textbf{ else } e_2 \end{bmatrix} \textbf{ else } y := \{e_1, e_2\}.$$

Thus, whenever counter $consult_\ell$ is positive, the program invokes the boolean-valued function $planner_\ell$ which determines whether the program should assign e_1 or e_2 to variable y. Each such "counseled" branch decrements the counter $consult_\ell$ by 1. When the counter is 0, the choice between e_1 and e_2 becomes purely non-deterministic. The function $planner_\ell$ can refer to all available variables. Its particular form depends on the property φ, and it is up to the user of this methodology to design a planner appropriate for the property at hand. This may require some ingenuity and a thorough understanding of the analyzed program.

[2] In fact, had we been considering the more general fair discrete systems, that are FBDSs without the restrictions on the data types, the planner would have transformed a probabilistic fair discrete system into a non-probabilistic one.

Finally, we augment the system with a parallel process, the *activator*, that non-deterministically sets all $consult_\ell$ variables to a constant value k. We equip this process with a justice requirement that guarantees that it replenishes the counters (activates the planners) infinitely many times.

Example 9. Consider Program *up–down* of Example 8, and assume $N=2$. To establish the P-validity of $\varphi : \Box\Diamond(y=0)$, we can take $k=4$ and define both planners to always yield 0, thus consistently choosing the second (decrementing) mode.

Example 9 demonstrates the planner strategy for non-parameterized systems. To extend the approach to parameterized systems, we should design a planner that can be abstracted. For example, if the parameterized system is a good candidate for counter abstraction, then the planner can be also counter abstracted and the resulting transformed system is a "regular" parameterized system that can be automatically verified by counter abstraction.

Example 10. Consider again Program *up–down*, and suppose we wish to verify the P-validity of $\varphi : \Box\Diamond(y=0)$ for any instantiation of $N \geqslant 1$. We can use the same planner as before, taking $k=4$ and defining $planner_0$ to always yield 0, thus consistently choosing the second (decrementing) mode.

The system can be counter-abstracted: each local state can be replicated five times, according to the number of remaining pre-determined choices, resulting in 10 local (countered) states. The counters can be restricted to 0 and 1, using 1 for "one or more", since there is no need here to distinguish between "one" and "more than one." We ran the abstracted system on TLV and successfully verified the property $\Box\Diamond(y=0)$ on the abstract system.

While the planners as described above are hand-crafted, at times it is possible to generate them automatically, with the assistance of SYMPMC: suppose we wish to verify $p \Rightarrow \Diamond q$ where p and q are state-assertions. SYMPMC constructs the graph corresponding to the system and then, in a bottom-up direction, removes maximal strongly connected components (MSCS) that respect all fairness requirements until the graph is empty (or a counter-example is obtained). It is possible to track the execution of SYMPMC and obtain information as to which probabilistic outcomes are "helpful." These can be processed through PROGEN to obtain an automatic planner.

In Program *up–down*, instantiating the system to, say, $N=4$, establishes that the Planner has to force four consecutive "decrement" modes, which leads to the Planner we obtained deductively.

6. Conclusion and future work

The paper reviews some of the recent research of the authors and their colleagues and students in the area of uniform verification of parameterized systems. Two main methodologies are described, the method of auxiliary constructs, and the method of counter abstraction. The method of auxiliary constructs allows for full automatic proofs of both safety and liveness properties of bounded-data parameterized systems, while the method of counter abstraction allows for automatic proofs of such properties for systems that consist of small modules that can communicate with one another. The

paper also shows how the methods can be utilized to aid the uniform verification of parameterized probabilistic systems.

There is no silver bullet for uniform verification of parameterized systems—the problem is undecidable. The most we can hope for is sound tools that will, at times, allow for verification. It is to be expected that new methodologies will be developed, and, hopefully, we will soon have a large enough toolset that will deal with many of the parameterized systems that we encounter.

Acknowledgements

The work reported here is a result of research performed over the past few years with several of our colleagues. In particular, we would like to thank Tamarah Arons, Yi Fang, Yonit Kesten, Nir Piterman, Sitvanit Ruah, and Jessie Xu for contributing to this research. Special thanks are due to the anonymous referees, who have done an excellent job catching numerous flaws in an earlier version of this paper. We would also like to thank the National Science Foundation and the John von Neumann Minverva Center for Verification of Reactive Systems.

References

[1] Apt KR, Kozen D. Limits for automatic verification of finite-state concurrent programs. Information Processing Letters 1986; 22(6): 307–309.
[2] Clarke E, Emerson E. Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proceedings of the IBM Workshop on Logics of Programs, Lecture Notes in Computer Science, vol. 131. Berlin: Springer; 1981. p. 52–71.
[3] Queille J, Sifakis J. Fairness and related properties in transition systems—a temporal logic to deal with fairness. Acta Informatica 1983;19:195–220.
[4] German S, Sistla A. Reasoning about systems with many processes. Journal of ACM 1992;39:675–735.
[5] Emerson E, Namjoshi K. Automatic verification of parameterized synchronous systems. In: Alur R, Henzinger T, editors. Proceedings of the eighth International Conference on Computer Aided Verification (CAV'96), Lecture Notes in Computer Science, vol. 1102. Berlin: Springer; 1996.
[6] Emerson E, Kahlon V. Reducing model checking of the many to the few. In: 17th International Conference on Automated Deduction (CADE-17), 2000. p. 236–55.
[7] Pnueli A, Ruah S, Zuck L. Automatic deductive verification with invisible invariants. In: Proceedings of the seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), vol. 2031. 2001. p. 82–97.
[8] Emerson EA, Namjoshi KS. Reasoning about rings. In: Proceedings of the 22nd ACM Conference on Principles of Programming Languages, POPL'95, San Francisco; 1995.
[9] Kurshan R, McMillan K. A structural induction theorem for processes. Information and Computation 1995;117: 1–11.
[10] Wolper P, Lovinfosse V. Verifying properties of large sets of processes with network invariants. In: Sifakis J, editor. Automatic verification methods for finite state systems. Lecture Notes in Computer Science, vol. 407. Berlin: Springer; 1989. p. 68–80.
[11] Halbwachs N, Lagnier F, Ratel C. An experience in proving regular networks of processes by modular model checking. Acta Informatica 1992;29(6/7):523–43.
[12] Lesens D, Halbwachs N, Raymond P. Automatic verification of parameterized linear networks of processes. In: 24th ACM Symposium on Principles of Programming Languages, POPL'97, Paris; 1997.
[13] Browne M, Clarke E, Grumberg O. Reasoning about networks with many finite state processes. In: Proceedings of the fifth ACM Symposium on Principles of Distributed Computing, 1986. p. 240–8.

[14] Shtadler Z, Grumberg O. Network grammars, communication behaviors and automatic verification. In: Sifakis J, editor. Automatic verification methods for finite state systems. Lecture Notes in Computer Science, vol. 407. Berlin: Springer; 1989. p. 151–65.
[15] Clarke E, Grumberg O, Jha S. Verifying parametrized networks using abstraction and regular languages. In: Sixth International Conference on Concurrency Theory (CONCUR'95), 1995. p. 395–407.
[16] Kesten Y, Pnueli A. Control and data abstractions: the cornerstones of practical formal verification. Software Tools for Technology Transfer 2000;4(2):328–42.
[17] Kesten Y, Pnueli A, Shahar E, Zuck L. Network invariants in action. In: Proceedings of Concur'02, Lecture Notes in Computer Science, vol. 2421. Berlin: Springer; 2002.
[18] Ip C, Dill D. Verifying systems with replicated components in Murφ. In: Alur R, Henzinger T, editors. Proceedings of the eighth International Conference on Computer Aided Verification (CAV'96) Lecture Notes in Computer Science, vol. 1102. Berlin: Springer; 1996.
[19] Jensen E, Lynch N. A proof of Burn's *n*-process mutual exclusion algorithm using abstraction. In: Steffen B, editor. Proceedings of the fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98). Lecture Notes in Computer Science, vol. 1384. Berlin: Springer; 1998. p. 409–23.
[20] Gribomont E, Zenner G. Automated verification of szymanski's algorithm. In: Steffen B, editor. Proceedings of the fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98). Lecture Notes in Computer Science, vol. 1384. Berlin: Springer; 1998. p. 424–38.
[21] Manna Z, Anuchitanukul A, Bjørner N, Browne A, Chang E, Colón M, Alfaro LD, Devarajan H, Sipma H, Uribe T. STeP: the Stanford temporal prover. Technical Report STAN-CS-TR-94-1518, Department of Computer Science, Stanford University, Stanford, California, 1994.
[22] Manna Z, Pnueli A. An exercise in the verification of multi-process programs. In: Feijen W, van Gasteren A, Gries D, Misra J, editors. Beauty is our business. Berlin: Springer; 1990. p. 289–301.
[23] Lesens D, Saidi H. Automatic verification of parameterized networks of processes by abstraction. In: Second International Workshop on the Verification of Infinite State Systems (INFINITY'97), 1997.
[24] Shankar N. Owre S, Rushby J. The PVS proof checker: a reference manual (draft). Technical Report Computer Science Laboratory, Menlo Park, CA: SRI International; 1993.
[25] Kesten Y, Maler O, Marcus M, Pnueli A, Shahar E. Symbolic model checking with rich assertional languages. In: Grumberg O, editor. Proceedings of the Ninth International Conference on Computer Aided Verification (CAV'97). Lecture Notes in Computer Science, vol. 1254. Berlin: Springer; 1997. p. 424–35.
[26] Abdulla P, Bouajjani A, Jonsson B, Nilsson M. Handling global conditions in parametrized system verification. In: Halbwachs N, Peled D, editors. Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99). Lecture Notes in Computer Science, vol. 1633. Berlin: Springer; 1999. p. 134–45.
[27] Jonsson B, Nilsson M. Transitive closures of regular relations for verifying infinite-state systems. In: Graf S, Schwartzbach M, editors. Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), Lecture Notes in Computer Science, vol. 1785. Berlin: Springer; 2000.
[28] Pnueli A, Shahar E. Liveness and acceleration in parameterized verification. In: Emerson A, Sistla PS, editors. Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00). Lecture Notes in Computer Science, vol. 1855. Berlin: Springer; 2000. p. 328–43.
[29] Emerson EA, Sistla AP. Symmetry and model checking. Formal Methods in System Design, vol. 9, No. 1/2; preliminary version appeared in 5th CAV, 1993.
[30] Emerson EA, Sistla AP. Utilizing symmetry when model checking under fairness assumptions. ACM Transactions on Programming Language System, vol. 19, No. 4; preliminary version appeared in 7th CAV, 1995.
[31] Clarke E, Enders R, Filkorn T, Jha S. Exploiting symmetry in temporal logic model checking. Formal Methods in System Design, vol. 9, No. 1/2; preliminary version appeared in 5th CAV, 1993.
[32] Gyuris V, Sistla AP. On-the-fly model checking under fairness that exploits symmetry. In: Grumberg O, editor. Proceedings of the Ninth International Conference on Computer Aided Verification (CAV'97), Lecture Notes in Computer Science, vol. 1254. Berlin: Springer; 1997.
[33] Arons T, Pnueli A, Ruah S, Xu J, Zuck L. Parameterized verification with automatically computed inductive assertions. In: Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), Lecture Notes in Computer Science, vol. 2102. Berlin: Springer; 2001. p. 221–34.

[34] Fang Y, Piterman N, Pnueli A, Zuck L. Liveness with invisible ranking. In: Proceedings of the fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Lecture Notes in Computer Science, vol. 2937. Berlin: Springer; 2004.
[35] Fang Y, Piterman N, Pnueli A, Zuck L. Liveness with incomprehesible 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. Berlin: Springer; 2004, p. 482–96.
[36] McMillan K. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Hu AJ, Vardi MY, editors. Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98). Lecture Notes in Computer Science, vol. 1427. Berlin: Springer; 1998. p. 110–21.
[37] Lubachevsky B. An approach to automating the verification of compact parallel coordination programs. Acta Informatica 1984; 21: 152–69.
[38] Pnueli A, Xu J, Zuck L. The $(0, 1, \infty)$ counter abstraction. In: Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02), Lecture Notes in Computer Science, vol. 2404. Berlin: Springer; 2002; http://www.cs.nyu.edu/~zuck/pubs/cav02.ps.
[39] Baukus K, Lakhnech Y, Stahl K. Verification of parameterized protocols. Journal of Universal Computer Science 2001;7(2):141–58.
[40] Graf S, Saidi H. Construction of abstract state graphs with PVS. In: Grumberg O, editor. Proceedings of the Ninth International Conference on Computer Aided Verification (CAV'97), vol. 1254. Berlin: Springer; 1997. p. 72–83. http://citeseer.nj.nec.com/graf97construction.html.
[41] Lehmann D, Rabin M. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem (exended abstract). In: Proceedings of the Eighth ACM Symposium on Principles of Programming Language, 1981. p. 133–8.
[42] Rabin M. The choice coordination problem. Acta Informatica 1982;17:121–34.
[43] Cohen S, Lehmann D, Pnueli A. Symmetric and economical solutions to the mutual exclusion problem in a distributed system. Theoretical Computer Science 1984;34:215–25.
[44] Zuck L, Pnueli A, Kesten Y. Automatic verification of probabilistic free choice. In: Proceedings of the Third workshop on Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 2294. Berlin: Springer; 2002.
[45] Arons T, Pnueli A, Zuck L. Parameterized verification by probabilistic abstraction. In: Gordon AD, editor. FoSSaCS. Lecture Notes in Computer Science, vol. 2620. Berlin: Springer; 2003. p. 87–102.
[46] Kwiatkowska M, Norman G, Parker D. Prism: probabilistic symbolic model checker. In: TOOLS 2002. Lecture Notes in Computer Science, vol. 2324. Berlin: Springer; 2002.
[47] Manna Z, Pnueli A. Temporal verification of reactive systems: safety. New York: Springer; 1995.
[48] Manna Z, Pnueli A. The temporal logic of reactive and concurrent systems: specification. New York: Springer; 1991.
[49] Burch JR, Dill DL. Automatic verification of pipelined microprocessor control. In: Dill D, editor. Proceedings of the Sixth International Conference on Computer Aided Verification (CAV'94). Lecture Notes in Computer Science, vol. 818. Berlin: Springer; 1994. p. 68–80.
[50] McMillan K. Getting started with SMV. Technical Report, Cadence Berkeley Labs, 1998.
[51] German S. Personal communication, 2000.
[52] Papamarcos M, Patel J. A low-overhead coherence solution for multiprocessors with private cache memories. In: Proceedings of the International Symposium on Shared Memory Multiprocessors (ISCA'84), 1984. p. 348–54.
[53] Delzanno G. Automatic verification of parametrized cache coherence protocols. In: Emerson A, Sistla PS, editors. Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00). Lecture Notes in Computer Science, vol. 1855. Berlin: Springer; 2000. p. 53–68.
[54] Szymanski BK. A simple solution to Lamport's concurrent programming problem with linear wait. In: Proceedings of the 1988 International Conference on Supercomputing Systems, St. Malo, France; 1988. p. 621–6.
[55] Henriksen J, Jensen J, Jørgensen M, Klarlund N, Paige B, Rauhe T, Sandholm A. Mona: monadic second-order logic in practice. In: Brinksma E, Cleaveland WR, Larsen KG, Margaria T, Steffen B, editors. Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95). Lecture Notes in Computer Science, vol. 1019. Berlin: Springer; 1995, also available through http://www.brics.dk/~klarlund/Mona/main.html.

[56] Lichtenstein O, Pnueli A. Checking that finite-state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM Symposium on Principles of Programming Language, 1985. p. 97–107.
[57] Feller W. An introduction to probability theory and its applications, vol. 1, 3rd ed. New York: Wiley; 1968.
[58] Pnueli A, Zuck L. Probabilistic verification. Information and Computation 1993;103(1):1–29.
[59] Vardi M, Wolper P. An automata-theoretic approach to automatic program verification. In: Proceedings of the First IEEE Symposium on Logic in Computer Science, 1986. p. 332–44.