Verification of Parameterized Systems with Combinations of Abstract Domains

Naghmeh Ghafari[1], Arie Gurfinkel[2], and Richard Trefler[1]

[1] David R. Cheriton School of Computer Science, University of Waterloo
[2] Software Engineering Institute, Carnegie Mellon University

Abstract. We present a framework for verifying safety properties of parameterized systems. Our framework is based on a combination of Abstract Interpretation and a backward-reachability algorithm. A parameterized system is a family of systems in which n processes execute the same program concurrently. The problem of parameterized verification is to decide whether for all values of n the system with n processes is correct. Despite well-known difficulties in analyzing such systems, they are of significant interest as they can describe a wide range of protocols from mutual-exclusion to transactional memory. We assume that neither the number of processes nor their statespaces are bounded a priori. Hence, each process may be *infinte-state*. Our key contribution is an abstract domain in which each element (a) represents the lower bound on the number of processes at a control location and (b) employs a numeric abstract domain to capture arithmetic relations between variables of the processes. We also provide an extrapolation operator for the domain to guarantee sound termination of the backward-reachability algorithm. Our abstract domain is generic enough to be instantiated by different well-known numeric abstract domains such as octagons and polyhedra. This makes the framework applicable to a wide range of parameterized systems.

1 Introduction

A parameterized system is a family of systems in which n processes execute the same program concurrently. The problem of parameterized verification is to verify whether for all values of n the system with n processes is correct. Such systems arise naturally in many important applications ranging from communication protocols such as mutual-exclusion and leader election, to distributed systems such as web-services, to cache coherence, resource sharing, transactional memory, and others.

Parameterized system verification is highly undecidable. Apt and Kozen [3] showed that even verification of parameterized systems of finite-state processes is undecidable. This negative result has naturally directed the research in parameterized analysis towards two directions: (i) studying decidability of restricted subclasses (e.g. [17, 16, 15]), and (ii) developing generally applicable but semi-automated proof principles that utilize induction (e.g. [10, 20]). In all of the cases above, it is assumed that each process is *finite-state*.

In this paper, we focus on the analysis of parameterized systems of infinite-state processes. This is a common setting in practice. For example, even in Lamport's bakery protocol [19] each process maintains an integer ticket, and, hence, has an infinite state-space. In this paper, we are interested in a sound, automated, and terminating procedure

for verifying safety properties of such systems. Since this problem is undecidable, such a procedure is necessarily incomplete.

Incomplete, but sound and terminating algorithms are commonly used for reasoning about single-process infinite-state programs. They are typically developed in the framework of Abstract Interpretation [13] (AI). In this paper, we apply such a technique to parameterized systems. We present a framework that combines AI-style reasoning with a backward-reachability algorithm. Our key contribution is an abstract domain in which each element (a) represents the lower bound on the number of processes at a control location and (b) employs a numeric abstract domain to capture arithmetic relations between variables of the processes. Our abstract domain is generic enough to be instantiated by different well-known numeric abstract domains such as octagons [22] and polyhedra [14].

We present an algorithm to over-approximate backward-reachability in a parameterized system using our abstract domain. In its initial form, the algorithm is sound but it is not guaranteed to terminate. We show that there are two reasons for divergence: one comes from the fact that the numeric domain is infinite, and the other is due to the existence of an unbounded number of processes in a parameterized system. We show that it is possible to enforce sound termination of the algorithm by combining numeric widening with a new approximation operator developed especially for our purpose. This results in an algorithm that is incomplete but sound and terminating. That is, if the algorithm does not find an error state, then the system is correct. However, if the algorithm finds an error state, it is uncertain that the error actually is present in the system and is not introduced by the over-approximation. We illustrate an implementation of our algorithm on a variant of Lamport's bakery mutual-exclusion protocol (Alg. 2 in [21]).

Related work. In recent years there has been substantial interest in verification of parameterized systems over a finite (or boolean) data domain. The proposed solutions range from exact model-checking and reachability analysis for restricted classes of systems [16, 15, 17], to generally applicable, sound, but incomplete procedures, e.g., network invariants [20, 11], and regular model checking [18, 6, 5]. Only a handful deal with both an infinite data domain and unbounded parameterization of processes [2, 8, 1, 4, 7].

Abdulla and Jonsson [2] consider the case of 1-clock timed systems. They show that the verification of a class of safety properties is decidable under some restrictions on the constraints used. Inspired by [2], Bozzano and Delzanno [8] present a safety verification technique for parameterized systems with unbounded *local* data variables. Their approach is based on assertions that combine multiset rewriting over first order formulas and constraints. Decidability is achieved by restricting constraints to a constant-free subclass of *difference constraints* (itself a subclass of linear arithmetic). In [1], the method of [8] is extended to GAP constraints. GAP constraints are linear constraints of the form: $x = y$, $x \leq y$, or $x + k < y$, where x and y are variables and k is a *positive* constant.

The method of [8] is generalized into an analysis framework in [7, 4] by using a constrained (multiset) rewriting system on words over an infinite alphabet. In this framework, each configuration is composed from a label over a finite set of symbols and a vector of data in a potentially infinite domain. The constraints are expressed in a logic

that is an extension of a monadic first order theory of the natural ordering on positive integers (corresponding to positions on the word). This logic is also parameterized with a first order theory on the considered data domain such as Presburger arithmetics. In [7, 4] the authors present decidability results for satisfiability of a particular fragment of this logic. They also prove that this fragment is closed under the computations of post- and pre-images. This result together with the decidability of the satisfiability problem can be used for deciding whether a given assertion is an inductive invariant of a system.

In this paper, we present an alternative framework to the multiset rewriting framework of [7, 4]. In our framework, we delegate the reasoning about constraints to Abstract Interpretation. The advantage is two-fold. First, our technique can use any constraints for which there are efficient abstract domains available. Second, the termination of the analysis is guaranteed by combining the widening operator of the abstract domain with a new approximation operator.

Many of the techniques above are based on *counter abstraction* (e.g. [23, 12]). The key idea of this abstraction is to keep track only of the upper bound on the number of processes that satisfy a certain property. For example, the number of processes in the critical section. To ensure that the abstract system is finite-state, the work of [23] restricts the value of counters to either 0, 1 or infinity. In [12], counter abstraction and predicate abstraction are combined together to achieve more flexibility. However, the system model is more restrictive than ours. Our abstract domain PD can be seen as a *variant* of counter abstraction that maintains the *lower* bound on the number of processes satisfying a certain condition.

In contrast to symbolic methods for finite collections of processes with local integer variables [9], our abstract domains are defined over an unbounded collection of variables. The number of variables during the backward-search is not bounded a priori. This allows us to reason about systems with global conditions over any number of processes.

Outline of the paper. The rest of the paper is organized as follows. Syntax and semantics of parameterized systems are defined in Sec. 2. The abstract domain for parameterized systems is introduced in Sec. 3, and is followed by the backward-reachability algorithm in Sec. 4. We discuss techniques to ensure termination of our algorithm and illustrate our algorithm on Lamport's bakery protocol in Sec. 5, followed by concluding remarks in Sec. 6.

2 Parameterized Systems

We describe the system model used in the rest of the paper.

Syntax. A parameterized system $\mathcal{P}$ is a triple (Q, V, T), where Q is a finite set of control locations, V is a finite set of variables, and T is a finite set of guarded commands (or rules). Each $\tau \in T$ is of the form:

$$\tau : q \xrightarrow{g} q' \qquad \text{(guarded command)}$$

where $q, q' \in Q$, and g is a guard. We allow for three types of guards: local, universal global, and existential global that are defined below.

We write V' for the set $\{x' \mid x \in V\}$, and $\texttt{self}.V$ and $\texttt{other}.V$ for the set $\{\texttt{self}.x \mid x \in V\}$ and $\{\texttt{other}.x \mid x \in V\}$, respectively. A *local guard* is an expression

$$\tau_1 : q_1 \xrightarrow{g_1} q_2,\ g_1 : (\texttt{self}.x' = \texttt{self}.x + 1) \wedge (\texttt{self}.y' = \texttt{self}.y)$$

$$\tau_2 : q_1 \xrightarrow{g_2} q_3,\ g_2 : \forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} = q_3) \wedge (\texttt{self}.x' = \texttt{self}.x) \wedge (\texttt{self}.y' = \texttt{self}.y - 2) \wedge (\texttt{other}.x > 0)$$

$$\tau_3 : q_3 \xrightarrow{g_3} q_1,\ g_3 : \exists \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} = q_3) \wedge (\texttt{self}.x' = \texttt{self}.x) \wedge (\texttt{self}.y' = \texttt{self}.y) \wedge (\texttt{other}.y - \texttt{other}.x > 2) \wedge (\texttt{other}.x > 1)$$

Fig. 1. An example of a parameterized system $\mathcal{P}_1 = (\{q_1, q_2, q_3\}, \{x, y\}, \{\tau_1, \tau_2, \tau_3\})$.

on $\texttt{self}.(V \cup V')$ constraining current and next local states of a single process. The *universal* and *existential global guards* are, respectively, expressions of the following form:

$$\forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} = q_o) \wedge \theta \qquad \exists \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} = q_o) \wedge \theta$$

where q_o is a control location in Q, $\texttt{other}.\mathbf{pc}$ is a special variable, and θ is an expression over $\texttt{self}.(V \cup V') \cup \texttt{other}.V$ variables. Intuitively, commands with local guards express how a process behaves independently of other processes in the system, commands with global guards allow a process to reference variables and control locations of the other processes in either universal or existential form. These three types of guarded commands are sufficient to express a wide variety of parameterized systems [1].

An example of a parameterized system where each process manipulates integer variables is shown in Fig. 1. It consists of three commands: τ_1 with a local guard g_1, τ_2 with a universal guard g_2, and τ_3 with an existential guard g_3. Informally, a process executing τ_1 changes its control location from q_1 to q_2, increments local variable x, and does not change local variable y. Similarly, a process executing τ_2 goes from q_1 to q_3 but only if all other currently executing processes are in q_3 and the value of their copies of the variable x are positive. Furthermore, execution of τ_2 decrements the y variable of the current process by 2. Finally, a process executing τ_3 changes its control location from q_3 to q_1 but only if there exists another process that is at q_3 and whose variable x is greater than 1 and the difference between variables y and x of that process is greater than 2. During this transition, variables x and y of the executing process do not change.

We formalize the semantics of parameterized systems using transition systems.

Semantics. A *process state* is a pair (q, v), where $q \in Q$ and v is a valuation assigning values to variables in V. We often treat a process state $u = (q, v)$ as a valuation of variables $V \cup \{\mathbf{pc}\}$ such that $u(\mathbf{pc}) = q$, and $u(y) = v(y)$ for all $y \in V$. An *n-process configuration* is a tuple $\langle u_1, \ldots, u_n \rangle$, where each u_i is a process state. We refer to the first (left-most) process in a configuration as P_1, to the second as P_2, etc, and refer to the number of the process as a process id (PID). So PID of P_1 is 1, PID of P_2 is 2, etc. For two configurations $c_1 = \langle u_1, \ldots, u_n \rangle$ and $c_2 = \langle w_1, \ldots, w_m \rangle$, we use $c_1 \cdot c_2$ to denote their concatenation $\langle u_1, \ldots, u_n, w_1, \ldots, w_m \rangle$.

For an expression θ, we write $\theta[x \leftarrow y]$ for the result of substituting y for x in θ. A valuation σ is a model of an expression θ over V, written $\sigma \models \theta$, if θ is satisfied by σ, i.e., $\theta[x \leftarrow \sigma(x) \mid x \in X]$ is valid. For example, let $\sigma = \{x \mapsto 5, y \mapsto 10\}$, then $\sigma \models (x < y)$, and $\sigma \not\models (x + y = 10)$. For a triple of valuations σ_c, σ_n, and σ_o

over V, we write $(\sigma_c, \sigma_n, \sigma_o)$ for a valuation σ over $\texttt{self}.V \cup \texttt{self}.V' \cup \texttt{other}.V$ defined as $\sigma(\texttt{self}.y) \triangleq \sigma_c(y)$, $\sigma(\texttt{self}.y') \triangleq \sigma_n(y)$, and $\sigma(\texttt{other}.y) \triangleq \sigma_o(y)$. We write (σ_c, σ_n) for short when σ_o is irrelevant.

Let n be a natural number and $\mathcal{P} = (Q, V, T)$ a parameterized system. An n-process instance of $\mathcal{P}$ is a transition system $\mathcal{T}_n(\mathcal{P}) = (C_n, \Delta_n)$, where C_n is the set of all n-process configurations, and $\Delta_n \subseteq C_n \times C_n$ is a transition relation. Intuitively, a pair of configurations c and c' are in Δ_n if c' is reachable from c via an execution of a guarded command by a single process. For each $\tau \in T$ of the form $q \xrightarrow{g} q'$, let Δ_n^τ be defined such that $(c, c') \in \Delta_n^\tau$ iff $c = c_1 \cdot \langle u \rangle \cdot c_2$, $c' = c_1 \cdot \langle u' \rangle \cdot c_2$, and the following holds:

- g is a local guard and $(u, u') \models g$, or
- g is a universal global guard and $\forall u_o \in (c_1 \cdot c_2) : (u, u', u_o) \models g$, or
- g is an existential global guard and $\exists u_o \in (c_1 \cdot c_2) : (u, u', u_o) \models g$.

Then, $\Delta_n \triangleq \bigcup_{\tau \in T} \Delta_n^\tau$.

For example, consider the parameterized system $\mathcal{P}_1$ given in Fig. 1. Let $c_1 = \langle (q_1, (x \mapsto 4, y \mapsto 6)) \rangle$ and $c_2 = \langle (q_2, (x \mapsto 5, y \mapsto 6)) \rangle$ be 1-process configurations. Then, $(c_1, c_2) \in \Delta_1^{\tau_1}$. Let $c_3 = \langle (q_3, (x \mapsto 4, y \mapsto 5)), (q_3, (x \mapsto 2, y \mapsto 7)) \rangle$ and $c_4 = \langle (q_1, (x \mapsto 4, y \mapsto 5)), (q_3, (x \mapsto 2, y \mapsto 7)) \rangle$ be 2-process configurations. Then, $(c_3, c_4) \in \Delta_2^{\tau_3}$.

In this paper, we work with a single transition system instead of many instances. We use $\mathcal{T}(\mathcal{P}) \triangleq (C, \Delta)$, where $C \triangleq \bigcup_{n \in \mathbb{N}} C_n$, and $\Delta \triangleq \bigcup_{n \in \mathbb{N}} \Delta_n$. Note that $\mathcal{T}(\mathcal{P})$ contains all n-instantiations of $\mathcal{P}$ as sub-systems.

Reachability Problem. The reachability problem of parameterized systems is: given a set of *initial* states $I \subseteq C$, and a set of *error* states $E \subseteq C$, decide whether there exist two configurations $c_i \in I$ and $c_e \in E$ such that there is a path from c_i to c_e in $\mathcal{T}$. This formulation is equivalent to a more common one of deciding whether there exists an $n \in \mathbb{N}$, such that an error configuration is reachable from an initial configuration in $\mathcal{T}_n(\mathcal{P})$. It is well-known that the verification of any safety property can be reduced to a reachability problem.

A backward-reachability-based algorithm is: given a set of error configurations E, compute an over-approximation of the set of all configurations that can reach E, denoted by R, then, decide whether the intersection of I and R is empty. In the rest of the paper, we only focus on computing R. All of the computation of our algorithm is done using a specialized abstract domain that we describe in the next section.

3 Abstract Domains for Parameterized Systems

We give a brief overview of numeric abstract domains and introduce our new domains for representing configurations of parameterized systems.

Abstract Domains. We provide a brief overview of the basics of Abstract Interpretation [13]. For the purpose of this paper, an *abstract domain* [13] A is a collection of elements equipped with a concretization function γ_A that maps each element of A to a set of concrete elements. We assume that A is equipped with two computable functions: an *abstract ordering* $\sqsubseteq_A$: $A \times A \to \{\text{true}, \text{false}\}$, and a *join* $\sqcup_A : A \times A \to A$ that over-approximate subset ordering and union, respectively:

$$a \sqsubseteq_A b \Rightarrow \gamma_A(a) \subseteq \gamma_A(b) \qquad a \sqcup_A b = c \Rightarrow (\gamma_A(a) \cup \gamma_A(b)) \subseteq \gamma(c) \qquad \text{(soundness)}$$

A well-known class of *numerical abstract domains* captures arithmetic (typically linear) relations between variables in a concrete domain. We use octagon [22] as an example of a numeric domain. For a set of variables V, elements of the *octagon* domain [22] $\text{OCT}(V)$ are conjunctions of constraints of the form $(\pm x \pm y \leq c)$, where $x, y \in V$ and c is a constant. The concretization γ_{OCT} maps a conjunction of constraints to a set of valuations, e.g., $\gamma_{\text{OCT}}(x \leq 3) = \{\sigma \in V \rightarrow \mathbb{N} \mid \sigma(x) \leq 3\}$. Abstract ordering is implemented with implication, e.g., $(x \leq 3) \sqsubseteq_{\text{OCT}} (x \leq 4)$ since $x \leq 3 \Rightarrow x \leq 4$. Join of two octagons is the smallest octagon containing their union. For example, $(x = 3) \sqcup_{\text{OCT}} (x = 5)$ is an octagon $3 \leq x \leq 5$ that can also be written as $-x \leq -3 \wedge x \leq 5$. We use this domain for all of the examples in the paper. However, our results extend to other domains such as polyhedra [14] (conjunctions of linear inequalities) and sets of octagons or polyhedra as well.

Parametric Abstract Domain PD. In this section, we define an abstract domain PD, called the *parametric domain*, that captures information about control locations of configurations of a parameterized system. In the rest of this section, we fix a parameterized system $\mathcal{P}$, and use Q to denote its control locations. Elements of PD are called *abstract locations*. Each element $s \in \text{PD}$ is a map $Q \rightarrow \mathbf{2}^{\mathbb{N}}$ such that $s[q]$ is finite for all $q \in Q$ and for $q, q' \in Q$, if $q \neq q'$ then $s[q] \cap s[q'] = \emptyset$. Intuitively, $s[q]$ represents the processes that are currently at q. For example, let

$$s_1 = (q_1 \mapsto \{1\}, q_2 \mapsto \{2, 3\}) \qquad (\star)$$

Intuitively, s_1 represents all concrete configurations in which there are *at least* three processes: one at q_1, and two at q_2. Note that the actual numeric PIDs are irrelevant and are only used for reference as we show below.

Let s be in PD. We write $\text{PROC}(s)$ for the set of all PIDs appearing in s. Formally, $\text{PROC}(s) \triangleq \bigcup_{q \in Q} s[q]$. We write $|s|$ for $|\text{PROC}(s)|$, and $\text{PC}(i, s)$ for the control location of process i, i.e., $\text{PC}(i, s) = q$ iff $i \in s[q]$. For example, for s_1 above, $\text{PROC}(s_1) = \{1, 2, 3\}$, $|s_1| = 3$, and $\text{PC}(1, s_1) = q_1$. Without loss of generality, we assume whenever $|s| = m$, then $\text{PROC}(s) = \{1, .., m\}$.

In the rest of this section, we formalize the definitions of concretization, abstract ordering, and join for this domain. Intuitively, $\gamma_{\text{PD}}(s)$ is the set of all configurations that have at least $|s[q]|$ processes at q, for all $q \in Q$. Formally, let $c = \langle (q_1, v_1), \ldots, (q_n, v_n) \rangle$ be a configuration, $s \in \text{PD}$ such that $|s| = m \leq n$, and $h : \{1, .., m\} \rightarrow \{1, .., n\}$ be an injection. We say that c satisfies s under h, written $c \models_h s$ iff

$$\forall i \in \text{PROC}(s) : \text{PC}(i, s) = q_{h(i)}$$

We define $\gamma_{\text{PD}}(s) \triangleq \{c \mid \exists h : c \models_h s\}$. It is easy to see that this definition captures our intuition. For example, let $c_1 = \langle (q_1, v_1), (q_2, v_2), (q_1, v_3), (q_2, v_4) \rangle$, where $\{v_i\}$ are arbitrary valuations, and $h = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 4\}$. Then, $c_1 \models_h s_1$; thus, $c_1 \in \gamma_{\text{PD}}(s_1)$.

For two abstract locations s and t, if for all $q \in Q$, $|t[q]| \leq |s[q]|$, then t approximates more concrete configurations. We define the ordering $\sqsubseteq_{\text{PD}}$ as:

$$s \sqsubseteq_{\text{PD}} t \Leftrightarrow (\forall q \in Q : |t[q]| \leq |s[q]|)$$

For example, let $s_2 = (q_1 \mapsto \{2\}, q_2 \mapsto \{1\})$ and $s_3 = (q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\})$. Then, $s_3 \sqsubseteq_{\text{PD}} s_2$, but $s_3 \not\sqsupseteq_{\text{PD}} s_1$ and $s_1 \not\sqsupseteq_{\text{PD}} s_3$. Note that the abstract domain PD is not a lattice. Thus, the Galois connection framework of AI (Example 4.6 in [13]) is not applicable. Therefore, we follow a more general framework of Abstract Interpretation [13] that allows for an abstract domain to be a pre-order.

Let $\top_{\text{PD}}$ be defined as an element s such that for all $q \in Q, s[q] = \emptyset$. Then, $\top_{\text{PD}}$ is the $\sqsubseteq_{\text{PD}}$-largest element of PD. For $s, t \in$ PD, we define the join as $s \sqcup_{\text{PD}} t = t$ if $s \sqsubseteq_{\text{PD}} t$ and $\top_{\text{PD}}$ otherwise. At a first glance, our definition of join may look too imprecise. However, our analysis algorithm (see BACKREACH in Sec. 4) only applies the join $s \sqcup_{\text{PD}} t$ under the assumption that $s \sqsubseteq_{\text{PD}} t$.

Theorem 1. *The abstract ordering $\sqsubseteq_{\text{PD}}$ and the join $\sqcup_{\text{PD}}$ are sound.*

The proof of the theorems can be found in the appendix. In the next section, we show how to extend the domain PD with a numeric (or even an arbitrary) abstract domain.

Abstract Domain PD(A). We combine the parametric domain PD with an abstract domain A. The new domain is called PD(A). For clarity of presentation, we assume that A is a numerical abstract domain. We call elements of PD(A) *abstract global states* (AGS). An AGS is of the form (s, ψ), where $s \in$ PD and $\psi \in A$. Intuitively, s captures the control location information and ψ captures numerical constraints on process variables. For an AGS $r = (s, \psi)$, we write $\text{loc}(r)$ for the abstract location s.

In the rest of the section, we fix a parameterized system $\mathcal{P} = (Q, V, T)$. For $x \in V$, we write $P_i.x$ to refer to the variable x of process i. We require that for every element $(s, \psi) \in$ PD(A), ψ is an expression over variables in the set $\{P_i.x \mid x \in V, i \in \text{PROC}(s)\}$. For example, an AGS $(s_1, P_1.x < P_2.y)$, where s_1 is as defined in $(\star)$, represents all concrete configurations that satisfy s_1 and, additionally, have a process i in state q_1 and a process j in state q_2 such that $P_i.x < P_j.y$. Note that i and j are not necessarily 1 and 2, since the PIDs in the abstract global states are only used for reference and do not directly correspond to PIDs in concrete configurations.

We now proceed to define $\gamma_{\text{PD(A)}}$ formally. For a function $h : \mathbb{N} \to \mathbb{N}$ and an expression ψ, we write $h(\psi)$ for the result of permuting all process references in ψ according to h, i.e., $h(\psi) \triangleq \psi[P_i \leftarrow P_{h(i)} \mid i \in \mathbb{N}]$. Let $c = \langle u_1, \ldots, u_n \rangle$ be a concrete configuration. We write σ_c for a valuation corresponding to the configuration c, defined as follows: $\sigma_c(P_j.x) \triangleq u_j(x)$. Let (s, ψ) be an AGS, such that $|s| = m$ and $m \leq n$, and $h : \{1, .., m\} \to \{1, .., n\}$ be an injection. We say that c satisfies (s, ψ) under h, written, $c \models_h (s, \psi)$ iff $c \models_h s \wedge \sigma_c \models h(\psi)$. Finally, we define $\gamma_{\text{PD(A)}}((s, \psi)) \triangleq \{c \mid \exists h : c \models_h (s, \psi)\}$.

We now describe the ordering $\sqsubseteq_{\text{PD(A)}}$. Let s, t be in PD, such that $s \sqsubseteq_{\text{PD}} t$. We write, $\mathcal{U}(s, t)$ for the set of all functions h such that (a) h is an injection from $\{1, .., |t|\}$ to $\{1, .., |s|\}$, and (b) for all $i \in \text{PROC}(t) : i \in t[q] \Rightarrow h(i) \in s[q]$. That is, h maps each process of t to an equivalent process of s. For example, let $s_4 = (q_1 \mapsto \{1, 2\})$, and $s_5 = (q_1 \mapsto \{1\})$, then $\mathcal{U}(s_4, s_5) = \{h_1, h_2\}$, where $h_1 = \{1 \mapsto 1\}$ and $h_2 = \{1 \mapsto 2\}$. Note that if $s \sqsubseteq_{\text{PD}} t$, then $\mathcal{U}(s, t)$ is not empty. The ordering $\sqsubseteq_{\text{PD(A)}}$ is defined as:

$$(s, \psi) \sqsubseteq_{\text{PD(A)}} (t, \varphi) \Leftrightarrow s \sqsubseteq_{\text{PD}} t \wedge \exists h \in \mathcal{U}(s, t) : \psi \sqsubseteq_A h(\varphi)$$

For example, let $\psi_1 = ((P_1.x > 0) \wedge (P_2.x > 4))$, and $\psi_2 = (P_1.x > 1)$, then $(s_4, \psi_1) \sqsubseteq_{\text{PD(A)}} (s_5, \psi_2)$, since ψ_1 implies $h_2(\psi_2) = (P_2.x > 1)$.

The $\sqsubseteq_{\text{PD(A)}}$-largest element is $(\top_{\text{PD}}, \top_A)$, where $\top_A$ is the $\sqsubseteq_A$-largest element of A. The join $\sqcup_{\text{PD(A)}}$ is defined as:

$$(s, \psi) \sqcup_{\text{PD(A)}} (t, \varphi) \triangleq \begin{cases} (s, \psi \sqcup_A h(\varphi)) & s \sqsubseteq_{\text{PD}} t \wedge t \sqsubseteq_{\text{PD}} s \\ \top_{\text{PD(A)}} & \text{otherwise} \end{cases}$$

where h is any injection in $\mathcal{U}(s, t)$. Intuitively, we use the join $\sqcup_A$ of A to join the constraints of the variables, while aligning PIDs between s and t. Note that a different choice for h affects precision but not soundness of the join. In practice, it is best to pick an h that leads to the $\sqsubseteq_{\text{PD(A)}}$-least result. As with PD, it is possible to define join more precisely, but it was not needed for our algorithm.

Theorem 2. *The abstract ordering $\sqsubseteq_{\text{PD(A)}}$ and the join $\sqcup_{\text{PD(A)}}$ are sound.*

Elements of PD(A) concisely represent (possibly infinite) sets of configurations of a concrete parameterized system. This domain is the basis of our backward-reachability algorithm that we present in the next section.

4 Backward-Reachability Analysis

We present the BACKREACH algorithm for over-approximating the backward-reachability in parameterized systems. We begin with an overview of the algorithm, then discuss its main step, i.e. computation of the pre-image, and conclude with an example.

Overview. The algorithm BACKREACH is shown in Fig. 2. As inputs, it takes a set `Trans` of guarded commands and an AGS `e`. The output is a set of AGSs that over-approximates all concrete configurations from which `e` is reachable.

The algorithm uses the list `RL` to keep track of all states seen so far, and a work list `WL` to keep track of all states to be explored. When `WL` becomes empty, the algorithm terminates. In each iteration, a state (s, ψ) is chosen from `WL` (lines 3–4), its predecessors are computed (lines 6–7), and are added to `RL` and `WL` lists if needed (lines 8–19). The computation of the predecessors is done using the function Pre, which is described in details below. The algorithm ensures that `RL` contains only one state for each abstract location by joining the AGSs with the same abstract locations (line 17).

In the rest of this section, we describe the implementation of the pre-image computation (line 7 of BACKREACH algorithm). First, we describe the operation for the domain PD, and then extend it to PD(A).

Pre-Image for PD. Let s be an element of PD, $\tau : q \xrightarrow{g} q'$ a guarded command, and i a PID. The result of pre-image operation $\text{Pre}_{\text{PD}}(s, \tau, i)$ is a set B of elements of PD that over-approximates all states from which a state in $\gamma(s)$ is reachable by process i executing τ. There are three cases, based on the type of the guard g.

Case 1 g is a local guard. If s is an abstract location obtained by process P_i executing τ, then, P_i is in state q' in s. Furthermore, P_i must have been in state q before executing τ. To formalize this, we define a helper function $\text{MOVEPROC}(s, i, q_1, q_2)$ that moves process i in s from location q_1 to location q_2: $\text{MOVEPROC}(s, i, q_1, q_2) \triangleq t$, where $t[q_1] = s[q_1] \setminus \{i\}$, $t[q_2] = s[q_2] \cup \{i\}$, and $t[q] = s[q]$ otherwise. Then,

1: **Set of AGS** BACKREACH (**Set** `Trans`, **AGS** `e`)
2: $\texttt{WL} \leftarrow \{\texttt{e}\}$, $\texttt{RL} \leftarrow \{\texttt{e}\}$
3: **forall** $(s, \psi) \in \texttt{WL}$ **do**
4: $\texttt{WL} \leftarrow \texttt{WL} \setminus \{(s, \psi)\}$
5: $P \leftarrow \emptyset$
6: **forall** $\{\tau \in \texttt{Trans}, i \in \text{PROC}(s) \mid \tau = (q \xrightarrow{g} q') \text{ and } i \in s[q']\}$ **do**
7: $P \leftarrow P \cup \text{Pre}((s, \psi), \tau, i)$
8: **forall** $r \in P$ **do**
9: $\texttt{skip} \leftarrow$ false, $\texttt{saved} \leftarrow$ null
10: **forall** $u \in \texttt{RL}$ **do**
11: **if** $r \sqsubseteq_{\text{PD(A)}} u$ **then**
12: $\texttt{skip} \leftarrow$ true, **break**
13: **if** $\text{loc}(r) = \text{loc}(u)$ **then**
14: $\texttt{saved} \leftarrow u$
15: **if** $\texttt{skip} =$ false **then**
16: **if** $\texttt{saved} \neq$ null **then**
17: $\texttt{RL} \leftarrow (\texttt{RL} \setminus \{\texttt{saved}\}) \cup \{\texttt{saved} \sqcup_{\text{PD(A)}} r\}, \texttt{WL} \leftarrow \texttt{WL} \cup \{\texttt{saved} \sqcup_{\text{PD(A)}} r\}$
18: **else**
19: $\texttt{RL} \leftarrow \texttt{RL} \cup \{r\}, \texttt{WL} \leftarrow \texttt{WL} \cup \{r\}$
20: **return** `RL`

Fig. 2. The BACKREACH algorithm.

$$\text{Pre}_{\text{PD}}(s, \tau, i) \triangleq \begin{cases} \{\text{MOVEPROC}(s, i, q', q)\} & \text{if } i \in s[q'] \\ \emptyset & \text{otherwise.} \end{cases}$$

For example, let $s_1 = (q_1 \mapsto \{1\}, q_2 \mapsto \{2, 3\})$, and $\tau = q_1 \xrightarrow{true} q_2$. Then, $\text{Pre}_{\text{PD}}(s_1, \tau, 1) = \emptyset$, and $\text{Pre}_{\text{PD}}(s_1, \tau, 2) = (q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3\})$.

Case 2 g is a universal global guard: $\forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\textbf{pc} = q_o) \wedge \theta$. Then, the pre-image computation is similar to Case 1 except that all processes other than i must be in control location q_o in s. Thus, $\text{Pre}_{\text{PD}}(s, \tau, i) \triangleq \{\text{MOVEPROC}(s, i, q', q)\}$, if $i \in s[q']$ and $\forall j \in \text{PROC}(s) \setminus \{i\} : \text{PC}(s, j) = q_o$, and $\emptyset$ otherwise.

Case 3 g is an existential global guard: $\exists \texttt{other} \neq \texttt{self} : (\texttt{other}.\textbf{pc} = q_o) \wedge \theta$. Then, τ can only be executed from an abstract location that has a process different from i at location q_o. The computation of Pre_{PD} is partitioned based on the choice of that `other` process. The `other` process can be either a process in $\text{PROC}(s)$, or a new process with PID ($|s| + 1$). Let

$$\text{Pre}_{\text{PD}}(s, \tau, i) \triangleq \bigcup_{j \in s[q_o] \setminus \{i\}} \text{OPre}_{\text{PD}}(s, \tau, i, j) \cup \text{OPre}_{\text{PD}}(s, \tau, i, |s| + 1)$$

where $\text{OPre}_{\text{PD}}(s, \tau, i, j)$ is the pre-image under the assumption that P_j is the `other` process. We define another helper function called $\text{MOVEADDPROC}(s, i, q_1, q_2, j, q_3)$ that in addition to moving process i from q_1 to q_2 adds a new process j to q_3: $\text{MOVEADDPROC}\ (s, i, q_1, q_2, j, q_3) \triangleq t$, where $t[q_1] = s[q_1] \setminus \{i\}$, $t[q_2] = s[q_2] \cup \{i\}$, $t[q_3] = s[q_3] \cup \{j\}$, and $t[q] = s[q]$ otherwise. Then,

$$\text{OPre}_{\text{PD}}(s, \tau, i, j) \triangleq \begin{cases} \{\text{MOVEPROC}(s, i, q', q)\} & \text{if } j \in s[q_o] \setminus \{i\} \\ \{\text{MOVEADDPROC}(s, i, q', q, j, q_o)\} & \text{if } j = |s| + 1 \\ \emptyset & \text{otherwise.} \end{cases}$$

For example, let $\tau_1 = q_1 \xrightarrow{g_1} q_2$ and $g_1 : \exists \texttt{other} \neq \texttt{self} : (\texttt{other}.\textbf{pc} = q_2) \wedge \theta$. Then, $\text{Pre}_{\text{PD}}(s_1, \tau_1, 2)$ is the union of $\text{OPre}_{\text{PD}}(s, \tau, 2, 3)$ and $\text{OPre}_{\text{PD}}(s, \tau, 2, 4)$ where $\text{OPre}_{\text{PD}}(s, \tau, 2, 3) = (q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3\})$ and $\text{OPre}_{\text{PD}}(s, \tau, 2, 4) = (q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3, 4\})$.

Theorem 3. *The pre-image operation of* PD *is sound.*

Pre-Image for PD(A). We assume that the domain A has a pre-image operation $\text{Pre}_A(\psi, R)$ that takes an element of the domain $\psi \in A$, and a relation R described by an expression over primed and unprimed variables. It returns an abstract element that over-approximates the pre-image of $\gamma_A(\psi)$ over R. Many numeric domains satisfy this assumption. For example, in OCT, $\text{Pre}_{\text{OCT}}(x \geq 1, x' = x + 1)$ is $x \geq 0$.

Let (s, ψ) be an element of PD(A) and $\tau : q \xrightarrow{g} q'$ a guarded command. The pre-image operation in PD(A) is defined using the following templates. If g is either local or universal, then

$$\text{Pre}_{\text{PD(A)}}((s, \psi), \tau, i) \triangleq \text{Pre}_{\text{PD}}(s, \tau, i) \times \text{Pre}_A(\psi, R_i)$$

and if g is existential then $\text{Pre}_{\text{PD(A)}}((s, \psi), \tau, i)$ is defined similar to Pre_{PD} where

$$\text{OPre}_{\text{PD(A)}}((s, \psi), \tau, i, j) \triangleq \text{OPre}_{\text{PD}}(s, \tau, i, j) \times \text{Pre}_A(\psi, R_{i,j})$$

where i, j are PIDs, and R_i, $R_{i,j}$ are relations defined based on g as described below.

Case 1 g is a local guard. Assume $g = \theta$, where θ is an expression over $\texttt{self}.(V \cup V')$. Let Θ_i and Γ_i be defined as follows:

$$\Theta_i \triangleq \theta[\texttt{self} \leftarrow P_i] \qquad \Gamma_i \triangleq \bigwedge_{j \in (\text{PROC}(s) \setminus \{i\})} \bigwedge_{x \in V} P_j.x' = P_j.x$$

Then, $R_i \triangleq \Theta_i \wedge \Gamma_i$. Intuitively, Θ_i instantiates the guard to process i, and Γ_i ensures that the variables of processes other than i are not affected. For example, let (s_1, ψ_1) be an AGS where s_1 is as defined in $(\star)$ and $\psi_1 = ((P_1.x > 0) \wedge (P_2.x > 1) \wedge (P_3.x > 2))$. Let $\tau_1 = q_1 \xrightarrow{g_1} q_2$ and $g_1 : x' = x + 1$. Then, $\text{Pre}_{\text{PD(A)}}((s_1, \psi_1), \tau, 2) = ((q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3\}), ((P_1.x > 0) \wedge (P_2.x > 0) \wedge (P_3.x > 2))$ since process P_2 is the `self` process.

Case 2 g is a universal global guard: $\forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\textbf{pc} = q_o) \wedge \theta$, where θ is an expression over $\texttt{self}.(V \cup V') \cup \texttt{other}.V$ variables. We need to instantiate θ with two PIDs: one for `self`, and one for `other`. Let $\Theta_{i,j}$ be defined as:

$$\Theta_{i,j} \triangleq \theta[\texttt{self} \leftarrow P_i, \texttt{other} \leftarrow P_j]$$

Then, $R_i \triangleq \bigwedge_{j \in (\text{PROC}(s) \setminus \{i\})} \Theta_{i,j} \wedge \Gamma_i$. Intuitively, R_i ensures that all processes other than i satisfy the global guard but only values of process i are affected during the transition.

Case 3 g is an existential guard: $\exists \texttt{other} \neq \texttt{self} : (\texttt{other}.\textbf{pc} = q_o) \wedge \theta$, where θ is again an expression over $\texttt{self}.(V \cup V') \cup \texttt{other}.V$ variables. However, in this case, the pre-image operator provides a PID j to instantiate the `other` process. Thus, $R_{i,j}$ is defined as $R_{i,j} \triangleq \Theta_{i,j} \wedge \Gamma_i$.

Name	Location	Constraints
1	$(q_2 \mapsto \{1\})$	$(P_1.x > 1) \wedge (P_1.y > 3)$
2	$(q_1 \mapsto \{1\})$	$(P_1.x > 0) \wedge (P_1.y > 3)$
3	$(q_3 \mapsto \{1, 2\})$	$(P_1.x > 0) \wedge (P_1.y > 3) \wedge (P_2.y - P_2.x > 2) \wedge (P_2.x > 1)$
4	$(q_1 \mapsto \{1\}, q_3 \mapsto \{2\})$	$(P_1.x > 0) \wedge (P_1.y > 5) \wedge (P_2.y - P_2.x > 2) \wedge (P_2.x > 1)$
5	$(q_1 \mapsto \{2\}, q_3 \mapsto \{1\})$	$(P_1.x > 0) \wedge (P_1.y > 3) \wedge (P_2.y - P_2.x > 4) \wedge (P_2.x > 1)$

Table 1. An example of a computation of BACKREACH.

Theorem 4. *The pre-image operation of* PD(A) *is sound.*

An Example. In this section, we illustrate a run of the BACKREACH algorithm on an example using abstract domain PD(OCT). We use the parameterized system shown in Fig. 1, and let e be $((q_2 \mapsto \{1\}), ((P_1.x > 1) \wedge (P_1.y > 3)))$.

We present the AGSs computed by the algorithm in Table 1. Each row in the table represents a single AGS (s, ψ) where the first column is a numeric reference, the second is the abstract location l, and the third is the octagon constraint ψ. Row 1 of the table corresponds to e defined above. We refer to the rows of Table 1 by numeric references.

In the first iteration, the algorithm computes $\text{Pre}(e, \tau_1, 1)$ that results in the AGS (s_2, ψ_2) shown in row 2. In the second iteration, the algorithm computes $(s_3, \psi_3) = \text{Pre}((s_2, \psi_2), \tau_3, 1)$ shown in row 3. In the third iteration, τ_2 is enabled twice: once for process P_1, and once for process P_2. Row 4 shows (s_4, ψ_4), the result of pre-image of τ_2 with respect to process P_1, i.e., $\text{Pre}((s_3, \psi_3), \tau_2, 1)$. This state is subsumed by (s_2, ψ_2) since $s_4 \sqsubseteq_{\text{PD}} s_2$ and $\psi_4 \Rightarrow \psi_2$. Thus, it is not added to the list RL. Row 5 shows (s_5, ψ_5), the result of pre-image of τ_2 with respect to process P_2, i.e., $\text{Pre}((s_3, \psi_3), \tau_2, 2)$. This state is subsumed by (s_2, ψ_2) as well. The reason is slightly more complicated. First, $s_5 \sqsubseteq_{\text{PD}} s_2$. Second, the process P_2 of s_5 corresponds to the process P_1 of s_2 and $\psi_5 \Rightarrow \psi_2[P_1 \leftarrow P_2]$. Thus, this AGS is not added to the list RL.

At this point, the work list WL becomes empty and the algorithm terminates. Thus, the RL contains only the AGSs shown in the first three rows of Table 1.

BACKREACH is sound: if it terminates, it always computes the correct result.

Theorem 5. *Let $\mathcal{P} = (Q, V, T)$ be a parameterized system and* e *be an abstract global state. If* BACKREACH(T, e) *terminates, it returns an over-approximation of the set of backward-reachable states from $\gamma_{\text{PD(A)}}$(e).*

BACKREACH is incomplete and may run forever. In the next section, we show how sound termination can be enforced.

5 Enforcing Convergence

There are two reasons for a possible divergence of BACKREACH. First, the numeric abstract domain A may be infinite (like octagons or polyhedra), thus BACKREACH may get stuck in an infinite numeric computation. Second, successive applications of pre-image to a transition with an existential guard may introduce unbounded numbers of processes. Here, we illustrate divergence of the BACKREACH algorithm through a set of examples and show how to enforce termination.

Numeric Divergence. We begin with an example that illustrates numeric divergence in the abstract domain PD(OCT). Let $\mathcal{P} = (Q, V, T)$ where $Q = \{q\}$, $V = \{x\}$, and

$T = \{\tau\}$ where τ is $q \xrightarrow{g} q$, $g : (x \geq 0) \Rightarrow (x' = x-1)$. Let $\mathtt{e}$ be $((q \mapsto \{1\}), (P_1.x = 5))$. Consider the execution of BACKREACH$(\mathtt{T}, \mathtt{e})$. In the first iteration, the algorithm computes the state $((q \mapsto \{1\}), (P_1.x = 6))$. It is joined to $\mathtt{e}$ at line 17, resulting in

$$((q \mapsto \{1\}), ((P_1.x = 5) \sqcup_{\text{OCT}} (P_1.x = 6))) = ((q \mapsto \{1\}), (5 \leq P_1.x \leq 6))$$

Similarly, the result of the second iteration is $((q \mapsto \{1\}), (5 \leq P_1.x \leq 7))$, etc. Thus, the BACKREACH$(\mathtt{T}, \mathtt{e})$ diverges.

In AI, a common approach to force sound convergence is to use *widening* instead of join to combine the reachable states. A *widening* operator [13], denoted by $\bigtriangledown_A$, is an operator that over-approximates join, i.e., $\forall x, y \in A : x \sqcup_A y \sqsubseteq_A x \bigtriangledown_A y$; additionally, for any increasing chain $x_0 \sqsubseteq_A x_1 \sqsubseteq_A \ldots \sqsubseteq_A x_n \ldots$ in A, the increasing chain $y_0 = x_0, \ldots, y_{n+1} = y_n \bigtriangledown_A x_{n+1}, \ldots$ stabilizes after a finite number of terms. Thus, replacing join with widening forces convergence of any least fixpoint computation.

We extend the widening operator of A to PD(A) in the following way. Given two abstract global states (s, ψ) and (t, φ), then

$$(s, \psi) \bigtriangledown_{\text{PD(A)}} (t, \varphi) \triangleq \begin{cases} (s, \psi \bigtriangledown_A h(\varphi)) & \text{if } s \sqsubseteq_{\text{PD}} t \wedge t \sqsubseteq_{\text{PD}} s \\ \top_{\text{PD(A)}} & \text{otherwise.} \end{cases}$$

Theorem 6. *The operator $\bigtriangledown_{\text{PD(A)}}$ is a widening on* PD(A).

In order to use this widening operator in our algorithm, we replace $\mathtt{saved} \sqcup_{\text{PD(A)}} r$ with $\mathtt{saved} \bigtriangledown_{\text{PD(A)}} (\mathtt{saved} \sqcup_{\text{PD(A)}} r)$ at line 17. We refer to the resulting algorithm as BACKREACH with widening.

Consider the previous example. With widening, the result of the first iteration is computed as follows:

$$\begin{aligned} & ((q \mapsto \{1\}), (P_1.x = 5)) \bigtriangledown_{\text{PD(OCT)}} ((q \mapsto \{1\}), (5 \leq P_1.x \leq 6)) \\ = {} & ((q \mapsto \{1\}), (P_1.x = 5) \bigtriangledown_{\text{OCT}} (5 \leq P_1.x \leq 6)) \\ = {} & ((q \mapsto \{1\}), (5 \leq P_1.x)) \end{aligned}$$

The algorithm converges after a single iteration. In this case, the result happens to be the exact set of all reachable states.

Successive applications of pre-image to transitions with only local or universal guards do not increase the number of processes in the reachable abstract global states. Therefore, systems with no existential guards may only experience numerical divergence. In such systems adding widening is sufficient to enforce convergence.

Theorem 7. *Let $\mathcal{P} = (Q, V, T)$ be a parameterized system with no existential transition and $\mathtt{e} \in$* PD(A). *The* BACKREACH$(\mathtt{T}, \mathtt{e})$ *with widening terminates and returns an over-approximation of the set of backward-reachable configurations from $\gamma_{\text{PD(A)}}(\mathtt{e})$.*

Parametric Divergence. Consider the following example. Assume the abstract domain is PD(OCT). Let $\mathcal{P} = (Q, V, T)$ where $Q = \{q_1, q_2\}$, $V = \{x\}$, and $T = \{\tau\}$ where

$$\tau : q_1 \xrightarrow{g} q_2 \ , \ g : \exists \mathtt{other} \neq \mathtt{self} : (\mathtt{other}.\mathbf{pc} = q_2) \wedge (\mathtt{other}.x = \mathtt{self}.x - 3)$$

Let $\mathtt{e} = ((q_2 \mapsto \{1\}), (2 \leq P_1.x \leq 5))$ as shown in row 1 of Table 2. The first iteration of BACKREACH$(\mathtt{T}, \mathtt{e})$ computes an AGS shown in row 2 of Table 2, the second,

Name	Location	Constraints
1	$(q_2 \mapsto \{1\})$	$(2 \leq P_1.x \leq 5)$
2	$(q_1 \mapsto \{1\}, q_2 \mapsto \{2\})$	$(2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8)$
3	$(q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\})$	$(2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (8 \leq P_3.x \leq 11)$
4	$(q_1 \mapsto \{1,2,3\}, q_2 \mapsto \{4\})$	$(2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (8 \leq P_3.x \leq 11) \wedge (11 \leq P_4.x \leq 14)$

Table 2. An example of a divergent computation of BACKREACH.

computes the AGS shown in row 3 of Table 2, etc. The algorithm does not terminate – each iteration adds a new AGS with one more process than in any AGS seen so far.

To mitigate this, we introduce an approximation operator called k-*compact*, $\rhd_k$, where $k \in \mathbb{N}$. Given an AGS (s, ψ) where $|s| > k$, $\rhd_k$ computes an AGS (t, φ) such that $(s, \psi) \sqsubseteq_{\text{PD(A)}} (t, \varphi)$ and $|t| = k$. The operator k-compact, $\rhd_k((s, \psi))$, is implemented by: (a) choosing a process, say i, in s, (b) removing i from s, and (c) existentially projecting away all variables of the form $P_i.x$ from ψ. Note that the choice of which process to drop only affects the precision and not the soundness of k-compact.

Theorem 8. *The approximation operator k-compact is sound.*

To incorporate $\rhd_k$ in the BACKREACH algorithm, we apply it after the pre-image computation at line 7. This ensures that the number of processes in each AGS never becomes larger than k.

Consider the previous example. Assume $k = 3$. Let ϕ denote the AGS computed in the third iteration (row 4 of Table 2). Assume $\rhd_3$ drops process P_3, then $\rhd_3(\phi)$ is

$$((q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\}), ((2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (11 \leq P_3.x \leq 14)))$$

The algorithm joins this AGS with the AGS computed in the second iteration (row 3 of Table 2) using widening and obtains

$$((q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\}), ((2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (8 \leq P_3.x)))$$

The algorithm terminates with an over-approximation of the set of reachable states.

Theorem 9. *Let $\mathcal{P} = (Q, V, T)$ be a parameterized system and $\mathtt{e} \in$ PD(A). The* BACKREACH$(\mathtt{T}, \mathtt{e})$ *algorithm with widening and k-compact operator always terminates and returns an over-approximation of the set of backward-reachable configurations from $\gamma_{\text{PD(A)}}(\mathtt{e})$.*

Lamport's Bakery Mutual-Exclusion Protocol. Fig. 3 shows a variant of Lamport's bakery mutual-exclusion protocol (Alg. 2 in [21]). The algorithm maintains two shared counters: *next* and *serv*, where *next* is the value of the next available ticket, and *serv* is the value of the ticket of the next process to be served. The shared variables belong to neither `self` nor `other`. We extend our framework to accommodate shared variables.

To enter the critical section, a process (i) obtains a ticket by incrementing *next* (as shown in τ_1), and storing its value in a local variable named *tick* (τ_2), (ii) picks a delay (τ_3) and spins for d steps (τ_4) and (τ_5), and (iii) enters its critical section when its ticket is being served (τ_6), i.e. its ticket value is equal to *serv*. When a process leaves the critical section, it goes back to the *idle* state and increments *serv* (τ_7).

$\tau_1 : idle \xrightarrow{g_1} choose \quad , g_1 : \forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\textbf{pc} \neq choose) \wedge (next' = next + 1)$

$\tau_2 : choose \xrightarrow{g_2} wait \quad , g_2 : \forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\textbf{pc} \neq choose) \wedge (\texttt{self}.tick' = next)$

$\tau_3 : wait \xrightarrow{g_3} pause \quad , g_3 : (\texttt{self}.d' = \texttt{self}.tick - serv)$

$\tau_4 : pause \xrightarrow{g_4} pause \; , g_4 : ((\texttt{self}.d > 0) \Rightarrow (\texttt{self}.d' = \texttt{self}.d - 1))$

$\tau_5 : pause \xrightarrow{g_5} wait \quad , g_5 : (\texttt{self}.d \leq 0) \wedge (\texttt{self}.tick > serv)$

$\tau_6 : pause \xrightarrow{g_6} use \quad , g_6 : (serv = \texttt{self}.tick) \wedge (\texttt{self}.d \leq 0)$

$\tau_7 : use \xrightarrow{g_7} idle \quad , g_7 : (next \geq serv + 1) \wedge (serv' = serv + 1)$

Fig. 3. Lamport's bakery mutual-exclusion protocol with proportional back-off.

The guards on τ_1 and τ_2 ensure that no other process changes *next* while a process is acquiring a ticket. A delay between consecutive reads of the *serv* is added to reduce network contention due to the polling of the common shared variable *serv*. In [21], the authors suggest that a reasonable delay is the number of processes already waiting to enter their critical section. The protocol ensures FIFO service by serving the processes in the same order in which they first requested it.

We have implemented the BACKREACH algorithm in JAVA using APRON library for octagon abstract domain[1].We have used this implementation to validate that the state $(idle \mapsto \{1, 2\})$ is not reachable from $(use \mapsto \{1, 2\})$. The experiments were performed on a P4 3.2 GHz machine running Linux SUSE 10.3. The computation with widening converges after 56 iterations and takes 3.475 seconds. The widening is crucial for handling τ_4 that is similar to the example in the beginning of this section.

6 Conclusion

We present a framework based on Abstract Interpretation for the analysis of safety properties of parameterized systems where each of the individual processes may be infinite-state. We introduce a new abstract domain for the parameterized systems that employs a numeric abstract domain. We describe an algorithm that over-approximates backward-reachability. We combine widening with an extrapolation operator developed for this abstract domain to enforce sound termination of the algorithm. We illustrate our technique by automatically verifying the mutual-exclusion property in a variant of Lamport's bakery protocol.

Safety verification of parameterized systems using AI-based frameworks introduces a whole family of new, sound, automatic, and terminating static analyses procedures for parameterized systems, each procedure varying the chosen abstraction and widening operator. We have implemented the BACKREACH algorithm and are currently investigating other protocols to which our analysis framework is applicable. One direction for future research is to consider other possible operators like k-compact that increase the precision of approximation by choosing the process to drop based on heuristics derived from the features of the analyzed system.

Acknowledgments. Ghafari and Trefler are supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

[1] Available at `http://www.swen.uwaterloo.ca/~nghafari/AIPMCTool`

References

1. P. A. Abdulla, G. Delzanno, and A. Rezine. "Parameterized Verification of Infinite-State Processes with Global Conditions". In *CAV'07*, volume 4590 of *LNCS*, pages 145–157, 2007.
2. P. A. Abdulla and B. Jonsson. "Verifying Networks of Timed Processes (Extended Abstract)". In *TACAS'98*, volume 1384 of *LNCS*, pages 298–312, 1998.
3. K. R. Apt and D. C. Kozen. "Limits for Automatic Verification of Finite-State Concurrent Systems". *Information Processing Letters*, 22(6):307–309, 1986.
4. A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. "Rewriting Systems with Data". In *FCT'07*, volume 4639 of *LNCS*, pages 1–22, 2007.
5. A. Bouajjani, P. Habermehl, and T. Vojnar. "Abstract Regular Model Checking". In *CAV'04*, volume 3114 of *LNCS*, pages 372–386, 2004.
6. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. "Regular Model Checking". In *CAV'00*, volume 1855 of *LNCS*, pages 403–418, 2000.
7. A. Bouajjani, Y. Jurski, and M. Sighireanu. "A Generic Framework for Reasoning About Dynamic Networks of Infinite-State Processes". In *TACAS'07*, volume 4424 of *LNCS*, pages 690–705, 2007.
8. M. Bozzano and G. Delzanno. "Beyond Parameterized Verification". In *TACAS'02*, volume 2280 of *LNCS*, pages 221–235, 2002.
9. T. Bultan, R. Gerber, and W. Pugh. "Model-Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations, and Experimental Results". *ACM Trans. on Programming Languages and Systems*, 21(4):747–789, 1999.
10. E. M. Clarke, O. Grumberg, and M. C. Browne. "Reasoning about Networks with Many Identical Finite-State Processes". In *PODC'86*, pages 240–248, 1986.
11. E. M. Clarke, O. Grumberg, and S. Jha. "Verifying Parameterized Networks". *ACM Trans. Program. Lang. Syst.*, 19(5):726–750, 1997.
12. E. M. Clarke, M. Talupur, and H. Veith. "Environment Abstraction for Parameterized Verification". In *VMCAI'06*, volume 3855 of *LNCS*, pages 126–141, 2006.
13. P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". *J. of Logic and Computation*, 2(4):511–547, 1992.
14. P. Cousot and N. Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In *POPL'78*, pages 84–97, 1978.
15. E. A. Emerson and V. Kahlon. "Reducing Model Checking of the Many to the Few". In *CADE'00*, volume 1831 of *LNCS*, pages 236–254, 2000.
16. E. A. Emerson and K. S. Namjoshi. "On Model Checking for Non-Deterministic Infinite-State Systems". In *LICS'98*, pages 70–80, 1998.
17. S. M. German and A. P. Sistla. "Reasoning about Systems with Many Processes". *J. of the ACM*, 39(3):675–735, 1992.
18. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. "Symbolic Model Checking with Rich Assertional Languages". In *CAV'97*, volume 1254 of *LNCS*, pages 424–435, 1997.
19. L. Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". *Communication of ACM*, 17(8):453–455, 1974.
20. D. Lesens, N. Halbwachs, and P. Raymond. "Automatic Verification of Parameterized Linear Networks of Processes". In *POPL'97*, pages 346–357, 1997.
21. J. M. Mellor-Crummey and M. L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Trans. on Computer Systems*, 9(1):21–65, 1991.
22. A. Miné. "The Octagon Abstract Domain". *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
23. A. Pnueli, J. Xu, and L. D. Zuck. "Liveness with (0, 1, ∞)-Counter Abstraction". In *CAV'02*, volume 2404 of *LNCS*, pages 107–122, 2002.