# Handling Parameterized Systems with Non-atomic Global Conditions

Parosh Aziz Abdulla[1], Noomene Ben Henda[1], Giorgio Delzanno[2], and Ahmed Rezine[1]

[1] Uppsala University, Sweden
parosh@it.uu.se, Noomene.BenHenda@it.uu.se, Rezine.Ahmed@it.uu.se
[2] Università di Genova, Italy
giorgio@disi.unige.it

**Abstract.** We consider verification of safety properties for parameterized systems with linear topologies. A process in the system is an extended automaton, where the transitions are guarded by both local and global conditions. The global conditions are non-atomic, i.e., a process allows arbitrary interleavings with other transitions while checking the states of all (or some) of the other processes. We translate the problem into model checking of infinite transition systems where each configuration is a labeled finite graph. We derive an over-approximation of the induced transition system, which leads to a symbolic scheme for analyzing safety properties. We have implemented a prototype and run it on several nontrivial case studies, namely non-atomic versions of Burn's protocol, Dijkstra's protocol, the Bakery algorithm, Lamport's distributed mutual exclusion protocol, and a two-phase commit protocol used for handling transactions in distributed systems. As far as we know, these protocols have not previously been verified in a fully automated framework.

## 1 Introduction

We consider verification of safety properties for *parameterized systems*. Typically, a parameterized system consists of an arbitrary number of processes organized in a linear array. The task is to verify correctness regardless of the number of processes. This amounts to the verification of an infinite family; namely one for each size of the system. An important feature in the behaviour of a parameterized system is the existence of *global conditions*. A global condition is either *universally* or *existentially* quantified. An example of a universal condition is that all processes to the left of a given process $i$ should satisfy a property $\theta$. Process $i$ can perform the transition only if all processes with indices $j < i$ satisfy $\theta$. In an existential condition we require that *some* (rather than *all*) processes satisfy $\theta$. Together with global conditions, we allow features such as shared variables, broadcast communication, and processes operating on unbounded variables.

All existing approaches to automatic verification of parameterized systems (e.g., [13, 5, 7, 9]) make the unrealistic assumption that a global condition is performed atomically, i.e., the process which is about to make the transition

F. Logozzo et al. (Eds.): VMCAI 2008, LNCS 4905, pp. 22–36, 2008.

checks the states of all the other processes and changes its own state, all in one step. However, almost all protocols (modeled as parameterized systems with global conditions) are implemented in a distributed manner, and therefore it is not feasible to test global conditions atomically.

In this paper, we propose a method for automatic verification of parameterized systems where the global conditions are not assumed to be atomic. The main idea is to translate the verification problem into model checking of systems where each configuration is a labeled *finite graph*. The labels of the nodes encode the local states of the processes, while the labels of the edges carry information about the data flow between the processes. Our verification method consists of three ingredients each of which is implemented by a fully automatic procedure: (i) a preprocessing phase in which a *refinement protocol* is used to translate the behaviour of a parameterized system with global conditions into a system with graph configurations; (ii) a *model checking phase* based on symbolic backward reachability analysis of systems with graph configurations; and (iii) an *over-approximation scheme* inspired by the ones proposed for systems with *atomic* global conditions in [4] and [2]. The over-approximation scheme is extended here in a non-trivial manner in order to cope with configurations which have graph structures. The over-approximation enables us to work with efficient symbolic representations (upward closed sets of configurations) in the backward reachability procedure. Below, we describe the three ingredients in detail.

In order to simplify the presentation, we consider a basic model. Nevertheless, the method can be generalized to deal with a number of features which are added to enrich the basic model, such as broadcast communication and shared variables (see [3] for details). In the basic model, a process is a finite-state automaton which operates on a set of local variables ranging over the Booleans. The transitions of the automaton are conditioned by the local state of the process, values of the local variables, and by global conditions. Transitions involving global conditions are not assumed to be atomic. Instead, they are implemented using an underlying protocol, here referred to as the *refinement protocol*. Several different versions of the protocol are possible. The one in the basic model works as follows. Let us consider a process, called the *initiator*, which is about to perform a transition with a global condition. Suppose that the global condition requires that all processes to the left of the initiator satisfy $\theta$. Then, the initiator sends a request asking the other processes whether they satisfy $\theta$ or not. A process sends an acknowledgment back to the initiator only if it satisfies $\theta$. The initiator performs the transition when it has received acknowledgments from all processes to its left. The acknowledgments are sent by the different processes independently. This means that the initiator may receive the acknowledgments in any arbitrary order, and that a given process may have time to change its local state and its local variables before the initiator has received its acknowledgment.

The refinement protocol induces a system with an infinite set of configurations each of which is a finite graph. The nodes of the graph contain information about the local states and the values of the local variables of the processes, while the edges represent the flow of request and acknowledgment messages used to

implement the refinement protocol. We observe that the graph representation defines a natural ordering on configurations, where a configuration is smaller than another configuration, if the graph of the former is contained in the graph of the latter (i.e., if there is a label-respecting injection from the smaller to the larger graph). To check safety properties, we perform backward reachability analysis on sets of configurations which are upward closed under the above mentioned ordering. Two attractive features of upward closed sets are (i) checking safety properties can almost always be reduced to the reachability of an upward closed set; and (ii) they are fully characterized by their minimal elements (which are finite graphs), and hence these graphs can be used as efficient symbolic representations of infinite sets of configurations. One problem is that upward closedness is not preserved in general when computing sets of predecessors. To solve the problem, we consider a transition relation which is an over-approximation of the one induced by the parameterized system. To do that, we modify the refinement protocols by eliminating the processes which have failed to acknowledge a universal global condition (either because they do not satisfy the condition or because they have not yet sent an acknowledgment). For instance in the above example, it is always the case that process $i$ will eventually perform the transition. However, when performing the transition, we eliminate each process $j$ (to the left of $i$) which has failed to acknowledge the request of $i$. The approximate transition system obtained in this manner is *monotonic* w. r. t. the ordering on configurations, in the sense that larger configurations can simulate smaller ones. The fact that the approximate transition relation is monotonic, means that upward closedness is maintained when computing predecessors. Therefore, all the sets which are generated during the backward reachability analysis procedure are upward closed, and can hence be represented by their minimal elements. Observe that if the approximate transition system satisfies a safety property then we can conclude that the original system satisfies the property too. The whole verification process is fully automatic since both the approximation and the reachability analysis are carried out without user intervention. Termination of the approximated backward reachability analysis is not guaranteed in general. However, the procedure terminates on all the examples we report in this paper.

In this paper, we will also describe how the method can be generalized to deal with the model where processes are infinite-state. More precisely, the processes may operate on variables which range over the natural numbers, and the transitions may be conditioned by *gap-order constraints*. Gap-order constraints [17] are a logical formalism in which one can express simple relations on variables such as lower and upper bounds on the values of individual variables; and equality, and gaps (minimal differences) between values of pairs of variables.

Another aspect of our method is that systems with graph configurations are interesting in their own right. The reason is that many protocols have inherently distributed designs, rather than having explicit references to global conditions. For instance, configurations in the Lamport distributed mutual exclusion protocol [15] or in the two-phase commit protocol of [12] are naturally modeled as graphs where the nodes represent the local states of the processes, and the edges

describe the data traveling between the processes. In such a manner, we get a model identical to the one extracted through the refinement protocol, and hence it can be analyzed using our method.

We have implemented a prototype and used it for verifying a number of challenging case studies such as parameterized non-atomic versions of Burn's protocol, Dijkstra's protocol, the Bakery algorithm, Lamport's distributed mutual exclusion protocol [15] , and the two-phase commit protocol used for handling transactions in [12]. As far as we know, none of these examples has previously been verified in a fully automated framework.

**Related Work.** We believe that this is the first work which can handle automatic verification of parameterized systems where global conditions are tested non-atomically. All existing automatic verification methods (e.g., [13, 5, 7, 9, 10, 4, 2]) are defined for parameterized systems where universal and existential conditions are evaluated atomically. Non-atomic versions of parameterized mutual exclusion protocols such as the Bakery algorithm and two-phase commit protocol have been studied with heuristics to discover invariants, ad-hoc abstractions, or semi-automated methods in [6, 14, 16, 8]. In contrast to these methods, our verification procedure is fully automated and is based on a generic approximation scheme for quantified conditions.

The method presented in this paper is related to those in [4, 2] in the sense that they also rely on combining over-approximation with symbolic backward reachability analysis. However, the papers [4,2] assume *atomic* global conditions. As described above, the passage from the atomic to the non-atomic semantics is not trivial. In particular, the translation induces models whose configurations are graphs, and are therefore well beyond the capabilities of the methods described in [4, 2] which operate on configurations with linear structures. Furthermore, the underlying graph model can be used in its own to analyze a large class of distributed protocols. This means that we can handle examples none of which can be analyzed within the framework of [4, 2].

## 2 Preliminaries

In this section, we define a basic model of parameterized systems.

For a natural number $n$, let $\overline{n}$ denote the set $\{1, \ldots, n\}$. We use $\mathcal{B}$ to denote the set $\{true, false\}$ of Boolean values. For a finite set $A$, we let $\mathbb{B}(A)$ denote the set of formulas which have members of $A$ as atomic formulas, and which are closed under the Boolean connectives $\neg, \wedge, \vee$. A *quantifier* is either *universal* or *existential*. A universal quantifier is of one of the forms $\forall_L$, $\forall_R$, $\forall_{LR}$. An existential quantifier is of one of the forms $\exists_L$, $\exists_R$, or $\exists_{LR}$. The subscripts $L$, $R$, and $LR$ stand for *Left*, *Right*, and *Left-Right* respectively. A *global condition* over $A$ is of the form $\Box\theta$ where $\Box$ is a quantifier and $\theta \in \mathbb{B}(A)$. A global condition is said to be *universal* (resp. *existential*) if its quantifier is *universal* (resp. *existential*). We use $\mathbb{G}(A)$ to denote the set of global conditions over $A$.

**Parameterized Systems.** A parameterized system consists of an arbitrary (but finite) number of identical processes, arranged in a linear array. Sometimes, we refer to processes by their indices, and say e.g., the process with index $i$ (or simply process $i$) to refer to the process with position $i$ in the array. Each process is a finite-state automaton which operates on a finite number of Boolean local variables. The transitions of the automaton are conditioned by the values of the local variables and by *global* conditions in which the process checks, for instance, the local states and variables of all processes to its left or to its right. The global transitions are not assumed to be atomic operations. A transition may change the value of any local variable inside the process. A parameterized system induces an infinite family of finite-state systems, namely one for each size of the array. The aim is to verify correctness of the systems for the whole family (regardless of the number of processes inside the system). A *parameterized system* $\mathcal{P}$ is a triple $(Q, X, T)$, where $Q$ is a set of *local states*, $X$ is a set of *local Boolean variables*, and $T$ is a set of *transition rules*. A transition rule $t$ is of the form

$$t : \left[\begin{array}{c} q \\ grd \rightarrow stmt \\ q' \end{array}\right] \tag{1}$$

where $q, q' \in Q$ and $grd \rightarrow stmt$ is a *guarded command*. Below we give the definition of a guarded command. Let $Y$ denote the set $X \cup Q$. A *guard* is a formula $grd \in \mathbb{B}(X) \cup \mathbb{G}(Y)$. In other words, the guard $grd$ constraints either the values of local variables inside the process (if $grd \in \mathbb{B}(X)$); or the local states and the values of local variables of other processes (if $grd \in \mathbb{G}(Y)$). A *statement* is a set of assignments of the form $x_1 = e_1; \ldots; x_n = e_n$, where $x_i \in X$, $e_i \in \mathcal{B}$, and $x_i \neq x_j$ if $i \neq j$. A *guarded command* is of the form $grd \rightarrow stmt$, where $grd$ is a guard and $stmt$ is a statement.

## 3 Transition System

In this section, we describe the induced transition system.

A *transition system* $\mathcal{T}$ is a pair $(D, \Longrightarrow)$, where $D$ is an (infinite) set of *configurations* and $\Longrightarrow$ is a binary relation on $D$. We use $\stackrel{*}{\Longrightarrow}$ to denote the reflexive transitive closure of $\Longrightarrow$. For sets of configurations $D_1, D_2 \subseteq D$ we use $D_1 \Longrightarrow D_2$ to denote that there are $c_1 \in D_1$ and $c_2 \in D_2$ with $c_1 \Longrightarrow c_2$. We will consider several transition systems in this paper.

First, a parameterized system $\mathcal{P} = (Q, X, T)$ induces a transition system $\mathcal{T}(\mathcal{P}) = (C, \longrightarrow)$ as follows. In order to reflect non-atomicity of global conditions, we use a protocol, called the *refinement protocol*, to refine (implement) these conditions. The refinement protocol uses a sequence of request and acknowledgment messages between processes. Therefore, a configuration is defined by (i) the local states and the values of the local variables of the different processes; and by (ii) the flow of requests and acknowledgments which are used to implement the refinement protocol. Below, we describe these two components, and then use them to define the set of configurations and the transition relation.

**Process States.** A *local variable state* $v$ is a mapping from $X$ to $\mathcal{B}$. For a local variable state $v$, and a formula $\theta \in \mathbb{B}(X)$, we evaluate $v \models \theta$ using the standard interpretation of the Boolean connectives. Given a statement *stmt* and a variable state $v$, we denote by $stmt(v)$ the variable state obtained from $v$ by mapping $x$ to $e$ if $(x = e) \in stmt$, $v(x)$ otherwise. We will also work with temporary states which we use to implement the refinement protocol. A *temporary* state is of the form $q^t$ where $q \in Q$ and $t \in T$. The state $q^t$ indicates that the process is waiting for acknowledgments from other processes while trying to perform transition $t$ (which contains a global condition). We use $Q^T$ to denote the set of temporary states, and define $Q^\bullet = Q \cup Q^T$. A *process state* $u$ is a pair $(q, v)$ where $q \in Q^\bullet$ and $v$ is a local variable state. We say that $u$ is *temporary* if $q \in Q^T$, i.e., if the local state is temporary. For a temporary process state $u = (q^t, v)$, we write $u^*$ to denote the process state $(q, v)$, i.e., we replace $q^t$ by the corresponding state $q$. If $u$ is not temporary then we define $u^* = u$.

Sometimes, abusing notation, we view a process state $(q, v)$ as a mapping $u : X \cup Q^\bullet \mapsto \mathcal{B}$, where $u(x) = v(x)$ for each $x \in X$, $u(q) = \mathit{true}$, and $u(q') = \mathit{false}$ for each $q' \in Q^\bullet - \{q\}$. The process state thus agrees with $v$ on the values of local variables, and maps all elements of $Q^\bullet$, except $q$, to *false*. For a formula $\theta \in \mathbb{B}(X \cup Q^\bullet)$ and a process state $u$, the relation $u \models \theta$ is then well-defined. This is true in particular if $\theta \in \mathbb{B}(X)$.

**The Refinement Protocol.** The refinement protocol is triggered by an *initiator* which is a process trying to perform a transition involving a global condition. The protocol consists of three phases described below. In the first phase, the initiator enters a temporary state and sends a request to the other processes asking whether they satisfy the global condition. In the second phase, the other processes are allowed to respond to the initiator. When a process receives the request, it sends an acknowledgment only if it satisfies the condition. The initiator remains in the temporary state until it receives acknowledgments from all relevant processes (e.g., all processes to its right if the quantifier is $\forall_R$, or some process to its left if the quantifier is $\exists_L$, etc). Then, the initiator performs the third phase which consists of leaving the temporary state, and changing its local state and variables according to the transition. The request of the initiator is received independently by the different processes. Also, the processes send their acknowledgments independently. In particular this means that the initiator may receive the acknowledgments in any arbitrary order. To model the status of the request and acknowledgments, we use *edges*. A *request edge* is of the form $i \xrightarrow{\texttt{req}}_t j$ where $i$ and $j$ are process indices and $t \in T$ is a transition. Such an edge indicates that process $i$ is in a temporary state trying to perform transition $t$ (which contains a global condition); and that it has issued a request which is yet to be acknowledged by process $j$. An *acknowledgment edge* is of the form $i \xleftarrow{\texttt{ack}}_t j$ with a similar interpretation, except that it indicates that the request of process $i$ has been acknowledged by process $j$. Observe that if a process is in a temporary state, then it must be an initiator.

**Configurations.** A *configuration* $c \in C$ is a pair $(U, E)$ where $U = u_1 \cdots u_n$ is a sequence of process states, and $E$ is a set of edges. We use $|c|$ to denote the number of processes inside $c$, i.e., $|c| = n$. Intuitively, the above configuration corresponds to an instance of the system with $n$ processes. Each pair $u_i = (q_i, v_i)$ gives the local state and the values of local variables of process $i$. We use $U[i]$ to denote $u_i$. The set $E$ encodes the current status of requests and acknowledgments among the processes. A configuration must also satisfy the following two invariants:

1. If $u_i$ is a temporary process state (for some transition $t$) then, for each $j : 1 \leq j \neq i \leq n$, the set $E$ contains either an edge of the form $i \xrightarrow{\texttt{req}}_t j$ or an edge of the form $i \xleftarrow{\texttt{ack}}_t j$ (but not both). This is done to keep track of the processes which have acknowledged request of $i$.
2. If $u_i$ is not a temporary process state then the set $E$ does not contain any edge of the form $i \xrightarrow{\texttt{req}}_t j$ or $i \xleftarrow{\texttt{ack}}_t j$, for any $t \in T$ and for any $j : 1 \leq j \neq i \leq n$. That is, if process $i$ is not in a temporary states, then it is not currently waiting for acknowledgments, and hence no edges of the above form need to be stored.

**Transition Relation.** Consider two configurations $c = (U, E)$ and $c' = (U', E')$ with $|c| = |c'| = n$. We describe how $c$ can perform a transition to obtain $c'$. Such a transition is performed by some process with index $i$ for some $i : 1 \leq i \leq n$. We write $c \xrightarrow{i} c'$ to denote that (i) $U[j] = U'[j]$ for each $j : 1 \leq j \neq i \leq n$ (i.e., only process $i$ changes state during the transition); and (ii) that there is a $t \in T$ of the form (1) such that one the following four conditions is satisfied (each condition corresponds to one type of transitions):

- *Local Transitions:* $grd \in \mathbb{B}(X)$, $U[i] = (q, v)$, $U'[i] = (q', v')$, $v \models grd$, $v' = stmt(v)$ , and $E' = E$. By $grd \in \mathbb{B}(X)$, we mean that $t$ is a local transition. The values of the local variables of the process should satisfy the guard $grd$, and they are modified according to $stmt$. The local states and variables of other processes are not relevant during the transition. Since the transition does not involve global conditions, the edges remains unchanged.
- *Refinement Protocol – First Phase:* $grd = \Box\theta \in \mathbb{G}(Y)$, $U[i] = (q, v)$, $U'[i] = (q^t, v)$, and $E' = E \cup \{i \xrightarrow{\texttt{req}}_t j | \ 1 \leq j \neq i \leq n\}$. Since $grd \in \mathbb{G}(Y)$, the transition $t$ contains a global condition. The initiator, which is process $i$, triggers the first phase of the refinement protocol. To do this, it moves to the temporary state $q^t$. It also sends a request to all other processes, which means that the new set of edges $E'$ should be modified accordingly. The local variables of the initiator are not changed during this step.
- *Refinement Protocol – Second Phase:* $grd = \Box\theta \in \mathbb{G}(Y)$, $U[i]$ is a temporary process state, $U'[i] = U[i]$, and there is a $j : 1 \leq j \neq i \leq n$ such that $U[j]^* \models \theta$ and $E' = E - \{i \xrightarrow{\texttt{req}}_t j\} \cup \{i \xleftarrow{\texttt{ack}}_t j\}$. A process (with index $j$) which satisfies the condition $\theta$ sends an acknowledgment to the initiator (process $i$). To reflect this, the relevant request edge is replaced by the corresponding acknowledgment edge. No local states or variables of any processes

are changed. Notice that we use $U[j]^*$ in the interpretation of the guard. This means that a process which is in the middle of checking a global condition, is assumed to be in its original local state until all the phases of the refinement protocol have successfully been carried out.

– *Refinement Protocol – Third Phase:* $grd = \Box\theta \in \mathbb{G}(Y), U[i] = (q^t, v), U'[i] = (q', v')$, $v' = stmt(v)$, $E' = E - \{i \xleftarrow{\texttt{ack}}_t j \mid 1 \leq j \neq i \leq n\} - \{i \xrightarrow{\texttt{req}}_t j \mid 1 \leq j \neq i \leq n\}$, and one of the following conditions holds:
  - $\Box = \forall_L$ and $(i \xleftarrow{\texttt{ack}}_t j) \in E$ for each $j : 1 \leq j < i$.
  - $\Box = \forall_R$ and $(i \xleftarrow{\texttt{ack}}_t j) \in E$ for each $j : i < j \leq n$.
  - $\Box = \forall_{LR}$ and $(i \xleftarrow{\texttt{ack}}_t j) \in E$ for each $j : 1 \leq j \neq i \leq n$.
  - $\Box = \exists_L$ and $(i \xleftarrow{\texttt{ack}}_t j) \in E$ for some $j : 1 \leq j < i$.
  - $\Box = \exists_R$ and $(i \xleftarrow{\texttt{ack}}_t j) \in E$ for some $j : i < j \leq n$.
  - $\Box = \exists_{LR}$ and $(i \xleftarrow{\texttt{ack}}_t j) \in E$ for some $j : 1 \leq j \neq i \leq n$.

  The initiator has received acknowledgments from the relevant processes. The set of relevant processes depends on the type of the quantifier. For instance, in case the quantifier is $\forall_L$ then the initiator waits for acknowledgments from all processes to its left (with indices smaller than $i$). Similarly, if the quantifier is $\exists_R$ then the initiator waits for an acknowledgment from some process to its right (with index larger than $i$), and so on. The initiator leaves its temporary state and moves to a new local state (the state $q'$) as required by the transition rule $t$. Also, the values of the local variables of the initiator are updated according to *stmt*. Since, process $i$ is not in a temporary state any more, all the corresponding edges are removed from the configuration.

We use $c \longrightarrow c'$ to denote that $c \xrightarrow{i} c'$ for some $i$.

**Variants of Refinement Protocols.** Our method can be modified to deal with several different variants of the refinement protocol described in Section 3. Observe that in the original version of the protocol, a process may either acknowledge a request or remain passive. One can consider a variant where we allow processes to explicitly refuse acknowledging of requests, by sending back a negative acknowledgment (a `nack`). We can also define different variants depending on the way a failure of a global condition is treated (in the third phase of the protocol). For instance, the initiator may be allowed to reset the protocol, by re-sending requests to all the processes (or only to the processes which have sent a negative acknowledgment).

## 4 Safety Properties

In order to analyze safety properties, we study the *coverability problem* defined below. Given a parameterized system $\mathcal{P} = (Q, X, T)$, we assume that, prior to starting the execution of the system, each process is in an (identical) *initial* process state $u_{init} = (q_{init}, v_{init})$. In the induced transition system $\mathcal{T}(\mathcal{P}) = (C, \longrightarrow)$, we use *Init* to denote the set of *initial* configurations, i.e., configurations

of the form $(U_{init}, E_{init})$, where $U_{init} = u_{init} \cdots u_{init}$ and $E_{init} = \emptyset$. In other words, all processes are in their initial states, and there are no edges between the processes. Notice that the set of initial configurations is infinite.

We define an ordering on configurations as follows. Given two configurations, $c = (U, E)$ with $|c| = m$, and $c' = (U', E')$ with $|c'| = n$, we write $c \preceq c'$ to denote that there is a strictly monotonic[1] injection $h$ from the set $\overline{m}$ to the set $\overline{n}$ such that the following conditions are satisfied for each $t \in T$ and $i, j : 1 \leq i \neq j \leq m$: (i) $u_i = u'_{h(i)}$, (ii) if $(i \xrightarrow{\texttt{req}}_t j) \in E$ then $(h(i) \xrightarrow{\texttt{req}}_t h(j)) \in E'$, and (iii) if $(i \xleftarrow{\texttt{ack}}_t j) \in E$ then $(h(i) \xleftarrow{\texttt{ack}}_t h(j)) \in E'$. In other words, for each process in $c$ there is a corresponding process in $c'$ with the same local state and with the same values of local variables. Furthermore, each request edge in $c$ is matched by a request edge between the corresponding processes in $c'$, while each acknowledgment edge in $c$ is matched by an acknowledgment edge between the corresponding processes in $c'$.

A set of configurations $D \subseteq C$ is *upward closed* (with respect to $\preceq$) if $c \in D$ and $c \preceq c'$ implies $c' \in D$. The *coverability problem* for parameterized systems is defined as follows:

PAR-COV
**Instance** A parameterized system $\mathcal{P} = (Q, X, T)$ and an upward closed set $C_F$ of configurations.
**Question** $Init \xrightarrow{*} C_F$ ?

It can be shown, using standard techniques (see e.g. [18,11]), that checking safety properties (expressed as regular languages) can be translated into instances of the coverability problem. Therefore, checking safety properties amounts to solving PAR-COV (i.e., to the reachability of upward closed sets). Intuitively, we use $C_F$ to denote a set of bad states which we do not want to occur during the execution of the system. For instance, in a mutual exclusion protocol, if the local state $q_{crit}$ corresponds to the process being in the critical section, then $C_F$ can be defined to be the set of all configurations where at least two processes are in $q_{crit}$. In such a case, $C_F$ is the set of bad configurations (those violating the mutual exclusion property). Notice that once a configuration has two processes in $q_{crit}$ then it will belong to $C_F$ regardless of the values of the local variables, the states of the rest of processes, and the edges between the processes. This implies that $C_F$ is upward closed.

## 5 Approximation

In this section, we introduce an over-approximation of the transition relation of a parameterized system. The aim of the over-approximations is to derive a new transition system which is *monotonic* with respect to the ordering $\preceq$ defined on configurations in Section 4. Formally, a transition system is monotonic with respect to the ordering $\preceq$, if for any configurations $c_1, c_2, c_3$ such that $c_1 \rightarrow c_2$

[1] $h : \overline{m} \rightarrow \overline{n}$ strictly monotonic means: $i < j \Rightarrow h(i) < h(j)$ for all $i, j : 1 \leq i, j \leq m$.

and $c_1 \preceq c_3$; there exists a configuration $c_4$ such that $c_3 \rightarrow c_4$ and $c_2 \preceq c_4$. The only transitions which violate monotonicity are those corresponding to the third phase of the refinement protocol when the quantifier is universal. Therefore, the approximate transition system modifies the behavior of the third phase in such a manner that monotonicity is maintained. More precisely, in the new semantics, we remove all processes in the configuration which have failed to acknowledge the request of the initiator (the corresponding edge is a request rather than an acknowledgment). Below we describe formally how this is done.

In Section 3, we mentioned that each parameterized system $\mathcal{P} = (Q, X, T)$ induces a transition system $\mathcal{T}(\mathcal{P}) = (C, \longrightarrow)$. A parameterized system $\mathcal{P}$ also induces an *approximate* transition system $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$; the set $C$ of configurations is identical to the one in $\mathcal{T}(\mathcal{P})$. We define $\rightsquigarrow = (\longrightarrow \cup \rightsquigarrow_1)$, where $\longrightarrow$ is defined in Section 3, and $\rightsquigarrow_1$ (which reflects the approximation of universal quantifiers in third phase of the refinement protocol) is defined as follows.

Consider two configurations $c = (U, E)$ and $c' = (U', E')$ with $|c| = n$ and $|c'| = m$. Suppose that $U[i] = (q^t, v)$ for some $i : 1 \leq i \leq n$ and some transition of the form of (1) where $grd = \Box\theta \in \mathbb{G}(Y)$ with $\Box \in \{\forall_L, \forall_R, \forall_{LR}\}$. In other words, in $c$, process $i$ is in a temporary state, performing the second phase of refinement protocol with respect to a universal quantifier. We write $c \overset{i}{\rightsquigarrow}_1 c'$ to denote that there is a strictly monotonic injection $h : \overline{m} \mapsto \overline{n}$ such that the following conditions are satisfied (the image of $h$ represents the indices of the processes we keep in the configuration):

- $j$ is in the image of $h$ iff one of the following conditions is satisfied: (i) $j = i$, (ii) $\Box = \forall_L$ and either $j > i$ or $(i \overset{\texttt{ack}}{\longleftarrow}_t j) \in E$, (iii) $\Box = \forall_R$ and either $j < i$ or $(i \overset{\texttt{ack}}{\longleftarrow}_t j) \in E$, (iv) $\Box = \forall_{LR}$ and $(i \overset{\texttt{ack}}{\longleftarrow}_t j) \in E$. That is we keep the initiator (process $i$) together with all the relevant processes who have acknowledged its request.
- $U'[h^{-1}(i)] = (q', stmt(v))$ and $U'[j] = U[h(j)]$ for $h(j) \neq i$, i.e., the local variables of process $i$ are updated according to *stmt* while the states and local variables of other processes are not changed.
- $E'$ is obtained from $E$ as follows. For all $j, k \in \overline{m}$ and $t' \in T$, (i) $(j \overset{\texttt{ack}}{\longleftarrow}_{t'} k) \in E'$ iff $(h(j) \overset{\texttt{ack}}{\longleftarrow}_{t'} h(k)) \in E$, and (ii) $(j \overset{\texttt{req}}{\longrightarrow}_{t'} k) \in E'$ iff $(h(j) \overset{\texttt{req}}{\longrightarrow}_{t'} h(k)) \in E$. In other words, we remove all edges connected to processes which are removed from the configuration $c$.

We use $c \rightsquigarrow_1 c'$ to denote that $c \overset{i}{\rightsquigarrow}_1 c'$ for some $i$.

**Lemma 1.** *The approximate transition system* $(C, \rightsquigarrow)$ *is monotonic w.r.t.* $\preceq$.

We define the coverability problem for the approximate system as follows.

APRX-PAR-COV
**Instance** A parameterized system $\mathcal{P} = (Q, X, T)$ and an upward closed set $C_F$ of configurations.
**Question** $Init \overset{*}{\rightsquigarrow} C_F$ ?

Since $\longrightarrow \subseteq \rightsquigarrow$, a negative answer to APRX-PAR-COV implies a negative answer to PAR-COV.

## 6 Backward Reachability Analysis

In this section, we present a scheme based on backward reachability analysis and we show how to instantiate it for solving APRX-PAR-COV. For the rest of this section, we assume a parameterized system $\mathcal{P} = (Q, X, T)$ and the induced approximate transition system $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$.

**Constraints.** The scheme operates on *constraints* which we use as a symbolic representation for sets of configurations. A constraint $\phi$ denotes an upward closed set $[\![\phi]\!] \subseteq C$ of configurations. The constraint $\phi$ represents minimal conditions on configurations. More precisely, $\phi$ specifies a minimum number of processes which should be in the configuration, and then imposes certain conditions on these processes. The conditions are formulated as specifications of the states and local variables of the processes, and as restrictions on the set of edges. A configuration $c$ which satisfies $\phi$ should have at least the number of processes specified by $\phi$. The local states and the values of the local variables should satisfy the conditions imposed by $\phi$. Furthermore, $c$ should contain at least the set of edges required by $\phi$. In such a case, $c$ may have any number of additional edges and processes (whose local states and local variables are then irrelevant for the satisfiability of $\phi$ by $c$). This definition implies that the interpretation $[\![\phi]\!]$ of a constraint $\phi$ is upward closed (a fact proved in Lemma 2). Below, we define the notion of a constraint formally.

A constraint is a pair $(\Theta, E)$ where $\Theta = \theta_1 \cdots \theta_m$ is a sequence with $\theta_i \in \mathbb{B}(X \cup Q^\bullet)$, and $E$ is a set of edges of the form $i \xrightarrow{\texttt{req}}_t j$ or $i \xleftarrow{\texttt{ack}}_t j$ with $t \in T$ and $1 \leq i, j \leq m$. We use $\Theta(i)$ to denote $\theta_i$ and $|\phi|$ to denote $m$. Intuitively, a configuration satisfying $\phi$ should contain at least $m$ processes, where the local state and variables of the $i^{th}$ process satisfy $\theta_i$. Furthermore the set $E$ defines the minimal set of edges which should exist in the configuration. More precisely, for a constraint $\phi = (\Theta, E_1)$ with $|\phi| = m$, and a configuration $c = (U, E_2)$ with $|c| = n$, we write $c \models \phi$ to denote that there is a strictly monotonic injection $h$ from the set $\overline{m}$ to the set $\overline{n}$ such that the following conditions are satisfied for each $t \in T$ and $i, j : 1 \leq i, j \leq m$: (i) $u_{h(i)} \models \theta_i$, (ii) if $(i \xrightarrow{\texttt{req}}_t j) \in E_1$ then $(h(i) \xrightarrow{\texttt{req}}_t h(j)) \in E_2$, and (iii) if $(i \xleftarrow{\texttt{ack}}_t j) \in E_1$ then $(h(i) \xleftarrow{\texttt{ack}}_t h(j)) \in E_2$. Given a constraint $\phi$, we let $[\![\phi]\!] = \{c \in C |\ c \models \phi\}$. Notice that if some $\theta_i$ is unsatisfiable then $[\![\phi]\!]$ is empty. Such a constraint can therefore be safely discarded if it arises in the algorithm. For a (finite) set of constraints $\Phi$, we define $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$. The following lemma follows from the definitions.

**Lemma 2.** *For each constraint $\phi$, the set $[\![\phi]\!]$ is upward closed.*

In all the examples we consider, the set $C_F$ in the definition of APRX-PAR-COV can be represented by a finite set $\Phi_F$ of constraints. The coverability question can then be answered by checking whether $\mathit{Init} \overset{*}{\rightsquigarrow} [\![\Phi_F]\!]$.

**Entailment and Predecessors.** To define our scheme we will use two operations on constraints; namely *entailment*, and *computing predecessors*, defined

below. We define an *entailment relation* $\sqsubseteq$ on constraints, where $\phi_1 \sqsubseteq \phi_2$ iff $[\![\phi_2]\!] \subseteq [\![\phi_1]\!]$. For sets $\Phi_1, \Phi_2$ of constraints, abusing notation, we let $\Phi_1 \sqsubseteq \Phi_2$ denote that for each $\phi_2 \in \Phi_2$ there is a $\phi_1 \in \Phi_1$ with $\phi_1 \sqsubseteq \phi_2$. Observe that $\Phi_1 \sqsubseteq \Phi_2$ implies that $[\![\Phi_2]\!] \subseteq [\![\Phi_1]\!]$. The lemma below, which follows from the definitions, gives a syntactic characterization which allows computing of the entailment relation.

**Lemma 3.** *For constraints $\phi = (\Theta, E)$ and $\phi' = (\Theta', E')$ of size $m$ and $n$ respectively , we have $\phi \sqsubseteq \phi'$ iff there exists a strictly monotonic injection $h : \overline{m} \to \overline{n}$ such that:*

1. *$\Theta'(h(i)) \Rightarrow \Theta(i)$ for each $i \in \overline{m}$, and*
2. *$\forall i, j : 1 \leq i, j \leq m$ and $\forall t \in T$, the following conditions holds: (i) if $i \xrightarrow{\texttt{req}}_t j \in E$ then $h(i) \xrightarrow{\texttt{req}}_t h(j) \in E'$, and (ii) if $i \xleftarrow{\texttt{ack}}_t j \in E$ then $h(i) \xleftarrow{\texttt{ack}}_t h(j) \in E'$.*

For a constraint $\phi$, we let $Pre(\phi)$ be a set of constraints, such that $[\![Pre(\phi)]\!] = \{c |\ \exists c' \in [\![\phi]\!] \,.\, c \rightsquigarrow c'\}$. In other words $Pre(\phi)$ characterizes the set of configurations from which we can reach a configuration in $\phi$ through the application of a single rule in the approximate transition relation. In the definition of *Pre* we rely on the fact that, in any monotonic transition system, upward-closedness is preserved under the computation of the set of predecessors (see e.g. [1]). From Lemma 2 we know that $[\![\phi]\!]$ is upward closed; by Lemma 1, $(C, \rightsquigarrow)$ is monotonic, we therefore know that $[\![Pre(\phi)]\!]$ is upward closed.

**Lemma 4.** *For any constraint $\phi$, $Pre(\phi)$ is computable.*

For a set $\Phi$ of constraints, we let $Pre(\Phi) = \bigcup_{\phi \in \Phi} Pre(\phi)$.

**Scheme.** Given a finite set $\Phi_F$ of constraints, the scheme checks whether $Init \stackrel{*}{\Longrightarrow} [\![\Phi_F]\!]$. We perform a backward reachability analysis, generating a sequence $\Phi_0 \sqsupseteq \Phi_1 \sqsupseteq \Phi_2 \sqsupseteq \cdots$ of finite sets of constraints such that $\Phi_0 = \Phi_F$, and $\Phi_{j+1} = \Phi_j \cup Pre(\Phi_j)$. Since $[\![\Phi_0]\!] \subseteq [\![\Phi_1]\!] \subseteq [\![\Phi_2]\!] \subseteq \cdots$, the procedure terminates when we reach a point $j$ where $\Phi_j \sqsubseteq \Phi_{j+1}$. Notice that the termination condition implies that $[\![\Phi_j]\!] = (\bigcup_{0 \leq i \leq j} [\![\Phi_i]\!])$. Consequently, $\Phi_j$ characterizes the set of all predecessors of $[\![\phi_F]\!]$. This means that $Init \stackrel{*}{\rightsquigarrow} [\![\Phi_F]\!]$ iff $(Init \bigcap [\![\Phi_j]\!]) \neq \emptyset$. In order to check emptiness of $(Init \bigcap [\![\Phi_j]\!])$, we rely on the result below which follows from the definitions. For a constraint $\phi = (\Theta, E)$, we have $(Init \bigcap [\![\phi]\!]) = \emptyset$ iff either $E \neq \emptyset$, or $u_{init} \not\models \Theta(i)$ for some $i \in \overline{n}$. Observe that, in order to implement the scheme we need to be able to (i) compute *Pre* (Lemma 4); (ii) check for entailment between constraints (Lemma 3); and (iii) check for emptiness of $(Init \bigcap [\![\phi]\!])$ for a constraint $\phi$ (as described above).

## 7 Unbounded Variables

In this section, we extend the basic model of Section 2 in two ways. First, we consider processes which operate on variables with unbounded domains. More

precisely, we allow local variables to range over the integers, and use a simple set of formulas, called *gap-order formulas*, to constrain the numerical variables in the guards. Furthermore, we allow nondeterministic assignment, where a variable may be assigned any value satisfying a guard. The new value of a variable may also depend on the values of variables of the other processes. Due to shortage of space we will only give an overview of the main ideas.

Consider a set $A$, partitioned into a set $A_{\mathcal{B}}$ of Boolean variables, and a set $A_{\mathbb{N}}$ of numerical variables. The set of *gap-order formulas* over $A_{\mathbb{N}}$, denoted $\mathbb{GF}(A_{\mathbb{N}})$, is the set of formulas which are either of the form $x = y$, $x \leq y$ or $x <_k y$, where $k \in \mathbb{N}$. Here $x <_k y$ stands for $x + k < y$ and specifies a *gap* of $k$ units between $y$ and $x$. We use $\mathbb{F}(A)$ to denote the set of formulas which have members of $\mathbb{B}(A_{\mathcal{B}})$ and of $\mathbb{GF}(A_{\mathbb{N}})$ as atomic formulas, and which is closed under the Boolean connectives $\wedge, \vee$. For a set $A$, we use $A^{next} = \{x^{next} | \ x \in A\}$ to refer to the *next-value* versions of the variables in $A$.

**Transitions.** In our extended model, the set of local variables $X$ is the union of a set $X_{\mathcal{B}}$ of Boolean variables and a set $X_{\mathbb{N}}$ of numerical variables. As mentioned above, variables may be assigned values which are derived from those of the other processes. To model this, we use the set $p{\cdot}Y = \{p{\cdot}x | x \in Y\}$ to refer to the local state and variables of process $p$. We consider a global condition to be of the form $\forall p : \theta$ where $\theta \in \mathbb{F}\,(X \cup p{\cdot}Y \cup X^{next})$. In other words, the formula checks the local variables of the initiator (through $X$), and the local states and variables of the other processes (through $p{\cdot}Y$). It also specifies how the local variables of the process in transition are updated (through $X^{next}$). Notice that the new values are defined in terms of the current values of variables and local states of all the other processes. Other types of transitions can be extended in an analogous manner. Values of next-variables not mentioned in $\theta$ remain unchanged.

*Example 1.* As an example, let the guard in the above transition rule be of the form $\forall p : p \cdot num < num^{next}$ where $num$ is a numerical variable. Then, this means that the process assigns to its variable $num$, a new value which is strictly greater than the values of $num$ in all other processes. Such a rule is used for instance in the Bakery algorithm to generate new tickets.

**The Refinement Protocol.** The first phase of the refinement protocol remains the same as in the basic model, i.e., the initiator sends requests to all other processes. The second phase is modified, so that an acknowledgment edge carries information about the responding process, i.e., the acknowledgment sent by process $p$ has the form $ack_p(u_p)$ where $u_p$ is the current local state of $p$. In the third phase, the initiator checks the global condition by looking at the values attached to the acknowledgments, and updates its own local variables accordingly. For instance, in the above example, the initiator receives the values of the variables $num$ of all the other processes on the acknowledgment edges. Then, it chooses a new value which is larger than all received values.

**Constraints.** The constraint system is modified so that we add gap-order constraints on the local variables of the processes and also on the values carried by

the acknowledgment edges. Performing operations such as checking entailment and computing predecessors on constraints with gap-orders can be carried out in a similar manner to [2].

## 8 Experimental Results

We have implemented our method in a prototype that we have run on several parameterized systems, including non-atomic refinements of Burn's protocol, Dijkstra's protocol and the Bakery's algorithm, as well as on the Lamport distributed Mutual exclusion protocol and the two-phase commit protocol. The Bakery and Lamport protocols have numerical local variables, while the rest have bounded local variables. The refinement $\mathcal{R}_1$ used for the first two algorithms corresponds to the refinement protocol introduced in Section 3. The refinements $\mathcal{R}_2$ and $\mathcal{R}_3$ are those introduced in the end of Section 3. More precisely, in $\mathcal{R}_2$, the initiator re-sends a request to all the processes whose values violate the global condition being tested. In $\mathcal{R}_3$, the initiator re-sends requests to all other processes.

The results, using a 2 GHZ computer with 1 GB of memory, are summarized in Table 1. We give for each case study, the number of iterations, the time, the number of constraints in the result, and an estimate of memory usage. Details of the case studies can be found in [3].

**Table 1.** Experimental results on several mutual exclusion algorithms

| | refine | iterat. | time | final constr. | memory |
|---|---|---|---|---|---|
| Burns | $\mathcal{R}_1$ | 26 | 0.5 sec | 44 | 1MB |
| Dijkstra | $\mathcal{R}_1$ | 93 | 0.5 sec | 41 | 1MB |
| Bakery | $\mathcal{R}_2$ | 4 | 0.06 sec | 12 | 1MB |
| Bakery | $\mathcal{R}_3$ | 4 | 0.06 sec | 12 | 1MB |
| Two Phase Commit | - | 6 | 0.03 sec | 9 | 1MB |
| Lamport | - | 29 | 30 mn | 4676 | 222MB |

## 9 Conclusions and Future Work

We have presented a method for automatic verification of parameterized systems. The main feature of the method is that it can handle global conditions which are not assumed to be atomic operations. We have built a prototype which we have successfully applied on a number of non-trivial mutual exclusion protocols. There are several interesting directions for future work. First, our algorithm operates essentially on infinite sets of graphs. Therefore, it seems feasible to extend the method to other classes of systems whose configurations can be modeled by graphs such as cache coherence protocols and dynamically allocated data structures. Furthermore, although the method works successfully on several examples, there is at least one protocol (namely the non-atomic version of Szymanski's protocol) where the method gives a false positive. We believe that this problem can

be solved by introducing a scheme which allows refining the abstraction (the over-approximation). Therefore, we plan to define a CEGAR (Counter-Example Guided Abstraction Refinement) scheme on more exact representations of sets of configurations.

## References

1. Abdulla, P., et al.: Algorithmic analysis of programs with well quasi-ordered domains. ICom 160, 109–127 (2000)
2. Abdulla, P., Delzanno, G., Rezine, A.: Parameterized Verification of Infinite-State Processes with Global Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
3. Abdulla, P., et al.: Handling parameterized systems with non-atomic global conditions. Technical Report 2007-030, it (2007)
4. Abdulla, P., et al.: Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
5. Abdulla, P., et al.: Regular Model Checking Made Simple and Efficient. In: Brim, L., et al. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 116–130. Springer, Heidelberg (2002)
6. Pnueli, A., et al.: Parameterized Verification with Automatically Computed Inductive Assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
7. Boigelot, B., Legay, A., Wolper, P.: Iterating Transducers in the Large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
8. Chkliaev, D., Hooman, J., van der Stok, P.: Mechanical verification of transaction processing systems. In: ICFEM 2000 (2000)
9. Clarke, E., Talupur, M., Veith, H.: Environment Abstraction for Parameterized Verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)
10. Delzanno, G.: Automatic verification of cache coherence protocols. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)
11. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. FMSD 2(2), 149–164 (1993)
12. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1992)
13. Kesten, Y., et al.: Symbolic model checking with rich assertional languages. TCS 256, 93–112 (2001)
14. Lahiri, S.K., Bryant, R.E.: Indexed Predicate Discovery for Unbounded System Verification. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
15. Lamport, L.: Time, clocks and the ordering of events in a distributed system. CACM 21(7), 558–565 (1978)
16. Manna, Z., et al.: STEP: the Stanford Temporal Prover. Draft Manuscript (1994)
17. Revesz, P.: A closed form evaluation for datalog queries with integer (gap)-order constraints. TCS 116, 117–149 (1993)
18. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. LICS 1986, pp. 332–344 (1986)