Handling Parameterized Systems with Non-Atomic Global Conditions

Parosh Aziz Abdulla
Uppsala University, Sweden.
parosh@it.uu.se

Noomene Ben Henda
Uppsala University, Sweden.
Noomene.BenHenda@it.uu.se

Giorgio Delzanno
Università di Genova, Italy.
giorgio@disi.unige.it

Ahmed Rezine
Uppsala University, Sweden.
Rezine.Ahmed@it.uu.se

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 θ. Process i can perform the transition only if all processes with indices $j < i$ satisfy θ. In an existential condition we require that *some* (rather than *all*) processes satisfy θ. 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., [12, 4, 6, 8]) make the unrealistic assumption that a global condition is performed atomically, i.e., the process which is about to make the transition 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 [3] 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 first start with a basic model, and then introduce additional features one by one. 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 θ. Then, the initiator sends a request asking the other processes whether they satisfy θ or not. A process sends an acknowledgment back to the initiator only if it satisfies θ. 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 acknowledgement). 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 shortly how the method can be generalized to deal with a number of features which are added to enrich the basic model (while still keeping the non-atomicity assumption). First, we consider parameterized systems where the 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. We will also describe different variants of the refinement protocol than the one described earlier. Finally, we explain how to handle other features such as shared variables, broadcast communication, and other variants of the refinement protocol.

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 [11] are naturally modelled as graphs where the nodes represent the local states of the processes, and the edges describe the data travelling 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 [11]. 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., [12, 4, 6, 8, 9, 3, 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 [5, 13, 16, 7]. 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 [3, 2] in the sense that they also rely on combining over-approximation with symbolic backward reachability analysis. However, the papers [3, 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 [3, 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 such as the ones mentioned earlier, none of which can be analyzed within the framework of [3, 2].

Outline In the next section, we give some preliminaries and define the basic model. In Section 3, we describe the induced transition system and the non-atomic semantics of global transitions (the refinement protocol). Section 4 introduces the coverability (safety) problem. In Section 5, we define the over-approximated transition system on which we run our algorithm. In Section 6, we present a generic scheme for deciding coverability and show how to instantiate it on our model. In Section 7, we consider a generalization of the model defined in Section 2 by considering processes which operate on variables with unbounded domains. Section 8 explains how to extend the model with additional features such as shared variables, broadcast transitions, and variants of the refinement protocol. In Section 9, we report results of our analysis on several mutual exclusion protocols. Section 10 concludes the paper and gives directions for future work. The appendix includes detailed descriptions of the case studies.

2 Preliminaries

In this section, we define a basic model of parameterized systems. This model will be enriched by additional features in Sections 7 and 8.

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

$$t : [q \,|\, grd \,\triangleright\, stmt \,|\, q'\,] \tag{1}$$

where $q, q' \in Q$ and $grd \to stmt$ 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 $grd \to stmt$, 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 $\overset{*}{\Longrightarrow}$ 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 $x \in X$, $u(q) = true$, and $u(q') = false$ 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 (see Figure 1). 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 θ 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:* $grd = \Box\theta \in \mathbb{G}(Y)$, $U[i] = (q^t, v)$, $U'[i] = (q', v')$, $v' = stmt(v)$, $E' = E - \{i \xleftarrow{\texttt{ack}}_t j |\ 1 \leq j \neq i \leq n\} - \{i \xrightarrow{\texttt{req}}_t j |\ 1 \leq j \neq i \leq n\}$, 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.

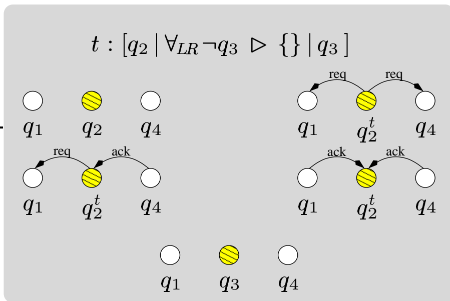


Figure 1. Refinement Protocol. The colored process initiates the first phase of the refinement protocol for t (first row) by moving to the temporary state q_2^t and sending request edges on t to all other processes (forward arrows). In the second phase of the protocol (second row), the white processes acknowledge the requests since they all satisfy the condition $\neg q_3$. Once all edges are acknowledged, the initiator (colored process) performs the third phase of the protocol by removing all edges and changing state to q_3 as illustrated in the last configuration.

Remark 1 *In the first phase of the protocol (described above), the initiator sends its request to all processes inside the system (including the ones whose replies are not relevant). However (in the third phase), it only takes into consideration acknowledgments from the relevant processes. For instance, if the quantifier is $\forall_L$ then the initiator waits only for acknowledgments from processes to its left. One could also have a variant of the protocol where the initiator only sends to the relevant processes. The two models have identical reachability properties, and our method can be extended in a straightforward manner to this alternative interpretation of global conditions.*

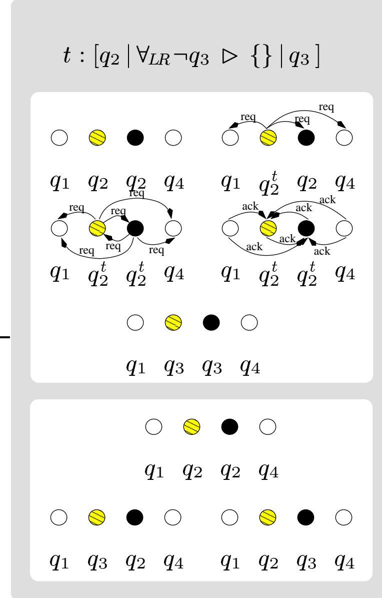


Figure 2. Atomic versus Non-Atomic. The higher side of the figure shows a possible non-atomic execution of the system: First, the two colored processes initiate the refinement protocol for transition t. In the second phase, the two colored processes mutually acknowledge the request edges on t since they both satisfy the condition $\neg q_3$. Once each of the colored processes receives all needed acknowledgments, it moves to q_3. Therefore, we end up by 2 states in q_3. This is opposed to the lower side execution were we assume that t is atomic. In this case, only one of the colored process can fire it and move to q_3. Once one of the processes executes t, the remaining process cannot fire it since the guard is not satisfied.

Remark 2 *In [3], we have considered the same type of parameterized systems, nevertheless, we assumed that global transitions are executed in an atomic manner. Observe here that under our current assumption, the resulting transition system induces more behaviors as depicted in Figure 2.*

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 $i, j : 1 \leq i \neq j \leq m$:

- $u_i = u'_{h(i)}$.
- If $(i \xrightarrow{\texttt{req}}_t j) \in E$ then $(h(i) \xrightarrow{\texttt{req}}_t h(j)) \in E'$.
- 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, 10]), 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.

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

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 $c_1 \to c_2$ and $c_1 \preceq c_3$; there exists a configuration c_4 such that $c_3 \to c_4$ 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:

  – $j = i$.

  – $\square = \forall_L$ and either $j > i$ or $(i \xleftarrow{\texttt{ack}}_t j) \in E$.

  – $\square = \forall_R$ and either $j < i$ or $(i \xleftarrow{\texttt{ack}}_t j) \in E$.

  – $\square = \forall_{LR}$ and $(i \xleftarrow{\texttt{ack}}_t j) \in E$.

  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 $t' \in T$,

  – $(j \xleftarrow{\texttt{ack}}_{t'} k) \in E'$ iff $(h(j) \xleftarrow{\texttt{ack}}_{t'} h(k)) \in E$, and

  – $(j \xrightarrow{\texttt{req}}_{t'} k) \in E'$ iff $(h(j) \xrightarrow{\texttt{req}}_{t'} h(k)) \in E$.

  In other words, we remove all edges connected to processes which are removed from the configuration c (see Figure 3).

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

Lemma 5.1 *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 $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$.

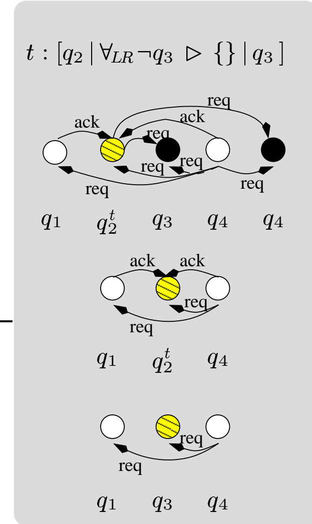


Figure 3. Approximation. In the top configuration, the marked process (the gray one) is in the second phase of the refinement protocol of the transition t. The second configuration is obtained by removing all processes that did not acknowledge the requests (the black processes). The third configuration is obtained by updating the state and the variables of the initiator with respect to t.

Constraints The scheme operates on *constraints* which we use as a symbolic representation for sets of configurations. A constraint ϕ denotes an upward closed set $[\![\phi]\!] \subseteq C$ of configurations. The constraint ϕ represents minimal conditions on configurations. More precisely, ϕ 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 ϕ should have at least the number of processes specified by ϕ. The local states and the values of the local variables should satisfy the conditions imposed by ϕ. Furthermore, c should contain at least the set of edges required by ϕ. 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 ϕ by c). This definition implies that the interpretation $[\![\phi]\!]$ of a constraint ϕ is upward closed (a fact proved in Lemma 6.1). Below, we define the notion of a constraint formally.

A constraint is a pair (Θ, 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 θ_i and $|\phi|$ to denote m. Intuitively, a configuration satisfying ϕ should contain at least m processes, where the local state and variables of the ith process satisfy θ_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 $i, j : 1 \leq i, j \leq m$:

- $u_{h(i)} \models \theta_i$.
- If $(i \xrightarrow{\texttt{req}}_t j) \in E_1$ then $(h(i) \xrightarrow{\texttt{req}}_t h(j)) \in E_2$.
- 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 ϕ, we let $[\![\phi]\!] = \{c \in C|\ c \models \phi\}$. Notice that if some θ_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 Φ, we define $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$. The following lemma follows from the definitions.

Lemma 6.1 *For each constraint ϕ, 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 Φ_F of constraints. The coverability question can then be answered by checking whether $Init \overset{*}{\rightsquigarrow} [\![\Phi_F]\!]$.

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 Φ_1, Φ_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 6.2 *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:*
   - *If $i \xrightarrow{\texttt{req}}_t j \in E$ then $h(i) \xrightarrow{\texttt{req}}_t h(j) \in E'$.*
   - *If $i \xleftarrow{\texttt{ack}}_t j \in E$ then $h(i) \xleftarrow{\texttt{ack}}_t h(j) \in E'$.*

For a constraint ϕ, 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 ϕ 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 6.1 we know that $[\![\phi]\!]$ is upward closed; by Lemma 5.1, $(C, \rightsquigarrow)$ is monotonic, we therefore know that $[\![Pre(\phi)]\!]$ is upward closed.

Lemma 6.3 *For any constraint ϕ, $Pre(\phi)$ is computable.*

(The construction of *Pre*(.) can be found in Appendix A.)

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

Scheme Given a finite set Φ_F of constraints, the scheme checks whether $Init \overset{*}{\Longrightarrow} [\![\Phi_F]\!]$. 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, Φ_j characterizes the set of all predecessors of $[\![\phi_F]\!]$. This means that $Init \overset{*}{\rightsquigarrow} [\![\Phi_F]\!]$ 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 6.3); (ii) check for entailment between constraints (Lemma 6.2); and (iii) check for emptiness of $(Init \bigcap [\![\phi]\!])$ for a constraint ϕ (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. Below, we show an examples of a global transition rule in the new model,

$$[\, q \,|\, \forall p : \theta \,|\, q' \,]$$

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 θ 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 Extensions

In this section, we describe briefly a number of features which can be added to the model of Section 2, and to which we can extend the verification method described in the previous sections.

Shared Variables. We extend the model with a finite set of *Boolean shared variables.* These variables are accessible to all processes. We modify the transitions by allowing conditions on the shared variables values in the guard, and assignments to these variables in the statement. We extend the definition of a configuration with a *shared variable state*; i.e., a mapping from the shared variables to the Boolean values. The relation $\preceq$ can be generalized by taking equality on the shared variable states. The definition of a constraint is now extended with a condition on the values of the shared variables. The entailment relation as well as the computation of *Pre* can be extended in the obvious manner.

Broadcast Transition. A *broadcast transition/rule* is a transition where an arbitrary number of processes simultaneously change states. A broadcast rule is a sequence of local transitions of the form

$$t : [q_0 \,|\, grd_0 \,\rhd\, stmt_0 \,|\, q_0'\,]\,[q_1 \,|\, grd_1 \,\rhd\, stmt_1 \,|\, q_1'\,]^* \cdots [q_m \,|\, grd_m \,\rhd\, stmt_m \,|\, q_m'\,]^* .$$

We use t_i to refer to the ith rule of t for $1 \leq i \leq m$. The broadcast transition is deterministic, i.e., for any rules

t_i, t_j where $1 \leq i \neq j \leq m$, $q_i \neq q_j$ or $grd_i \wedge grd_j = false$. The semantics of a broadcast transition of the above form is the following. A process, *the sender*, in state q_0 and satisfying the condition grd_0, changes its state to q_0' and changes its variables according to $stmt_0$. Any other process, *a receiver*, in some process state $u = (q, v)$ proceeds as follows: (i) either it fires t_i if $q = q_i \wedge u \models grd_i$ for some $i : 1 \leq i \leq m$ in which case the receiver is called *active*; or, (ii) it remains *passive* otherwise.

The refinement protocol for broadcast transitions is similar to the one described for global conditions. The only difference lies in the second phase of the protocol. More precisely, an active receiver changes its local state and variables according to the rule, and sends back an acknowledgment. A passive receiver only sends back an acknowledgment. We extend our approximation to broadcast rules, by removing all processes that have not acknowledged the request of the initiator. The computation of *Pre* can be extended in a straightforward manner.

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).

9 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 Section 8. 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.

Table 1. Experimental results on several mutual exclusion algorithms.

	ref	*it*	time	final constr.	memory
Burns	$\mathcal{R}_1$	26	< 1 s	44	1MB
Dijkstra	$\mathcal{R}_1$	93	< 1 s	41	1MB
Bakery	$\mathcal{R}_2$	4	< 1 s	12	1MB
Bakery	$\mathcal{R}_3$	4	< 1 s	12	1MB
2-phase Commit	-	6	< 1 s	9	1MB
Lamport	-	29	30 m	4676	222MB

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.

A detailed description of the case studies can be found in Appendix B.

10 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] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[2] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *Proc.* 19th *Int. Conf. on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 145–157, 2007.

[3] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *Proc. TACAS '07,* 13th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 721–736. Springer Verlag, 2007.

[4] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002,* 13th *Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.

[5] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In Berry, Comon, and Finkel, editors, *Proc.* 13th *Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234, 2001.

[6] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc.* 15th *Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.

[7] D. Chkliaev, J. Hooman, and P. van der Stok. Mechanical verification of transaction processing systems. In *ICFEM 2000*, 2000.

[8] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. VMCAI '06,* 7th *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141, 2006.

[9] G. Delzanno. Automatic verification of cache coherence protocols. In Emerson and Sistla, editors, *Proc.* 12th *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Verlag, 2000.

[10] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1992.

[12] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.

[13] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV 2004*, pages 135–147, 2004.

[14] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[15] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[16] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. chang, M. Colón, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STEP: the Stanford Temporal Prover. Draft Manuscript, June 1994.

[17] P. Revesz. A closed form evaluation for datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1):117–149, 1993.

[18] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1st *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

A Appendix – Construction of Pre

Auxiliary Definitions The following definitions are needed in the sequel. For a transition rule t of the form (1), we define the function $[t]$ on $X \cup Q^\bullet$ as follows: for each $x \in X$, $[t](x) = stmt(x)$ if x occurs in $stmt$ and $[t](x) = x$ otherwise. For each $q'' \in Q^\bullet$, $[t](q'') = true$ if $q'' = q'$, and $false$ otherwise. For $\theta \in \mathbb{B}(X \cup Q)$, we use $\theta[t]$ to denote the formula obtained from θ by substituting all occurrences of elements in θ by their corresponding $[t]$-images.

For a constraint ϕ', we define $Pre(\phi') = \bigcup_{t \in T} Pre_t(\phi')$, i.e., we compute the set of predecessor constraints with respect to each transition rule $t \in T$. First, we define a simple operator. For natural numbers $j, k \geq 1$, we define $k \odot j$ to be k if $k < j$ and $k+1$ if $k \geq j$. For instance $2 \odot 4 = 2$ and $7 \odot 4 = 8$.

Computing Predecessors In the following, assume t to be a transition rule of the form (1). Consider a constraint $\phi' = (\Theta', E')$ with $|\phi| = n$. We define $Pre_t(\phi')$ to be the set of all constraints $\phi = (\Theta, E)$ such that there is an $i : 1 \leq i \leq n$ and one of the following conditions are satisfied:

1. $grd \in \mathbb{B}(X)$ (i.e. grd is a local condition), $|\phi| = n$, $\forall j : 1 \leq j \neq i \leq n : \Theta(j) = \Theta'(j)$, $\Theta(i) = \Theta'(i)[t] \wedge grd \wedge q$, and $E = E'$. This case corresponds to running a transition with a local condition backwards. The transition is performed by process i. The local state and variables of process i are changed according to the transition t. This is reflected in the definition of $\Theta(i)$ which is essentially the weakest precondition of $\Theta'(i)$ with respect to t. The local states and variables of the other processes are not changed and hence $\Theta(j) = \Theta'(j)$ for $j \neq i$. The edges are not changed either, and hence $E = E'$ (Figure 4).

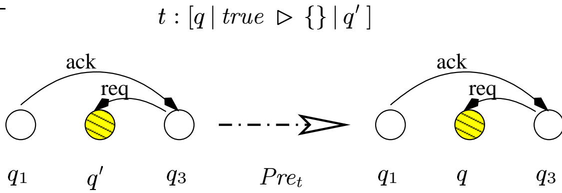


Figure 4. Computing Pre **for local transitions.**

2. $grd = \Box\theta$, $\Box \in \{\forall_L, \forall_R, \forall_{LR}\}$, $|\phi| = n$, $\Theta(i) = \Theta'(i)[t] \wedge q^t$, $\forall j : 1 \leq j \neq i \leq n : \Theta(j) = \Theta'(j)$, and depending on $\Box$, one of the following conditions holds:

   (a) $\Box = \forall_L$ and $E = E' \cup \{i \xleftarrow{\mathsf{ack}}_t j \,|\, 1 \leq j < i\}$.

   (b) $\Box = \forall_R$ and $E = E' \cup \{i \xleftarrow{\mathsf{ack}}_t j \,|\, i < j \leq n\}$.

   (c) $\Box = \forall_{LR}$ and $E = E' \cup \{i \xleftarrow{\mathsf{ack}}_t j \,|\, 1 \leq i \neq j \leq n\}$.

   This case corresponds to running the third phase of the refinement protocol backwards when the global condition is universally quantified. The local state of process i is changed back to the temporary state q^t, and the assignment in t is performed backwards. The local states and variables of the other processes are not changed. All the relevant processes should have an acknowledgment edge to process i (since all relevant processes not having such an edge are eliminated according to the approximate transition relation). Notice that the resulting constraint does not put constraints on the edges of irrelevant processes (e.g. processes with $j > i$ for $\forall_L$), i.e., its denotation contains all possible choices for these edges. For instance if the quantifier is $\forall_L$ then we add acknowledgment edges from all processes to the left of process i. The cases of $\forall_R$ and $\forall_{LR}$ are analogous (Figure 5).

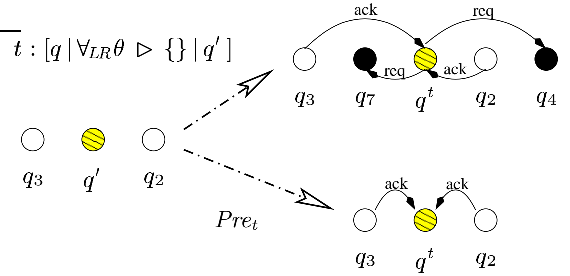


Figure 5. Computing Pre **for the third phase of the refinement protocol in the case of a universally quantified global transition.**

3. $grd = \Box\theta$, $\Box \in \{\exists_L, \exists_R, \exists_{LR}\}$, $|\phi| = n$, $\Theta(i) = \Theta'(i)[t] \wedge q^t$, $\forall j : 1 \leq j \neq i \leq n : \Theta(j) = \Theta'(j)$, and depending on $\Box$, one of the following conditions holds:

   (a) $\Box = \exists_L$ and $E = E' \cup \{i \xleftarrow{\mathsf{ack}}_t j\}$ for some $j : 1 \leq j < i$.

   (b) $\Box = \exists_R$ and $E = E' \cup \{i \xleftarrow{\mathsf{ack}}_t j\}$ for some $j : i < j \leq n$.

   (c) $\Box = \exists_{LR}$ and $E = E' \cup \{i \xleftarrow{\mathsf{ack}}_t j\}$ for some $j : 1 \leq i \neq j \leq n$.

   This is one of the two cases which correspond to running the third phase of the refinement protocol

backwards when the global condition is existentially quantified. The case is similar to that of universal quantifiers, except that we require only one relevant process (rather than all) to have an acknowledgment edge (Figure 6, the bottom-left configuration).

4. $grd = \Box\theta$, $\Box \in \{\exists_L, \exists_R, \exists_{LR}\}$, $|\phi| = n+1$, there is a $j : 1 \le j \le n+1$ such that:
   (a) $\Theta(i \odot j) = \Theta'(i)[t] \wedge q^t$.
   (b) $\Theta(j) = true$.
   (c) $\Theta(k \odot j) = \Theta'(k)$ for each $k : 1 \le k \ne i \le n$.
   (d) e is the smallest set of edges containing the following elements:
      i. $(k_1 \odot j) \xleftarrow{\texttt{ack}}_t (k_2 \odot j)$ if $\left(k_1 \xleftarrow{\texttt{ack}}_t k_2\right) \in E'$ for all $k_1, k_2 : 1 \le k_1 \ne k_2 \le n$.
      ii. $(k_1 \odot j) \xrightarrow{\texttt{req}}_{t'} (k_2 \odot j)$ if $\left(k_1 \xrightarrow{\texttt{req}}_{t'} k_2\right) \in E'$ for all $k_1, k_2 : 1 \le k_1 \ne k_2 \le n$ and $t' \in T$.
      iii. $(i \odot j) \xleftarrow{\texttt{ack}}_t j$.
   (e) Depending on $\Box$, one of the following conditions holds:
      i. $\Box = \exists_L$ and $j \le i$,
      ii. $\Box = \exists_R$ and $j > i$,
      iii. $\Box = \exists_R$.

This is the second case which corresponds to running the third phase of the refinement protocol backwards when the global condition is existentially quantified. The difference compared to case 3 is that the process which sends an acknowledgment is not part of the representation of ϕ'. Therefore, we add a new process with index j to ϕ. The state of the initiator is changed according to case 4a (in a similar manner to cases 2 and 4). The only required condition on the new process is that it has sent an acknowledgment to the initiator. The local state and variables of that process are constrained (case 4b). The states and local variables of the other processes are not changed (case 4c). Existing edges are maintained when going backwards (cases 4(d)i and 4(d)ii), while the new acknowledgment edge is added from the new process to the initiator (case 4(d)iii) (Figure 6, the top-left configuration).

5. $grd = \Box\theta$, $|\phi| = n$, $\nexists j : 1 \le j \ne i \le n : i \xleftarrow{\texttt{ack}}_t j \in E'$, $\forall j : 1 \le j \ne i \le n : \Theta(j) = \Theta'(j)$, $\Theta(i) = \Theta'(i)[t_q] \wedge q$, where t_q is the transition

$$[\,q \mid true \rhd \{\} \mid q^t\,]$$

that represents the update of the local state, and $E = E' - \{i \xrightarrow{\texttt{req}}_t j \mid i \ne j\}$ (Figure 7).

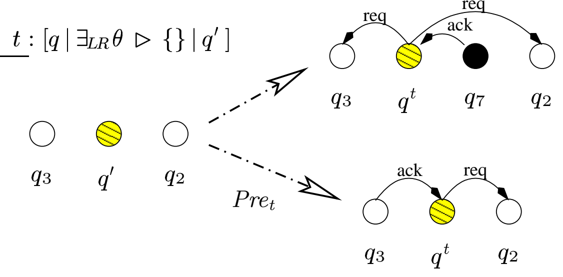


Figure 6. **Computing *Pre* for the third phase of the refinement protocol in the case of an existentially quantified global transition.**

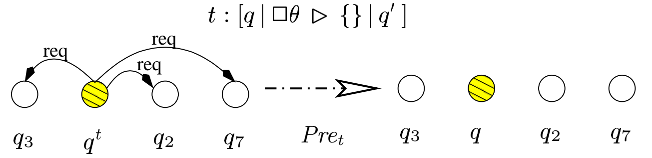


Figure 7. **Computing *Pre* for the first phase of the refinement protocol.**

6. $grd = \Box\theta$, $|\phi| = n$, $\exists i, j : 1 \le j \ne i \le n : i \xleftarrow{\texttt{ack}}_t j \in E'$, $\forall k : 1 \le k \ne j \le n : \Theta(k) = \Theta'(k)$, $\Theta(j) = \Theta'(j) \wedge \theta$, and $E = E' - \{i \xleftarrow{\texttt{ack}}_t j\} \cup \{i \xrightarrow{\texttt{req}}_t j\}$ (Figure 8).

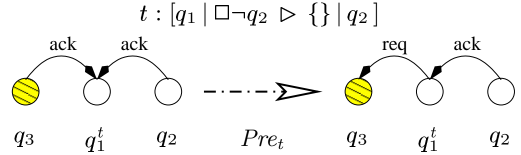


Figure 8. **Computing *Pre* for the second phase of the refinement protocol.**

Observe that in case (4), the length of constraint ϕ is larger than that of ϕ'. This means that the sizes of the constraints which arise during the analysis are not a priory bounded.

B Appendix – Case Studies

In the following section, we give detailed descriptions of the case studies. For each example, we give the original model followed by the corresponding refinement protocol. In the description of the original model, we use the atomic syntax, while the refinement protocol describes how the corresponding global transitions are implemented.

Let $\mathcal{P} = (Q, X, T)$ be a parameterized system and let $t \in T$ be a transition of the form:

$$t : [q \mid grd \rhd stmt \mid q'] \tag{2}$$

where $grd = \Box\theta$ is a global condition. We use the following notations for the three phases of the refinement protocol of a transition t of the form (2).

First phase: $\texttt{sendreq}_t\ q$ means that a process in control state q initiates the first phase of refinement protocol for t by broadcasting request edges to all other processes and changing its state to q^t.

Second phase: $\texttt{sendack}_t\ \texttt{if}\ \theta$ denotes that a process satisfying the formula θ acknowledges a request edge on t where he is the target.

Third phase: $\texttt{test}_t\ \Box\ \texttt{ack goto}\ q'\ \texttt{with}\ stmt$ denotes that if all edges (or at least one depending on the quantifier $\Box$) in the range of $\Box$ have been acknowledged, then the initiator moves to q', changes its variables according to the the statement $stmt$, and removes all edges on the transition t. We omit $stmt$ when it is not relevant.

We use respectively **tt** and **ff** to denote the Boolean values $true$ and $false$.

B.1 Burn's Algorithm

In order to model Burn's mutual exclusion algorithm, we consider a parameterized system where each process has a local Boolean variable f (for $flag$). The local state ranges over $\{q_1, \ldots, q_7\}$ where q_6 represents the critical section. The transitions are described below.

Burn's Algorithm

Instance

Q: $q_1, \ldots, q_7$

X: $f \in \mathcal{B}$

T:

$t_1 : [q_1 \mid \mathbf{tt} \rhd f = \mathbf{ff} \mid q_2]$ $\quad t_2 : [q_2 \mid \exists_L f \rhd \{\} \mid q_1]$

$t_3 : [q_2 \mid \forall_L \neg f \rhd \{\} \mid q_3]$ $\quad t_4 : [q_3 \mid \mathbf{tt} \rhd f = \mathbf{tt} \mid q_4]$

$t_5 : [q_4 \mid \exists_L f \rhd \{\} \mid q_1]$ $\quad t_6 : [q_4 \mid \forall_L \neg f \rhd \{\} \mid q_5]$

$t_7 : [q_5 \mid \forall_R \neg f \rhd \{\} \mid q_6]$ $\quad t_8 : [q_6 \mid \mathbf{tt} \rhd f = \mathbf{ff} \mid q_7]$

$t_9 : [q_7 \mid \mathbf{tt} \rhd \{\} \mid q_1]$

Initial Process State u_{init}: $q_1, f \mapsto \mathbf{ff}$

Final Constraints Φ_F: $(q_6 q_6, \emptyset)$

The refinement protocol for the global transitions is as follows.

Burn's Refinement

$$t_2 : \begin{bmatrix} \texttt{sendreq}_{t_2}\ q_2 \\ \texttt{sendack}_{t_2}\ \texttt{if}\ f \\ \texttt{test}_{t_2}\ \exists_L\ \texttt{ack goto}\ q_1 \end{bmatrix}$$

$$t_3 : \begin{bmatrix} \texttt{sendreq}_{t_3}\ q_2 \\ \texttt{sendack}_{t_3}\ \texttt{if}\ \neg f \\ \texttt{test}_{t_3}\ \forall_L\ \texttt{ack goto}\ q_3 \end{bmatrix}$$

$$t_5 : \begin{bmatrix} \texttt{sendreq}_{t_5}\ q_4 \\ \texttt{sendack}_{t_5}\ \texttt{if}\ f \\ \texttt{test}_{t_5}\ \exists_L\ \texttt{ack goto}\ q_1 \end{bmatrix}$$

$$t_6 : \begin{bmatrix} \texttt{sendreq}_{t_6}\ q_4 \\ \texttt{sendack}_{t_6}\ \texttt{if}\ \neg f \\ \texttt{test}_{t_6}\ \forall_L\ \texttt{ack goto}\ q_5 \end{bmatrix}$$

$$t_7 : \begin{bmatrix} \texttt{sendreq}_{t_7}\ q_5 \\ \texttt{sendack}_{t_7}\ \texttt{if}\ \neg f \\ \texttt{test}_{t_7}\ \forall_R\ \texttt{ack goto}\ q_6 \end{bmatrix}$$

Any process in state q_2 initiates non-deterministically the refinement protocol for one of the transitions t_2 or t_3. In both cases, the initiator broadcast request edges on the transition he has chosen and moves to the corresponding waiting state. In case of transition t_2, the process moves to q_1 if at least one edge with a process to its left has been acknowledged; while in case of transition t_3, the process moves to q_3 if all edges with all processes to its left have been acknowledged. In state q_4, a similar situation occurs meaning that a process at q_4 chooses non-deterministically between initiating t_5 or t_6. The refinement protocol of transition t_7 implies that a process in state q_5 can move to the critical section (q_6) only if the second phase succeeds with all processes to its right; i.e., only if all the processes to its right have

acknowledged the corresponding requests.

B.2 Dijkstra's Algorithm

In Dijkstra's model, a process has seven states $q_1, \ldots, q_7$ where q_6 represents the critical section. The local variables are the Boolean p for *pointer* and the number $f \in \{0, 1, 2\}$ for $flag$.

Dijkstra's Algorithm
Instance
Q: $q_1, \ldots, q_7$
X: $p \in \mathcal{B}, f \in [0..2]$
T:

$t_1 : [q_1 \,|\, \mathbf{tt} \rhd f = 1 \,|\, q_2\,]$
$t_2 : [q_2 \,|\, \forall_{LR}(f = 0 \vee \neg p) \rhd \{\} \,|\, q_3\,]$
$t_3 : [q_3 \,|\, \mathbf{tt} \rhd p = \mathbf{tt} \,|\, q_4\,]\,[q_1 \,|\, \mathbf{tt} \rhd p = \mathbf{ff} \,|\, q_1\,]^*$
$\ldots [q_7 \,|\, \mathbf{tt} \rhd p = \mathbf{ff} \,|\, q_7\,]^*$
$t_4 : [q_4 \,|\, \mathbf{tt} \rhd f = 2 \,|\, q_5\,]$ $t_5 : [q_5 \,|\, \forall_{LR} f \neq 2 \rhd \{\} \,|\, q_6\,]$
$t_6 : [q_5 \,|\, \exists_{LR} f = 2 \rhd \{\} \,|\, q_1\,]$ $t_7 : [q_6 \,|\, \mathbf{tt} \rhd f = 0 \,|\, q_7\,]$
$t_8 : [q_7 \,|\, \mathbf{tt} \rhd \{\} \,|\, q_1\,]$

Initial Process State u_{init}: $q_1, f \mapsto 0, p \mapsto \mathbf{ff}$
Final Constraints Φ_F: $(q_6 q_6, \emptyset)$

Below, we give the refinement protocol of the global and the broadcast transitions.

Dijkstra's Refinement

$$t_2 : \left[\begin{array}{l} \texttt{sendreq}_{t_2}\ q_2 \\ \texttt{sendack}_{t_2}\ \texttt{if}\ f = 0 \vee \neg p \\ \texttt{test}_{t_2}\ \forall_{LR}\ \texttt{ack goto}\ q_3 \end{array}\right]$$

$$t_3 : \left[\begin{array}{l} \texttt{sendreq}_{t_3}\ q_3 \\ \texttt{sendack}_{t_3}\ \texttt{if}\ \mathbf{tt}\ \texttt{and}\ p := \mathbf{ff} \\ \texttt{test}_{t_3}\ \forall_{LR}\ \texttt{ack goto}\ q_4\ \texttt{with}\ p = \mathbf{tt} \end{array}\right]$$

$$t_5 : \left[\begin{array}{l} \texttt{sendreq}_{t_5}\ q_5 \\ \texttt{sendack}_{t_5}\ \texttt{if}\ f \neq 2 \\ \texttt{test}_{t_5}\ \forall_{LR}\ \texttt{ack goto}\ q_6 \end{array}\right]$$

$$t_6 : \left[\begin{array}{l} \texttt{sendreq}_{t_6}\ q_5 \\ \texttt{sendack}_{t_6}\ \texttt{if}\ f = 2 \\ \texttt{test}_{t_6}\ \exists_{LR}\ \texttt{ack goto}\ q_1 \end{array}\right]$$

In the original algorithm, a pointer, i.e., a variable ranging over process indices is used. We model this by the local Boolean variable p. A process has the variable p equal to **tt** iff it is being pointed to by the pointer variable. In order to simulate the pointer changes, we use the broadcast transition t_3. By firing t_3, a process changes state from q_3 to q_4, changes its local variable p to **tt** and simultaneously changes p to **ff** in all other processes. Observe that in the refinement protocol of broadcast transitions, any other process changes its state (or variables depending on the transition) in the second phase; i.e., whenever the process acknowledges the corresponding request edge.

A process in state q_3 fires t_3 by executing the three phases of the refinement protocol as described below. First, the the process (the initiator) sends request edges on t_3 to all other processes and moves to the temporary state $q_3^{t_3}$. Any process that receives such a request changes it is local variable p to **ff** and acknowledges the request. Once all edges on t_3 are acknowledged, the initiator moves to q_4 and changes its variable p to **tt**. The refinement protocol of the remaining global transitions is as described in the main text.

B.3 Lamport's Bakery Algorithm

Lamport's Bakery Algorithm is a well-known solution to the critical section problem for an arbitrary, finite number of processes [14].

var

- choosing: shared array [0..n-1] of boolean;
- number: shared array [0..n-1] of integer;

```
1.  Process P[i] :=
2.  loop forever
3.    choosing[i] := true;
4.    number[i] := max(number[0],number[1],...,number[n-1]) + 1;
5.    choosing[i] := false;
6.    for j := 0 to n-1 do begin
7.      while choosing[j] do nothing;
8.      while number[j] ≠ 0 and
9.        (number[j], j) ≪ (number[i],i) do
10.         nothing;
11.       end;
12. critical section
13. number[i] := 0;
14. remainder section
```

Figure 9. Lamport's bakery algorithm: $(a, b) \ll (c, d)$ **iff** $a < c$ **or** $(a = c$ **and** $b < d)$.

The rationale behind the algorithm is as follows. Each process has a local variable *num* in which it stores a ticket. Initially *num* is set to zero. When a process is interested in entering the critical section, it sets *num* to a value strictly greater than the tickets (i.e., value of *num*) of all other processes in the system. A possible way to implement this step is by taking the maximum value of *num* in all processes and then incrementing it by one. More in general, we just need to generate a *fresh* ticket. After the choosing step, the process waits until its ticket is less than the tickets of all other processes and then enters the critical section. When it

releases the critical section, it resets its ticket to zero. The pseudo code of the algorithm is shown in Figure 9.

The original Lamport's Bakery algorithm does not take any atomicity assumption on the computation of the fresh tickets. Since two processes may take the same number, in the entry condition we have to compare tickets and identifiers: if two processes have the same ticket, the one with smaller identifier has higher priority. We assume that all identifiers are different. Furthermore, to protect the test in the entry section from race conditions in the choosing phase, the algorithm uses one Boolean flag per process (an array *choosing*) to mark the assignment to the local variable num. The flag is set to true before starting the assignment and to false after its completion. A process willing to enter the critical section is forced to wait until all processes have concluded their choosing phase before comparing its ticket with the other ones.

The bakery algorithm can be specified using the parameterized system with global conditions shown below. Individual processes have two local variables id (identifier) and num (ticket) that range over natural numbers and four possible states $idle, choose, wait, use$. In the model description we let $\theta(p)$ denote the following formula.

$$\begin{pmatrix} p{\cdot}num = 0 \ \vee \ num < p{\cdot}num \ \vee \\ \vee \ (num = p{\cdot}num \ \wedge \ id < p{\cdot}id) \end{pmatrix}$$

Local variables: $X = \{id, num \in \mathbb{N}\}$
States: $Q = \{idle, choose, wait, use\}$
Transitions:
$t_1 : [\,idle \mid \mathbf{tt} \mid choose\,]$
$t_2 : \left[\,choose \mid \forall p : (num^{next} > p{\cdot}num) \mid wait\,\right]$
$t_3 : [\,wait \mid \forall p : (\neg p{\cdot}choose \wedge \theta(p)) \mid use\,]$
$t_4 : \left[\,use \mid num^{next} = 0 \mid idle\,\right]$

Initially, the state is *idle* and the value of num is *zero* for each process. In t_1 a process moves from *idle* to *choose* to start the choosing phase. In t_2 a process moves from *choose* to *wait* and assigns to num a new ticket, i.e., a value strictly greater than the value of num in all other process ($pnum$). In t_3 a process moves from *wait* to *use* only if the entry condition holds for each other process. In rule t_4 a process jumps back to *idle* and resets num to zero.

We are interested in verifying mutual exclusion for this protocol. Unsafe states here are configurations in which at least two processes are inside the critical section (in state *use*).

In the rest of the section we discuss the analysis of this protocol with different variant of refinement protocols for the transitions t_2 and t_3. All refinement protocols make use of a *local* universal global condition involving process states attached to acknowledgements. In order to distinguish them from non-atomic conditions, we write them as $\forall^{local} p.\ ack_p(u) \ \wedge \ \theta$. This condition is used by the initiator to check if all acknowledgments have been received. We use the approximation scheme described in the paper to deal with this kind of conditions.

B.3.1 Refinement Protocol $\mathcal{R}_2$

As a first formal model of the bakery algorithm with non-atomic conditions, we consider one of the refinement protocols of Section 8. Here, the initiator i sends again a request to receptor j in case the global condition is violated by the values returned by j. In this case study, the refinement protocol works as follows. Let us first consider transition t_2. The initiator is a process in state *choose*.

- The initiator sends a request for reading the local states of the other processes and then moves to the temporary state $choose^{t_2}$. We recall that Boolean conditions are evaluated on state $choose^{t_2}$ as for state *choose*.
- A responder p sends the current local state along with an acknowledgment, i.e., a message $ack^1_p(s_p, id_p, num_p)$ where s_p is the current state, id_p and num_p the value of the local variables of the responder.
- The initiator checks that all acknowledgments have been received, applies the global condition on the values attached to messages, and then moves to state *wait*. Specifically, the initiator with local variable num checks the *local* condition

$$\forall^{local} p.\ (ack^1_p(s_p, id_p, num_p) \ \wedge \ num^{next} > num_p)$$

Observe that the processes acknowledge the request in any order, and that the initiator tests the condition on the set of messages sent by other processes (i.e. other operations may occur between the time a responder sends an *ack* and the time the initiator checks the condition).

Let us now consider transition t_3. The initiator is now a process in state *wait*.

- The initiator sends requests for reading the local states of the other processes and then moves to the temporary state $wait^{t_3}$.
- A responder p sends the current local state along with an acknowledgment, i.e., a message $ack^2_p(s_p, id_p, num_p)$.

- The initiator checks that all acknowledgments have been received. Then, it applies the entry condition on the values attached to messages. Specifically, the initiator, with local variables *num* and *id*, tests the following condition.

$$\forall^{local} p \cdot \begin{pmatrix} ack_p^2(s_p, id_p, num_p) \wedge \ s_p \neq choose \wedge \\ (num_p = 0 \vee num < num_p \vee \\ (num = num_p \wedge id < id_p)) \end{pmatrix}$$

  - If the condition succeeds, the initiator moves to state *use*.
  - If the condition fails for a process p, the initiator sends a new request to process p, i.e., the acknowledgment edge from p is replaced by a new request edge.

Notice that the initiator does not change state until the entry condition is satisfied. Thus, this protocol models a busy wait for acknowledgments with good values.

B.3.2 Refinement Protocol $\mathcal{R}_3$

The second refinement protocol we consider is a slight variation of the previous one. Specifically, in the refinement of transition t_3, when the entry condition fails for some process p, the initiator resets all edges and jumps back to state *wait*. This operation restarts the refinement protocol for the same transition t_3.

B.4 Lamport's Distributed Mutual Exclusion Algorithm

Unlike the algorithms discussed so far, Lamport's distributed mutual exclusion algorithm [15] was originally defined for processes communicating by message passing. This means that non-atomicity is inherent in this model.

A process in this model has three states: $idle, wait$ and *use* (for critical section). Each process i has a logical clock $clock_i$ that ranges over the natural numbers, and a local queue Q_i in which i stores the timestamped requests. Mutual exclusion is guaranteed by letting only the process with the "earliest" request access the critical section.

In order to determine which request is the earliest, the algorithm makes use of the well known Lamport's *happened-before* partial ordering. A total order is then derived by combining the natural order on *id*'s (process indices) with the happened-before order. Now in each queue, requests are ordered by their associated timestamps, and, in case the timestamps are equal, by the *id*'s of the senders.

Depending on its current state, a process i may perform one of the following transitions. Observe that when performing these transitions the process increases its clock.

Idle: Process i broadcasts the request $req_i(clock_i)$, stores a copy of $[i, clock_i]$ in its own queue Q_i and moves to state $wait$.

Wait: Process i moves to state use, i.e. accesses the critical section, when replies are collected from all other processes, and i's own request is the smallest in the local queue Q_i.

Use: Process i exits from use to $idle$ by removing all copies $[i, clock]$ (where $clock$ is an arbitrary value) from its local queue, and broadcasting a release message $rel_i(clock_i)$.

Each time a process j receives a message with timestamp ts, it updates its local clock to a value strictly larger than $\max(clock, ts)$, and, depending on the type of the message, does the following.

Request: When the process j receives a request $req_i(ts)$ from a process i, it adds (i, ts) to its own local queue Q_j, and sends back a reply message $ack_j(clock_j)$.

Acknowledgment: In case process j receives the reply $ack_i(ts)$ from process i, it updates its local clock.

Release: When the process j receives a release message $rel_i(ts)$, it removes all pairs $[i, clock]$ (for any value clock) from its local queue Q_j.

Our model of the distributed Lamport algorithm consists of 8 rules. We used unbounded variables carried by the graph edges to keep track of the content of local queues. During the analysis, our prototype generated 4676 constraints after 29 iterations.

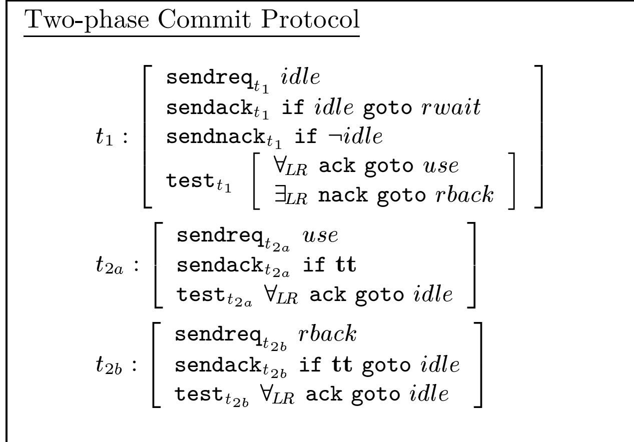
B.5 Distributed Two-phase Commit Protocol

Protocols for controlling transactions in distributed database systems can naturally be viewed as refinement protocols for atomic transitions. To illustrate this, let us consider the following simple parameterized system

$$\begin{aligned} t_1 &: [idle \mid \forall_{LR}\ idle \rhd \{\} \mid use] \\ t_2 &: [use \mid \mathbf{tt} \rhd \{\} \mid idle] \end{aligned}$$

where *use* indicates that a process is in the critical section. In a distributed environment in which global conditions cannot be tested atomically, the access to

the critical section can be viewed as a special type of "all-or-nothing" transaction, i.e., either it can be executed atomically or the state of the system rolls back to a safe one. Protocols like the *two-phase commit algorithm* [11] can thus be applied to safely implement our protocol. We consider here a *distributed* version of the two-phase commit protocol scheme in which the initiator is selected dynamically. The refinement protocol based on the two-phase commit algorithm for our simple mutex is defined then as follows.



The intuition here is as follows. Initially all agents are in state *idle*. Any *idle* agent can nondeterministically initiate the protocol. The initiator and the other processes (cohorts) then execute the following steps.

Phase 1 The initiator sends a request to enter the critical section to all other processes (cohorts) and then waits for a message from each cohort. A cohort replies with an agreement message if his/her state is *idle* or with an abort message if his/her state is different from *idle*.

Phase 2: Success If the initiator receives agreement messages from all cohorts, he/she sends a commit message to all the cohorts and enters the critical section. When the initiator releases the critical section, he/she sends a notification to all cohorts. A cohort sends an agreement and moves to state *idle*. When the initiator receives all agreements, he/she moves to state *idle*.

Phase 2: Failure If any cohort sends an abort message during Phase 1, the initiator sends a rollback message to all cohorts. A cohort sends an agreement and moves to state *idle*. When the initiator receives all agreements, he/she moves to state *idle*.

The protocol must ensure mutual exclusion, i.e., in every reachable state there cannot be two agents in their critical section at the same time. This protocol can naturally be encoded in our specification language by adding `nack` edges to the request graphs.