Monotonic Abstraction in Action

(Automatic Verification of Distributed Mutex Algorithms)

Parosh Aziz Abdulla[1] parosh@it.uu.se,
Giorgio Delzanno[2] giorgio@disi.unige.it, and
Ahmed Rezine[1] Rezine.Ahmed@it.uu.se

[1] Uppsala University, Sweden
[2] Università di Genova, Italy.

Abstract. We consider verification of safety properties for *parameterized distributed protocols*. Such a protocol consists of an arbitrary number of (infinite-state) processes that communicate asynchronously over FIFO channels. The aim is to perform *parameterized verification*, i.e., showing correctness regardless of the number of processes inside the system. We consider two non-trivial case studies: the distributed Lamport and Ricart-Agrawala mutual exclusion protocols. We adapt the method of *monotonic abstraction* that considers an over-approximation of the system, in which the behavior is monotonic with respect to a given pre-order on the set of configurations. We report on an implementation which is able to fully automatically verify mutual exclusion for both protocols.

1 Introduction

In this paper, we consider automatic verification of safety properties for *parameterized distributed protocols*. Such a protocol consists of an arbitrary number of concurrent processes communicating asynchronously. The aim is to prove correctness of the protocol regardless of the number of processes.

Several aspects of the behavior of distributed protocols make them extremely difficult to analyze. First, the processes communicate asynchronously through channels and shared variables. Each process may operate on *heterogeneous* data types such as Boolean, integers, counters, logical clocks, time stamps, tickets, etc. Furthermore, such protocols often involve *quantified* conditions. For instance, a process may need to receive acknowledgments from *all* the other processes inside the system, before it is allowed to perform a transition. Finally, these protocols are often *parameterized* meaning we have to verify correctness of an infinite family of systems each of which is an infinite-state system. Here, we refine the method of [3, 2] based on *monotonic abstractions* to perform fully automatic verification of two difficult examples; namely the distributed mutual exclusion algorithm by Lamport [17]; and its modification by Ricart and Agrawala [21].

We model a parametrized distributed system (or a distributed system for short) as consisting of an arbitrary number of processes. Each process is an extended finite-state automaton which operates on a number of Boolean and numerical (natural number) variables. Each pair of processes is connected through

a number of bounded FIFO-channels which the processes use to interchange messages. A transition inside a process may be conditioned by the local variables and the messages fetched from the heads of the channels accessible to the process. The conditions on the numerical variables are stated as *gap-order constraints*. Gap-order constraints [19] are a logical formalism in which we 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. Also, as mentioned above, one important aspect in the behavior of distributed protocols is the existence of *quantified* conditions. This feature is present for instance in the Lamport and Ricart-Agrawala protocols. Here, the process which is about to perform a transition needs to know (or receive) information (e.g., acknowledgments) from the other processes inside the system. Since a process cannot communicate directly with the other processes, it keeps instead information locally about them. This local information is stored through a number of variables which we call *record variables*. A process has a copy of each record variable corresponding to each other process inside the system. As an example, consider a system with n processes. Suppose that, in the protocol, a process needs to receive acknowledgments from all the other processes. The protocol then uses a Boolean variable *ack* to record information about received acknowledgments. Then, a process (say process i) will have $n-1$ copies of the variable *ack*, where each copy corresponds to another process (the copy corresponding to process j records whether process i has received an acknowledgment from process j). When process i receives an acknowledgment from process j through the relevant channel, it assigns *true* to its copy of *ack* corresponding to process j. Process i performs the transition by *universally* quantifying over all its copies of *ack*, i.e., checking that all of them are set to true. We can also have existential quantification in which case the process checks that *some* local copy has a certain value (rather than *all* local copies).

In this paper, we report on two case studies where we use our model of distributed systems to describe parameterized versions of the distributed Lamport and Ricart-Agrawala protocols. We have verified fully automatically the protocols, using a tool which adapts the method of *monotonic abstractions* reported in [3, 2]. The idea of monotonic abstraction is to make use of the theory of *monotonic programs*. In fact, one of the widely adopted frameworks for infinite-state verification is based on the concept of transition systems which are monotonic with respect to a given pre-order on the set of configurations. This framework provides a scheme for symbolic backward reachability analysis, and it has been used for the design of verification algorithms for various models including Petri nets, lossy channel systems, timed Petri nets, broadcast protocols, etc. (see, e.g., [5, 12, 13, 1]). The main advantage of the method is that it allows to work on (infinite) sets of configurations which are upward closed with respect to the pre-order. These sets have often very efficient symbolic representations (each upward closed set can be uniquely characterized by its minimal elements) which makes them attractive to use in reachability analysis. Unfortunately, many systems do not fit into this framework, in the sense that there is no nontrivial (useful) order-

ing for which these systems are monotonic. The idea of *monotonic abstractions* [3, 2] is to compute an over-approximation of the transition relation. Given a preorder $\preceq$, we define an abstract semantics of the considered systems which ensures their monotonicity. Basically, the idea is to consider that a transition is possible from a configuration c_1 to c_2 if it is possible from c_1 to a larger configuration $c_3 \succeq c_2$. The whole verification process is fully automatic since both the approximation and the reachability analysis are carried out without user intervention. Observe that if the approximate transition system satisfies a safety property then we can safely conclude that the original system satisfies the property, too. Based on the method, we have implemented a prototype and applied it for *fully automatic* verification of the distributed Lamport and Ricart-Agrawala protocols. Termination of the approximated backward reachability analysis is not guaranteed in general.

Related Work This paper gives detailed descriptions of two non-trivial case studies, where we adapt monotonic abstraction [2–4] to the case of distributed protocols. Compared to the methods of [3] and [2, 4] which operate on simple Boolean and integer variables respectively, our formalism allows the modeling of heterogeneous data types such as FIFO queues, logical clocks, etc, which are very common in the deigns of distributed protocols.

In [24], the authors consider distributed protocols with a bounded number of processes, and also build for heterogeneous systems (e.g., with Booleans and integers) on top of the Omega-based solver. Here, we have a tool for heterogeneous data types built on top of our verification method [2–4] which allows to deal with unbounded numbers of components. There have been several works on verification of parameterized systems of *finite-state processes*, e.g., regular model checking [15, 6, 8] and counter abstraction methods [11, 14, 12, 13]. In our case, the processes are infinite-state, and therefore our examples cannot be analyzed with these methods unless they are combined with additional abstractions. Furthermore all existing automatic parameterized verification methods (e.g., [15, 6, 8, **?**,11, 3, 2]) are defined for systems under the (practically unreasonable) assumption that quantified conditions are performed atomically (globally). In other words, the process is assumed to be able to check the states of all the other processes in one atomic step. On the other hand, in our quantified conditions, the process can only check variables which are local to the process. Non-atomic versions of parameterized mutual exclusion protocols such as the Bakery algorithm have been studied with heuristics to discover invariants, ad-hoc abstractions, or semi-automated methods in [7, 16, 18, 9, 10]. In contrast to these methods, our verification procedure is fully automated and is based on a more realistic model.

A parameterized formulation of the Ricart-Agrawala algorithm has been verified semi-automatically in [22], where the STeP prover is used to discharge some of the verification conditions needed in the proof. We are not aware of other attempts of fully automatic verification of parameterized versions of the Ricart-Agrawala algorithm or of the distributed version of Lamport's distributed algorithm.

Outline In the next section, we give preliminaries, and in Section 3 we describe our model for distributed systems (protocols). In Section 4, we give the operational semantics by describing the (infinite-state) transition system induced by a distributed system. In Section 5, we introduce an ordering on the set of configurations of the system, and explain how to specify safety properties (such as mutual exclusion) as reachability of a set which is upward closed with respect to the ordering. In Section 6 and 7 we give the modeling of the distributed Lamport and the Ricart-Agrawala protocols respectively. In Section 8, we give an overview of the method of monotonic abstractions used to perform reachability analysis. Section 9 reports the result of applying our prototype on the two case studies. Finally, we give some conclusions and directions for future research.

2 Preliminaries

We use $\mathcal{B}$ to denote the set $\{true, false\}$ of Boolean values; and use $\mathcal{N}$ to denote the set of natural numbers. We assume an element $\perp \notin \mathcal{B} \cup \mathcal{N}$ and use $\mathcal{B}_\perp$ and $\mathcal{N}_\perp$ to denote $\mathcal{B} \cup \{\perp\}$ and $\mathcal{N} \cup \{\perp\}$ respectively. For a natural number n, let $\overline{n}$ denote the set $\{1, \ldots, n\}$. We will work with sets of variables. Such a set A is often partitioned into two subsets: *Boolean* variables $A_\mathcal{B}$ which range over $\mathcal{B}$, and *numerical* variables $A_\mathcal{N}$ which range over $\mathcal{N}$. We denote by $\mathbb{B}(A_\mathcal{B})$ the set of Boolean formulas over $A_\mathcal{B}$. We will also use a simple set of formulas, called *gap formulas*, to constrain the numerical variables. More precisely, we let $\mathbb{G}(A_\mathcal{N})$ be the set of formulas which are either of the form $x = y$ or of the form $x \sim_k y$ where $\sim \in \{<, \leq\}$, $x, y \in A_\mathcal{N}$, and $k \in \mathcal{N}$. Here $x <_k y$ stands for $x + k < y$. We use $\mathbb{F}(A)$ to denote the set of formulas which has members of $\mathbb{B}(A_\mathcal{B})$ and of $\mathbb{G}(A_\mathcal{N})$ as atomic formulas, and which is closed under the Boolean connectives $\wedge, \vee$. For instance, if $A_\mathcal{B} = \{a, b\}$ and $A_\mathcal{N} = \{x, y\}$ then $\theta = (a \supset b) \wedge (x + 3 < y)$ is in $\mathbb{F}(A)$. Sometimes, we write a formula as $\theta(y_1, \ldots, y_k)$ where $y_1, \ldots, y_k$ are the variables which may occur in θ; so we can write the above formula as $\theta(x, y, a, b)$.

A *substitution* is a set $\{x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n\}$ of pairs where x_i are variables, and e_i are all constants or all variables. For each $i : 1 \leq i \leq n$, e_i is of the same type as x_i. Here, we assume that all the variables are distinct, i.e., $x_i \neq x_j$ if $i \neq j$. For a formula θ and a substitution S, we use $\theta[S]$ to denote the formula we get from θ by simultaneously replacing all occurrences of the variables $x_1, \ldots, x_n$ by $e_1, \ldots, e_n$ respectively. Observe that, if $e_1, \ldots, e_n$ are constants, then all variables appearing in θ will be replaced. In such a case, the formula $\theta[S]$ evaluates either to *true* or to *false*. Sometimes, we may write $\theta[S_1][S_2] \cdots [S_m]$ instead of $\theta[S_1 \cup S_2 \cup \cdots \cup S_m]$. As an example, if $\theta = (x_1 < x_2) \wedge (x_3 <_2 x_4)$ then $\theta[x_1 \leftarrow y_2, x_4 \leftarrow x_3][x_2 \leftarrow x_3] = (y_2 < x_3) \wedge (x_3 <_2 x_3)$.

3 Parameterized Distributed Systems

In this section, we introduce a basic model for parameterized distributed systems with heterogeneous data types.

A *parameterized distributed system* (or *distributed system* for short) consists of an arbitrary (but finite) number n of identical processes. Each process has a number of *local* variables and communicates asynchronously with the other processes through a set of bounded *FIFO channels*. To simplify, we assume, in this and in the next section, that each channel is of size one. It is straightforward to extend the results to the case of channels of any (finite) size. Furthermore, a process maintains a number of *record* variables which are used to store information about the local states and values of local variables of the other processes. All the variables and channels are assigned either Boolean or integer variables. A process is modeled as an extended finite-state automaton where the transitions check and update the values of the variables and channels accessible to the process. A transition is of one of three types. A *local* transition involves only the local variables of the process. In a *quantified* transition, the process may also check and update the values of the record variables. Such a transition is called *quantified* since (as we shall see below) it may involve an arbitrary number of variables. Finally, in a *communication* transition, also the contents of the channels can be checked and updated. A distributed system, described in this manner, induces an infinite family of (infinite-state) systems, namely one for each size n. The aim is to verify correctness of the systems for the whole family.

To simplify, we assume that each process is indexed by a natural number $i : 1 \leq i \leq n$. The index of the process does not appear in the transition rules, and hence has no relevance for the behavior of the process. Sometimes, we simply write "process i" to refer to the process with index i. In the sequel, we assume the sets L, R, and Ch of *local*, *record*, and *channel* variables, respectively. The set L is partitioned into $L_{\mathcal{B}}$ (which range over $\mathcal{B}$) and $L_{\mathcal{N}}$ (which range over $\mathcal{N}$). A variable in L assumes values in $\mathcal{B}$ or $\mathcal{N}$ depending on its type. Also, the other sets R and Ch are partitioned in a similar manner. In case of a channel variable, the variable will take values from $\mathcal{B}_\perp$ and $\mathcal{N}_\perp$ where the value $\perp$ indicates that the channel is empty. Each process i has one copy of the set L. Also, for each record variable $x \in R$ and pairs of processes i and j, process i has a local copy of x corresponding to j. Process i then uses that particular copy of x to record information about the state of process j. Finally, for each channel variable $x \in Ch$ and pairs of processes i and j, there is one copy of x which i can write to and j can read from; and (symmetrically) another copy which j can write to and i can read from. Notice that, for an instance of n processes, there will be n copies of L and $n(n-1)$ copies of R and Ch.

To describe the transitions of the system, we introduce the set $L^{next} = \{x^{next} | \, x \in L\}$ which contains the *next-value* versions of the variables in L. A variable $x^{next} \in L^{next}$ represents the next value of x when performing a transition. The sets R^{next} and Ch^{next} are defined in a similar manner. Formally, a *distributed system* $\mathcal{D}$ is a pair (Q, T), where Q is a finite set of *local states*, and T is a finite set of *transition rules*. A transition is of the form

$$t : \left[q \rightarrow q' \;\; \rhd \;\; \theta \right] \tag{1}$$

where $q, q' \in Q$ and θ is either a *local*, a *quantified*, or a *communication condition*. Intuitively, the process which makes the transition changes its local state from

q to q'. In the meantime, the values of the variables and channels accessible to the process are checked and updated according to θ. Below, we describe how we define local, quantified, and communication conditions.

A *local condition* is a formula in $\mathbb{F}(L \cup L^{next})$. The formula specifies how the local variables of the process are updated with respect to their current values. A *quantified condition* θ is either of the form $\forall \cdot \theta_1$ (i.e., it is universal), or of the form $\exists \cdot \theta_1$ (i.e., it is existential), where $\theta_1 \in \mathbb{F}\left(L \cup L^{next} \cup R \cup R^{next}\right)$. The universal condition checks the local variables of the process (say with index i) which is about to make the transition (through L), and the copies of the record variables inside i corresponding to *all* the other processes (through R). It also specifies how these variables are updated (through L^{next} and R^{next}). The existential case can be explained analogously, with the difference that the record variables corresponding so *some* other (unspecified) process (rather than all other processes) will be checked and updated. A *communication condition* θ is of the form $Com{\cdot}\theta_1$ where θ_1 belongs to $\mathbb{F}\left(L \cup L^{next} \cup R \cup R^{next} \cup Ch \cup Ch^{next}\right)$. Intuitively, the process (say with index i) chooses some other process (say with index j). Process i performs the transition checking and updating its local variables and its copies of the record variables corresponding to process j (in a similar manner to above). Furthermore, process i can read the values of the channels to which j can write and i can read (through Ch); and update the channels to which i can write and j can read (through Ch^{next}). Here, we assume that the transition is enabled only if it does not try to read the value of an empty channel, or to write to a channel which is full (occupied). Notice that the transition is implicitly existentially quantified, in the sense that process i checks and updates record variables and channel contents corresponding only to one other process.

Remark 1 (Finite Variables). The case where the variables range over finite domains can be handled in a straightforward manner.

Example Assume local states q_1, q_2 and q_3, a local numerical variable *clock*, a Boolean record variable *checked* and a numerical channel variable c. In the rest of the paper, we introduce some syntactic sugar to improve readability. We assume that non-mentioned *next-value* forms of local and record variables equal their current value. Follow examples of local, universally quantified and communication transitions.
Local. The process changes local state from q_1 to q_2. It assigns a new value to the local (numerical) variable *clock* which is larger than its current value.

$$q_1 \rightarrow q_2 \triangleright (clock < clock^{next})$$

Universally quantified. The process changes local state from q_2 to q_3. It also changes the value of *clock* as above, and checks whether the values of all copies of the record variable *checked* are equal to *true*. Furthermore, the process resets all these values to *false*.

$$q_2 \rightarrow q_3 \triangleright \forall\, (clock < clock^{next} \ \wedge \ checked \ \wedge \ \neg(checked^{next}))$$

Communication. A process (say with index i) at local state q_2 changes the value of *clock* as above, and chooses some other process (say with index j). Process i checks whether its copy of *checked* corresponding to process j is *false*. In such a case, it sets *checked* to *true*, and sends the value of its updated logical clock to process j along the relevant copy of channel c (the copy to which process i writes to and process j reads from).

$$q_2 \rightarrow q_2 \triangleright Com \cdot \begin{pmatrix} clock < clock^{next} \\ \wedge\ \neg checked\ \wedge\ checked^{next}\ \wedge\ c^{next} = clock^{next} \end{pmatrix}$$

4 Operational Semantics

In this section, we define the transition system associated with a distributed system. In general, 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. A distributed system $\mathcal{D} = (Q, T)$ induces a transition system $\mathcal{T}(\mathcal{D}) = (C, \longrightarrow)$ as follows. A configuration is defined by the local states and the values of the local variables in the processes, the values of the record variables, and the contents of the channels. Formally, a *configuration* c (of *size* n) is a tuple (n, s, u, v, w) where

- s is a mapping $\overline{n} \rightarrow Q$. For each process (with index i) the value of $s(i)$ defines the local state of the process.
- u is a mapping $\overline{n} \rightarrow L \rightarrow (\mathcal{B} \cup \mathcal{N})$. For each process (with index i) and local variable x, the value of $u(i)(x)$ defines the value of the copy of x in process i. The value may be in $\mathcal{B}$ or $\mathcal{N}$ depending on the type of x.
- v is a mapping $\overline{n} \rightarrow \overline{n} \rightarrow R \rightarrow (\mathcal{B} \cup \mathcal{N})$. For processes (with indices i and j), and record variable x, $v(i)(j)(x)$ defines the value of the copy of x in process i corresponding to process j.
- w is a mapping $\overline{n} \rightarrow \overline{n} \rightarrow Ch \rightarrow (\mathcal{B}_\perp \cup \mathcal{N}_\perp)$. For processes (with indices i and j), and channel variable x, the value of $w(i)(j)(x)$ defines the content of the copy of channel x to which i can write and j can read. If $w(i)(j)(x) = \perp$ then the channel is empty.

Now, we are ready to define the transition relation $\longrightarrow$. Consider two configurations $c_1 = (n, s_1, u_1, v_1, w_1)$ and $c_2 = (n, s_2, u_2, v_2, w_2)$ of the same size n. Consider a transition t rule of the form of (1) and a natural number $1 \leq i \leq n$. Intuitively, we will describe the effect of process i performing transition t. We write $c_1 \xrightarrow{t,i} c_2$ to denote that the following conditions are satisfied:

- $s_2(j) = s_1(j)$, $u_2(j) = u_1(j)$, $v_2(j) = v_1(j)$ for each $j : 1 \leq j \neq i \leq n$. Furthermore, $w_2(j)(k) = w_1(j)(k)$, if $1 \leq j \neq i \leq n$ and $1 \leq k \neq i \leq n$, and $j \neq k$. The other processes do not change their local states, local variables, or their record variables. The channels which cannot be read from or written to by process i are not changed either.
- $s_1(i) = q$ and $s_2(i) = q'$. The current and new local states of process i should be consistent with those given in the transition rule.

– One of the following conditions holds:
  - θ is a local condition and the formula $\theta\,[\rho_1]\,[\rho_2]$ holds, where the substitutions are defined by $\rho_1 = \{x \leftarrow u_1(i)(x) |\ x \in L\}$, and by $\rho_2 = \{x^{next} \leftarrow u_2(i)(x) |\ x \in L\}$. Furthermore, $v_2 = v_1$ and $w_2 = w_1$. The current and new values of the local variables of i are consistent with θ.
  - $\theta = \forall \cdot \theta_1$ is a universal quantified condition and $\theta_1\,[\rho_1]\,[\rho_2]\left[\rho_3^j\right]\left[\rho_4^j\right]$ holds for each $j : 1 \leq j \neq i \leq n$. The substitutions ρ_1 and ρ_2 are defined as in the previous case, while $\rho_3^j = \{x \leftarrow v_1(i)(j)(x) |\ x \in R\}$, and $\rho_4^j = \{x^{next} \leftarrow v_2(i)(j)(x) |\ x \in R\}$. Furthermore $w_2 = w_1$. In addition to the local variables, process i may check and update the values of its record variables. The manner in which the variables are changed should be consistent with the condition for each other process j.
  - $\theta = \exists \cdot \theta_1$ is an existential quantified condition and $\theta_1\,[\rho_1]\,[\rho_2]\left[\rho_3^j\right]\left[\rho_4^j\right]$ holds for some $j : 1 \leq j \neq i \leq n$. Furthermore $w_2 = w_1$. All the substitutions are defined as in the previous case. The difference is that the variable changes should be consistent with the condition of the transition for *some* other process j (rather than all other processes).
  - $\theta = Com \cdot \theta_1$ is a communication condition. In this case, the formula $\theta_1\,[\rho_1]\,[\rho_2]\left[\rho_3^j\right]\left[\rho_4^j\right]\left[\rho_5^j\right]\left[\rho_6^j\right]$ holds for some $j : 1 \leq j \neq i \leq n$. The substitutions ρ_1, ρ_2, ρ_3^j, and ρ_4^j are defined as above, while ρ_5^j is defined by $\{x \leftarrow w_1(j)(i)(x) |\ x \in Ch\}$ and ρ_6^j by $\{x^{next} \leftarrow w_2(i)(j)(x) |\ x \in Ch\}$. Furthermore the following conditions are satisfied for each $x \in Ch$:
    * either x does not occur in θ_1 or both $w_1(j)(i)(x) \neq \perp$ and $w_2(j)(i) = \perp$. The channel can be read only if it is not empty. After the reading operation, the channel becomes empty.
    * either x^{next} does not occur in θ_1 or $w_1(i)(j)(x) = \perp$. A channel can be written to only if it is empty.

We write $c_1 \longrightarrow c_2$ to denote that $c_1 \xrightarrow{t,\,i} c_2$ for some t and i.

5 Safety Properties

Following the methodology of [3, 2], we introduce an ordering on configurations, which we use to define the safety problem. Assume a distributed system $\mathcal{D} = (Q, T)$. We assume that, the system starts executing from an *initial* configuration, where each process starts running from an (identical) *initial* local state, with predefined initial values in the local and record variables, and with empty channels. In the induced transition system $\mathcal{T}(\mathcal{D}) = (C, \longrightarrow)$, we use *Init* to denote the set of initial configurations. Notice that this set is infinite, since there is a different initial configuration for each instance (size) of the system.

We define an ordering on configurations. To do that, we first introduce a notation. Consider a configuration $c = (n, s, u, v, w)$, a variable $x \in L \cup R \cup Ch$, and i, j where $1 \leq i \neq j \leq n$. Abusing notation, we define $c(x)(i)(j)$ to be $u(i)(x)$ if $x \in L$, $v(i)(j)(x)$ if $x \in R$, and $w(i)(j)(x)$ if $x \in Ch$. Consider two

configurations $c_1 = (n_1, s_1, u_1, v_1, w_1)$ and $c_2 = (n_2, s_2, u_2, v_2, w_2)$. We write $c_1 \preceq c_2$ to denote that there is an injection $h : \overline{n_1} \to \overline{n_2}$ such that the following conditions are satisfied for each $i, j, l, m : 1 \leq i, j, l, m \leq n_1$:

1. $s_1(i) = s_2(h(i))$.
2. $c_1(i)(j)(x) = \bot$ iff $c_2(h(i))(h(j))(x) = \bot$ for all $x \in Ch$.
3. $c_1(i)(j)(x) = \mathit{true}$ iff $c_2(h(i))(h(j))(x) = \mathit{true}$ for all $x \in L_{\mathcal{B}} \cup R_{\mathcal{B}} \cup Ch_{\mathcal{B}}$.
4. $c_1(i)(j)(x) = c_1(l)(m)(y)$ iff $c_2(h(i))(h(j))(x) = c_2(h(l))(h(m))(y)$ for all $x, y \in L_{\mathcal{N}} \cup R_{\mathcal{N}} \cup Ch_{\mathcal{N}}$.
5. $c_1(i)(j)(x) <_{k_1} c_1(l)(m)(y)$[3] implies that there is a $k_2 \geq k_1$ such that $c_2(h(i))(h(j))(x) <_{k_2} c_2(h(l))(h(m))(y)$ for all $x, y \in L_{\mathcal{N}} \cup R_{\mathcal{N}} \cup Ch_{\mathcal{N}}$.

A set of configurations $D \subseteq C$ is *upward closed* (with respect to the ordering $\preceq$) if $c \in D$ and $c \preceq c'$ implies $c' \in D$. For sets of configurations $D, D' \subseteq C$ we use $D \longrightarrow D'$ to denote that there are $c \in D$ and $c' \in D'$ with $c \longrightarrow c'$.

The *coverability problem* for parameterized systems is defined as follows:

PAR-COV
Instance
- A distributed system $\mathcal{D} = (Q, T)$.
- Two sets of configurations *Init* and C_F, with C_F upward closed.

Question $\mathit{Init} \xrightarrow{*} C_F$?

It can be shown, using standard techniques (see e.g. [23]), that checking many classes of safety properties, e.g. mutual exclusion, 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).

6 Distributed Mutex by Lamport

We describe the distributed mutual exclusion algorithm by Lamport [17] in our model. In this algorithm, a number of processes compete for a shared resource and communicate by message passing. The protocol guarantees mutual exclusion by allowing only the process with the *earliest* request to access its critical section. Here, *earliest* is defined by means of *logical clocks* [17], one per process. A logical clock is a local numerical variable that is strictly increased each time a process performs a transition. The value of the local logical clock is appended to each sent message. Each time a process receives a time-stamped message, it updates its logical clock to a value that is strictly larger than the maximum of the time stamp in the message, and of the previous value of the clock. Ties are broken by giving priority to the process to the left. Here we model the relative positions of the processes by introducing a Boolean local variable *right* that is unmodified once initialized. This gives a total ordering that uniquely defines the process with the earliest request.

[3] recall $x <_{k_1} y$ iff $x + k_1 < y$.

In our model (table 1) of the algorithm, each process is in one of five local states, namely $idle, ask, wait, use$ and $free$. The logical clock of a process is represented by a numerical local variable $clock$. The process has a local variable $last$ which it uses to record the value of its logical clock at the time when it last started sending requests to other processes. The process has also a Boolean record variable $checked$ which it uses to keep track of other processes to (from) which it has already sent (received) messages such as requests, acknowledgments, etc. Another record variable, namely $Queue$, is used to store the time stamps associated with the requests received from other processes. Finally, the system has two channel variable c and ts. A process uses its copies of the channels to send timed-stamped messages. For instance, when a process wants to send a time-stamped request to another process, then it puts the message req to the relevant copy of c (the one writable by the current process and readable by the other process) and the time stamp to the relevant copy of ts.

Each time a process takes a transition, its logical clock is increased using the formula $clock < clock^{next}$. Initially, all processes are in their initial local state $idle$. When a process wants to enter the critical section, it first sends requests to all other processes. This is done in three steps (transitions t_1, t_2, and t_3). In t_1, the process moves from local state $idle$ to local state ask. In doing this, it also records the new value of its logical clock in the local variable $last$. Notice that t_1 is a local transition. In ask, the process loops sending requests to the other processes, one at a time. This is done through t_2 which is a communication transition. In each execution of t_2, the process chooses another process , and checks whether it has already sent a request to that process (using the record variable $checked$). If this is not the case then it sets $checked$ to true, and sends a time-stamped request to the other process on channels c and ts. More precisely, it sends the request message req through c and the new value of its logical clock through ts. In t_3, the process checks whether it has sent a request to all the other processes, by testing that all its copies of the record variable $checked$ are equal to $true$. Observe that t_3 is a universally quantified transition.

A process can at any time receive a request from another process. This is done by transition t_9 which is a communication transition. The process receives a request from another process through channel c, and the associated time stamp through channel ts. It assigns to its logical clock a new value which is strictly larger than both the old value of the logical clock (the formula $clock < clock^{next}$) and the received time stamp (the formula $ts < clock^{next}$). It assigns the time stamp to the copy of the record variable $Queue$ corresponding to the other process; then it sends back an acknowledgment to the other process together with a time stamp which is equal to the new value of its logical clock.

After sending the requests, a process starts collecting acknowledgments (in state $wait$). This is done by the communication transition t_4. The process receives an acknowledgment from the copy of channel c corresponding to another process together with the corresponding time stamp from channel ts. It updates its logical clock, and marks it has received an acknowledgment from the other process in a similar manner to above (see e.g., the explanation of t_2).

The process enters the critical section (transition t_5) only if it has received acknowledgments from all other processes, and if its request is the earliest among the received requests. The process request is the earliest if for each other process j in the system, one of the three following conditions holds; either (i) no request was received from process j (checked with $Queue = zero$); or the time stamp associated with the received request (stored in $Queue$) is (ii) strictly larger than $last$; or (ii) equal to $last$ but process j is to the right of the current process ($right \ \wedge \ last = Queue$).

Finally a process releases the resource by sending a release message to all the other processes in the system. This is done in three steps (transitions t_6, t_7 and t_8). These steps are similar to the three steps of sending requests to the other processes (transitions t_1, t_2 and t_3). A process that receives a release message (transition t_{10}), updates its local clock, and removes from its local queue the corresponding request.

Table 1: Lamport Distributed Mutex

States: $Q = \{idle, ask, wait, use, free\}, any \in Q$

Local:	$clock, last$ are *naturals originally zero*
Record:	$checked$ is a *Boolean originally false*
	$Queue$ is a *natural originally zero*
Channel:	c is in $\{req, ack\}_\perp$ *originally* $\perp$
	ts is in $\mathcal{N}_\perp$ *originally* $\perp$

$t_1 : \ idle \ \rightarrow \ ask \triangleright \left(clock < clock^{next}\right) \ \wedge \ \left(last^{next} = clock^{next}\right)$
$t_2 : \ ask \ \rightarrow \ ask \triangleright Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge \ \left(c^{next} = req\right) \ \wedge \ \left(ts^{next} = clock^{next}\right) \\ \wedge \ \left(\neg checked \ \wedge \ checked^{next}\right) \end{pmatrix}$
$t_3 : \ ask \ \rightarrow \ wait \triangleright \forall \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge \ \left(checked \ \wedge \ \neg(checked^{next})\right) \end{pmatrix}$
$t_4 : \ wait \ \rightarrow \ wait \triangleright Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge \ (c = ack) \wedge \left(ts < clock^{next}\right) \\ \wedge \ \left(\neg checked \ \wedge \ checked^{next}\right) \end{pmatrix}$
$t_5 : \ wait \ \rightarrow \ use \triangleright \forall \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge \ \left(checked \ \wedge \ \neg(checked^{next})\right) \\ \wedge \ \begin{pmatrix} (Queue = zero) \vee (last < Queue) \\ \vee \ (right \wedge (last = Queue)) \end{pmatrix} \end{pmatrix}$
$t_6 : \ use \ \rightarrow \ free \triangleright \ \left(clock < clock^{next}\right)$
$t_7 : \ free \ \rightarrow \ free \triangleright Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge \left(c^{next} = rel\right) \wedge \left(ts^{next} = clock^{next}\right) \\ \wedge \ \left(\neg checked \ \wedge \ checked^{next}\right) \end{pmatrix}$
$t_8 : \ free \ \rightarrow \ idle \triangleright \forall \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge \ \left(checked \ \wedge \ \neg(checked^{next})\right) \end{pmatrix}$

$$t_9: \quad any \;\rightarrow\; any \triangleright Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge\ (c = req) \wedge \left(ts < clock^{next}\right) \\ \wedge\ \left(Queue^{next} = ts\right) \\ \wedge \left(c^{next} = ack\right) \wedge \left(ts^{next} = clock^{next}\right) \end{pmatrix}$$

$$t_{10}: \quad any \;\rightarrow\; any \triangleright Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge\ (c = rel) \wedge \left(ts < clock^{next}\right) \\ \wedge\ \left(Queue^{next} = zero\right) \end{pmatrix}$$

The set of configurations violating mutual exclusion is the set where at least two processes are in state *use*.

7 Distributed Mutex by Ricart-Agrawala

The Ricart-Agrawala algorithm [21] is a modification of Lamport's distributed mutex. The modification aims at diminishing the number of exchanged messages per entry to the critical section. This is achieved by not sending release messages, and modifying the conditions for sending acknowledgment messages. In a similar manner to our model of Lamport's algorithm, each process can have one of the five states: *idle, ask, wait, use* and *free*. We use two local numerical variables *clock* and *last*, three Boolean record variables $checked, right$ and $deferred$, and two channel variables c and ts. Except for $deferred$, all variables play the same roles as in Lamport's algorithm. This variable is used to remember the processes to which an acknowledgment should be sent when releasing the critical section.

Like in Section 6, a process sends requests by means of three transitions (t_1, t_2 and t_3). A process that receives a request while in state *idle* or *free*, updates its logical clock and sends back an acknowledgment (transition t_9). If a process receives the request when in states *ask, wait* or *use*, then it may take one of two actions. If the time stamp received with the request is (i) strictly smaller than the value of the local variable *last*, or is (ii) equal to *last* and the sender of the request is to the left of the receiver, then the receiver sends back an acknowledgment (transition t_{10}). Otherwise, this is deferred (transition t_{11}).

After sending the requests, a process collects acknowledgments (transition t_4). A process that did receive acknowledgments from all other processes can access its critical section (transition t_5). Finally a process releases the resource by sending an acknowledgment message to each other process with a deferred request (transitions t_6, t_7 and t_8).

Table 2: Ricart-Agrawala Distributed Mutex

States: $Q = \{idle, ask, wait, use, free\}$,
$grant \in \{idle, free\}, hold \in \{ask, wait, use\}$

Local: *clock, last* are *naturals originally zero*
Record: *checked, deferred* are *Booleans originally false*
Channel: c is in $\{req, ack\}_\perp$ *originally* $\perp$
ts is in $\mathcal{N}_\perp$ *originally* $\perp$

$$
\begin{array}{|l|}
\hline
t_1 :\ idle \ \rightarrow\ \ ask \ \triangleright\ \ \left(clock < clock^{next}\right)\ \wedge\ \left(last^{next} = clock^{next}\right) \\
\hline
t_2 :\ \ ask \ \rightarrow\ \ ask \ \triangleright\ Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge\ \left(c^{next} = req\right)\ \wedge\ \left(ts^{next} = clock^{next}\right) \\ \wedge\ \left(\neg checked\ \wedge\ checked^{next}\right) \end{pmatrix} \\
\hline
t_3 :\ \ ask \ \rightarrow\ \ wait \ \triangleright\ \forall \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge\ checked\ \wedge\ \neg(checked^{next}) \end{pmatrix} \\
\hline
t_4 :\ wait \ \rightarrow\ \ wait \ \triangleright\ Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge\ (c = ack) \wedge \left(ts < clock^{next}\right) \\ \wedge\ \left(\neg checked\ \wedge\ checked^{next}\right) \end{pmatrix} \\
\hline
t_5 :\ wait \ \rightarrow\ \ use \ \triangleright\ \forall \left(\left(clock < clock^{next}\right) \wedge checked\ \wedge\ \neg(checked^{next})\right) \\
\hline
t_6 :\ \ use \ \rightarrow\ \ free \ \triangleright\ \ \left(clock < clock^{next}\right) \\
\hline
t_7 :\ free \ \rightarrow\ \ free \ \triangleright\ Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge \left(c^{next} = ack\right) \wedge \left(ts^{next} = clock^{next}\right) \\ \wedge\ \left(deferred\ \wedge\ \neg(deferred)^{next}\right) \end{pmatrix} \\
\hline
t_8 :\ free \ \rightarrow\ \ idle \ \triangleright\ \forall \left(\left(clock < clock^{next}\right) \wedge (\neg deferred)\right) \\
\hline
t_9 : grant \rightarrow grant \ \triangleright\ Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge (c = req) \wedge \left(ts < clock^{next}\right) \\ \wedge\ \left(c^{next} = ack\right) \wedge \left(ts^{next} = clock^{next}\right) \end{pmatrix} \\
\hline
t_{10} :\ hold \rightarrow\ \ hold \ \triangleright\ Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge (c = req) \wedge \left(ch1_{ts} < clock^{next}\right) \\ \wedge\ ((ts < last) \vee (ts = last \wedge \neg right)) \\ \wedge\ \left(c^{next} = ack\right) \wedge \left(ts^{next} = clock^{next}\right) \end{pmatrix} \\
\hline
t_{11} :\ hold \rightarrow\ \ hold \ \triangleright\ Com \cdot \begin{pmatrix} \left(clock < clock^{next}\right) \\ \wedge\ (c = req) \wedge \left(ts < clock^{next}\right) \\ \wedge\ ((last < ts) \vee (ts = last \wedge right)) \\ \wedge\ \left(\neg deferred\ \wedge\ deferred^{next}\right) \end{pmatrix} \\
\hline
\end{array}
$$

The set of configurations violating mutual exclusion is the set where at least two processes are in state *use*.

8 Approximation and scheme overview

In this section, we use a methodology introduced in [3, 2] for solving PAR-COV. The methodology consists in over-approximating the transition relation $\longrightarrow$ of Section 4 by a new monotonic transition relation $\rightsquigarrow\ = \longrightarrow \cup \rightsquigarrow_1$. Intuitively, the relation $\rightsquigarrow_1$ corresponds to the deletion of all processes (together with the corresponding record and channel variables) violating a condition θ_1 when taking a quantified universal transition $t = \forall \cdot \theta_1$. Observe that a negative answer to $Init \overset{*}{\rightsquigarrow} C_F$ implies a negative answer to PAR-COV. We check $Init \overset{*}{\rightsquigarrow} C_F$ using a scheme based on backward reachability analysis. The scheme symbolically represents sets of configurations by constraints. We write $[\![\phi]\!]$ to refer to the (infinite) upward closed set of configurations represented by a constraint ϕ. For a (finite) set of constraints Φ, we define $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$. We also write $Pre(\phi)$ to mean a set of constraints, such that $[\![Pre(\phi)]\!] = \{c |\ \exists c' \in [\![\phi]\!] \,.\, c \rightsquigarrow c'\}$. The

set $[\![Pre(\phi)]\!]$ needs to be upward closed in order to be represented by a set of constraints. The monotonicity of $\rightsquigarrow$ ensures upward closedness.

Scheme. Given a finite set Φ_F of constraints representing the set C_F, we check whether $Init \stackrel{*}{\rightsquigarrow} [\![\Phi_F]\!]$. We perform backward reachability analysis, generating a sequence $[\![\Phi_0]\!] \subseteq [\![\Phi_1]\!] \subseteq [\![\Phi_2]\!] \subseteq \cdots$, of finite sets of constraints such that $\Phi_0 = \Phi_F$, and $\Phi_{j+1} = \Phi_j \cup Pre(\Phi_j)$. The procedure terminates when we reach a point j where $[\![\Phi_j]\!] \supseteq [\![\Phi_{j+1}]\!]$. Notice that the termination condition implies that Φ_j characterizes the set of all predecessors of $[\![\phi_F]\!]$. This means that $Init \stackrel{*}{\rightsquigarrow} [\![\Phi_F]\!]$ iff $(Init \bigcap [\![\Phi_j]\!]) \neq \emptyset$. Observe that, in order to implement the scheme (i.e., transform it into an algorithm), we need to be able (for any constraints ϕ, ϕ') to (i) check that $(Init \bigcap [\![\phi]\!]) = \emptyset$, (ii) compute the set $Pre(\phi)$; (iii) check that $[\![\phi]\!] \subseteq [\![\phi']\!]$. The definitions of constraints and the operations on them are similar to [20] and are introduced in the appendix.

9 Experimental Results

The method has been implemented in a prototype. The tool starts from specifications of bad states (at least two processes in state *use*). We report on the obtained results, using a 1.6 Ghz laptop with 1G of memory. Mutual exclusion

	# iterations	# constraints	time(sec)	memory(MB)
Distr. Lamport	30	4676	85	18
Distr. Ricart-Agrawala	32	1205	13	< 5

Table 3. Obtained results

of both distributed algorithms has been checked fully automatically. We give the number of iterations and constraints (in the final set resulting from the fixpoint analysis), together with the required time in seconds and memory in megabytes.

10 Conclusions and Future Research

We have shown how to instantiate the monotonic abstraction scheme [3, 2] for automatic verification of parameterized distributed protocols. We have described how the method works on two non-trivial case studies, namely the distributed Lamport and the Ricart-Agrawala mutex protocols. Both protocols are verified automatically in our prototype without the need for manual intervention.

An interesting direction for future work is to extend the method to systems whose configurations can be modeled by graphs such as cache coherence protocols and dynamically allocated data structures. There are also several other interesting classes on problems for which monotonic abstraction seems to be relevant. For instance, we are currently working on applying monotonic abstraction to perform *shape analysis* on memory heaps. The idea is to find suitable pre-orders which allow to perform an abstract (over-approximate) reachability analysis using upward-closed sets of heap graphs.

References

1. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with wqo 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. CAV*, pages 145–157, 2007.
3. P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Regular model checking without transducers. In *Proc. TACAS*, pages 721–736, 2007.
4. P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Proc. VMCAI '08*.
5. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.
6. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR*, pages 116–130, 2002.
7. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. CAV*, pages 221–234, 2001.
8. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc. CAV*, pages 223–235, 2003.
9. D. Chkliaev, J. Hooman, and P. van der Stok. Mechanical verification of transaction processing systems. In *ICFEM*, 2000.
10. E. Clarke, M. Talupur, and H. Veith. Proving ptolemy right: Environment abstraction principle for model checking concurrent system. In *Proc. TACAS '08*.
11. G. Delzanno. Automatic verification of cache coherence protocols. In *Proc. CAV*, pages 53–68, 2000.
12. E. Emerson and K. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. LICS*, pages 70–80, 1998.
13. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS '99*.
14. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.
15. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *TCS*, 256:93–112, 2001.
16. S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Proc. CAV*, pages 135–147, 2004.
17. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
18. 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.
19. P. Revesz. A closed form evaluation for datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1):117–149, 1993.
20. A. Rezine. *Parameterized Systems: Generalizing and Simplifying Automatic Verification*. PhD thesis, Uppsala University, 2008.
21. G. Ricart and A. K. Agrawal. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.
22. E. Sedletsky, A. Pnueli, and M. Ben-Ari. Formal verification of the ricart-agrawala algorithm. In *Proc. CFSTTCS '00*.
23. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS*, pages 332–344, June 1986.
24. T. Yavuz-Kahveci and T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. *STTT*, 5(1), 2003.