Handling Global Conditions in Parametrized System Verification

Parosh Aziz Abdulla
Dept. of Computer Systems
P.O. Box 325
S-751 05 Uppsala, Sweden
parosh@docs.uu.se

Ahmed Bouajjani
VERIMAG, Centre Equation, 2
av. de Vignate
38610 Gieres, France
Ahmed.Bouajjani@imag.fr

Bengt Jonsson
Dept. of Computer Systems
P.O. Box 325
S-751 05 Uppsala, Sweden
bengt@docs.uu.se

Marcus Nilsson
Dept. of Computer Systems
P.O. Box 325
S-751 05 Uppsala, Sweden
marcusn@docs.uu.se

Abstract

We consider symbolic verification for a class of parametrized systems, where a system consists of a linear array of processes, and where an action of a process may in general be guarded by both *local conditions* restricting the state of the process about to perform the action, and *global conditions* defining the *context* in which the action is enabled. Such a model captures the behaviour, e.g., of idealized versions of mutual exclusion protocols, such as the bakery and ticket algorithms by Lamport, Burn's protocol, Dijkstra's algorithm, and Szymanski's algorithm. The presence of both local and global conditions makes these protocols infeasible to analyze, using existing model checking methods for parametrized systems. In all these methods the actions are guarded only by local conditions involving the states of a finite set of processes.

We perform verification using the standard symbolic reachability algorithm enhanced by an operation to speed up the search of the state space. The speed up operation computes the effect of an arbitrary number of applications of an action, rather than a single application. This is crucial for obtaining termination e.g. when applying the algorithm to the above protocols.

We illustrate the use of our method through applications to Szymanski's algorithm and Lamport's bakery protocol.

Note: The extended abstract contains, in addition to 10 pages of text, an appendix which may be read at the discretion of the program committee.

1 Introduction

Much attention has recently been paid to extending the applicability of model checking to infinite-state systems. One reason why a program may have an infinite state space is that it operates on unbounded data structures. Examples of such systems include timed automata [ACD90], hybrid automata [Hen95], data-independent systems [JP93, Wol86], relational automata [Čer94], counter machines [AČ98], pushdown processes [BS95], and lossy channel systems [AJ96]. Another reason is that the program has an infinite control part. This is the case e.g. in Petri nets [Esp95, Jan90], and parametrized systems, in which the topology of the system is parametrized by the number of processes inside the system. In verification of parameterized systems, we are often interested in proving the correctness of the system regardless of the number of processes. Verification algorithms for systems consisting of an unbounded number of similar or identical finite-state processes include [GS92, AJ98, KMM+97], and (using a manually supplied induction hypothesis) [CGJ95, KM89, WL89].

In this paper we consider algorithmic verification of a class of parametrized systems, intended to capture at least the behaviours of several mutual-exclusion algorithms that can be found in the literature. Examples of mutual exclusion algorithms that work for an arbitrary number of processes are: the bakery and ticket algorithms by Lamport, Burn's protocol, Dijkstra's algorithm, and Szymanski's algorithm. These algorithms are implemented on systems with an arbitrary number of processes with linearly ordered identities. The ordering of the processes may reflect the actual physical ordering (e.g. Szymanski's algorithm), or the values assigned to local variables inside processes (e.g. the ticket given to each process during the execution of Lamport's bakery protocol). A configuration of the system can be described as a string representing the local states of the processes. A common feature which places these protocols outside the scope of existing model checking methods, is that an action of a process is in general guarded by both *local* and *global* conditions on the processes. Local conditions restrict the state of the process which is about to perform the action. Global conditions define the *context* in which the action is allowed to occur. A *context* is typically stated as a formula which is quantified over the set of processes inside the system. Examples of contexts are "all processes with lower identities should have local states belonging to given set", or "there should be at least one process with a higher identity which has a local state included in a given set", etc. We propose a model which combines both types of conditions. An action involves the change of local state of a process, and may be conditioned on both the local state, and the context in which the action is performed. We define a class of languages for specifying contexts, which is sufficiently general to capture all the above examples.

To verify our protocols we perform a standard symbolic forward reachability analysis, using regular expressions as representations of (possibly infinite) sets of configurations. It is well-known that checking most safety properties (including satisfiability of mutual exclusion) can be reduced to checking the reachability of a set of "bad" configurations (in our case specified as a regular expression). However, the presence of both local and global guards implies that the standard reachability algorithm will not terminate when applied to any of the earlier mentioned protocols. A main contribution of this paper is that we define an operation to speed-up the search through the state space. The speed-up operator computes the effect of an arbitrary number of applications of an action, rather than the effect of only a single application. This is crucial for obtaining termination during the analysis of any of the above protocols. Notice that the algorithm is incomplete and may in general still fail to terminate.

Related Work There are several results on verification of parametrized systems [GS92, AJ98, CGJ95, KMM+97]. In all these works the actions are guarded only by local conditions involving the states of a finite set of processes. A work, which is close in spirit to ours is [KMM+97]. The authors propose to use regular sets of strings to represent states of parametrized arrays of processes, and to represent the effect of performing an action by a predicate transformer (transducer). However, the work in [KMM+97] considers only transducers that represent the effect of a single application of a transition. This means that their approach will not terminate if applied to reachability analysis for the protocols we consider in this paper. In contrast, we introduce the speed-up operator for actions with both local and global contexts, meaning that reachability analysis will terminate. Applications of speed up operations are reported in the context of communicating finite state automata [BG96, BGWW97, BH97, ABJ98]. The speed up operation is applied to transitions of different types than in our work, namely those that iterate a single loop in the control part of a program, rather than repetitive applications of a transition to different processes in the system. There has also been a number of case studies in verification of mutual exclusion protocols such as Burn's protocol [JL98] and Szymanski's algorithm [GZ98, MAB+94, MP90]. The verification in each case is dependent on abstraction functions or lemmas explicitly provided by the user.

Outline In the next section, we define the class of system models that we consider and illustrate it by an idealized version of Szymanski's mutual exclusion algorithm. In Section 3 we define composition and acceleration of actions. In Section 4 we show how they can be used in verification of safety properties, illustrated by a verification of Szymanski's algorithm. Section 5 contains conclusions and some non-resolved problems. The appendix contains proofs of some theorems, a further example, and a general transducer construction.

2 Preliminaries

In this section, we will introduce a generic system model which is intended to capture the behaviour of many existing mutual exclusion protocols, e.g. Dijkstra's mutual exclusion problem, Lamport's bakery algorithm, Burn's protocol, Szymanski's algorithm. In our model, a program consists of an arbitrary number of identical processes, ordered in a linear array. The process behaviours are defined through a finite set of *actions*. An action represents a change of local state of a process. An action may be conditioned on both the local state of the process, and the *context* in which it may take place. The context represents a global condition on the local states of the rest of processes inside the system. The ordering of the processes may reflect the actual physical ordering (e.g. Szymanski's algorithm), or the values assigned to local variables inside processes (e.g. the ticket given to each process during the execution of Lamport's bakery protocol).

An idealized version of Szymanski's mutual exclusion algorithm can be given as follows. In the algorithm, an arbitrary number of processes compete for a critical section. The local state of each process consists of a control state ranging over the integers from 1 to 7 and of two boolean flags, w and s. A pseudo-code version of the actions of any process i could look as follows:

$$
\begin{array}{ll}
1: & \textbf{await } \forall j : j \neq i : \neg s_j \\
2: & w_i, s_i := \mathit{true}, \mathit{true} \\
3: & \textbf{if } \exists j : j \neq i : (pc_j \neq 1) \wedge \neg w_j \\
 & \quad \textbf{then } s_i := \mathit{false},\ \text{goto } 4 \\
 & \quad \textbf{else } w_i := \mathit{false},\ \text{goto } 5 \\
4: & \textbf{await } \exists j : j \neq i : s_j \wedge \neg w_j \textbf{ then } w_i, s_i := \mathit{false}, \mathit{true} \\
5: & \textbf{await } \forall j : j \neq i : \neg w_j \\
6: & \textbf{await } \forall j : j < i : \neg s_j \\
7: & s_i := \mathit{false},\ \text{goto } 1
\end{array}
$$

For instance, according to the code at line 6, if the control state of a process i is 6, and if the context is that the value of s is *false* in all processes to the left, then the control state of i may be changed to 7. In a similar manner, according to the code at line 4, if the control state of a process i is 4, and if the context is that there is at least another process (either to the right or to the left of i) where the value of s is *true* and the value of w is *false*, then the control state and the values of w and s in i may be changed to 5, *false*, and *true*, respectively. In fact in almost all the protocols that we have considered, contexts are defined by existentially or universally quantified formulas restricting the local states of processes to the left or to the right. In our model we work with a particular subclass of regular languages, which can capture such contexts.

A *left context* is a regular language which can be accepted by a deterministic finite-state automaton with a unique accepting state, and where all outgoing transitions from the accepting state are self-loops. (transitions with identical source and target states). A *right context* is a language such that the language of reversed strings is a left context. The *tail* of a left context is the set of symbols that label self-loops from the accepting state. The tail of a right context is the tail of the left context which is its reverse language.

Examples of left contexts are regular expressions of the form

$$e_1 f_1 e_2 f_2 \cdots e_n f_n e_{n+1}$$

where each e_i is of form $(a_1 + \cdots + a_m)^*$, where each f_i is of form $(b_1 + \cdots + b_k)$ such that b_j does not occur in the expression e_i, for any $j = i, \ldots, k$.

Now, we give the formal definition of our model. We use a finite set C of *colours* to model the local states of processes. A *program* is a triple $\mathcal{P} = \langle C, \phi_I, \mathcal{A} \rangle$ where

C is a finite set of colours,

ϕ_I is a regular set of *initial configurations* over C, and

$\mathcal{A}$ is a finite set of *actions*. An *action* is a triple of the form

$$\phi_L \;;\; \tau(c, c') \;;\; \phi_R$$

where ϕ_L is a left context, ϕ_R is a right context, and $\tau(c, c')$ is a an idempotent binary relation on C.

A *configuration* γ of $\mathcal{P}$ is a string $\gamma[1]\,\gamma[2]\,\cdots\,\gamma[n]$ over C, where $\gamma[i]$ denotes the local state of process i. For $i, j : 1 \le i \le j \le n$, we use $\gamma[i \,..\, j]$ to denote the substring $\gamma[i]\,\gamma[i+1]\,\cdots\,\gamma[j]$. An action

$$\alpha = \phi_L \;;\; \tau(c, c') \;;\; \phi_R$$

defines a relation α on configurations such that $\alpha(\gamma, \gamma')$ holds if γ and γ' are of equal length n, and there is an i with $1 \leq i \leq n$ such that $\tau(\gamma[i], \gamma'[i])$ holds, $\gamma[1 .. i-1] = \gamma'[1 .. i-1] \in \phi_L$, and $\gamma[i+1 .. n] = \gamma'[i+1 .. n] \in \phi_R$. Thus, an action corresponds to a (possibly nondeterministic) program statement in which the colour at one position i can be changed from some colour c to some colour c', provided that $\tau(c, c')$ holds and that the string to the left of i is in ϕ_L and that the string to the right of i is in ϕ_R. We write $\gamma_1 \longrightarrow \gamma_2$ to denote that $\alpha(\gamma_1, \gamma_2)$ for some action $\alpha \in \mathcal{A}$. We use α^* and $\stackrel{*}{\longrightarrow}$ to denote the transitive closures of α and $\longrightarrow$ respectively. A configuration γ is said to be *reachable* if there is a configuration $\gamma_I \in \phi_I$ such that $\gamma_I \stackrel{*}{\longrightarrow} \gamma$.

The *reachability problem* is defined as follows.

Instance A program $\mathcal{P}$ and a set of configurations of $\mathcal{P}$ represented by a regular expression ϕ_F.

Question Is γ reachable?

In Figure 1 we represent Szymanski's algorithm as a program in our framework. To simplify the notation, we introduce the following syntactical notations. We let a colour be a triple $\langle pc, w, s \rangle$, where $pc \in \{1, \ldots, 7\}$, and w and s are boolean. We use predicates to define colours. For example, the predicate $(\neg s)$ denotes the set of colours where the value of s is equal to *false*, that is the set $\{\langle pc, w, \mathit{false} \rangle \ : \ pc \in \{1, \ldots, 7\}$ and $w \in \{\mathit{true}, \mathit{false}\}\}$. We use the predicate *true* to denote the set of all colours. We use guarded commands to represent binary relations on colours. For instance, the command $(pc = 1) \longrightarrow pc := 2$ represents the relation $\{\langle \langle pc_1, w, s \rangle, \langle pc_2, w, s \rangle \rangle \ :$ $(pc_1 = 1)$ and $(pc_2 = 2)\}$. Notice that e.g. at line 3 the left context $((pc = 1) \ \vee \ w)^*((pc \neq 1) \ \wedge \ \neg w)\mathit{true}^*$ is equivalent to $\mathit{true}^*((pc \neq 1) \ \wedge \ \neg w)\mathit{true}^*$; however, we use the previous expression in order to be consistent with the definition of a left context.

3 Composition and Acceleration of Actions

In this section we define an operation which computes the effect of an unbounded number of executions of an action. We also describe a local sequential composition of actions.

Acceleration. For an action α, let α^* be the action constructed by repeating the action α an arbitrary number of times. More precisely, α^* denotes the set of pairs (γ, γ') of configurations such that there exists a sequence $\gamma_0 \gamma_1 \gamma_2 \cdots \gamma_n$ of configurations with $n \geq 0$ such that $\gamma = \gamma_0$, $\gamma' = \gamma_n$, and such that $\alpha(\gamma_i, \gamma_{i+1})$ for

1 :	$(\neg s)^* \ ; \ (pc = 1) \longrightarrow pc := 2 \ ; \ (\neg s)^*$
2 :	$true^* \ ; \ (pc = 2) \longrightarrow pc, w, s := 3, true, true \ ; \ true^*$
3 :	$((pc = 1) \ \vee \ w)^*((pc \neq 1) \ \wedge \ \neg w)true^* \ ; \ (pc = 3) \longrightarrow pc, s := 4, false \ ; \ true^*$
4 :	$true^* \ ; \ (pc = 3) \longrightarrow pc, s := 4, false \ ; \ ((pc = 1) \ \vee \ w)^*((pc \neq 1) \ \wedge \ \neg w)true^*$
5 :	$((pc \neq 1) \ \vee \ \neg w)^* \ ; \ (pc = 3) \longrightarrow pc, w := 5, false \ ; \ ((pc \neq 1) \ \vee \ \neg w)^*$
6 :	$(\neg s \ \vee \ w)^*(s \ \wedge \ \neg w)true^* \ ; \ (pc = 4) \longrightarrow pc, w, s := 5, false, false \ ; \ true^*$
7 :	$true^* \ ; \ (pc = 4) \longrightarrow pc, s := 5, false, false \ ; \ (\neg s \ \vee \ w)^*(s \ \wedge \ \neg w)true^*$
8 :	$(\neg w)^* \ ; \ (pc = 5) \longrightarrow pc := 6 \ ; \ (\neg w)^*$
9 :	$(\neg s)^* \ ; \ (pc = 6) \longrightarrow pc := 7 \ ; \ true^*$
10 :	$true^* \ ; \ (pc = 7) \longrightarrow pc, s := 0, false \ ; \ true^*$

Figure 1: Actions for Modelling Szymanski's Algorithm

$i = 0, 1, \ldots, n-1$. Similarly, we let α^+ be the action constructed by repeating the action α one or more times. We shall characterize the relation between configurations given by α^*. For the characterization, we shall use the following properties of contexts:

Lemma 3.1 *Let ϕ_L be a left context with tail Σ_L. If $w_1 \in \phi_L$ then $w_1 \ w_2 \in \phi_L$ if and only if $w_2 \in \Sigma_L^*$.*

Lemma 3.2 *Let ϕ_R be a right context with tail Σ_R. If $w_1 \in \phi_R$ then $w_2 \ w_1 \in \phi_R$ if and only if $w_2 \in \Sigma_R^*$.*

Proofs of these lemmas can be found in Appendix A.

We shall now characterize α^+ for any action α. A characterization of α^* can be obtained from a characteriztion of α^+ by taking the union with the identity relation.

Theorem 3.3 *Let α be an action of the form*

$$\alpha = \phi_L \ ; \ \tau(c, c') \ ; \ \phi_R$$

where $\tau(c, c')$ is idempotent, and where Σ_L (Σ_R) is the tail of ϕ_L (ϕ_R). Then α^+ consists of the set of pairs (γ, γ') of configurations of equal length (say n), such that there are indices i, j with $1 \leq i \leq j \leq n$ such that

1. $\gamma[1..i-1] = \gamma'[1..i-1] \in \phi_L$,
2. $\gamma[j+1..n] = \gamma'[j+1..n] \in \phi_R$,

3. *For each k with $i \leq k \leq j$ we have $\gamma[k] = \gamma'[k]$ or $\tau(\gamma[k], \gamma'[k])$.*

4. *For each k with $i < k \leq j$ we have $\gamma[k] \in \Sigma_R$ or $\gamma'[k] \in \Sigma_R$.*

5. *For each k with $i \leq k < j$ we have $\gamma[k] \in \Sigma_L$ or $\gamma'[k] \in \Sigma_L$.*

6. *For all indices k_1, k_2 with $i \leq k_1 < k_2 \leq j$ we have $\gamma[k_1] \in \Sigma_L \ \vee \ \gamma[k_2] \in \Sigma_R$ and $\gamma'[k_1] \in \Sigma_L \ \vee \ \gamma'[k_2] \in \Sigma_R$.* □

The proof of the theorem can be found in Appendix A.

In the symbolic reachability analysis (described in Section 4), we use regular expressions as representations of sets of configurations. The characterization of Theorem 3.3 can be used to model the effect of (repetitive applications of) actions on regular sets by using *finite-state transducers*. This approach is proposed in [KMM+97], where however acceleration is not considered.

We recall that an action α denotes a set of pairs (γ, γ') of configurations. Equivalently, we can represent the action as a set of finite strings over $C \times C$, namely as the strings $(c_1, c_1') \ (c_2, c_2') \ \cdots \ (c_n, c_n')$ such that $(c_1 c_2 \cdots c_n, c_1' c_2' \cdots c_n') \in \alpha$. It is easy to see that each action can be represented by a finite-state transducer.

More importantly, if the action α satisfies the conditions of Theorem 3.3, then the characterization of Theorem 3.3 can be used to find a representation of α^+ in a straight-forward way, since α^+ can be represented as a regular language over $c \times c$. As an example, in Figure 2 we show the transducer which accepts α^* where α is the action at line 3 in Figure 1. We note that the transducer in Figure 2 need not use the full generality of the characterization of Theorem 3.3, since the alphabets Σ_L and Σ_R both are equal to the set of all colours. In the figure we use *copy* to

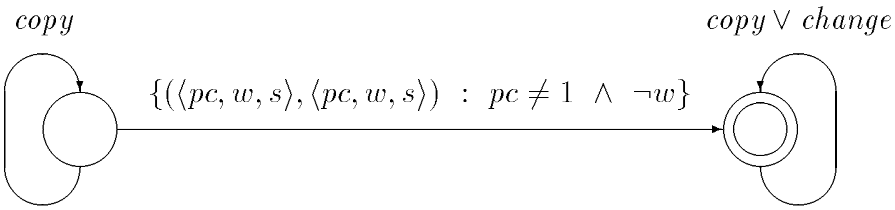

Figure 2: Transducer for α^* from line 3 of Figure 1.

denote pairs $(\langle pc, w, s\rangle, \langle pc, w, s\rangle)$ of identical tuples, and *change* to denote pairs $\{(\langle pc, w, s\rangle, \langle pc', w', s'\rangle) \ : \ pc = 3 \ \wedge \ pc' = 4 \ \wedge \ w' = w \ \wedge \ s' = \mathit{false}\}$ that represent a change of local control state.

The general construction of a transducer which characterizes α^+ for any action α can be found in Appendix C.

For a regular expression ϕ and an action α, we use $\alpha^*(\phi)$ to denote the regular expression we get by computing (in the usual way) the product of ϕ and the transducer corresponding to α^*.

Composition. Let $\alpha = \phi_L \ ; \ \tau \ ; \ \phi_R$ and $\alpha' = \phi'_L \ ; \ \tau' \ ; \ \phi'_R$ be two actions of the same program. We define their local sequential composition, denoted $\alpha; \alpha'$, as

$$\alpha; \alpha' = \phi_L \cap \phi'_L \ ; \ \tau; \tau' \ ; \ \phi_R \cap \phi'_R$$

where $\tau; \tau'$ is the set (c, c') of pairs of colours such that there is a colour c'' with $\tau(c, c'')$ and $\tau'(c'', c')$. We note that the intersection of two left contexts is always a left context, and similarly for right contexts. In general, $\tau; \tau'$ need not be idempotent, so the composition is defined only if $\tau; \tau'$ is idempotent.

As an isolated operation, sequential composition does not add to the power of reachability analysis. However, if used in combination with the acceleration operation, i.e. applying $(\alpha; \alpha')^*$ (in addition to α^*), it can often give extra power to the reachability algorithm.

4 Verification

In this section we show how to verify our protocols using an enhanced version of the standard algorithm for performing symbolic forward reachability analysis. We use regular expressions as symbolic representations of (potentially infinite) sets of configurations. Given a set of configurations, the standard algorithm produces, at each step, the set of successors corresponding to a single application of an action. However, as we shall illustrate later in the section, the algorithm will not terminate when applied to any of the protocols which are of interest for us in this paper. To solve this problem, we use the operation α^* (defined in Section 3) to speed up the search of the state space. We recall that α^* computes the set of successors corresponding to an arbitrary number of applications of an action (rather than a single application).

Suppose we are given a program $\mathcal{P} = \langle C, \phi_I, \mathcal{A} \rangle$ and a regular expression ϕ_F, and want to check whether some state $\gamma_F \in \phi_F$ is reachable in $\mathcal{P}$. We maintain a set V which we use to store regular expressions which are generated during the search. At the start, the set V contains the single regular expression ϕ_I. From each unexplored element ϕ in V, we compute a set of new elements namely $\alpha^*(\phi)$ for each action α. When a new element ϕ is generated, it is compared with those which are already in V. If $\phi \subseteq \phi'$ for some $\phi' \in V$, then ϕ is discarded, since it will not add new configurations to the searched state space (it is actually sufficient that $\phi \subseteq \sum_{\phi' \in V} \phi'$ for ϕ to be safely discarded). In fact, we can also discard all $\phi' \in V$ with $\phi' \subseteq \phi$. It is also checked whether ϕ has a non-empty intersection with ϕ_F. If the intersection is non-empty, the algorithm terminates. Otherwise, the algorithm is terminated when no new regular expressions can be generated. Obviously, our algorithm is incomplete in the sense that while it will always find reachable configurations in ϕ_F, it will not necessarily terminate if all configurations in ϕ_F are unreachable.

We illustrate our algorithm through an application to Szymanski's protocol. To simplify the notation we use the coding of colours shown in Table 1, so e.g. c_2 corresponds to the colour $\langle 2, \mathit{false}, \mathit{false} \rangle$. We start from a set of initial states represented by a regular expression $\phi_0 = c_1^*$.

First, we observe that if we apply the standard reachability algorithm, then we get an infinite loop as follows. Form ϕ_0 can apply action 1 getting $c_1^*\, c_2\, c_1^*$. Applying action 1 again gives us $c_1^*\, c_2\, c_1^*\, c_2\, c_1^*$, etc.

Although the standard algorithm fails, using the speed-up operation leads to termination. In Table 2 we describe a simulation of our algorithm. We start from the set of initial configurations ϕ_0. For each regular expression ϕ_i and action α, we compute $\alpha^*(\phi_i)$ and add the result to the set of existing expressions. In case of ϕ_0, only action 1 is enabled, giving ϕ_1. Whenever an expression is entailed by another one (e.g. $\phi_4 \subseteq \phi_8$), we indicate that in the table. In such a case, the constraint (ϕ_4 here) will not be analyzed further in the algorithm. The algorithm terminates in 13 steps. Notice that in this example, it is sufficient to accelerate single actions in order to achieve termination, and hence there is no need to use the sequential composition operator. However, in the case of the bakery algorithm (Appendix B), we need to use both operators.

c_1	$\langle 1, \mathit{false}, \mathit{false} \rangle$
c_2	$\langle 2, \mathit{false}, \mathit{false} \rangle$
c_3	$\langle 3, \mathit{true}, \mathit{true} \rangle$
c_4	$\langle 4, \mathit{true}, \mathit{false} \rangle$
c_5	$\langle 5, \mathit{false}, \mathit{true} \rangle$
c_6	$\langle 6, \mathit{false}, \mathit{true} \rangle$
c_7	$\langle 7, \mathit{false}, \mathit{true} \rangle$

Table 1: Coding of colours in analysis of Szymanski's algorithm

ϕ_0	c_1^*	ϕ_1	
ϕ_1	$(c_1 + c_2)^*$	ϕ_2	
ϕ_2	$(c_1 + c_2 + c_3)^*$	ϕ_3, ϕ_4	
ϕ_3	$(c_1 + c_2 + c_3 + c_4)^*\ c_2\ (c_1 + c_2 + c_3 + c_4)^*$	ϕ_5	
ϕ_4	$(c_1 + c_3)^* c_5 (c_1 + c_3)^*$		Subs ϕ_8
ϕ_5	$(c_1 + c_2 + c_3 + c_4)^*\ c_3\ (c_1 + c_2 + c_3 + c_4)^*$	ϕ_6, ϕ_7	
ϕ_6	$(c_1 + c_2 + c_3 + c_4)^*\ c_2\ (c_1 + c_2 + c_3 + c_4)^*\ c_4\ (c_1 + c_2 + c_3 + c_4)^*$ $+$ $(c_1 + c_2 + c_3 + c_4)^*\ c_4\ (c_1 + c_2 + c_3 + c_4)^*\ c_2\ (c_1 + c_2 + c_3 + c_4)^*$		Subs ϕ_3
ϕ_7	$(c_1 + c_3 + c_4)^*\ c_5\ (c_1 + c_3 + c_4)^*$	ϕ_8	Subs ϕ_8
ϕ_8	$(c_1 + c_3 + c_4 + c_5)^*\ c_5\ (c_1 + c_3 + c_4 + c_5)^*$	ϕ_9	
ϕ_9	$(c_1 + c_5 + c_6)^*\ c_6\ (c_1 + c_5 + c_6)^*$	ϕ_{10}	
ϕ_{10}	$c_1^*\ c_7\ (c_1 + c_5 + c_6)^*$	ϕ_{11}	
ϕ_{11}	$c_1^*\ c_1\ (c_1 + c_5 + c_6)^*$	ϕ_{12}, ϕ_{13}	
ϕ_{12}	$(c_1 + c_2)^*\ c_2\ (c_1 + c_2)^*$		Subs ϕ_2
ϕ_{13}	$c_1^*\ c_1\ c_7\ (c_1 + c_5 + c_6)^*$		Subs ϕ_{10}

Table 2: Reachability Analysis of Szymanski's Algorithm

5 Conclusions

In the paper, we have presented techniques for reachability analysis of parametrized systems where a configuration of the system can be described by a string representing the local states of the processes. We have found that naive symbolic reachability analysis does not converge for such systems, and propose to use acceleration of actions, possibly in combination with composition. We showed that with this extension, symbolic reachability analysis does indeed terminate for idealized versions of parametrized mutual algorithms, such as Szymanski's algorithm, Burn's and Dijkstra's mutual exclusion algorithms, and the bakery and ticket algorithms by Lamport. We should emphasize that we do not in general guarantee termination.

We further note that we have considered idealized versions of the mutual exclusion algorithms. In most implementations of these algorithms, a global guard (such as e.g., $\forall j : j < i : \neg s_j$) is not atomic, but implemented as a loop which checks the states of other processes. We have not considered how to treat non-atomic global guards in this paper.

References

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Proc.* 10^{th} *Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318, 1998.

[AČ98] Parosh Aziz Abdulla and Karlis Čerāns. Simulation is decidable for one-counter nets. In *Proc. CONCUR '96,* 9^{th} *Int. Conf. on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 253–268, 1998.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc.* 5^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 414–425, Philadelphia, 1990.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[AJ98] Parosh Aziz Abdulla and Bengt Jonsson. Verifying networks of timed processes. In Bernhard Steffen, editor, *Proc. TACAS '98,* 7^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 298–312, 1998.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, 1996.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. of the Fourth International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer Verlag, 1997.

[BH97] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. In *Proc. ICALP '97*, number 1256 in Lecture Notes in Computer Science, 1997.

[BS95] O. Burkart and B. Steffen. Composition, decomposition, and model checking of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.

[Čer94] K. Čerāns. Deciding properties of integral relational automata. In Abiteboul and Shamir, editors, *Proc. ICALP '94*, volume 820 of *Lecture Notes in Computer Science*, pages 35–46. Springer Verlag, 1994.

[CGJ95] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In Lee and Smolka, editors, *Proc. CONCUR '95, 6^{th} Int. Conf. on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 395–407. Springer Verlag, 1995.

[Esp95] J. Esparza. Petri nets, commutative context-free grammers, and basic parallel processes. In *Proc. Fundementals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 221–232, 1995.

[GS92] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of Szymanski's algorithm. In *Proc. TACAS '98, 7^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 424–438, 1998.

[Hen95] T.A. Henzinger. Hybrid automata with finite bisimulations. In *Proc. ICALP '95*, 1995.

[Jan90] P. Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Computer Science*, 74:71–93, 1990.

[JL98] E. Jensen and N. A. Lynch. A proof of Burn's n-process mutual exclusion algorithm using abstraction. In *Proc. TACAS '98, 7^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 409–423, 1998.

[JP93] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2):272–302, Dec. 1993.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proc. 8^{th} ACM Symp. on Principles of Distributed Computing, Canada*, pages 239–247, Edmonton, Alberta, 1989.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254, pages 424–435, Haifa, Israel, 1997. Springer Verlag.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. chang, M. Colón, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STEP: the stanfor temporal prover. Draft Manuscript, June 1994.

[MP90] Z. Manna and A. Pnueli. An exercise in the verification of multi – process programs. In W.H.J. Feijen, A.J.M van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business*, pages 289–301. Springer-Verlag, 1990.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants (extended abstract). In Sifakis, editor, *Proc. Workshop on Computer Aided Verification*, number 407 in Lecture Notes in Computer Science, pages 68–80, 1989.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc. 13^{th} ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.

A Proof of Some Lemmas

Proof of Lemma 3.1

Let us check whether $w_1\ w_2 \in \phi_L$ by running the automaton $\mathcal{A}_{\phi_L}$ on $w_1\ w_2$. When the automaton has read w_1 it is in its final accepting state. Thereafter it will accept $w_1\ w_2$ if and only if it thereafter remains in its final accepting state, i.e., if and only if $w_2 \in \Sigma_L$.

Proof of Theorem 3.3

Proof: We first prove that the above characterization is necessary. So, let α^+ contain the pair (γ, γ'). Let i be the lowest index at which γ and γ' differ and let j be the highest index at which γ and γ' differ. Remember that we assume that at most one change occurs at each position in γ. If there is only one change, then $i = j$, otherwise $i < j$. The fact that some change occurred at position i implies the first condition, and the fact that some change occurred at position j implies the second condition. The third criterion is obvious. If criterion 4 is false, then it will never be possible to carry out any change at position i, since the string between $i+1$ and j will never be in Σ_R^* and therefore (by Lemma 3.2) the string between $i+1$ and n will never be in ϕ_R. Analogously, criterion 5 must be true. To derive the last criterion, assume that for $i \leq k_1 < k_2 \leq j$ we have $\gamma[k_1] \notin \Sigma_L$ and $\gamma[k_2] \notin \Sigma_R$. Then we cannot perform a change at k_1 before a change at k_2 or vice versa. Hence we cannot perform a change at either k_1 or k_2 which contradicts either criterion 4 or criterion 5. Analogously, we must have $\gamma'[k_1] \in \Sigma_L \ \vee \ \gamma'[k_2] \in \Sigma_R$.

We next prove that the above requirements are sufficient. Let γ and γ' be configurations of length n that satisfy requirements 1 - 6. Let $k_1, \ldots, k_m$ be the positions with $i = k_1 < k_2 < \cdots < k_m = j$ at which γ and γ' differ. We shall prove that there exists an ordering between these indices at which α can be performed in order to transform γ into γ'. More precisely, we shall find a permutation $l_1\ l_2\ \cdots\ l_m$ of $k_1\ k_2\ \cdots\ k_m$ which generates a sequence $\tilde{\gamma}_0\ \tilde{\gamma}_1\ \tilde{\gamma}_2\ \cdots\ \tilde{\gamma}_m$ of configurations such that $\tilde{\gamma}_0 = \gamma$ and $\tilde{\gamma}_m = \gamma'$, and such that for each $p = 0, \ldots, m-1$

- $\tilde{\gamma}_p[1..l_p - 1] \in \phi_L$
- $\tilde{\gamma}_p[l_p + 1..n] \in \phi_R$

- $\tilde{\gamma}_{p+1} = \tilde{\gamma}_p[l_p \mapsto \gamma'[l_p]]$ (i.e., $\tilde{\gamma}_{p+1}$ is obtained from $\tilde{\gamma}_p$ by replacing the element at position l_p by $\gamma'[l_p]$).

When now γ and γ' are fixed, let us introduce the notation I_L to denote the set k of indices such that $\gamma[k] \in \Sigma_L$ and I'_L to denote the set k of indices such that $\gamma'[k] \in \Sigma_L$. The sets I_R and I'_R are defined similarly.

The first half of requirement 6 implies that the sequence $k_1\ k_2\ \cdots\ k_m$ consists of an initial part with indices in I_L followed by an index k_p which need neither be in I_L or I_R, followed by a remaining part with indices in I_R. Similarly, the second half of requirement 6 implies that the sequence $k_1\ k_2\ \cdots\ k_m$ consists of an initial part with indices in I'_L followed by an index k_q which need neither be in I'_L or I'_R, followed by a remaining part with indices in I'_R.

In the case that $k_p \notin (I_L \cup I_R)$ and $k_q \notin (I'_L \cup I'_R)$ and k_p occurs before k_q, then the sequence $k_1\ k_2\ \cdots\ k_m$ can be written as

$$k_1\ \cdots\ k_{p-1}\ k_p\ k_{p+1}\ \cdots\ k_{q-1}\ k_q\ k_{q+1}\ \cdots\ k_m$$

where

1. $k_1\ \cdots\ k_{p-1}$ are in $I_L \cap I'_L$. By requirement 4 they must also (except for k_1) be in $I_R \cup I'_R$. We can thus view the sequence $k_1\ \cdots\ k_{p-1}$ as an interleaving of a sequence $k^{(1)}_1\ \cdots\ k^{(1)}_{m_1}$ in $I_L \cap I'_L \cap I'_R$ and a sequence $k^{(2)}_1\ \cdots\ k^{(2)}_{m_2}$ in $I_L \cap I'_L \cap I_R$ (with the obvious adjustment for k_1).
2. $k_p \in I'_L \cap I'_R$ (If $p = 1$ then we only require $k_p \in I'_L$).
3. $k_{p+1}\ \cdots\ k_{q-1}$ are in $I_R \cap I'_L$.
4. $k_q \in I_L \cap I_R$ (If $q = m$ then we only require $k_q \in I_R$).
5. $k_{q+1}\ \cdots\ k_m$ are in $I_R \cap I'_R$. By requirement 5 they must also (except for k_m) be in $I_L \cup I'_L$. We can thus view the sequence $k_{q+1}\ \cdots\ k_m$ as an interleaving of a sequence $k^{(3)}_1\ \cdots\ k^{(3)}_{m_3}$ in $I_R \cap I'_R \cap I'_L$ and a sequence Let $k^{(4)}_1\ \cdots\ k^{(4)}_{m_4}$ in $I_R \cap I'_R \cap I_L$ (with the obvious adjustment for k_m).

We can now construct the sequence $l_1\ l_2\ \cdots\ l_m$ as follows:

1. First we take k_p followed by $k^{(1)}_{m_1}\ k^{(1)}_{m_1-1}\ \cdots\ k^{(1)}_1$. We note that after these changes, all indices k with $i < k \leq j$ have colours in Σ_R.
2. We then take the sequence $k^{(2)}_1\ \cdots\ k^{(2)}_{m_2}$ followed by $k_{p+1}\ \cdots\ k_{q-1}$ followed by $k^{(3)}_1\ \cdots\ k^{(3)}_{m_3}$. We note that after these changes, all indices k with $i \leq k < j$ have colours in Σ_L.

3. Finally, we take the sequence $k^{(4)}_{m_4}\ k^{(4)}_{m_4-1}\ \cdots\ k^{(4)}_1$ followed by k_q.

There are also some special cases. E.g., if $k_p \in (I_R \cup I'_L)$ then it can be said to belong to the sequence $k_{p+1}\ \cdots\ k_{q-1}$ so that the sequence of changes start with $k^{(1)}_{m_1}\ k^{(1)}_{m_1-1}\ \cdots\ k^{(1)}_1$. If $k_p \in (I_L \cup I'_L \cup I'_R)$ then it can be said to belong to the sequence $k_1\ \cdots\ k_{p-1}$ with a similar treatment.

The case where $p \notin (I_L \cup I_R)$ and $q \notin (I'_L \cup I'_R)$ and p occurs after q is analogous. □

B The Bakery Algorithm

An idealized version of the Bakery mutual exclusion algorithm can be given as the following pseudo-code for any process. Each process p has an integer-valued variable $ticket_p$, whose value is initially 0.

Idle :	$ticket_p := 1 + \max_q ticket_q$;
Waiting :	$\mathbf{await}\ \forall q \neq p : (ticket_p < ticket_q\ \vee\ ticket_q = 0)$;
Critical :	$ticket_p := 0$

We can model this algorithm as follows: Since only positive values of tickets are interesting, we represent the global state of the system by an array of colours with indices from 1 up to some suitablly large number. The colour at each index is either W (for Waiting), C (for Critical), and $\perp$ (process neither in Waiting nor Critical). The initial configuration is in the set $\perp^*$. We represent binary relations on colours by guarded commands, where the local colour is held in the variable c. Note that the first pseudocode

$$
\begin{array}{ll}
1a: & \varepsilon;\ c = \perp \longrightarrow c := W\ ;\ (\perp)^* \\
1b: & (\perp + C + W)^*(C + W)(C + W)^*\ ;\ c = \perp \longrightarrow c := W\ ;\ (\perp)^* \\
2: & (\perp)^*\ ;\ c = W \longrightarrow c := C\ ;\ (\perp + C + W)^* \\
3: & (\perp + C + W)^*\ ;\ c = C \longrightarrow c := \perp\ ;\ (\perp + C + W)^*
\end{array}
$$

Figure 3: Actions for Modeling the Bakery Algorithm

statement is represented by two actions: The first action (1a) for the case that there are no other process in states Waiting or Critical with tickets, and the second action (1b) for the case that there are some process(es) in states

Waiting or Critical with tickets, and that the new ticket will be chosen larger than the existing ones.

We can perform a forward reachability analysis to generate the set of reachable states of the algorithm. In order for this analysis to terminate, we must perform a speedup on the composition $2;3$ of transitions 2 and 3 in the above program. This transition is

$$(\perp)^* \; ; \; c = W \longrightarrow c := \perp \; ; \; (\perp + C + W)^*$$

The reachability analysis converges in a few steps, according to Table 3. In this table, each row contains from left to right: the name of a regular expression, a regular expression, an action which is explored from this regular expression, and the name of a regular expression denoting the result of that exploration.

ϕ_0	$\perp^*$	1a	ϕ_1	
ϕ_1	$W \; \perp^*$	1b*	ϕ_2	Subs ϕ_2
ϕ_2	$W \; W^* \; \perp^*$	$(2;3)^*$	ϕ_3	
ϕ_3	$\perp^* \; W^* \; \perp^*$	(2)	ϕ_4	
ϕ_4	$\perp^* \; C \; W^* \; \perp^*$			

Table 3: Reachability Analysis of the Bakery Algorithm

C General Transducer

In order to illustrate that the conditions in Theorem 3.3 characterizes a regular relation between configurations, we show a representation of this relation in terms of a finite-state transducer. We show the part which is inserted between the accepting state q_L of an automaton that copies strings in ϕ_L and the initial state q_R of an automaton that copies strings in ϕ_R. In Figure 4, we show the general construction. Edges are labeled by predicates on pairs (c, c') of colours that are read. We use the abbreviations c_L for $c \in \Sigma_L$, c'_L for $c' \in \Sigma_L$, c_R for $c \in \Sigma_R$, and c'_R for $c' \in \Sigma_R$. In addition to the transitions in the figure, there are self-loop at states q_1, q_2, q_3, and q_4 labeled $c = c' \in \Sigma_L \cap \Sigma_R$. Informally, the states correspond to the following situations.

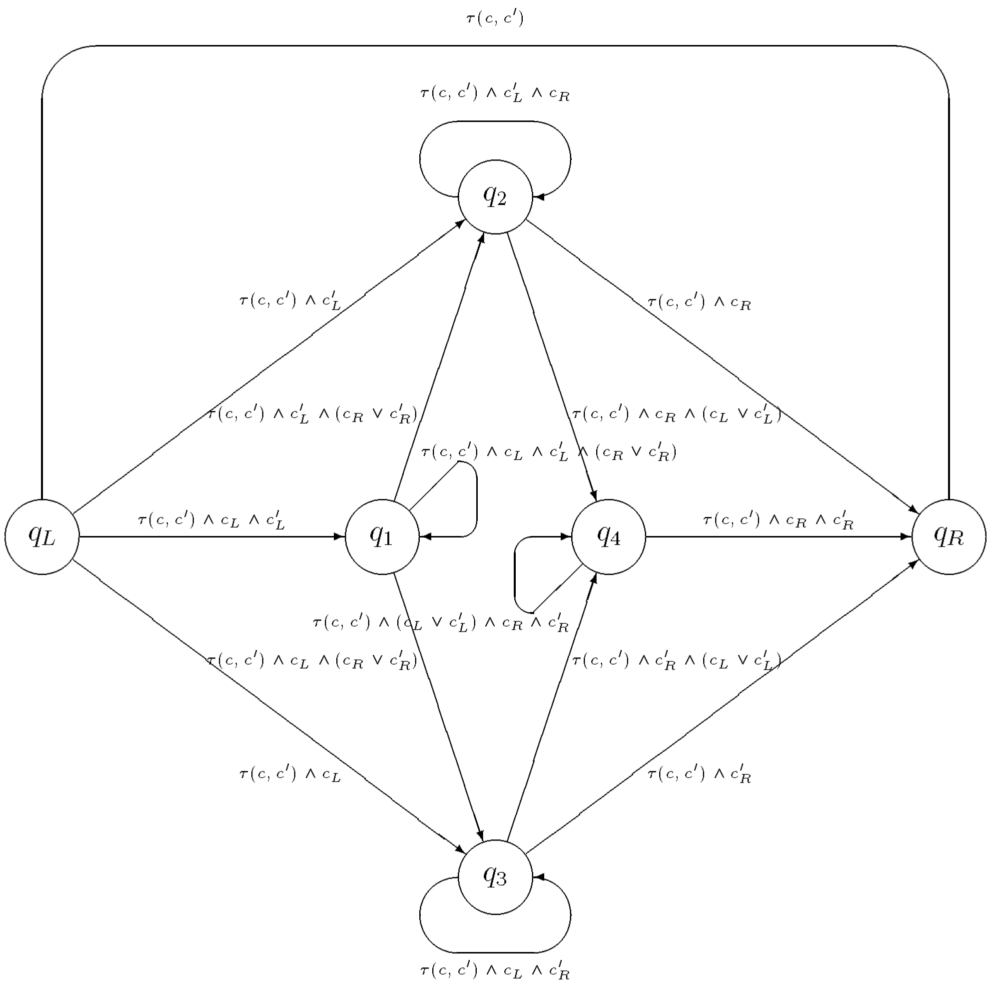

Figure 4: General transducer for α^+

- q_1 corresponds to a state when the transducer has read an index where some change has occurred, but where so far there has been no index with change at which $c \notin \Sigma_L \ \vee \ c' \notin \Sigma_L$.

- q_2 corresponds to a state when the transducer has read an index where some change has occurred where $c \notin \Sigma_L$, but where so far there has been no index with change at which $c' \notin \Sigma_L$.

- q_3 corresponds to a state when the transducer has read an index where some change has occurred where $c' \notin \Sigma_L$, but where so far there has been no index with change at which $c \notin \Sigma_L$.

- q_4 corresponds to a state when the transducer has read an index where some change has occurred where $c' \notin \Sigma_L$ and some index where some change has occurred where $c \notin \Sigma_L$.