Monotonic Abstraction in Parameterized Verification

Parosh Aziz Abdulla[1]

Department of Information Technology
Uppsala University
Sweden

Giorgio Delzanno[2]

Dipartimento Informatica e Scienze dell´Informazione
Università di Genova
Italy

Ahmed Rezine[3]

LIAFA
University of Paris 7
France

Abstract

We present a tutorial on verification of safety properties for parameterized systems. Such a system consists of an arbitrary number of processes which are organized in a linear array. The aim is to prove correctness of the system regardless of the number of processes inside the system. We give an overview of the method of *monotonic abstraction*, which provides an over-approximation of the transition system induced by a parameterized system. The over-approximation gives a transition system which is monotonic with respect to a well quasi-ordering on the set of configurations. This makes it possible to use existing methods for verification of well quasi-ordered programs.

Keywords: Model Checking, Automatic Verification, Parameterized Systems, Safety Properties

1 Introduction

In this tutorial, we explain the basic ideas of the *monotonic abstraction*, a technique which we have introduced recently [4,3,5] for automatic verification of *parameterized systems*. A parameterized system consists of an arbitrary number number of

[1] Email: parosh@it.uu.se

[2] Email: giorgio@disi.unige.it

[3] Email: rezine.ahmed@liafa.jussieu.fr

processes usually organized as a linear array. In fact, a parameterized system represents an infinite family of systems, namely one for each size of the system. We are interested in *parameterized verification*, i.e., verifying correctness regardless of the number of processes inside the system. The term *parameterized* refers to the fact that the size of the system is (implicitly) a parameter of the verification problem. Examples of parameterized systems include mutual exclusion algorithms, bus protocols, telecommunication protocols, and cache coherence protocols.

The main techniques used for verification of parameterized systems have been defined within the paradigm of *regular model checking* [14,7,9]. In regular model checking, states are represented by words where each element of the word corresponds to the local state of one process. The ordering of the elements inside the word reflects the ordering of the corresponding processes inside the array. Using words to represent configurations allows us to use finite automata (or regular expressions) to represent (infinite) sets of configurations. A configuration belongs to the set if its word encoding is accepted by the automaton. We can also encode the transition relation by *finite-state transducers.* A transducer is an extension of an automaton, where each run of the transducer both inputs and outputs a word (rather than only inputting a word). A transition from a configuration c_1 to a configuration c_2 can be encoded by a run of the transducer where we input the word encoding of c_1, and output the word encoding of c_2. Safety properties can be checked through performing reachability analysis, which amounts to applying the transducer relation iteratively to the set of initial states. The main problem with transducer-based techniques is the difficulty of computing the transitive closure. Existing methods are heavy and usually rely on several layers of computationally expensive automata-theoretic constructions; in many cases making them very inefficient and severely limiting their applicability.

In parallel, there has been an extensive research on the verification of infinite-state systems which are *monotonic* w.r.t. a *well quasi-ordering* on the set of configurations [2]. The main idea is to perform symbolic backward reachability analysis to check safety properties for such systems. The method was first reported in [6] and applied to analyze safety properties for lossy channel systems. Concretely, we define a pre-order $\preceq$ on the set of configurations such that (1) $\preceq$ is a simulation with respect to the transition relation (i.e., the transition relation is monotonic w.r.t. $\preceq$), and (2) $\preceq$ is a well-quasi ordering (WQO for short). Given such a pre-order, we can derive a backward algorithm for checking reachability of sets of configurations which are upward closed w.r.t. $\preceq$. Upward closed sets are attractive to use in this setting for several reasons. First, we are interested in safety properties, in which we check the reachability of a set of *bad configurations.* These are configurations which we do not want to occur during the execution of the system. For instance, in mutual exclusion protocols, the bad configurations are those in which at least two processes are in their critical sections. This means that checking safety properties amounts to checking reachability of upward closed sets of configurations. The second attractive feature of upward closed sets is that they can be characterized by their minimal elements, which often makes it possible to have efficient symbolic representations of infinite sets of configurations.

We start from the set of bad configurations, and then compute the sets of predecessors, i.e., sets of configurations which correspond to going one step backwards along the transition relation. Monotonicity implies that, for any upward-closed set, the set of its predecessors is an upward-closed set. Since the set of bad configurations is upward closed, it follows that all the sets which are generated are also upward closed. This procedure is guaranteed to terminate by the well quasi-ordering of the relation on the set of configurations.

Since its first application to lossy channel systems [6], the method has been used for the design of verification algorithms for a wide range of models such as Petri nets, timed Petri nets, broadcast protocols, cache coherence protocols, etc. (see, e.g., [8,10,11,12]).

Unfortunately, parametrized systems do not quite fit into this framework, in the sense that there is no nontrivial (useful) WQO for which these systems are monotonic. The ordering $\preceq$ amounts to the subword relation on words. The main obstacle is that parameterized systems usually use universal global conditions in which a process may need to check states of all other processes inside the system. Universal conditions are inherently non-monotonic, since having larger configurations may lead to the violation of the universal condition. In this tutorial, we give an overview of the method of *monotonic abstraction* [4,3,5,1], which attempts to overcome this problem by defining an abstract semantics which *forces* 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 any smaller configuration $c_1' \preceq c_1$ to c_2. More precisely, the abstraction kills (deletes) all the processes inside the configuration which violate the universal condition. Since the abstract transition relation is an over-approximation of the original one, proving a safety property in the abstract system implies that the property also holds in the original system.

Monotonic abstraction has been used for performing shape analysis [1], and for parametrized verification of mutual exclusion and cache coherence protocols [4,3,5]. Surprisingly, it leads to quite efficient analysis which can handle *fully automatically* several non-trivial examples of such systems [4,3].

Outline

In the next section, we give an overview of parameterized systems, using a simple protocol that we use as a running example throughout this tutorial. In Section 3, we introduce the (infinite) transition systems which arise from parameterized systems. In Section 4 we introduce our ordering on the set of configurations, and then we define our abstraction in Section 5. We describe our approximated algorithm in Section 6. In Section 7, we simulate the reachability algorithm on a simple example. Finally, in Section 8, we describe several features which can be used to enrich the basic model of Section 2.

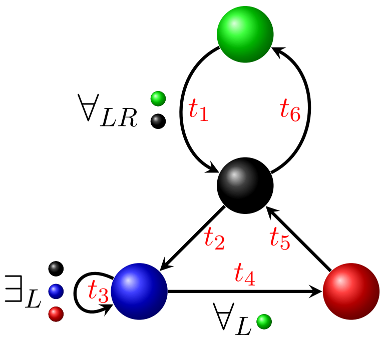

Fig. 1. One process in the protocol.

2 Parameterized Systems

In this section, we introduce the concept of *parameterized systems*, through a simple example of a protocol which implements mutual exclusion among an arbitrary number of processes.

A *parameterized system* consists of an arbitrary number of processes each of which is a finite-state process. The processes are (usually) organized as a linear array. In each step in the execution of a parameterized system, one process, called the *active* process, changes state. The rest of the processes, called the *passive* processes, do not change states. We call the passive processes to the left of the active process the *left context* of the active process. The *right context* is defined analogously. The active process may perform a *local transition* in which it changes its state independently of the states of the passive processes. The active process may also perform a *global transition* in which it checks the states of the passive processes. A global transition is either *universally* or *existentially* quantified. An example of a universal condition is that *all* processes in the left context of the active process should be in certain states. In an existential transition we require that *some* (rather than *all*) processes should be in certain states.

In our example, each process (depicted in Figure 2) has four local states, namely the green, black, blue, and red states. We represent these states by coloured balls 🟢, ⚫, 🔵, and 🔴. Sometimes, we refer to a process in a configuration by its state, so we say e.g. "the red process" rather than "the process in its red state".

Initially, all the processes are green (they are idle). When a process becomes interested in accessing the critical section (which corresponds to the red state), it declares its interest by moving to the black state. This is described by the global universal transition rule t_1 in which the move is allowed only if all other processes are in their green or black states. The universal quantifier labeling t_1 encodes the condition that all other processes (whether in the left or the right context – hence the index LR of the quantifier) of the active process should be green or black.

In the black state, the process may move to the blue state through the local transition t_2 (in which the process does not need to check the states of the other processes). Notice that any number of processes my cross from the initial (green) state to the black state. However, once the first process has crossed to the blue state, it “closes the door” on the processes which are still in their green states. These processes will no longer be able to leave their green states until the door is opened again (when no process is blue or red). From the set of processes which have declared interest in accessing the critical section (those which have left their green states and are now black or blue) the leftmost process has the highest priority. This is encoded by the global universal transition t_4 where a process may move from its blue state to its red state only subject to the universal condition that all processes in its left context are green (the index L of the quantifier stands for “Left”). If the process finds out, through the existential global condition, that there are other processes that are black, blue, or red, then it loops back to the blue state through the existential transition t_3. Once the process leaves the critical section, it will return back to the black state through the local transition t_5. In the black state, the process chooses either to try to reach the critical section again, or to become idle (through the local transition t_6).

Formally, we represent a parameterized systems $\mathcal{P}$ by a pair (Q, T), where Q is the set of the local states of the processes, and T is the set of transition rules which define the behaviour of each process. In the above example, the set Q consists of four states (green, black, blue, and red), while the set T consists of six rules, namely three local rules (t_2, t_5, and t_6), two universal rules (t_1 and t_4), and one existential rule (t_3).

3 Transition Systems

A parameterized systems $\mathcal{P} = (Q, T)$ induces a transition systems $\mathcal{T} = (C, \longrightarrow)$, where C is the set of *configurations* and $\longrightarrow$ is a transition relation on C. A configuration is a word in Q^*, where each element of the word represents the local state of one process.

Let us consider the example of Section 2. The word 🟢🔵🔴🔵⚫ represents a configuration in an instance of the system with five processes that are in their green, blue, red, blue, and black states, in that order. Since there is no bound on the size of configurations, the set of configuration is infinite. We define the transition relation $\longrightarrow := \bigcup_{t \in T} \xrightarrow{t}$, where $\xrightarrow{t}$ is a relation on configurations that captures the effect of the transition rule t. The definition of $\xrightarrow{t}$ depends on the type of t (whether it is local, existential, or universal). We will consider three transition rules from Figure 2 to illustrate the idea.

The local rule t_2 induces transitions of the form

🔵🟢⚫🔴⚫ $\xrightarrow{t_2}$ 🔵🟢🔵🔴⚫

Here the active process changes its local state from black to blue.

The existential rule t_3 induces transitions of the form

⚫🟢🔵🔴⚫ $\xrightarrow{t_3}$ ⚫🟢🔵🔴⚫

The blue process can perform the transition since there is a black process in its left context. However, the transition is not enabled from the configuration 🟢🟢🔵🔴

⚫, since there are no red, blue, or black processes in the left context of the process trying to perform the transition.

The universal rule t_4 induces transitions of the form

🟢🟢🔵🔵⚫ $\xrightarrow{t_4}$ 🟢🟢🔴🔵⚫

The active process 🔵 can perform the transition since all processes in its left context are green. On the other hand, neither of the blue processes can perform the transition form the configuration 🔴🟢🔵🔵⚫ since, for each one of them, there is at least one process in its left context which is not green.

We use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For sets C_1 and C_2 of configurations, we use $C_1 \xrightarrow{*} C_2$ to denote that there are configurations $c_1 \in C_1$ and $c_2 \in C_2$ such that $c_1 \xrightarrow{*} c_2$.

An *initial configuration* is one in which all processes are in their initial (green) states. We use *Init* to denote the set of initial configurations. Examples of initial configurations are 🟢🟢 and 🟢🟢🟢🟢 corresponding to instances of the system with two and four processes respectively. Notice that there is an infinite set of initial configurations, namely one for each size of the system.

As mentioned in Section 2, the protocol is intended to observe mutual exclusion. In other words, we are interested in verifying a *safety property*. To do this we characterize the set *Bad* of configurations: all configurations which contain at least two red processes. Examples of configurations in *Bad* are 🔴🔵🔴 and 🟢🔴🔵🔴🔴. Showing the safety property amounts to proving that the protocol, starting from an initial configuration, will never reach a bad configuration. In other words, we want to answer the question whether $\mathit{Init} \xrightarrow{*} \mathit{Bad}$.

4 Ordering

In this section, we define an ordering on configurations, which we use to define bad sets of configurations, and hence also to formulate the class of safety properties which we consider in this tutorial.

For configurations c_1 and c_2, we use $c_1 \preceq c_2$ to denote that c_1 is a (not necessarily contiguous) subword of c_2. For instance, we have 🔵🔴 $\preceq$ ⚫🔵🟢🔴🔵. A set U of configurations is said to be *upward closed*, if whenever $c \in U$ and $c \preceq c'$ then $c' \in U$. For a configuration c, we use $\widehat{c}$ to denote the upward closed set $U := \{c' |\, c \preceq c'\}$, i.e., $\widehat{c}$ contains all configurations which are larger than c w.r.t. the ordering $\preceq$. In such a case, we call c the *generator* of U.

We are interested in upward closed sets for two reasons. First, all sets of bad configurations which we work with are upward closed. For instance, in the example of Section 2-3, the set *Bad* of configurations violating mutual exclusion are those which contain at least two red processes. The set is upward closed since whenever a configuration contains two red processes then any larger configuration will also contain (at least) two red processes.

The second reason why we are interested in upward closed sets is that they have an efficient symbolic representation. In fact, it can be shown that each upward closed set can be characterized by a *finite* set of generators. More precisely, for an upward closed set U, there are configurations $c_1, \ldots, c_n$ with $U = \widehat{c_1} \cup \cdots \cup \widehat{c_n}$. For instance, the set *Bad* above has a single generator, namely 🔴 🔴. Thus, operations which manipulate upward closed sets can be translated into operations which manipulate words. In this manner we avoid using heavy machinery, such as regular languages, when performing reachability analysis. This makes our approach much more efficient in practice compared to automata-based methods such as regular model checking [14,7,9].

We will check safety properties using backward reachability analysis. For a set C of configurations, we use $Pre(C) := \{c |\ \exists c' \in C \cdot c \longrightarrow c'\}$. In other words, the set $Pre(C)$ contains exactly all configurations from which a configuration in C can be reached through a single application of the transition relation.

To solve the safety problem, we present a scheme for backward reachability analysis. We start with the set *Bad* of bad configurations which is upward closed. Then, we apply the function *Pre* repeatedly generating a sequence $U_0, U_1, U_2, \ldots$ of sets of configurations, where $U_0 := Bad$, and $U_{i+1} := U_i \cup Pre(U_i)$ for $i \geq 0$. We observe that the set U_i characterizes the set of configurations from which the set *Bad* is reachable within i steps. We would like the sets U_i to be upward (so that we can represent them by their finite sets of generators). In order to achieve that, we introduce a sufficient condition, namely that of *monotonicity.* Monotonicity implies that $Pre(U)$ is upward closed whenever U is upward closed. Since U_0 is upward closed by definition, monotonicity would imply that all the sets U_i are upward closed.

A transition system is said to be *monotonic* if $\preceq$ forms a simulation on the set of configurations. In other words, for all configurations c_1, c_2, c_3, whenever $c_1 \preceq c_2$ and $c_1 \longrightarrow c_3$ then $c_2 \longrightarrow c_4$ for some $c_4 \succeq c_3$.

Monotonicity implies that upward closedness is preserved through the application of *Pre* as follows. Consider an upward closed set U. Let c_1 be a member of $Pre(U)$ and let $c_2 \succeq c_1$. We will show that c_2 is also a member of $Pre(U)$. Since $c_1 \in Pre(U)$, we know by definition that there is a $c_3 \in U$ such that $c_1 \longrightarrow c_3$. By monotonicity it follows that there is a c_4 such that $c_3 \preceq c_4$ and $c_2 \longrightarrow c_4$. From $c_3 \in U$ and $c_3 \preceq c_4$ it follows that $c_4 \in U$. This means that we have found a configuration $c_4 \in U$ such that $c_2 \longrightarrow c_4$, which implies that $c_2 \in Pre(U)$.

5 Monotonic Abstraction

In this section, we define an abstraction that generates an over-approximation of the transition system. The abstract transition system is monotonic, thus allowing to work with upward closed sets. In fact, we first show that local and existential transitions are monotonic, and hence need not be approximated. Therefore, we only provide an over-approximation for universal transitions.

Consider the local transition

$c_1 =$ 🟢 ⚫ 🔴 $\xrightarrow{t_2}$ 🟢 🔵 🔴 $= c_3$

in which a process changes state from black to blue. Consider the configuration $c_2 =$ 🟢🔵🟢⚫🔴⚫ that is larger than c_1. Clearly, c_2 can perform the local transition

$c_2 =$ 🟢🔵🟢⚫🔴⚫ $\xrightarrow{t_2}$ 🟢🔵🟢🔵🔴⚫ c_4

leading to $c_4 \succeq c_2$. Local transitions are monotonic, since the active process in the small configuration (the black process in c_1) also exists in the larger configuration (i.e., c_2). A local transition does not check or change the states of the passive processes; and hence the larger configuration c_2 is also able to perform the transition, while maintaining the ordering $c_3 \preceq c_4$.

Consider the existential transition

$c_1 =$ ⚫🟢🔵🔴⚫ $\xrightarrow{t_3}$ ⚫🟢🔵🔴⚫ $= c_3$

Let us observe that the configuration c_1 can be divided into three parts: the active process 🔵, the left context ⚫🟢, and the right context 🔴⚫. Furthermore, the left context contains a witness ⚫ which enables the transition. Consider the configuration $c_2 =$ 🟢⚫🟢🔵🔴⚫🔴 that is larger than c_1. Also, the configuration c_2 can be divided into three parts: the active process 🔵, the *left context* 🟢⚫🟢, and the *right context* 🔴⚫🔴. Notice that the left context of c_2 is larger than the left context of c_1, and hence the former will also contain the witness ⚫. This means that c_2 can perform the same transition

$c_2 =$ 🟢⚫🟢🔵🔴⚫🔴 $\xrightarrow{t_3}$ 🟢⚫🟢🔵🔴⚫🔴 $= c_4$

leading to $c_4 \succeq c_3$.

Next, we motivate why universal transitions are not monotonic. Consider the universal transition

$c_1 =$ 🟢🟢🔵🔵⚫ $\xrightarrow{t_4}$ 🟢🟢🔴🔵⚫ $= c_3$.

The transition is enabled since all processes in the left context of the active process satisfy the condition of the transition (they are green). Consider the configuration $c_2 =$ 🟢⚫🟢🔴🔵🔵⚫. Although $c_1 \preceq c_2$, the universal transition t_4 is not enabled from c_2 since the left context of the active process contains processes that violate the condition of the transition. This implies that universal transitions are not monotonic.

In order to deal with non-monotonicity of universal transitions, we will work with an abstract transition relation $\longrightarrow_A$ that is an over-approximation of the concrete transition relation $\longrightarrow$. We call $\longrightarrow_A$ the *monotonic abstraction* of $\longrightarrow$. The relations $\xrightarrow{t}_A$ coincides with $\xrightarrow{t}$ in case of local and existential transitions (in these two cases, the relation is monotonic and hence no over-approximation is needed). In case t is universal, we have $c_1 \xrightarrow{t}_A c_2$ iff $c_1' \xrightarrow{t} c_2$ for some $c_1' \preceq c_1$. In other words, $c_1 \xrightarrow{t}_A c_2$ if c_1 can first "transform" to a smaller configuration from which it can perform the transition. This means for instance that

🟢⚫🟢🔴🔵🔵⚫ $\xrightarrow{t_4}_A$ 🟢🟢🔴🔵⚫

since

$$\text{🟢⚫🟢🔴🔵🔵⚫} \preceq \text{🟢🟢🔵🔵⚫} \xrightarrow{t_4} \text{🟢🟢🔴🔵⚫}$$

The abstract transition relation is monotonic also w.r.t. universal transitions, since $c_1 \preceq c_2$ and $c_1 \longrightarrow_A c_3$ implies that $c_2 \longrightarrow_A c_3$. Notice that in the over-approximation, we delete those processes in the configuration that violate the condition of the universal transition.

Since the abstract transition relation $\longrightarrow_A$ is an over-approximation of the original transition relation $\longrightarrow$, it follows that if a safety property holds in the abstract model, then it will also hold in the concrete model.

6 Backward Reachability Algorithm

We present a backward algorithm for approximated reachability analysis. The idea is that we compute the function *Pre* w.r.t. the abstract relation $\longrightarrow_A$ rather than the concrete relation $\longrightarrow$. This means that we can work with upward closed sets in the scheme for backward reachability analysis presented in Section 4. Recall that we generate a sequence $U_0, U_1, U_2, \ldots$ of sets of configurations, where $U_0 := \mathit{Bad}$, and $U_{i+1} := U_i \cup \mathit{Pre}(U_i)$ for $i \geq 0$. Since U_0 is upward closed by definition, and $\longrightarrow_A$ is monotonic, all the sets U_i are upward closed.

Several properties of upward closed sets enable us to transform the backward reachability scheme into an algorithm. First, each set can be represented by its finite set of generators. Given a configuration c, we show below how to compute the set of generators for the set $\mathit{Pre}(\widehat{c})$. This means that we only need to work with generators (configurations) as a symbolic representation of the sets which arise in the algorithm.

Now, we show that the algorithm is guaranteed to terminate. Suppose that the algorithm, during its execution, produces two generators c_1, c_2 such that $c_1 \preceq c_2$. Since $\widehat{c_2} \subseteq \widehat{c_1}$, we can safely discard c_2 from the analysis without the loss of any information. In such a case, we say that c_2 is *subsumed* by c_1. Discarding configurations in this manner makes it possible to apply the methodology of [2]. According to [2], termination of the algorithm is guaranteed since $\preceq$ is a so called *well quasi-ordering* [13]. That $\preceq$ is a well quasi-ordering means that for any infinite sequence $c_0, c_1, c_2, \ldots$ of configurations, there are $i < j$ such that $c_i \preceq c_j$.

It remains to show that we can compute the generators of $\mathit{Pre}(\widehat{c})$ for any configuration c. We define $\mathit{Pre}(\widehat{c}) := \bigcup_{t \in T} \mathit{Pre}_t(\widehat{c})$ where Pre_t gives the generators of the set of configurations from which we can reach $\widehat{c}$ through one application of the transition rule t. The definition of Pre_t depends on the type of t (whether it is local, existential, or universal). We will consider different transition rules in Figure 2 to illustrate how to compute $\mathit{Pre}_t(\widehat{c})$.

For the local rule t_5 in Figure 2, we have

$$\mathit{Pre}_{t_5}\left(\widehat{\text{🟢⚫🔵}}\right) = \left\{\text{🟢🔴🔵}\right\}$$

In other words, the predecessor set is characterized by one generator, namely 🟢🔴

🔵. Strictly speaking, the set contains also a number of other configurations such as 🟢⚫🔴🔵. However such configurations are subsumed by the original configuration 🟢⚫🔵, and therefore we will not include them in the set.

For existential transitions, there are two cases depending on whether a witness exists or not in the configuration. Consider the existential rule t_3 in Figure 2. We have

$$Pre_{t_3}\left(\text{⚫}\widehat{\text{🔵}}\text{🔴}\right) = \left\{\text{⚫🔵🔴}\right\}$$

In the above case, there is a witness ⚫ in the left context of the active process 🔵. On the other hand, we have

$$Pre_{t_3}\left(\text{🟢}\widehat{\text{🔵}}\text{🔴}\right) = \left\{\begin{array}{ll} \text{⚫🟢🔵🔴}, & \text{🟢⚫🔵🔴} \\ \text{🔴🟢🔵🔴}, & \text{🟢🔴🔵🔴} \\ \text{🔵🟢🔵🔴}, & \text{🟢🔵🔵🔴} \end{array}\right\}$$

In this case there is no witness available in the left context of the active process. Therefore, we add a witness explicitly in each possible state (⚫, 🔴, or ⚫), and each possible place in the left context of the active process. Notice that the sizes of the new generators (four processes) is larger than the size of the original configuration (three processes). This means that the sizes of the configurations generated by the backward algorithm may increase, and hence there is no bound *a priori* on the sizes of the configurations. However, termination is still guaranteed to the well quasi-ordering of $\preceq$.

For universal conditions, we check whether there are any processes in the configuration violating the condition. Consider the universal rule t_4 in Figure 2. Then

$$Pre_{t_4}\left(\text{🟢⚫}\widehat{\text{🟢}}\text{🔴🔵}\right) = \emptyset$$

since there is a black process in the left context of the potential active process (which is in state 🔴). On the other hand

$$Pre_{t_4}\left(\text{🟢}\widehat{\text{🟢🔴}}\text{🔵}\right) = \left\{\text{🟢🟢🔵🔵}\right\}$$

since all processes in the left context of the active process are in their green states.

7 Example

We show how the backward reachability algorithm runs on our running example. We start by the generator

$$g_0 = \text{🔴🔴}$$

of the set of bad configuration. The only transition which might be enabled backwards from a red state, is the one induced by the rule t_4. From the two red processes in g_0 only the left one can perform t_4 backwards (the right process cannot perform t_4 backwards since its left context contains a process not satisfying the condition of the quantifier):

$$Pre_{t_4}\left(g_0\right) = \left\{ g_1 = \text{🔵🔴} \right\}$$

From g_1, two rules are enabled backwards (both from the blue process): the local rule t_2

$$Pre_{t_2}\left(g_1\right) = \left\{ g_2 = \text{⚫🔴} \right\}$$

and the existential rule t_3

$$Pre_{t_3}\left(g_1\right) = \left\{ \text{⚫🔵🔴}, \text{🔵🔵🔴}, \text{🔴🔵🔴} \right\}$$

Since a witness is missing in the left context, we add it explicitly. All the three generators in $Pre_{t_3}\left(g_1\right)$ are subsumed by g_1.

One rule is enabled backwards from g_2, namely the local rule t_5 from the black process:

$$Pre_{t_5}\left(g_2\right) = \{g_0\}$$

Notice that the universal transition t_1 is not enabled from the black process, since there is another process (the red process) in the configuration that violates the condition of the quantifier. At this point, the algorithm terminates, since it is not possible to provide any new generators which are not subsumed by the existing ones.

Since there is no initial configuration (with only green processes) in $\widehat{g_0} \cup \widehat{g_1} \cup \widehat{g_2}$, the set of bad configurations is not reachable from the set of initial configurations in the abstract semantics. Therefore, the set of bad configurations is not reachable from the set of initial configurations in the concrete semantics, either.

8 Extensions

In our earlier works [4,3], we have considered the following extensions of the basic model of parameterized systems we present in this paper.

Local and Global Variables

We allow each process to have a set of variables which range over finite domains, and which the process can read and write locally. We also allow a finite number of global variables which all the processes may read and write.

Broadcast and Binary Communication

In a broadcast transition, an arbitrary number of processes change states simultaneously. The broadcast is initiated by a process, called the *initiator*. Together with the initiator, an arbitrary number of processes change state simultaneously. In *binary communication* two processes perform a *rendez-vous* changing states simultaneously.

Dynamic Behaviour

We allow dynamic creation and deletion of processes.

Numerical Variables

We consider parameterized systems where the individual processes operate on numerical variables which range over the natural numbers. The conditions on the numerical variables are stated as *gap-order constraints*. Gap-order constraints [15] are a logical formalism in which we can express simple relations such as lower and upper bounds on the values of individual variables; and equality, and gaps (minimal differences) between values of pairs of variables.

References

[1] Parosh Aziz Abdulla, Ahmed Bouajjani, Jonathan Cederberg, Fréd'eric Haziza, and Ahmed Rezine. Monotonic abstraction for programs with dynamic memory heaps. In *Proc. 20th Int. Conf. on Computer Aided Verification*, 2008.

[2] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96, 11th IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.

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

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

[5] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In *Proc. VMCAI '08, 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, 2008.

[6] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *Proc. LICS '93, 8th IEEE Int. Symp. on Logic in Computer Science*, pages 160–170, 1993.

[7] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002, 13th Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.

[8] Parosh Aziz Abdulla and Aletta Nylén. Timed Petri nets and BQOs. In *Proc. ICATPN'2001: 22nd Int. Conf. on application and theory of Petri nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 53 –70, 2001.

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

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

[11] E.A. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. LICS '98, 13th IEEE Int. Symp. on Logic in Computer Science*, pages 70–80, 1998.

[12] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS '99, 14th IEEE Int. Symp. on Logic in Computer Science*, 1999.

[13] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7):326–336, 1952.

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

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