Forcing Monotonicity in Parameterized Verification: From Multisets to Words

Parosh Aziz Abdulla

Uppsala University
Department of Information Technology
P.O. Box 337
751 05 Uppsala
Sweden
http://user.it.uu.se/~parosh/

Abstract. We present a tutorial on verification of safety properties for parameterized systems. Such a system consists of an arbitrary number of processes; the aim is to prove correctness of the system regardless of the number of processes inside the system. First, we consider a class of parameterized systems whose behaviours can be captured exactly as Petri nets using *counter abstraction*. This allows analysis using the framework of *monotonic* transition systems introduced in [1]. Then, we consider parameterized systems for which there is no natural ordering which allows monotonicity. We describe the method of *monotonic abstraction* which provides an over-approximation of the transition system. We consider both systems where the over-approximation gives rise to reset Petri nets, and systems where the abstract transition relation is a set of rewriting rules on words over a finite alphabet.

1 Introduction

One of the widely adopted frameworks in the context of infinite-state verification is based on the concept of *monotonic systems wrt. a well-quasi ordering* [1], which provides a scheme for proving the termination of backward reachability analysis. The method was first used for the verification of lossy channel systems [6] and then extended to a general methodology in [1]. Since its introduction in [1], the framework has been extended and used for the design of verification algorithms for various models including Petri nets, cache protocols, timed Petri nets, broadcast protocols, etc. (see, e.g., [2, 11, 9, 10, 7]). The idea is to define, for a given class of models, a preorder $\preceq$ on the configuration space such that (1) $\preceq$ is a simulation relation on the considered models, and (2) $\preceq$ is a well-quasi ordering (wqo for short). If such a preorder can be defined, then it can be proved that the reachability problem of an upward-closed set of configurations (w.r.t. $\preceq$) is decidable. Indeed, (1) monotonicity implies that for any upward-closed set, the set of its predecessors is an upward-closed set, and (2) the fact that $\preceq$ is a wqo implies that every upward-closed set can be characterized by a *finite* set of minimal elements. Therefore, starting from an upward-closed set of configurations U, the iterative computation of the backward reachable configurations from U necessarily terminates since only a finite number of steps are needed to capture all minimal elements of the set of predecessors of U. Obviously, this requires that upward-closed sets can be effectively represented and manipulated (i.e., there are procedures for, e.g., computing immediate predecessors and unions, and for checking entailment). This general scheme can be applied for the verification of safety properties since this problem can be reduced to checking the reachability of a set of bad configurations which is typically an upward-closed set w.r.t. the considered preorder. (For instance, mutual exclusion is violated as soon as there are (at least) two processes in the critical section.)

Unfortunately, many systems do not fit into this framework, in the sense that there is no nontrivial (useful) wqo for which these systems are monotonic. Nevertheless, a natural approach to overcome this problem is *monotonic abstraction*. Given a preorder $\preceq$, we consider an abstract semantics which *forces* monotonicity for the considered system. In this paper, we introduce the basic ideas through a sequence of simple *parameterized systems*. A parameterized system consists of an arbitrary number of processes. Consequently, it 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. Parametrized systems do not quite fit into the wqo framework since they induce transition relations which are not monotonic. The main obstacle is that they usually use universal global conditions in which a process may need to check the states of all the 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 the case of parametrized systems, monotonic abstraction amounts to killing (deleting) all the processes inside the configuration which violate the universal condition. The abstract transition relation is an over-approximation of the original one. Hence, proving a safety property in the abstract system implies that the property also holds in the original system. For a more technical description of the method and its application to non-trivial examples see, e.g., [4, 3, 13, 5].

2 Parameterized Systems

In this section, we introduce the concept of *parameterized systems*. For this purpose, we use 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 components each of which is a finite-state process. In our example, access to the critical section is controlled by a global lock. The system is supposed to satisfy *mutual exclusion*, i.e., at most one process may have access to the global resource at any given time. In each step in the execution of a parameterized system, one process, called the *active* process performs a *local transition* changing its state. The rest of the processes, called the *passive* processes, do not change states. A process (depicted in Figure 1) has two local states, namely `I` where the process is idle and `C` where the process is in its critical section. The resource is guarded by a lock `L` whose value is equal to 1 when the lock is free and 0 otherwise. When a process wants to access the critical section, it must first acquire the lock. This can be done only if no other process has already acquired the lock. Concretely, when the process moves from `I` to `C` it checks whether the lock is free, makes the move and at the same time acquires the lock (makes it busy). Acquiring the lock is encoded by decrementing the value of the lock (the value of the lock is not allowed to become negative, and hence the process is blocked in case $\mathtt{L} = 0$). From the critical section, the process eventually releases the lock moving back to the idle state `I`. We require that the system should never reach a configuration where two or more processes are in the state `C`. Recall that we are interested in *parameterized verification*, i.e., verifying that this property is satisfied regardless of the number of competing processes.

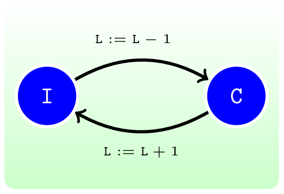

Fig. 1. One process in the simple protocol.

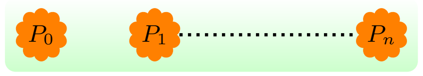

Fig. 2. A parameterized system consisting of an arbitrary number of processes.

3 Counter Abstraction

In this section, we describe how to capture the behaviour of certain classes of parameterized systems by Petri nets through the use of *counter abstraction*.

3.1 Petri Nets

A Petri net $\mathcal{N}$ is a tuple (P, T, F), where P is a finite set of *places*, T is a finite set of *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. If $(p, t) \in F$ then p is said to be an *input* place of t; and if $(t, p) \in F$ then p is said to be an *output* place of t. We use $\mathit{In}(t) := \{p|\ (p, t) \in F\}$ and $\mathit{Out}(t) := \{p|\ (t, p) \in F\}$ to denote the sets of input places and output places of t respectively.

Figure 3 shows an example of a Petri net with three places (drawn as circles), namely `L`, `W`, and `C`; and two transitions (drawn as rectangles), namely t_1 and t_2. The flow relation is represented by edges from places to transitions, and from transitions to places. For instance, the flow relation in the example includes the

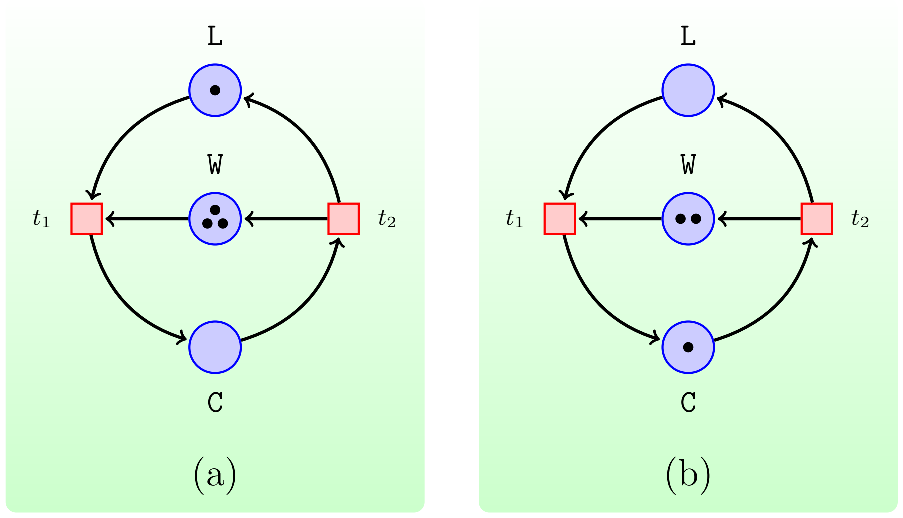

Fig. 3. (a) A simple Petri net. (b) The result of firing t_1.

pairs $(\mathtt{L}, t_1)$ and $(t_2, \mathtt{W})$, i.e., $\mathtt{L}$ is an input place of t_1, and $\mathtt{W}$ is an output place of t_2.

The transition system induced by a Petri net is defined by the set of *configurations* together with the *transition relation* defined on them. A *configuration* c of a Petri net [1] is a multiset over P. The configuration c defines the number of *tokens* in each place. Figure 3 (a) shows a configuration where there is one token in place $\mathtt{L}$, three tokens in place $\mathtt{W}$, and no token in place $\mathtt{C}$. The configuration corresponds to the multiset $\left[\mathtt{L}, \mathtt{W}^3\right]$.

The operational semantics of a Petri net is defined through the notion of *firing* transitions. This gives a transition relation on the set of configurations. More precisely, when a transition t is fired, then a token is removed from each input place, and a token is added to each output place of t. The transition is fired only if each input place has at least one token. Formally, we write $c_1 \longrightarrow c_2$ to denote that there is a transition $t \in T$ such that $c_1 \geq \mathit{In}(t)$ and $c_2 = c_1 - \mathit{In}(t) + \mathit{Out}(t)$ (where $+$ and $-$ are the usual operations defined on multisets). For sets C_1, C_2 of configurations, we write $C_1 \longrightarrow C_2$ to denote that $c_1 \longrightarrow c_2$ for some $c_1 \in C_1$ and $c_2 \in C_2$. We define $\xrightarrow{*}$ to the reflexive transitive closure of $\longrightarrow$.

3.2 Counter Abstraction

We can use *counter abstraction* to capture the behaviour of the parameterized version of the simple mutual exclusion protocol described in Section 2 as a Petri net (shown in Figure 3). The idea is to *count* the number of processes in each given local state. More precisely, we devote a place in the Petri net for each

[1] A configuration in a Petri net is often called a *marking* in the literature.

process state in the protocol. The numbers of tokens in places `I` and `C` represent the number of processes in their idle states and critical sections respectively. Absence of tokens in `L` means that the lock is currently taken by some process. Each transition of the Petri net corresponds to one of the processes performing a local transition: the transition t_1 corresponds to a process moving from `I` to `C` (thus decreasing the number of processes in the state `I`, increasing the number of processes in `C`, and taking the lock); and the transition t_2 corresponds to a process moving from `C` to `I` (thus increasing the number of processes in the state `I`, decreasing the number of processes in `C`, and releasing the lock).

3.3 Safety Properties

We are interested in checking a safety property for the Petri net in Figure 3. In a safety property, we want to show that "nothing bad happens" during the execution of the system. Typically, we define a set *Bad* of configurations, i.e., configurations which we do not want to occur during the execution of the system. In this particular example, we are interested in proving mutual exclusion. The set *Bad* contains those configurations that violate mutual exclusion, i.e., configurations in which at least two processes are in their critical sections. These configurations are of the form $\left[\mathtt{L}^k, \mathtt{W}^m, C^n\right]$ where $n \geq 2$. The set C_{init} of *initial configurations* are those where all processes are idle. Examples of initial configurations are $\left[\mathtt{I}^2\right]$ and $\left[\mathtt{I}^5\right]$, corresponding to instances of the system with two and five processes respectively. Notice that there are infinitely many initial configurations (one for each possible size of the system). Checking the safety property can be carried out by checking whether we can fire a sequence of transitions taking us from an initial configuration to a bad configuration, i.e., we check whether the set *Bad* is reachable (i.e., whether $C_{init} \xrightarrow{*} Bad$).

3.4 Ordering

We define the ordering $\leq$ on configurations to be the standard one on multisets, i.e., $c_1 \leq c_2$ if $c_1(p) \leq c_2(p)$ for each $p \in P$. According to Dickson's lemma [8], the relation $\leq$ is a *well quasi-ordering* (*wqo* for short), i.e., for each infinite sequence $c_0, c_1, c_2, \ldots$ of configurations there are i and j such that $i < j$ and $c_i \leq c_j$.

We will work with sets of configurations which are upward closed with respect to $\leq$. For a configuration c, we define $\widehat{c}$ to be the set of configurations which are larger than c wrt. $\leq$, i.e., $\widehat{c} = \{c' |\ c \leq c'\}$. For a set C, we define $\widehat{C} := \cup_{c \in C} \widehat{c}$. For an upward closed set U, we define the *generator* of U to be the set of minimal elements of U, i.e., the set G such that

- $\widehat{G} = U$, i.e., U can be generated from G by taking the upward closure of G wrt. $\leq$.
- $a \leq b$ implies $a = b$ for all $a, b \in G$. In other words, the set G is canonical in the sense that all its elements are incomparable wrt. $\leq$.

We use $gen\,(U)$ to denote the set G. Upward closed sets are interesting in our setting for two reasons:

- The set $gen\,(U)$ is finite; otherwise we would have an infinite set of incomparable elements which contradicts the wqo property. This means that each upward closed set U can be characterized by a *finite* set of configurations, namely its generator $gen\,(U)$. The set $gen\,(U) = \{a_1, \ldots, a_n\}$ is a finite characterization of U in the sense that $U = \widehat{a_1} \cup \cdots \cup \widehat{a_n}$.
- Sets of bad configurations are almost always upward closed. For instance, in our example, whenever a configuration contains two processes in their critical sections then any larger configuration will also contain (at least) two processes in their critical sections, so the set *Bad* is upward closed. In this manner, checking the safety property amounts to deciding reachability of an upward closed set.

3.5 Monotonicity

Consider the ordering $\leq$ on the configurations of the Petri net. It follows from the definitions that the transition relation $\longrightarrow$ is *monotonic* wrt. $\leq$. In other words, given configurations c_1, c_2, and c_3, if $c_1 \longrightarrow c_2$ and $c_1 \leq c_3$, then there is a configuration c_4 such that $c_2 \leq c_4$ and $c_3 \longrightarrow c_4$.

3.6 Computing predecessors

Consider an upward closed set U of configurations. By monotonicity it follows that the set $\{c |\; c \longrightarrow U\}$ is upward closed. For a configuration c and a transition t, we define $Pre(t)(c)$ to be the set $\{c_1, \ldots, c_n\}$ which is the generator of the set of configurations from which we can reach $\widehat{c}$ through a single firing of t.

3.7 Backward Reachability Analysis

As mentioned above, we are interested in checking whether it is the case that the set *Bad* of configurations is reachable. The safety property is violated iff the question has a positive answer. The algorithm, illustrated in Figure 4, starts from the set of bad configurations, and tries to find a path backwards through the transition relation to the set of initial configurations. The algorithm operates on upward closed sets of configurations. An upward closed set is symbolically represented by a finite set of configurations, namely the members of its generator. In the above example, the set $gen\,(Bad)$ is the singleton $\left\{\left[\mathtt{C}^2\right]\right\}$. Therefore, the algorithm starts from the configuration $c_0 = \left[\mathtt{C}^2\right]$, and repeatedly computes predecessors through applying the function *Pre*. From the configuration c_0, we go backwards and derive the generator of the set of configurations from which we can fire a transition and reach a configuration in $Bad = \widehat{c_0}$. Transition t_1 gives the configuration $c_1 = [\mathtt{L}, \mathtt{W}, \mathtt{C}]$, since $\widehat{c_1}$ contains exactly those configurations from which we can fire t_1 and reach a configuration in $\widehat{c_0}$. Analogously, transition t_2 gives the configuration $c_2 = \left[\mathtt{C}^3\right]$, since $\widehat{c_2}$ contains exactly those configurations from which we can fire t_2 and reach a configuration in $\widehat{c_0}$. Since $c_0 \leq c_2$, it follows that $\widehat{c_2} \subseteq \widehat{c_0}$. In such a case, we say that c_2 is *subsumed* by c_0. Since $\widehat{c_2} \subseteq \widehat{c_0}$, we can discard c_2 safely from the analysis without the loss of any information. Now, we repeat the procedure on c_1, and obtain the configurations $c_3 = \left[\mathtt{L}^2, \mathtt{W}^2\right]$ (via t_1), and $c_4 = \left[\mathtt{C}^2\right]$ (via t_2), where c_4 is subsumed by c_0. Finally, from c_3 we obtain the configurations $c_5 = \left[\mathtt{L}^3, \mathtt{W}^3\right]$ (via t_1), and $c_6 = [\mathtt{L}, \mathtt{W}, \mathtt{C}]$ (via t_2). The configurations c_5 and c_6 are subsumed by c_3 and c_1 respectively. The iteration terminates at this point since all the newly generated configurations were subsumed by existing ones, and hence there are no more new configurations to consider. In fact, the set $B = \left\{\left[\mathtt{C}^2\right], [\mathtt{L}, \mathtt{W}, \mathtt{C}], \left[\mathtt{L}^2, \mathtt{W}^2\right]\right\}$ is the generator of the set of configurations from which we can reach a bad configuration. The three members in B are those configurations which are not discarded in the analysis (they were not subsumed by other configurations). To check whether *Bad* is reachable, we check the intersection $\widehat{B} \cap C_{init}$. Since the intersection is empty, we conclude that *Bad* is not reachable, and hence the safety property is satisfied by the system.

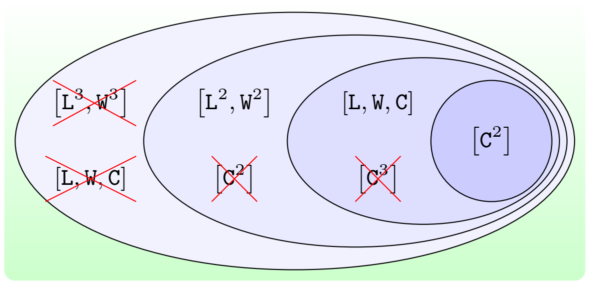
$\cancel{[\mathtt{L}^3, \mathtt{W}^3]}$
$\cancel{[\mathtt{L}, \mathtt{W}, \mathtt{C}]}$
$[\mathtt{L}^2, \mathtt{W}^2]$
$\cancel{[\mathtt{C}^2]}$
$[\mathtt{L}, \mathtt{W}, \mathtt{C}]$
$\cancel{[\mathtt{C}^3]}$
$[\mathtt{C}^2]$

Fig. 4. Running the backward reachability algorithm on the example Petri net. Each ellipse contains the configurations generated during one iteration. The subsumed configurations are crossed over.

3.8 Sufficient Conditions

We summarize the properties needed in order to derive the above algorithm:

1. *Monotonicity*. This implies that the predecessor set of an upward closed set of configurations is upward closed.
2. $\preceq$ is a wqo. We need this property for two reasons: to represent upward closed sets by a finite set of configurations (a generator of the set); and to guarantee termination of the algorithm.
3. For each c, we can compute the (finite) set $gen\left(\{c' \mid c' \longrightarrow \widehat{c}\}\right)$. In fact, $Pre(t)(c) = (c \ominus Out\,(t)) + In\,(t)$, where $\ominus$ rounds negative values up to 0 (i.e., $y \ominus x = 0$ if $x > y$ and $y \ominus x = y - x$ otherwise).
4. For each c, we can check whether there is a $c' \in C_{init}$ such that $c \preceq c'$. This is needed to check the emptiness of the intersection $\widehat{B} \cap C_{init}$.

4 Monotonic Abstraction

We consider parameterized systems, where the local transitions of a processes may be constrained by global conditions, i.e., the process may have to check the states of all the other processes before proceeding with the transition. To capture such conditions we need a more powerful model than standard Petri nets, namely Petri nets with *inhibitor arcs*. We introduce the concept of *monotonic abstraction* and describe how it transforms inhibitor arcs into *reset* arcs.

We consider a parametrized version of a simple reader-writer protocol. The system consists of an arbitrary number of processes which may read from or write to a global variable, and are supposed to satisfy the *reader-writer* property, i.e., writing should be exclusive to one process (at any point of time, if a process is writing, then no other process should be reading or writing). Notice that several different processes may be reading at the same time.

A process (depicted in Figure 5) has three local states, namely `I` where the process is idle, `R` where the process is reading, and `W` where the process is writing. Writing to the global variable is controlled by a lock whose value is equal to 1 when the lock is free and 0 otherwise. When a process wants to start reading, it checks whether the lock is free. If this is the case, the process moves from `I` to `R` without changing the value of the lock. From `R`, the process eventually moves back to `I`.

When a process wants to start writing, it checks whether there are other processes reading the global variable (encoded by the condition $\#\mathtt{R} = 0?$). It acquires the lock by decreasing the value of `L` by one. If the lock is not free, then $\mathtt{L} = 0$ and the transition is blocked. From `W`, the process eventually moves back to `I` releasing the lock (by increasing the value of `L` by one).

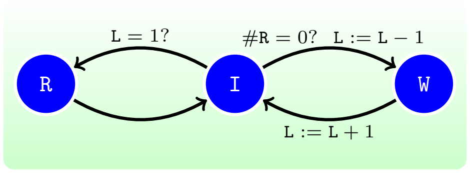

Fig. 5. One process in the reader-writer protocol.

4.1 Petri Nets with Inhibitor Arcs

A Petri net with *inhibitor arcs* is a generalization of Petri nets in the sense that an arc form a place p to a transition t may be declared to be an *inhibitor*. In such a case, the transition t may only be fired from configurations in which p is empty (does not contain any tokens). For the arcs which are not inhibitors the standard rules for firing transitions in Petri nets hold.

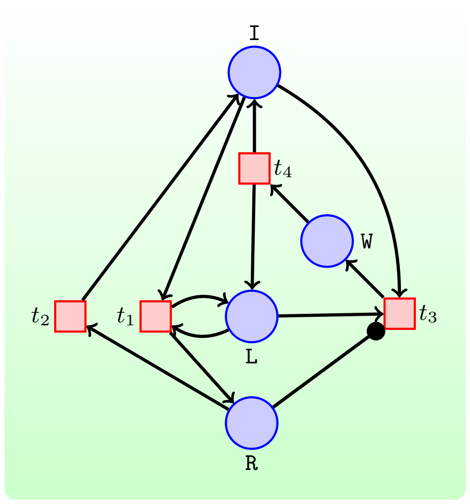

Fig. 6. A Petri net with one inhibitor arc.

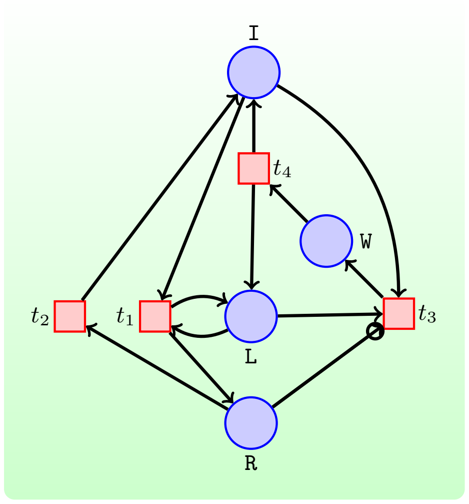

Fig. 7. A Petri net with one reset arc.

Figure 6 shows an example of a Petri net with one inhibitor arc (represented by the arrow whose head is a filled circle) between `R` and t_3.

4.2 Counter Abstraction

In a similar manner to Section 3, we can capture the behaviour of the parameterized reader-writer protocol as a Petri net with inhibitor arcs (Figure 6). The numbers of tokens in places `I`, `R`, and `W` represent the number of processes in their idle, read, and write states respectively. Absence of tokens in `L` means that there is currently a process writing to the global variable. The transitions of the Petri net are interpreted as follows. The transition t_1 represents a process moving from `I` to `R`. The process checks the state of the lock but does not change its value (this is represented by the two arcs between the place `L` and t_1). The transition t_2 corresponds to a process moving back from `R` to `I`. The transition t_3 means that an idle process becomes a writer. Here, we need an inhibitor arc to encode the condition that there are no processes currently reading the variable. This is done by checking that place `R` is empty. Finally, transition t_4 represents a process leaving the `W` state and becoming idle again.

4.3 Forcing Monotonicity

A Petri net with inhibitor arcs is not monotonic. For instance, consider the configurations $c_1 = [\mathtt{L}, \mathtt{I}]$, $c_2 = [\mathtt{W}]$, and $c_3 = [\mathtt{L}, \mathtt{I}, \mathtt{R}]$. Then, we have $c_1 \leq c_3$ and $c_1 \longrightarrow c_2$ (by firing the transition t_3), but there is no c_4 such that $c_3 \longrightarrow c_4$ and $c_2 \leq c_4$. The inhibitor arc does not allow taking t_3 from c_3 since the place `R` is not empty. In fact, the only transitions enabled from c_3 are t_1 and t_2 leading to the configurations $\left[\mathtt{L}, \mathtt{R}^2\right]$ resp. $\left[\mathtt{L}, \mathtt{I}^2\right]$ (none of which is larger than $[\mathtt{W}]$).

That Petri nets with inhibitor arcs are not monotonic is not surprising given that they are Turing-powerful. We revert therefore to abstraction, where we compute an over-approximation which is monotonic. The only transitions which violate monotonicity are those with inhibitor arcs. In our abstraction, we change the semantics of the Petri net, by replacing inhibitor arcs with *reset arcs*. A reset arc does not disable the transition. Instead, the reset arc removes all the tokens from the input place thus making it empty. One important property of reset nets is that they are monotonic. Thus we generate an abstraction which is not exact (as in Section 3) but which nevertheless is monotonic.

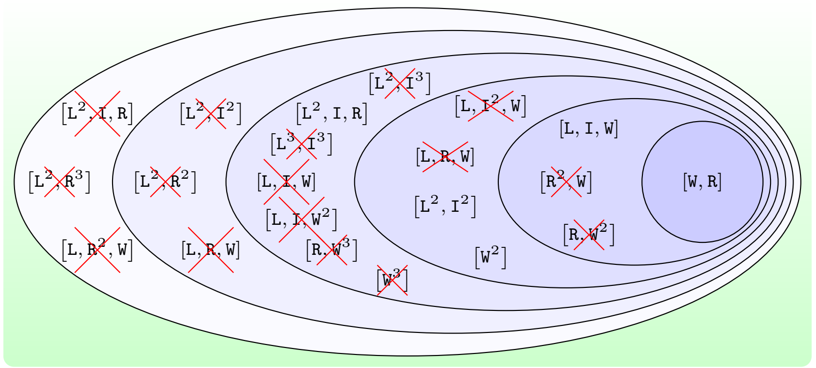

Fig. 8. Running the backward reachability algorithm on the example Petri net with reset arcs in Figure 7.

The existence of tokens in $\mathtt{R}$ means that there are processes in the configuration which violate an enabling condition thus blocking the transition t_3. In other words, the existence of readers prevents a process moving from its idle to its writing state. Our abstraction means that we "kill" all the processes violating the condition, thus enabling the transition again. Since the abstract transition relation is an over-approximation of the original transition relation, it follows that if a safety property holds in the abstract model, then it will also hold in the concrete model.

Figure 7 show the Petri net with reset arcs we get as an abstraction of the Petri net with inhibitor arcs in Figure 6. The inhibitor arc is replaced by a reset arc (with a head which is an empty circle). Figure 8 shows the result of running the backward reachability algorithm on the reset Petri net of Fig 7.

5 Linear Topologies

In many cases, the components of a parameterized systems are organized as a linear array. Configurations of the system can then be represented by *words* over a finite alphabet rather than multisets. Each alphabet symbol inside the word represents the local state of one process. The ordering on the symbols reflects the ordering on the processes. As mentioned earlier, one important (and difficult) feature in the behaviour of parameterized systems is the existence of *global condition* in which a process may have to check the states of the other processes inside the systems before performing a transition. A global transition is either *universally* or *existentially* quantified. An example of a universal condition is that *all* processes in the left context[2] 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. We have already seen an example of a global condition in the reader-writer protocol of Section 4. A process in the protocol changes states from `I` to `W` only if the number of reader processes is equal to zero. This is equivalent to the universal condition that all the other processes should be either idle or writing.

5.1 Simple Example

We introduce our method through a simple example of a protocol which implements mutual exclusion among an arbitrary number of processes. Each process (depicted in Figure 9) has four local states, namely the idle (`I`), requesting (`R`), waiting (`W`), and critical (`C`) states.

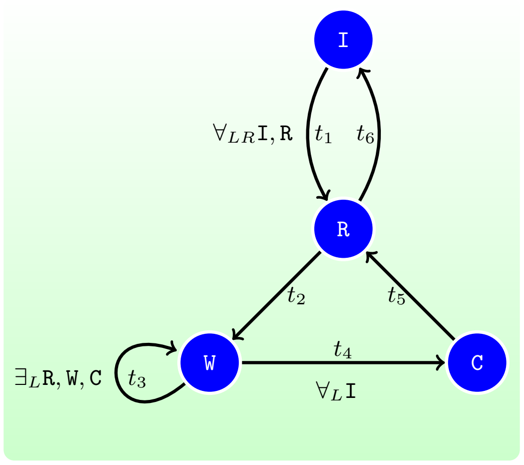

Fig. 9. One process in the mutual exclusion protocol with linear topology.

[2] The left context of the active process contains all the process which are to its left inside the configuration.

Initially, all the processes are idle (in state `I`). When a process becomes interested in accessing the critical section (which corresponds to the state `C`), it declares its interest by moving to the requesting state `R`.

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 idle or requesting 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 `I` or `R`. In the requesting state, the process may move to the waiting state `W` 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 may cross from the initial (idle) state to the requesting state. However, once the first process has crossed to the waiting state, it "closes the door" on the processes which are still in their initial states. These processes will no longer be able to leave their initial states until the door is opened again (when no process is in `W` or `C`). From the set of processes which have declared interest in accessing the critical section (those which have left their idle states and are now in the requesting or waiting states) the leftmost process has the highest priority. This is encoded by the global universal transition t_4 where a process may move from its waiting state to its critical section only subject to the universal condition that all processes in its left context are idle (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 requesting, in their waiting states, or in their critical sections, then it loops back to the waiting state through the existential transition t_3. Once the process leaves the critical section, it will return back to the requesting state through the local transition t_5. In the requesting state, the process chooses either to try to reach the critical section again, or to become idle (through the local transition t_6).

5.2 Abstraction

Since the ordering among the processes in the system is relevant, we can no longer use multisets to describe the configurations of the system. This means that we have to go beyond Petri nets in order to produce an abstraction of the system. As mentioned above, a configuration will now be represented as a word over a finite alphabet representing the local states of the processes. In our example this alphabet is given by the set $\{\mathtt{I}, \mathtt{R}, \mathtt{W}, \mathtt{C}\}$. For instance the configuration `IWCWR` represents a configuration in an instance of the system with five processes that are in their idle, waiting, critical, waiting, and requesting states in that order. The definition of the transition relation $\longrightarrow$ depends on the type of t (whether it is local, existential, or universal). We will consider three transition rules from Figure 9 to illustrate the idea. The local rule t_2 induces transitions of the form `WIRCR` $\longrightarrow$ `WIWCR`. Here the active process changes its local state from requesting to waiting. The existential rule t_3 induces transitions of the form `RIWCR` $\longrightarrow$ `RIWCR`. The waiting process can perform the transition since there is a requesting process in its left context. However, the same transition is not enabled from the configuration `IIWCR`, since there are no critical, waiting, or requesting processes in the left context of the process trying to perform the transition. The universal rule t_4 induces transitions of the form IIWWR $\longrightarrow$ IICWR. The active process (in the waiting state) can perform the transition since all processes in its left context are idle. On the other hand, neither of the waiting processes can perform the transition form the configuration CIWWR since, for each one of them, there is at least one process in its left context which is not idle.

An *initial configuration* is one in which all processes are in their initial states. Examples of initial configurations are II and IIIII, corresponding to instances of the system with two and five processes respectively. As mentioned above, the protocol is intended to guarantee mutual exclusion. In other words, we are interested in verifying a *safety property*. To do this we characterize the set of bad configurations: all configurations which contain at least two processes in their critical sections. Examples of bad configurations are CRC and ICRCWC. Showing the safety property amounts to proving that the protocol, starting from an initial configuration, will never reach a bad configuration.

5.3 Monotonic Abstraction

We define an ordering on configurations where $c_1 \preceq c_2$ if c_1 is a (not necessarily contiguous) subword of c_2. For instance, WC $\preceq$ RWICW. The relation $\preceq$ is a wqo by Higman's lemma [12]. In a similar manner to Section 3 and Section 4, 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 rule t_2 and the induced transition $c_1 =$ IRC $\longrightarrow$ IWC $= c_2$ in which a process changes state from requesting to waiting. Consider the configuration $c_3 =$ IWIRCR that is larger than c_1. Clearly, c_3 can perform the local transition $c_3 =$ IWIRCR $\longrightarrow$ IWIWCR $= c_4$ leading to $c_4 \succeq c_2$. Local transitions are monotonic, since the active process in the small configuration (the requesting process in c_1) also exists in the larger configuration (i.e., c_3). A local transition does not check or change the states of the passive processes; and hence the larger configuration c_3 is also able to perform the transition, while maintaining the ordering $c_2 \preceq c_4$.

Consider the local rule t_3 and the induced transition $c_1 =$ RIWCR $\longrightarrow$ RIWCR $= c_2$. Let us observe that the configuration c_1 can be divided into three parts: the active process in the waiting state, the left context RI, and the right context CR. Furthermore, the left context contains a witness (the process in the requesting state) which enables the transition. Consider the configuration $c_3 =$ IRIWCRC that is larger than c_1. Also, the configuration c_3 can be divided into three parts: the active process in the waiting state, the left context IRI, and the right context CRC. Notice that the left context of c_3 is larger than the left context of c_1, and hence the former will also contain the witness. This means that c_3 can perform the same transition $c_3 =$ IRIWCRC $\longrightarrow$ IRIWCRC $= c_4$ leading to $c_4 \succeq c_2$.

Next, we motivate why universal transitions are not monotonic. Consider the universal rule t_4 and the induced transition $c_1 = \texttt{IIWWR} \longrightarrow \texttt{IICWR} = c_2$. The transition is enabled since all processes in the left context of the active process satisfy the condition of the transition (they are idle). Consider the configuration $c_3 = \texttt{IRICWWR}$. Although $c_1 \preceq c_3$, the universal transition t_4 is not enabled from c_3 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 change the semantics of the system using the same idea as the one in Section 4. More precisely, we delete all the processes violating the condition of the universal rule. This means for instance that we have a transition of the form $\texttt{IRICWWR} \longrightarrow \texttt{IICWR}$ since we can first delete the two processes in the requesting and critical states and then perform the transition.

5.4 Computing predecessors

For a configuration c and a transition rule t, we define $\mathit{Pre}(t)(c)$ to be the set $\{c_1, \ldots, c_n\}$ which is the generator of the set of configurations from which we can reach $\widehat{c}$ through one application of t. We will consider different transition rules in Figure 9 to illustrate how to compute *Pre*. For the local rule t_5 in Figure 9, we have $\mathit{Pre}(t_5)(\texttt{IRW}) = \{\texttt{ICW}\}$. In other words, the predecessor set is characterized by one configuration, namely ICW. Strictly speaking, the set contains also a number of other configurations such as IRCW. However such configurations are subsumed by the original configuration IRW, and therefore we will for simplicity 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 9. We have $\mathit{Pre}(t_3)(\texttt{RWC}) = \{\texttt{RWC}\}$. In this case, there is a witness (a requesting process) in the left context of the active process. On the other hand, we have $\mathit{Pre}(t_3)(\texttt{IWC}) = \{\texttt{RIWC}, \texttt{IRWC}, \texttt{WIWC}, \texttt{IWWC}, \texttt{CIWC}, \texttt{ICWC}\}$. 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 (requesting, waiting, or critical), at each possible place in the left context of the active process.

Notice that the sizes of the new configurations (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 due 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 9. Then $\mathit{Pre}(\texttt{IRICW}) = \emptyset$ since there is a requesting process in the left context of the potential active process (which is in the critical section). On the other hand, $\mathit{Pre}(\texttt{IICW}) = \texttt{IIWW}$ since all processes in the left context of the active process are in their idle states.

5.5 Backward Reachability Algorithm

We show how the backward reachability algorithm runs on our example (Figure 10). We start by the generator of the set of bad configurations, namely $\{\texttt{CC}\}$. The only transition which is enabled backwards from a critical state, is the one induced by the rule t_4. From the two processes in $\texttt{CC}$ 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): $Pre(t_4)(\texttt{CC}) = \{\texttt{WC}\}$. From $\texttt{WC}$, two rules are enabled backwards (both from the waiting process): the local rule t_2: $Pre(t_2)(\texttt{WC}) = \{\texttt{RC}\}$; and the existential rule t_3: $Pre(t_3)(\texttt{WC}) = \{\texttt{RWC}, \texttt{WWC}, \texttt{CWC}\}$. All the three configurations in $Pre(t_3)\,(\texttt{WC})$ are subsumed by $\texttt{WC}$. One rule is enabled backwards from $\texttt{RC}$, namely the local rule t_5 from the requesting process: $Pre_{t_5}\,(\texttt{RC}) = \{\texttt{CC}\}$. Notice that the universal transition t_1 is not enabled from the requesting process, since there is another process (the critical 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 configurations which are not subsumed by the existing ones.

Since there is no initial configuration (with only idle processes) in $\widehat{\texttt{CC}} \cup \widehat{\texttt{WC}} \cup \widehat{\texttt{RC}}$, 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.

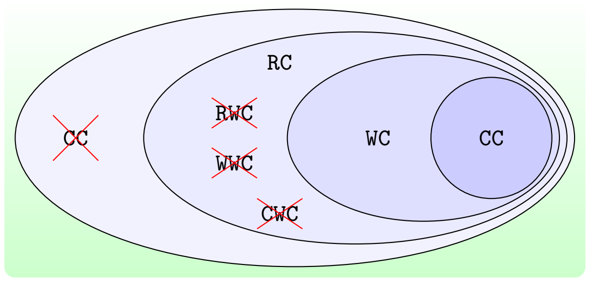

Fig. 10. Running the backward reachability algorithm on the example Protocol.

References

1. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96,* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.
2. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

3. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *Proc.* 19th *Int. Conf. on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 145–157, 2007.
4. P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *Proc. TACAS '07,* 13th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 721–736. Springer Verlag, 2007.
5. P. A. Abdulla, N. B. Henda, G. Delzanno, and A. 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. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *Proc. LICS '93,* 8th *IEEE Int. Symp. on Logic in Computer Science*, pages 160–170, 1993.
7. P. A. Abdulla and B. Jonsson. Model checking of systems with many identical timed processes. *Theoretical Computer Science*, 290(1):241–264, 2003.
8. L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. J. Math.*, 35:413–422, 1913.
9. E. Emerson and K. 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.
10. 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.
11. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
12. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7):326–336, 1952.
13. N. Yonesaki and T. Katayama. Functional specification of synchronized processes based on modal logic. In *IEEE 6th International Conference on Software Engineering*, pages 208–217, 1982.