Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Lightweight Regular Model Checking Approach for Parameterized Systems

Giorgio Delzanno[1] `giorgio@disi.unige.it`,
Ahmed Rezine[2] `rahmed@it.uu.se`

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

The date of receipt and acceptance will be inserted by the editor

Abstract. In recent years we have designed a lightweight approach to regular model checking specifically designed for parameterized systems with global conditions. Our approach combines the strength of regular languages, used for representing infinite sets of configurations, with symbolic model checking and approximations. In this paper we give a uniform presentation of several variations of a symbolic backward reachability scheme in which different classes of regular expressions are used in place of BDDs. The classification of the proposed methods is based on the precision of the resulting approximated analysis.

1 Introduction

In this paper we consider verification of safety properties for parameterized systems with universal and existential global conditions. Typically, a parameterized system consists of an arbitrary number of identical processes, each with a finite number of control states, organized in a linear array. Configurations of the parameterized system can be seen as words with the set of control states as alphabet. Global conditions are used here as guards for state transitions of an individual process. An example of a universally quantified global condition is that all processes to the left of a given process i should satisfy a property φ. Process i can perform the transition only if all processes with indices $j < i$ satisfy φ. In an existential condition we require that *some* (rather than *all*) processes satisfy φ. The task is to verify correctness regardless of the number of processes. Parameterized linear systems with universal global conditions are Turing equivalent. All interesting verification problems are undecidable for this class of infinite-state systems.

In this paper we describe a lightweight regular model checking approach that can be used to provide either semi-decision procedures or approximated algorithms. The method is the result of a joint effort of research groups in Genova and Uppsala [4–8]. One of the initial goals of the research project was to avoid the use of heavy machinery like transducers in regular model checking for infinite-state systems. In our approach we use regular languages in in a more classical symbolic model checking approach. More precisely, we define symbolic least fixpoint computations in which regular languages are used in place of BDDs. Regular languages are used here to encode infinite sets of configurations that are upward closed with respect to a given pre-order. The symbolic operations needed to define this kind of approach are classical set operations like union, intersection, inclusion, and membership and an operation to compute predecessors of configurations represented by a regular language. For all these operations, we provide specialized algorithms adapted to the class of regular expressions used in the analysis. To attack the problem of termination of the resulting procedures, it comes natural to evaluate the application of the theory of well-structured transition systems introduced in [2]. In this context the use of well-quasi orderings on the set of configurations, lifted then to symbolic representations called constraints, is central to obtain sufficient conditions for the termination of the analysis. Different classes of regular languages can be adapted to work in the generic backward scheme. In this paper we discuss pros and cons of some of them, supporting the different definitions by means of an experimental evaluation obtained via prototype implementations of libraries of symbolic operations for each one of the considered classes.

More in detail, the paper is organized as follows. In Section 2 we first introduce a model for linearly ordered parameterized systems in which transitions are guarded by universal and existential global conditions. The expressive power of the model is equivalent to that of Turing machines. We add in the appendix a detailed description of several examples specified using our modeling language. These examples include the Illinois and the DEC Firefly cache coherence protocols from [18], the Bakery and Burns mutual exclusion algorithms used in [4]; a model of Szymanski's algorithm with atomicity conditions from [10,31], refinements of Szymanski's algorithm taken from [28] and [22]. In the same section we recall the main requirement to apply the generic backward scheme for infinite-state systems based on the theory of well-quasi orderings taken from [2].

In Sect. 3 we describe a class of regular expressions that can be used for exact backward analysis of our parameterized models. The resulting symbolic model checking approach is necessarily a semi-decision procedure. Despite this fact, we present examples of parameterized systems taken from the literature for which the exact analysis terminates and verifies non trivial correctness properties. Unfortunately, for several classical problems, e.g., models of mutual exclusion protocols, an exact analysis diverges.

In Sect. 4 we introduce a class of regular expressions that can be used for an approximated analysis of our parameterized models. The class of regular expressions considered here denotes upward closed sets of configurations ordered via the subword relation. The use of this ordering, known to be a well-quasi ordering, allows us to ensure the termination of the resulting symbolic model checking algorithms, thus avoiding, at least in theory, the divergence problems of an exact analysis. The type of approximation applied here corresponds to the monotonic abstraction introduced in [4]. This abstraction approximates the set of predecessors of a given upward closed set of configurations by using the smallest upward closed set containing it. Monotonic abstractions work well in most of the parameterized models taken from the literature. However there is a special class of parameterized systems, including a model of Szymanski's algorithm with non-atomic updates of variables, for which monotonic abstraction returns false positives. This model is presented in a seminal paper of Manna and Pnueli [28] on semi-automated verification of parameterized systems.[1]

Finally, in Sect. 5 we introduce regular expressions, called *context-sensitive constraints*, that can be used for refining the precision of monotonic abstraction. The theory of well-quasi ordering can still be used for ensuring the termination of the resulting symbolic model checking algorithm. Notably the resulting approximated algorithms can be applied to verify all our benchmark examples of (ordered and unordered) parameterized systems taken from the literature.

[1] We would like to take the opportunity to remember once again Prof. Amir Pnueli. Prof. Pnueli pointed out to us the paper [28] after a talk at VMCAI '08 in San Francisco. Furthermore, he made it available on his personal web page (the paper dates back to 1990). Needless to say that reading his paper enlightened our research with new challenging problems.

2 Preliminaries

2.1 Parameterized Systems with Global Conditions

For a set A, we use A^* to denote the set of finite words over A, and use w_1w_2 to denote the concatenation of two words w_1 and w_2 in A. For a natural number n, we use $\overline{n}$ to denote the set $\{1, \ldots, n\}$.

Formally, a *parameterized system* is a pair $\mathcal{P} = (Q, T)$, where Q is a finite set of *local states*, and T is a finite set of *transitions*. A transition is either *local* or *global*. A local transition is of the form $q \rightarrow q'$, where the process changes a local state from q to q' independently of the local states of the other processes. A global transition is of the form $q \rightarrow q' : \mathbb{Q}P$, where $\mathbb{Q} \in \{\exists_L, \exists_R, \exists_{LR}, \forall_L, \forall_R, \forall_{LR}\}$ and $P \subseteq Q$. Here, the process checks also the local states of the other processes when it makes the move. For instance, the condition $\forall_L P$ means that "all processes to the left should be in local states which belong to the set P"; the condition $\forall_{LR} P$ means that "all other processes (whether to the left or to the right) should be in local states which belong to the set P"; and so on. Sometimes we write $\exists$ and $\forall$ instead of $\exists_{LR}$ and $\forall_{LR}$ respectively.

Given Q and T, a parameterized system $\mathcal{P} = (Q, T)$ induces an infinite-state transition system $(C, \longrightarrow)$ where $C = Q^*$ is the set of *configurations* and $\longrightarrow$ is a transition relation on C. For a configuration $c = q_1q_2 \cdots q_n$, we define $c^\bullet$ as the set $\{q_1, \ldots, q_n\}$. For configurations $c = c_1qc_2$, $c' = c_1q'c_2$, and a transition $t \in T$, we write $c \xrightarrow{t} c'$ to denote that one of the following conditions is satisfied:

- t is a local transition of the form $q \rightarrow q'$.
- t is a global transition of the form $q \rightarrow q' : \mathbb{Q}P$, and one of the following conditions is satisfied:
  - either $\mathbb{Q}P = \exists_L P$ and ${c_1}^\bullet \cap P \neq \emptyset$, or $\mathbb{Q}P = \exists_R P$ and ${c_2}^\bullet \cap P \neq \emptyset$, or $\mathbb{Q}P = \exists_{LR} P$ and $({c_1}^\bullet \cup {c_2}^\bullet) \cap P \neq \emptyset$.
  - or $\mathbb{Q}P = \forall_L P$ and ${c_1}^\bullet \subseteq P$, or $\mathbb{Q}P = \forall_R P$ and ${c_2}^\bullet \subseteq P$, or $\mathbb{Q}P = \forall_{LR} P$ and $({c_1}^\bullet \cup {c_2}^\bullet) \subseteq P$.

We use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$.

We define an ordering $\preceq$ on configurations as follows. Let $c = q_1 \cdots q_m$ and $c' = q'_1 \cdots q'_n$ be configurations. Then, $c \preceq c'$ if c is a subword of c', i.e., there is a strictly increasing injection h from $\overline{m}$ to $\overline{n}$ such that $q_i = q'_{h(i)}$ for all $i : 1 \leq i \leq n$.

Given a parameterized system, we assume that, prior to starting the execution of the system, each process is in an (identical) *initial* state q_{init}. We use *Init* to denote the set of *initial* configurations, i.e., configurations of the form $q_{init} \cdots q_{init}$ (all processes are in their initial states). Notice that the set *Init* is infinite.

A set of configurations $U \subseteq C$ is *upward closed* with respect to $\preceq$ if $c \in U$ and $c \preceq c'$ imply $c' \in U$. For a configuration c, we use $\widehat{c}$ to denote the upward closure of c, i.e., the set $\{c' \mid c \preceq c'\}$. We also use $\widehat{D}$ to denote the upward closure of the set of configurations D, i.e., the set $\{c' \mid \exists c \in D \text{ s.t. } c \preceq c'\}$. For sets of configurations $D, D' \subseteq C$ we use $D \longrightarrow D'$ (respectively $D \xrightarrow{*} D'$) to denote that there are $c \in D$ and $c' \in D'$ with $c \longrightarrow c'$ (respectively $c \xrightarrow{*} c'$). The *coverability problem* for parameterized systems is defined as follows:

PAR-COV
Instance
– A parameterized system $\mathcal{P} = (Q, T)$.
– A finite set C_F of configurations.
Question $Init \xrightarrow{*} \widehat{C_F}$?

It can be shown, using standard techniques (see e.g. [32]), that checking safety properties (expressed as regular languages) can be translated into instances of the coverability problem. Typically, $\widehat{C_F}$ (which is an infinite set) is used to characterize sets of *bad* configurations which we do not want to occur during the execution of the system. In such a case, the system is safe iff $\widehat{C_F}$ is not reachable. Therefore, checking safety properties amounts to solving PAR-COV (i.e., to the reachability of upward closed sets). We give an example in the next section.

2.2 A Mutual Exclusion Protocol

Consider a parameterized system in which processes have the state diagram represented in Fig. 1. Each process has five local states $q_0, \ldots, q_4$. All the processes are initially in state q_0. A process in the critical section is represented by state q_4. The set of configurations violating mutual exclusion contains exactly configurations with at least two occurrences of symbol q_4. Processes start crossing from q_0 to q_1, and then to state q_2. Once the first process has crossed to state q_2 it "closes the door" on the processes which are still in q_0. These processes will no longer be able to leave q_0 until the door is opened again (when no process is in state q_2 or q_3). Furthermore, a process is allowed to cross from q_3 to state q_4 only if there is at least one process still in state q_2 (i.e., the door is still closed on the processes in state q_0). This is to prevent a process first reaching q_4 and then a process to its left starting to move from q_0 all the way to state q_4 (thus violating mutual exclusion). From the set of processes which have left state q_0 (and which are now in state q_1 or q_2) the leftmost process has the highest priority. This is encoded by the global condition that a process may move from q_2 to q_3 only subject to the global condition that all processes to its left are in state q_0 (this condition is encoded by the universal quantifier $\forall_L$, where "L" stands for "Left"). A typical run of the system is of the form $q_0q_0q_0q_0 \longrightarrow q_0q_1q_0q_0 \longrightarrow q_0q_1q_1q_0 \longrightarrow q_0q_2q_1q_0 \longrightarrow q_0q_2q_2q_0 \longrightarrow q_0q_3q_2q_0 \longrightarrow q_0q_4q_2q_0 \longrightarrow q_0q_0q_2q_0$. The protocol satisfies mutual exclusion.

2.3 Extensions and other case studies

We now extend our model with synchronization mechanisms like broadcast communication and read and write operations on globally shared variables ranging in a finite domain or in the natural numbers. Operations on the latter type can be obtained by synchronization using processes to encode the value of the shared variable ranging over natural numbers. For this, we assume that an arbitrary number of processes moves from state q_{init} to state *zero*. Thereafter, these processes can move between the two states *zero* and *one*. Increment is modeled via synchronization with a *zero* process that moves to *one*, decrement via synchronization with a *one* process that moves to *zero*, and zero test is modeled via a global condition "*there are no processes with state one*". The current value of the shared variable is the number of occurrence of processes in state *one*. Thus, this kind of variables may range over an unbounded set of natural numbers.

Other examples, sometimes using these extensions, are described in the rest of the paper and in the appendix. More specifically, the examples we consider are: the Illinois and the DEC Firefly cache coherence protocols from [18]; the Bakery and Burns' mutual exclusion algorithms used in [4]; a compact model of Szymanski's algorithm with atomicity conditions from [10, 31], a refinement of Szymanski's algorithm from [28] (see Fig. 2), and Gribomont-Zenner's mutex algorithm from [22]. Several synchronization and reference counting examples using unbounded integer counters are also considered. These include an abstract model of the reference counting example for page allocation in [19], and solutions to the readers and writers problem from [33] with priorities to readers or writers. We remark that in all examples global conditions are evaluated atomically.

2.4 A Generic Backward Scheme for Symbolic Model Checking

In this section, we recall a generic scheme from [2] for performing symbolic backward reachability analysis based on the notion of constraint as a way to symbolically represent sets of configurations. In the subsequent sections we will instantiate the general scheme by using special types of regular expressions as constraints.

Assume a transition system $(D, \longrightarrow)$ with a set *Init* of initial states. We will work with a set of constraints defined over D. A *constraint* ϕ denotes a potentially infinite set $[\![\phi]\!]$ of configurations (i.e. $[\![\phi]\!] \subseteq D$). For a finite set Φ of constraints, we let $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$.

We define an *entailment relation* $\sqsubseteq$ on constraints, where $\phi_1 \sqsubseteq \phi_2$ iff $[\![\phi_2]\!] \subseteq [\![\phi_1]\!]$. For sets Φ_1, Φ_2 of constraints, abusing notation, we let $\Phi_1 \sqsubseteq \Phi_2$ denote that for each $\phi_2 \in \Phi_2$ there is a $\phi_1 \in \Phi_1$ with $\phi_1 \sqsubseteq \phi_2$. Notice that $\Phi_1 \sqsubseteq \Phi_2$ implies that $[\![\Phi_2]\!] \subseteq [\![\Phi_1]\!]$ (although the converse is not true in general).

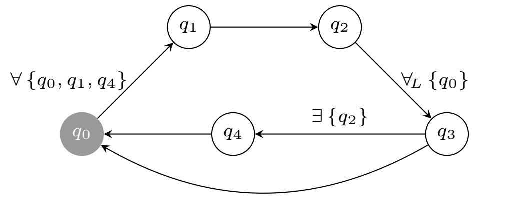

Fig. 1. State diagram of an individual process with initial state q_0

For a constraint ϕ, we let $Pre(\phi)$ be a finite set of constraints, such that $[\![Pre(\phi)]\!] = \{c \mid \exists c' \in [\![\phi]\!] \,.\, c \longrightarrow c'\}$. In other words $Pre(\phi)$ characterizes the set of configurations from which we can reach a configuration in $[\![\phi]\!]$ through the application of a single transition rule. For our class of systems, we will show that such a set always exists and is in fact computable. For a set Φ of constraints, we let $Pre(\Phi) = \bigcup_{\phi \in \Phi} Pre(\phi)$. Below we present a scheme for a symbolic algorithm which, given a finite set Φ_F of constraints, checks whether $Init \xrightarrow{*} [\![\Phi_F]\!]$.

In the scheme, we perform a backward reachability analysis, generating a sequence $\Phi_0 \subseteq \Phi_1 \subseteq \cdots$ of finite sets of constraints such that $\Phi_0 = \Phi_F$, and $\Phi_{j+1} = \Phi_j \cup Pre(\Phi_j)$. Since $[\![\Phi_0]\!] \subseteq [\![\Phi_1]\!] \subseteq \cdots$, the procedure terminates if we reach a point j where $\Phi_j \sqsubseteq \Phi_{j+1}$. Notice that the termination condition implies that $[\![\Phi_j]\!] = (\bigcup_{0 \leq i \leq j} [\![\Phi_i]\!])$. Consequently, Φ_j characterizes the set of all predecessors of $[\![\Phi_F]\!]$. This means that $Init \xrightarrow{*} [\![\Phi_F]\!]$ iff $(Init \bigcap [\![\Phi_j]\!]) \neq \emptyset$.

Observe that, in order to implement the scheme (i.e., transform it into an algorithm), we need to be able to (i) compute Pre; (ii) check for entailment between constraints; and (iii) check for emptiness of $Init \bigcap [\![\phi]\!]$ for a given constraint ϕ. A constraint system satisfying these three conditions is said to be *effective*. Moreover, in [2], it is shown that termination is guaranteed in case the constraint system is *well quasi-ordered (wqo)* with respect to $\sqsubseteq$, i.e., for each infinite sequence $\phi_0, \phi_1, \phi_2, \ldots$ of constraints, there are $i < j$ with $\phi_i \sqsubseteq \phi_j$.

3 Exact Analysis

Assume a parameterized system $\mathcal{P} = (Q, T)$, where Q is a finite set of states. In order to finitely represent infinite sets of system configurations (e.g. configurations of arbitrary size) we use the *context-sensitive constraints* defined in this section. We work with words in $\mathbb{A}^*$, where $\mathbb{A} = Q \cup \mathbb{P}(Q)$, and $\mathbb{P}(Q)$ denotes the powerset of Q. We use $p, q, \ldots$ to denote states in Q, and $P, R, \ldots$ to denote sets of states in $\mathbb{P}(Q)$. Furthermore, for $w \in \mathbb{A}^*$ we use $w^\bullet$ to denote the union of all states in Q occurring in w either as one of its letters or listed in one of its sets. As an example, for $R = \{q_1, q_2\}$ we have that $(Rq_3R)^\bullet = \{q_1, q_2, q_3\}$.

A *c*ontext-sensitive (CC-)constraint is a word in $\mathbb{A}^*$ of the form

$$R_0 q_1 R_1 \ldots q_n R_n$$

where $q_i \in Q$ for $i : 1 \leq i \leq n$ and $R_i \subseteq Q$ for $i : 0 \leq i \leq n$. The configuration $q_1 \ldots q_n$ is called the *basis* and each of the sets R_i is called a *context*. The denotation of a context-sensitive constraint ϕ, written $[\![\phi]\!]$, is defined as the set of configurations of the form $c_0 q_1 c_1 \ldots q_n c_n$ where $c_i \in R_i^*$ for $i : 0 \leq i \leq n$.

The denotation of a CC $R_0 q_1 R_1 \ldots q_n R_n$ can be defined via the regular expression

$$C_0 \cdot q_1 \cdot C_1 \ldots q_n \cdot C_n$$

where $C_i = (q_{i1} + \ldots + q_{in_i})^*$ for $R_i = \{q_{i1}, \ldots, q_{in_i}\}$. Here we use $\cdot$ and $+$ to denote concatenation and union of sets of strings, and A^* to denote strings obtained by any number of repetitions of the strings in A. The expression C_i represents constraints on the context in between pairs of explicit process states q_{i-1} and q_i for $i : 1 < i \leq n$, and to the left (respectively right) of q_1 (respectively q_n).

As an example, assume $Q = \{q_1, q_2, q_3\}$, $R_0 = R_1 = \{q_2, q_3\}$ and $R_2 = \{q_1, q_3\}$. The constraint ϕ defined as $R_0 q_1 R_1 q_2 R_2$ denotes all configurations of the form $c_0 q_1 c_1 q_2 c_2$ such that sub-configurations c_0 and c_1 cannot contain processes of type q_1 and sub-configuration c_2 cannot contain occurrences of processes of type q_2. Therefore, configurations $q_3 q_1 q_3 q_2 q_1$ and $q_3 q_1 q_3 q_3 q_2 q_1$ belong to $[\![\phi]\!]$, whereas $q_1 q_1 q_2$ and $q_1 q_3 q_2 q_2$ do not. Notice that CC's of the form $Qq_1Q \ldots q_nQ$ denote upward closed sets of states with respect to word inclusion (there are no constraints on the contexts). For instance, the set of bad configurations in Example 1 can be characterized by the CC-formula Qq_4Qq_4Q where $Q = \{q_0, q_1, q_2, q_3, q_4\}$.

3.1 Symbolic Operations

We now define the symbolic operations we use in our analysis, namely the entailment and the predecessor computation on context-sensitive constraints. These respectively correspond to the application, without any loss of precision, of the inclusion and the *pre* operation on the associated denotations (sets of configurations).

Entailment. For constraints $\phi = R_0 q_1 \ldots q_n R_n$ and $\phi' = R'_0 q'_1 \ldots q'_m R'_m$, we define $\phi \sqsubseteq \phi'$ iff there exists a monotonic injection $h : \overline{n} \to \overline{m}$ such that $q_i = q'_{h(i)}$ for $i : 1 \leq i \leq n$ (the basis of ϕ is a subword of the basis of ϕ') and the following conditions hold:

– $(R'_0 q'_1 \ldots q'_{h(1)-1} R'_{h(1)-1})^\bullet \subseteq R_0^\bullet$
– $(R'_{h(i)} q'_{h(i)+1} \ldots q'_{h(i+1)-1} R'_{h(i+1)-1})^\bullet \subseteq R_i^\bullet$ for $i : 1 \leq i \leq n-1$;
– $(R'_{h(n)} q'_{h(n)+1} \ldots q'_m R'_m)^\bullet \subseteq R_n^\bullet$.

We have that $\phi_1 \sqsubseteq \phi_2$ if and only if $[\![\phi_2]\!] \subseteq [\![\phi_1]\!]$ (ϕ_1 is weaker than ϕ_2).

Computing Predecessors. Given a set S of CC's, it is possible to define a *symbolic predecessor operator Pre* that effectively computes, when applied to S, a set $S' = Pre(S)$ of CC's such that $[\![S']\!] \longrightarrow [\![S]\!]$.

We first introduce the symbolic predecessor computation for a $\forall_L$-rule, and then describe the case of the other transitions. Consider a transition t of the form $q \to q' \ : \ \forall_L P$ with $P \subseteq Q$. Then, $Pre_t(\phi)$ is the set $\{\phi' \mid \phi \rightsquigarrow_t \phi'\}$ where $\rightsquigarrow_t$ is the minimal relation that satisfies the following conditions. Let $\phi = R_0 q_1 \ldots q_n R_n$:

1. if there exists i s.t. $q_i = q'$ with $q_j \in P$ for each $j : 1 \leq j < i$, then $\phi \rightsquigarrow_t (R_0 \cap P) q_1 \ldots q_{i-1} (R_{i-1} \cap P) q R_i q_{i+1} \ldots q_n R_n$
2. if there exists i s.t. $q' \in R_i$ with $q_j \in P$ for each $j : 1 \leq j \leq i$, then $\phi \rightsquigarrow_t (R_0 \cap P) q_1 \ldots q_i (R_i \cap P) q R_i q_{i+1} \ldots q_n R_n$

Notice that: in (1) the length of the new basis and the number of contexts are the same as in ϕ, whereas in the new constraint produced in (2) we add a new process as well as a new context.

The case of $\forall_R$-rules is similar to that for $\forall_L$-rules. The remaining cases are given below.

Forall Let $\phi = R_0 q_1 \ldots q_n R_n$. Consider a transition t of the form $q \to q' \ : \ \forall_{LR} P$ with $P \subseteq Q$. Then, $\rightsquigarrow_t$ is the minimal relation that satisfies the following conditions.

1. if there exists i s.t. $q_i = q'$ with $q_j \in P$ for each $j : (1 \leq j \neq i \leq n)$, then $\phi \rightsquigarrow_t (R_0 \cap P) q_1 \ldots q_{i-1} (R_{i-1} \cap P) q (R_i \cap P) q_{i+1} \ldots q_n (R_n \cap P)$.
2. if there exists i s.t. $q' \in R_i$ with $q_j \in P$ for each $j : 1 \leq j \leq n$, then $\phi \rightsquigarrow_t (R_0 \cap P) q_1 \ldots q_i (R_i \cap P) q (R_i \cap P) q_{i+1} \ldots q_n (R_n \cap P)$.

In the first case we apply a backward rewrite step to state q' occurring in the basis and restrict the corresponding left and right expressions in accord with the guard $(R_i \cap P)$. The rule cannot be applied if some of the other elements in the basis is not in P (it violates the guard). In the second case we first select the process state q' from context R_i and then proceed as in the first case. **Local** Let t be a local rule $q \to q'$, $\rightsquigarrow_t$ is the minimal relation that satisfies the following conditions:

1. if there exists $E_1, E_2 \in \mathbb{A}^*$ s.t. $\phi = E_1 q' E_2$, then $\phi \rightsquigarrow_t E_1 q E_2$.
2. if there exists $E_1, E_2 \in \mathbb{A}^*$ and $R \subseteq Q$ s.t. $\phi = E_1 R E_2$ and $q' \in R$, then $\phi \rightsquigarrow_t E_1 R q R E_2$.

In the first case we apply a backward rewrite step to an element occurring in the basis. In the second case we first select the process state q from context R and then proceed as in the first case. **Exist.** Let t be the rule $q \to q' \ : \ \exists_L P$, $\rightsquigarrow_t$ is the minimal relation that satisfies the following conditions:

1. if there exists $E_1, E_2, E_3 \in \mathbb{A}^*$ s.t. $\phi = E_1 p E_2 q' E_3$ with $p \in P$, then $\phi \rightsquigarrow_t E_1 p E_2 q E_3$.
2. if there exists $E_1, E_2, E_3 \in \mathbb{A}^*$, $R \subseteq Q$ s.t. $p \in R \cap P$ and $\phi = E_1 R E_2 q' E_3$, then $\phi \rightsquigarrow_t E_1 R p R E_2 q E_3$.
3. if there exists $E_1, E_2, E_3 \in \mathbb{A}^*$, $R \subseteq Q$ s.t. $p \in P$, $q' \in R$ and $\phi = E_1 p E_2 R E_3$, then $\phi \rightsquigarrow_t E_1 p E_2 R q R E_3$.
4. if there exists $E_1, E_2, E_3 \in \mathbb{A}^*$, with $R, S \subseteq Q$ s.t. $p \in S \cap P$, $q' \in R$ and $\phi = E_1 S E_2 R E_3$, then $\phi \rightsquigarrow_t E_1 S p S E_2 R q R E_3$.
5. if there exists $E_1, E_2 \in \mathbb{A}^*$, $R \subseteq Q$ s.t. $p \in R \cap P$, $q' \in R$ and $\phi = E_1 R E_2$, then $\phi \rightsquigarrow_t E_1 R p R q R E_3$.

In the first rule we apply the rule backwards to an explicit occurrence of q'. An explicit occurrence of p satisfies the guard. In the second rule we apply the rule backwards to an explicit occurrence of q' and select a context R (that intersects with P) to satisfy the guard. As precondition we add the subexpression RpR. In the third rule we reason in a similar way w.r.t. to an explicit occurrence of q', i.e., we assume that it occurs in a context R and add the subexpression RqR as precondition. In the fourth rule we apply the same reasoning both to the selected process q' and to the witness process p by assuming that they occur in two distinct contexts R and S. In the precondition we split both R and S and add the subexpressions RpR and SqS. In the last case we assume that p and q both occur in the same context R. In the precondition we add the subexpression $RpRqR$.

The rules for computing predecessors with respect to rules with guards of the form $\exists_R$ and $\exists_{LR}$ can be derived in a manner similar to the above described cases.

3.2 Experimental Results

We have implemented a prototype version of the backward scheme in Section 2.4 in which constraints are instantiated with CC regular expressions. In the simplest case studies described in the appendix like the (simplified) Bakery algorithm and the Illinois protocol the CC-based analysis automatically verifies mutual exclusion as shown in Table 1. We recall that parameterized systems with universal conditions are as powerful as counter and Turing machines, thus termination of exact analysis cannot be guaranteed. Exact analysis may diverge even on simple examples. Such a case occurs when testing mutual exclusion for the *dirty* cache line state (i.e. there cannot

Model	# iter	# constr	ex-time	verified
Bakery [4]	4	3	0.01s	$\surd$
Illinois [18]	2	17	0.05s	$\surd$

Table 1. Experimental results with exact analysis based on CC regular expressions.

be more than one dirty cache line for the same memory location) in the DEC Firefly model of [18]. A similar behavior was already observed with HyTech [23] (a tool manipulating polyhedra that can be used for unordered models) in [18]. In the more complicated examples like the algorithms of Burns and Szymanski exact analysis does not terminate.

4 Regular Expressions for Monotonic Abstraction

As discussed in the previous section, backward analysis with exact representation of predecessors leads to practical (not only theoretical) divergence in most of the examples of parameterized systems listed in the appendix.

One way of attacking verification in the other case studies is based on a reformulation of the monotonic abstraction introduced in [4] via a class of regular expressions that is less powerful than CC-expressions (hence they are used to compute an over-approximated analysis).

The monotonic abstraction we introduced in [4] is based on the idea of approximating predecessors of a set of configurations by using the smallest upward closed set that contains it. We recall that, as mentioned in the preliminaries, an upward closed set is defined in terms of subword inclusion. More precisely, we say that $c_1 \preceq c_2$ if c_1 is a subword of c_2. For instance, $aabb \preceq acabddba$ whereas $abc \not\preceq ab$ and $abc \not\preceq cab$. An upward closed set of configurations is a set S s.t. if $c \in S$ then for every c' s.t. $c \preceq c'$, $c' \in S$. It is well known that $\preceq$ is a well quasi ordering (wqo), i.e., for any infinite sequence $w_1 w_2 \ldots$ of words there exist $i < j$ s.t. $w_i \preceq w_j$. This property ensures that every upward closed set is finitely representable by a finite set of minimal words (the generators of the set w.r.t. $\preceq$). This is the reason why upward closed sets defined w.r.t. a wqo are particularly interesting for symbolic backward analysis. The finite-basis property provides a natural way to represent an infinite upward closed set of configurations (words) by just taking the corresponding finite set of generators. The generators can be then viewed as constraints in the generic backward scheme of Section 2.4.

Let us first now reformulate these notions in our setting. Assume a parameterized system $\mathcal{P} = (Q, T)$, where Q is a finite set of states. In order to finitely represent upward closed sets of system configurations we use the subclass of CC expressions defined as follows. We still work with words in $\mathbb{A}^*$, where $\mathbb{A} = Q \cup \mathbb{P}(Q)$ and $\mathbb{P}(Q)$ denotes the set of subsets of Q.

An *upward-closed (*UC-)constraint is a special case of CC-constraints of the form

$$Qq_1Q \ldots q_nQ$$

where $q_i \in Q$ for $i : 1 \leq i \leq n$.

The denotation of a UC-constraint ϕ, written $[\![\phi]\!]$, is defined as the set of configurations of the form

$$c_0q_1c_1 \ldots q_nc_n$$

where $c_i \in Q^*$ for $i : 0 \leq i \leq n$. It is immediate to verify that $[\![\phi]\!]$ is upward closed w.r.t. the subword relation $\preceq$ and that $q_1 \ldots q_n$ is the generator of the set. As an example, given $Q = \{q_1, q_2, q_3\}$ the constraint ϕ defined as Qq_1Qq_2Q denotes all configurations of the form $c_0q_1c_1q_2c_2$ such that sub-configurations c_0, c_1 and c_2 are arbitrary words in Q^*. As another example, notice the set of bad configurations in Example 1 is upward closed and can be characterized by the UC Qq_4Qq_4Q. Actually, in all examples considered in the appendix, bad configurations that violate a safety property can be represented as upward closed sets (i.e. as UC expressions). This modeling evidence shows that a backward approach based on upward closed sets has often a favorable initial seed (the bad configurations) for carrying out the analysis.

The denotation of a CC $Qq_1Q \ldots q_nQ$ can also be defined via the regular expression

$$C_0 \cdot q_1 \cdot C_1 \ldots q_n \cdot C_n$$

where $C_i = (q_1 + \ldots + q_k)^*$ for $Q = \{q_1, \ldots, q_k\}$.

Since each context always corresponds to the whole alphabet Q, we can simplify the notation and write a UC expression by simply using $Qq_1Q \ldots q_nQ$ the corresponding basis $q_1 \ldots q_n$. This is the notation used, e.g., in [4], where an upward closed set of configurations is represented via the finite set of words that generates it.

4.1 Symbolic Operations

The symbolic operations for handling UC-expressions can be derived from those of CC-expressions. Still, it is interesting to make some observations on entailment and predecessor computation.

Entailment. We first observe that for constraints $\phi = Qq_1 \ldots q_n Q$ and $\phi' = Qq'_1 \ldots q'_m Q$, the entailment relation is directly defined via the subword relation of the corresponding basis, i.e., $\phi \sqsubseteq \phi'$ iff $w_1 = q_1 \ldots q_n \preceq w_2 = q'_1 \ldots q'_m$ (w_1 is a subword of w_2). Thus inclusion of the upward closed sets $[\![\phi]\!]$ and $[\![\phi']\!]$ reduces to the subword relation of the corresponding generator words. Since the subword relation is a well quasi ordering, working with UC regular expressions ensures the termination of the backward scheme described in Section 2.4.

Computing Predecessors. As mentioned in the introduction, the monotonic abstraction framework is based on the idea of approximating predecessors $\mathit{Pre}(S)$ of a given set of configurations S (in our case S is upward closed) via the smallest upward closed set that contains it. Thus, given a UC constraint ϕ, computing predecessors amounts to approximate the denotation of $\mathit{Pre}(\phi)$, defined using the definition given for CC constraints, with the UC constraint ϕ' s.t. $[\![\mathit{Pre}(\phi)]\!] \subseteq \phi'$ and such that no other UC constraint has smaller set of denotations that still contains those of $\mathit{Pre}(\phi)$.

Given an upward closed set S of configurations, it is easy to verify that $\mathit{Pre}(S)$ remains an upward closed set when computed for local or existentially quantified transitions. Thus, in this case the symbolic computation of *Pre* coincides with that defined for CC expressions.

The subtle case is for transitions with universal global conditions. In this case $\mathit{Pre}(S)$ may not be upward closed. Consider for instance, the rule $q_2 \to q_3$ with global condition $\forall_{LR} \{q_0\}$. Given a CC expression Qq_3Qq_0Q that represents any word containing the subword q_3q_0, the set of predecessors is the set of words of the form Rq_2Rq_0R where $R = \{q_0\}$, i.e., all words with one or more occurrences of q_0, and exactly one occurrence of q_2 to the left of q_0. Clearly, this set is not upward closed w.r.t. subword inclusion (e.g. $q_3q_3q_0$ contains the subword q_3q_0 but it is not a valid predecessor). Even though the set of predecessors is not upward closed, we can approximate it with the smallest upward closed set that contains it. Such a set is obtained by relaxing R into Q, i.e., it corresponds to the UC expression Qq_2Qq_0Q.

In general, the UC constraint that represents the monotonic abstraction of a rule with universal global condition $q \to q' : \forall P$ starting from a UC constraint

$$Qq_1 \ldots q_{i-1}Qq'Qq_{i+1}Q \ldots q_nQ$$

is the UC constraint

$$Qq_1 \ldots QqQ \ldots q_nQ$$

if and only if $\{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n\} \subseteq P$. If some q_j with $j \neq i$ is not in P, then the set of predecessors is empty. In the previous example we proved that this abstraction may give a strict superset of the predecessor configurations. However implementing it is easy and avoids the use of transducers or expensive case analysis as for CC expressions.

In [4] we have shown that, when interpreted over the operational semantics, monotonic abstraction corresponds to an approximated transition system in which transitions with universal global conditions are treated in the following way. Universal transitions are always enabled but all processes that do not comply with the universal condition are deleted.

Monotonic abstraction works well in several examples (see discussion in the next section). However it may return false positives. To understand the type of false positives returned by monotonic abstraction, consider the abstract transition system associated to the system in Fig. 1 computed by applying monotonic abstraction. Consider the following run in the original (non-abstract) system: $q_0q_0q_0q_0 \longrightarrow q_0q_1q_0q_0 \longrightarrow q_0q_1q_1q_0 \longrightarrow q_0q_2q_1q_0 \longrightarrow q_0q_2q_2q_0 \longrightarrow q_0q_3q_2q_0 \longrightarrow q_0q_4q_2q_0 \longrightarrow q_0q_1q_2q_0$.

From the next-to-last configuration, the left most process can move (in the abstract system) to q_1. More precisely, the run may continue as follows in the abstract system. $q_0q_4q_2q_0 \longrightarrow q_1q_4q_0 \longrightarrow q_1q_4q_1 \longrightarrow q_2q_4q_1 \longrightarrow q_2q_4q_2 \longrightarrow q_3q_4q_2 \longrightarrow q_4q_4q_2$. Notice that monotonic abstraction removes the guard (the process in state q_2) since it does not satisfy the global condition of the rule $q_0 \to q_1 : \forall \{q_0, q_1, q_4\}$. With this abstraction the door is opened again. This allows processes in q_0 to move again, enabling one of them to eventually join the process which is already in the critical section. This gives a false positive.

We found similar false positives in the analysis of some of the case studies listed in the appendix. We discuss this point in the next section.

4.2 Experimental Results

The backward scheme in Section 2.4 in which constraints are instantiated with UC regular expressions is at the core of the PFS model checker described in [4].

As shown in Table 2 monotonic abstraction has enough precision to verify the considered properties for a wide range of examples. However, it returns false positives for some of the protocols in Table 2. One of these examples is the model for Szymanski's algorithm taken from [28] and shown in Fig. 2. Other examples are: a different formulation of Szymanski's algorithm taken from [22], a model for reference counting in virtual memory [19], and variants of the readers/writers mutual exclusion protocol. All examples are described in the appendix.

Concerning the model of Szymanski's algorithm with non-atomic updates of Fig. 2, the main steps of the spurious error trace returned by PFS (monotonic abstraction) on the algorithm of Fig. 2 are described below.

$$(s_0, s_0, s_0) \to^* (s_1, s_1, s_1) \to^* (s_1, s_1, s_3) \to (s_2, s_1, s_3) \to (s_3, s_1, s_3) \to (s_3, s_1, s_4) \to^* (s_5, s_2, s_4) \to (s_9, s_2, s_4) \to^* (s_9, s_2, s_7) \longrightarrow_A (s_3, s_7) \to^* (s_{10}, s_{10})$$

Model	# iter	# constr	ex-time	spurious trace	verified
Bakery [4]	2	2	0.01s		$\surd$
Illinois [18]	5	33	0.02s		$\surd$
Burns [4]	14	40	0.05s		$\surd$
DEC Firefly [18]	3	11	0.01s		$\surd$
Compact Szymanski's alg. [10,31]	10	17	0.1s		$\surd$
Refined Szymanski's alg. [28]	24	658	1.5 s	$\surd$	
Gribomont-Zenner [22]	36	197	0.2 s	$\surd$	
Ref. counting [19]	7	15	0.02s	$\surd$	
Readers/writers[33]	(10:5)	(31:28)	(0.05s:0.02s)	(:$\surd$)	($\surd$:)
Readers/writers (locks:no locks)	(21:7)	(125:67)	(0.4s:0.6s)	(:$\surd$)	($\surd$:)
Readers/writers (locks:no locks)	(22:9)	(683:219)	(9.4s:0.3s)	(:$\surd$)	($\surd$:)
Light control [33]	13	96	0.06s	$\surd$	

Table 2. Experimental results with UC regular expressions (monotonic abstraction).

```
var f : array[N] of {0,1,2,3,4}
f := (0,...,0);
process p[i] =
1   non critical;
2   f[i] := 1;
3   await ∀j ≠ i.f[j] < 3;
4   f[i] := 3;
5   if ∃j ≠ i.f[j] = 1
      then
6         f[i] := 2;
7         await ∃ j ≠ i.f[j] = 4;
8         f[i] := 4;
9     else  f[i] := 4;
10  await ∀j < i.f[j] < 2;
11  critical section;
12  await ∀j > i.f[j] < 2 ∨ f[j] > 3;
13  f[i] := 0;
```

```
States :  Q = {s0, s1, ..., s11}
Transitions :
 instruction : transition
           1 : s0 → s1
           2 : s1 → s2
           3 : s2 → s3 : ∀LR{s0, s1, s2, s3, s7, s8}
           4 : s3 → s4
      5 then : s4 → s6 : ∃LR{s2, s3}
           6 : s6 → s7
           7 : s7 → s8 : ∃LR{s9, s10, s11}
           8 : s8 → s9
      5 else : s4 → s5 : ∀LR¬{s2, s3}
           9 : s5 → s9
          10 : s9 → s10 : ∀L{s0, s1, s2, s3}
          11 : s10 → s11 : ∀R¬{s4, s5, s6, s7, s8}
          12 : s11 → s0

Initial state :  s0
Bad configurations :  φ = (s10 s10, Q*)
```

Fig. 2. Szymanski's Algorithm [28] (left), and its parameterized model (right).

The step indicated with $\longrightarrow_A$ corresponds to the deletion of a process violating the universal condition of the third instruction in Fig. 2(right). The spurious error trace is due to the fact that with monotonic abstraction we cannot distinguish between the existence and non-existence of a process in a given state in the contexts between two explicit states (contexts contain all possible strings over Q). Therefore, to rule out the spurious trace we have to refine the information represented in the contexts.

5 Regular Expressions for Context-sensitive Analysis

In order to eliminate false positives like those obtained during the analysis of Szymanski's algorithm, we need to improve the precision of the symbolic representation of configurations, preferably retaining the guarantee of termination provided by the UC class of regular expressions. For this purpose, in [8] we have defined a special class of constraints called *simple context-sensitive (SCC-) constraints.*

A simple context-sensitive (SCC-)constraint is a word in $\mathbb{A}^*$ of the form

$$Rq_1R \ldots q_nR$$

in which $\{q_1, \ldots, q_n\} \subseteq R \subseteq Q$.

Since the same constraint is uniformly applied to each context in the basis, we can simplify the notation and represent an SCC as a pair (c, R), where $c \in Q^*$ and $c^\bullet \subseteq R \subseteq Q$. We refer to R as the *padding set.* As we discuss later in this section, the requirement that the basis c is included in the padding set has two consequences: it allows us to apply the theory of well-quasi ordering to ensure termination of the backward analysis; it gives us a natural way to define *accelerations* to speed up the symbolic computation of predecessors. Notice that an SCC need not represent an upward closed set of configurations. Indeed, the environment R may be a strict subset of the set of all states. For instance, if $Q = \{a, b, c\}$ then the denotation of the SCC $(aa, \{a, b\})$ contains strings like $aa, aba, abab, \ldots$ but it does not include any strings with c even if they contain aa as a substring (i.e. $aca, abac, \ldots$ are not in its denotation).

A CC $\phi = R_0q_0 \ldots q_nR_n$ can naturally be approximated by the following SCC:

$$\phi^{\#} = (q_0 \dots q_n, \phi^\bullet)$$

Indeed, it is immediate to check that $[\![\phi]\!] \subseteq [\![\phi^{\#}]\!]$; intuitively, we will over-approximate the CC constraints generated during the analysis using the $^{\#}$ operator. Let us now reconsider the symbolic operations (discussed in Section 3 for CC's) needed for implementing an SCC-based symbolic backward analysis.

Entailment The entailment relation for SCC's can now be simplified as follows. For $\phi = (c, R)$ and $\phi' = (c', R')$, we have that $\phi \sqsubseteq \phi'$ iff $c \preceq c'$ and $R \supseteq R'$. We recall that $\phi \sqsubseteq \phi'$ implies $[\![\phi']\!] \subseteq [\![\phi]\!]$.

Furthermore, the following property holds.

Lemma 1. *$\sqsubseteq$ is a well quasi-ordering (WQO) for SCC's.*

Proof. We prove that for any infinite sequence of constraints $\phi_0, \phi_1, \dots$, there exist $i < j$ such that $\phi_i \sqsubseteq \phi_j$. Indeed, let ϕ_i be of the form (c_i, R_i). Since Q is finite and $R_i \subseteq Q$ for all i, it follows that there is an infinite subsequence $\phi_{i_0}, \phi_{i_1}, \phi_{i_2}, \dots$ such that $R_{i_j} = R_{i_k}$ for all j, k. By Higman's lemma [24] (which implies that $\preceq$ is a WQO on Q^*), there are $j < k$ such that $c_{i_j} \preceq c_{i_k}$, and hence $\phi_{i_j} \sqsubseteq \phi_{i_k}$. From this it follows that $\sqsubseteq$ is a well-quasi ordering on the set of SCC'sexpressions.

We extend the relation $\sqsubseteq$ to sets of SCC's such that $\Phi_1 \sqsubseteq \Phi_2$ if for each $\phi_2 \in \Phi_2$ there is a $\phi_1 \in \Phi_1$ with $\phi_1 \sqsubseteq \phi_2$. Notice that $\Phi_1 \sqsubseteq \Phi_2$ implies that $[\![\Phi_2]\!] \subseteq [\![\Phi_1]\!]$.

As an example, consider the SCC $\phi = (pq, \{p, q, r\})$. Examples of configurations in $[\![\phi]\!]$ are prq and $rprprqr$. The set of bad configurations in Example 1 can be characterized by the SCC's $(q_4 q_4, \{q_0, q_1, q_2, q_3, q_4\})$. Also, for the SCC $\phi = (pq, \{p, q, r, s\})$ and $\phi' = (qpprqp, \{p, q, r\})$, we have $\phi \sqsubseteq \phi'$.

5.1 Computing Predecessors

The abstract predecessor operator $Pre^{\#}$ is obtained as the composition of Pre and of the abstraction $^{\#}$, i.e., $Pre^{\#}(\phi) = (Pre(\phi))^{\#}$. However, it would be inefficient to implement it in this way. Indeed, in general Pre requires the analysis and generation of several cases (as for $\exists_L$-rules). The large number of generated constraints makes the exact analysis unfeasible even on simple examples. For this reason, we show next how to directly define $Pre^{\#}$ as an operator working on SCC's-constraints.

First, we introduce some notations. For a basis c and a state q, we write $c \otimes q$ to denote the set $\{c_1 q c_2 \mid c = c_1 c_2\}$. The operation adds the singleton q in an arbitrary position inside c. We define $Pre^{\#}$ by means of a relation $\overset{t}{\leadsto}$ defined as follows. For a transition t, we define $\overset{t}{\leadsto}$ to be the smallest relation on constraints containing the following elements:

Local: If t is a local transition of the form $q \to q'$ then

- $(c_1 q' c_2, R) \overset{t}{\leadsto} (c_1 q c_2, R \cup \{q\})$.
- $(c, R) \overset{t}{\leadsto} (c_1, R \cup \{q\})$ if $q' \in R$ and $c_1 \in (c \otimes q)$

In the first case, a process in the basis of the constraint performs a local transition from q to q'. We add q to the padding set as required by the well-formedness of SCC's-constraints (the basis must always be contained in the padding set). From an operational perspective, augmenting the padding set with q has an effect similar to *widening* operators used in relational analysis for unordered parameterized systems (e.g. based on polyhedra in [18]). To illustrate this, consider the rule $p \to q$ and the constraint (r, R) where $R = \{q, r\}$. The exact predecessor computation would compute an infinite sequence of the form $RpRrR$, $RrRpR$ (one occurrence of p), $RrRpRpR$, $RpRrRpR$, $RpRpRrR$ (two occurrences of p), ... Our approximated operator computes instead in one step the limit of the sequence, i.e., $(rp, R \cup \{p\}), (pr, R \cup \{p\})$ (*at least* one occurrence of p). Thus, our abstraction plays here the role of a *widening* step for *ordered* configurations.

Exists: if t is a global transition of the form $q \to q' : \exists_L P$ then

- $(c_1 q' c_2, R) \overset{t}{\leadsto} (c_1 q c_2, R \cup \{q\})$ if $P \cap c_1{}^\bullet \neq \emptyset$.
- $(c_1 q' c_2, R) \overset{t}{\leadsto} (c_3 q c_2, R \cup \{q\})$ if $p \in P \cap R$, $p \notin c_1{}^\bullet$, $c_3 \in (c_1 \otimes p)$.
- $(c_1 p c_2, R) \overset{t}{\leadsto} (c_1 p c_3, R \cup \{q\})$ if $p \in P$, $q' \in R$, and $c_3 \in (c_2 \otimes q)$.
- $(c_1 c_2, R) \overset{t}{\leadsto} (c_1 p c_3, R \cup \{q\})$ if $p \in P \cap R$, $p \notin c_1{}^\bullet$, $q' \in R$, and $c_3 \in (c_2 \otimes q)$.

In the first case, a process in the basis of the constraint performs an existential global transition from q to q'. The transition is performed if there is a witness which is to the left of the process and which is inside the basis of the constraint. The second case is similar to the first case, except that the witness is in the padding set (and not in the left part of the basis). Therefore, we add the witness explicitly in an arbitrary position to the left of the process. In the third case, a number of processes (at least one process) in the padding set perform the transition. There is a witness which enables the transition inside the basis. The witness should be to the left of the process making the transition. In the fourth case, both the witness and the process making the transition are in the padding set. This case is similar to the third case, except that we need to add the witness process to the basis. In a similar manner to the local transition case, we add q to the padding set.
If t is a global transition of the form $q \to q' : \exists_R P$ or $q \to q' : \exists_{LR} P$, then analogous conditions to the previous case hold.

Forall: t is a global transition of the form $q \to q' : \forall_{LR} P$, then

- $(c_1 q' c_2, R) \overset{t}{\leadsto} (c_1 q c_2, (R \cap P) \cup \{q\})$, if $(c_1 c_2)^\bullet \subseteq P$.

– $(c_1c_2, R) \overset{t}{\rightsquigarrow} (c_1qc_2, (R \cap P) \cup \{q\})$, if $q' \in R$, $q \notin R$ and $(c_1c_2)^\bullet \subseteq P$.

In the first case, a process in the basis moves from q to q'. The remaining processes in the basis must be in P. Furthermore, we restrict the padding set to those processes within P. In the second case, a process of type q in the padding set moves to q'. Notice that in both cases, the state q is added to the padding set.
If t is a global transition of the form $q \to q' : \forall_L P$, then

– $(c_1q'c_2, R) \overset{t}{\rightsquigarrow} (c_1qc_2, R \cup \{q\})$, if $c_1{}^\bullet \subseteq P$.
– $(c_1c_2, R) \overset{t}{\rightsquigarrow} (c_1qc_2, R \cup \{q\})$, if $q' \in R$ and $c_1{}^\bullet \subseteq P$.

In the first case, a process in the basis moves from q to q'. The remaining processes in the basis belong to R. In our constraints we use a single padding set to define the constraints on processes to the left and to the right of the process that makes the transition. Thus, to compute the precondition of the universal condition on the padding set we have to apply an over-approximation and use R as constraints on contexts (processes to the left should be restricted to $R \cap P$). In the second case, a process q from the padding set moves to q'. Once again, the state q is added to the padding. The second case is similar to the first case, except that the process that performs the transition is selected from the padding set.
If t is a global transition of the form $q \to q' : \forall_R P$, then analogous conditions to the previous case hold.

5.2 Experimental Results

We have implemented the backward scheme of Section 2.4 using SCC expressions. Since these expressions are used to compute overapproximations of predecessor configurations, the backward algorithm is guaranteed to terminate and to compute an overapproximation of the set of predecessors of the bad configurations given in input. Furthermore, the resulting fixpoint is included in the fixpoint computed using UC constraints, i.e., with SCC expressions we improve precision without losing termination. Clearly, the price is an increased complexity in the computation of predecessors (more cases to analyze w.r.t. UC constraints), and thus a possible combinatorial explosion during the analysis.

Despite of this additional complexity, the approximated SCC-based algorithm terminates without detecting error traces, i.e., mutual exclusion is verified for all the examples considered in the appendix. This collection of benchmark includes most of the examples studied in alternative approaches for parameterized verification. For instance, the compact model studied in [10,31] can be verified using both UC and SCC.

6 Conclusions and Related Work

We have presented a survey on recent advances on parameterized verification carried out in the last years in the context of a fruitful collaboration between research groups at the University of Genova and Uppsala. In the paper we have given a uniform approach of different algorithms, approximations and heuristics using the common language of regular expressions. Our approach is a lightweight form of regular model checking as it tries to combine the benefits of using regular languages as a symbolic representation of configurations with those of abstraction to efficiently compute predecessors and entailment. As future research we plan to further investigate verification methods and efficient data structures for SCC's-like context-sensitive constraints that can help to lift the non-atomicity assumptions. We also plan to come up with refinement schemes that are similar to the one developed in [3] for the unordered case.

Related Work The constraints used for the exact analysis are similar to the APC regular expressions studied in [13]. The verification method proposed in [13] is complementary to ours. Indeed, it is based on symbolic forward exploration with accelerations and without guarantying termination; whereas we consider here an over-approximation (based on simple context-sensitive constraints) that ensures the termination of symbolic backward exploration.

Other parameterized verification methods based on reductions to finite-state models have been applied to safety properties of mutual exclusion protocols like Szymanski's algorithm. Among these, we mention the *invisible invariants* method [10,30] and the *environment abstraction* method [15,31]. In [31] environment abstraction is applied to a formulation of Szymanski's algorithm with the same assumptions as the model in [10], called *compact Szymanski* in Table 2. This model can be verified using monotonic abstraction. The refined model [28] we consider is different in that atomic instructions do not contain both tests and assignments. This potentially introduces new race conditions making verification a harder task. It is not clear whether the refined models of Szymanski's algorithm considered in the present paper can be automatically verified using the methods suggested in [10,15].

The infinite-state reference counting example we consider in this paper is inspired by a finite-state abstraction studied in [19]: in contrast to the predicate-abstraction approach used in [19], we model reference counting for a physical page under observation via an unbounded integer shared variable, with increment, decrement, and zero-test.

Unordered models with counters can be modeled with systems working on unbounded integer variables such as in ALV [33,34] (based on the Omega library) and HyTech [23] (based on Halbwachs's polyhedra library). In these approaches extrapolation and widening operators are needed to enforce termination. This is typical for polyhedra-based methods when applied to models like DEC firefly and readers/writers. In contrast to methods

Model	**# iter**	**# constr**	**ex-time**	**verified**
Bakery [4]	3	2	0.01s	$\surd$
Illinois [18]	7	53	0.18s	$\surd$
Burns [4]	15	48	0.02s	$\surd$
DEC Firefly [18]	5	10	0.03s	$\surd$
Compact Szymanski's alg. [10,31]	24	162	3.35s	$\surd$
Refined Szymanski's alg. [28]	34	641	1m	$\surd$
Gribomont-Zenner [22]	56	863	5m	$\surd$
Ref. counting [19]	7	8	0.01s	$\surd$
Readers/writers [33] (locks:no locks)	(7:7)	(12:8)	(0.02s:0.01s)	($\surd$:$\surd$)
Readers/writers (locks:no locks) pr=readers	(25:12)	(128:34)	(1.7s:0.06s)	($\surd$:$\surd$)
Readers/writers (locks:no locks) pr=writers	(27:9)	(646:19)	(17.2s:0.03s)	($\surd$:$\surd$)
Light control [33]	9	29	0.02s	$\surd$

Table 3. Experimental results with SCC regular expressions.

like HyTech and ALV, the algorithm presented in this paper incorporates accelerations that can be applied both to ordered and unordered parameterized systems without losing termination.

Finally, an SMT-based implementation of the symbolic backward reachability scheme is proposed in [21]. In [21] parameterized systems are represented as unbounded arrays. First-order formulas are used to represent minimal elements of upward closed sets. An SMT-solver is interleaved with an ad hoc predecessor operator in order to simplify and discard formulas during a least fixpoint computation. Invariant generation is used as an heuristics to accelerate the computation. The performance obtained with the SMT-based implementation is very promising. Integrating heuristics like those used in [21] in our symbolic method could be an interesting direction for future work.

References

1. P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic abstraction for programs with dynamic memory heaps. CAV 2008: 341-354.
2. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. LICS 1996: 313–321.
3. P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine. Constrained monotonic abstraction: A cegar for parameterized verification. CONCUR 2010: 86–101.
4. P. A. Abdulla, N. Ben Henda, G. Delzanno, and A. Rezine. Regular model checking without transducers. TACAS 2007: 721–736.
5. P. A. Abdulla, N. Ben Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. VMCAI 2008: 22-36.
6. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. CAV 2007: 145–157.
7. P. A. Abdulla, G. Delzanno, F. Haziza, and A. Rezine. Parameterized tree systems. FORTE '08: 69–83.
8. P. A. Abdulla, G. Delzanno, and A. Rezine. Approximated Context-sensitive Analysis for Parameterized Verification. . FMOODS/FORTE 2009: 41-56.
9. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. CONCUR 2002: 116–130.
10. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. CAV 2001: 221–234.
11. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. CAV 2003: 223–235.
12. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. CAV 2004: 372–386.
13. A. Bouajjani, A. Muscholl, and T. Touili. Permutation Rewriting and Algorithmic Verification. Inf. and Comp., 205(2): 199-224, 2007.
14. T. Bultan, R. Gerber, W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. TOPLAS. 21(4): 747-789, 1999.
15. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. VMCAI 2006: 126-141.
16. P.-J. Courtois, F. Heymans, D. Lorge Parnas. Concurrent Control with "Readers" and "Writers". CACM 14(10): 667-668, 1971.
17. D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. CAV 2001: 286-297.
18. G. Delzanno. Constraint-Based Verification of Parameterized Cache Coherence Protocols. FMSD 23(3): 257-301 (2003)
19. M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counted objects. TACAS 2009: 352-367.
20. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! TCS 256(1-2):63–92, 2001.
21. S. Ghilardi, S. Ranise Backward Reachability of Array-based Systems by SMT Solving: Termination and Invariant Synthesis, Logical Methods in Computer Science, to appear, 2010.
22. E. Gribomont and G. Zenner. Automated verification of Szymanski's algorithm. TACAS 1998: 424–438.
23. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. STTT 1:110-122, 1997.

24. G. Higman. Ordering by divisibility in abstract algebras. London Math. Soc. (3), 2(7):326–336, 1952.
25. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. TCS 256: 93–112, 2001.
26. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
27. Z. Manna et al. STEP: the Stanford Temporal Prover, 1994.
28. Z. Manna and A. Pnueli. An exercise in the verification of multi-process programs. Beauty is Our Business: 289–301, 1990.
29. M. Nilsson. *Regular Model Checking.* PhD thesis, Uppsala University, 2005.
30. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. TACAS 2001: 82–97.
31. M. Talupur. *Abstraction techniques for parameterized verification.* PhD thesis, CMU, 2006.
32. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. LICS 1986: 332–344.
33. T. Yavuz-Kahveci, T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. STTT 5(1): 15-33, 2003.
34. T. Yavuz-Kahveci, T. Bultan. Verification of parameterized hierarchical state machines using action language verifier. MEMOCODE 2005: 79-88

A Detailed Description of Case studies

For sake of completeness, we include in this appendix a detailed description of all examples tested with our algorithm.

We describe mutual exclusion algorithms, cache coherence protocols, and counter based synchronization schemes on which we tried our approach to verify typical safety properties for the induced infinite-state transition system. We use rules like $[q \rightarrow q'] : \phi$ to denote an update from q to q' of a local state with condition (global conditions, formulas on shared variables) ϕ. For shared variables we used unprimed(primed) variables to indicate pre-(post-)conditions. For brevity, we often omit to indicate preconditions for variables that are unconstrained. For instance, we use *true* if all variables can take any value. Furthermore, if a variable does not occur in the postcondition of a transition, then we assume that its value remains unchanged. For example, $[q \rightarrow q'] : \neg v$ states that a process can move from state q to state q' if the shared boolean variable v evaluates to false and regardless of the values of the other variables; in this transition, no variable is altered in the system. $[q_1 \rightarrow q_2]\,[q_3 \rightarrow q_4] : \phi$ to indicate a synchronization step (two process simultaneously update their states when ϕ is satisfied). We also use * in rules like

$$[q_1 \rightarrow q_2]\,[q_3 \rightarrow q_4]^* \ldots : \phi$$

$States:\ Q = \{q_1, q_2, q_3\}$

$Transitions:$
$t_1:\ [q_1 \rightarrow q_2] : \forall_R\{q_1\}$
$t_2:\ [q_2 \rightarrow q_3] : \forall_L\{q_1\}$
$t_3:\ [q_3 \rightarrow q_1]$

$Initial\ configurations:\ q_1^*$
$Bad\ configurations:\ \phi = (q_3 q_3, Q^*)$

Fig. 3. Bakery algorithm

to indicate broadcast rules, i.e., when a process moves from q_1 to q_2 and all processes in q_3 move to q_4. Finally, we often use $\cdot$ instead of state to denote dynamic creation and deletion of processes. E.g. $[\cdot \rightarrow q]$ denotes dynamic creation of a process with initial state q.

A.1 Bakery mutex

Figure 3 describes a simplified version of the original Bakery algorithm [26]. In this version [29], processes have states that range over $\{q_1, q_2, q_3\}$. A process gets a ticket with a value strictly higher than the ticket value of any process in the queue (transition t_1). A process accesses the critical section if it has a ticket with the lowest value among the existing tickets (transition t_2). Finally, a process leaves the critical section, freeing its ticket (transition t_3). Mutual exclusion violation (detected with transition t_4) corresponds to configurations where more than one process is in state q_3. Transition t_4 is a binary communication. It can be easily encoded using a shared variable and an $\exists_{LR}$ global transition.

A.2 Burns mutex

In this algorithm (Fig.4), processes have states that range over (q, f) where q is in $\{q_1, ...q_7\}$ and f is Boolean. Each process interested in accessing the critical section checks twice to its left if there are other interested processes. If there are, it returns to q_1 (transitions t_2, t_5), otherwise, it continues (transitions t_3, t_6). Once at q_6, all processes successively access the critical section (q_7) starting with the right most ones.

Mutual exclusion is violated in case more than one process is at state $(q_7, .)$.

A.3 Compact version of Szymanski's algorithm

In this algorithm (Fig. 5), processes have states that range over $\{q_0, \ldots q_7\}$. The state q_0 is initial, and q_7 is the critical section. Once processes take transition t_2, they are ensured to access the critical section. At transition t_4, processes go to state q_4 where they wait for processes at state q_1, q_2, if any. Otherwise, transition t_6 to q_5 is fired. Once a process is at state q_5, no other process can

States : $Q = \{(q_1, f) | q_1 \in \{q_1, \dots, q_7\}, f \in Bool\}$
Transitions :

$t_1 : [q_1 \to q_2, \neg f]$	$t_2 : [q_2 \to q_1] : \exists_L\{f\}$	$t_3 : [q_2 \to q_1] : \forall_L\{\neg f\}$
$t_4 : [q_3 \to q_4, f]$	$t_5 : [q_4 \to q_1] : \exists_L\{f\}$	$t_6 : [q_4 \to q_5] : \forall_L\{\neg f\}$
$t_7 : [q_5 \to q_6] : \forall_R\{\neg f\}$	$t_8 : [q_6 \to q_7, \neg f]$	$t_9 : [q_7 \to q_1]$

Initial configurations : $(q_1, \neg f)^*$
Bad configurations : $\phi = (q_7 q_7, Q^*)$

Fig. 4. Burns algorithm.

States : $Q = \{q_0, \dots q_7\}$
Transitions :

$t_1 : [q_0 \to q_1]$	$t_2 : [q_1 \to q_2] : \forall_L\{q_0, q_1, q_2, q_4\}$	$t_3 : [q_2 \to q_3]$
$t_4 : [q_3 \to q_4] : \exists_{LR}\{q_1, q_2\}$	$t_5 : [q_3 \to q_5] : \forall_{LR}\{q_0, q_3, q_4, q_5, q_6, q_7\}$	
$t_6 : [q_4 \to q_5] : \exists_{LR}\{q_5, q_6, q_7\}$	$t_7 : [q_5 \to q_6] : \forall_{LR}\{q_0, q_1, q_2, q_5, q_6, q_7\}$	
$t_8 : [q_6 \to q_7] : \forall_L\{q_0, q_1, q_2\}$	$t_9 : [q_7 \to q_0]$	

Initial configurations : q_0^*
Bad configurations : $\phi = (q_7 q_7, Q^*)$

Fig. 5. Compact Szymanski's Algorithm.

fire t_2, and all processes waiting at state q_4 can get to state q_5. Once all processes that fired t_2 have gathered at state q_5 they can get to state q_6 from which they can access the critical section q_7 with priority to processes to the left. Configurations violating mutual exclusion are those with at least two processes at state q_7.

A.4 Griboment-Zenner Mutex

This algorithm (Fig.6) could be seen as a version of Szymanski's algorithm (Fig.5), with transitions that are finer grained in the sense that tests and assignments are split over different atomic transitions. In this model, process states range over $\{q_1, \dots q_{13}\}$, with q_1 as initial state.

Configurations not satisfying mutual exclusion are those where at least two processes are at state q_{12}.

A.5 Illinois cache coherence

In this protocol (Fig. 7), a cache line may be at one of the four states: *invalid*, *dirty*, *shared*, and *valid*. Transitions t_1, t_2 and t_3 correspond to read misses, t_4, t_5 to write hits, and t_6 to a write miss, and transitions t_7, t_8 and t_9 to cache line replacements. Transitions t_1, t_{10} and t_{11} are binary communications. Transitions t_4 and t_6 are broadcasts. Transition t_3 is also a broadcast, and says that if there is a cache line at state *shared* or *valid*, then an invalid cache can move to state *shared*, while, simultaneously, all caches at state *valid* move to state *shared*. Cache lines are not coherent if a cache line in state dirty coexists with another line in state dirty or shared.

A.6 DEC Firefly cache coherence

In this protocol (Fig. 8), a cache line may be in one of the following states: *invalid*, *exclusive*, *shared* and *dirty*. Transitions t_1, t_2 and t_3 correspond to read misses, t_4, t_6 to write hits, and t_5 to a write miss. Cache lines are not coherent if a *dirty* line coexists with a *shared* a *dirty* or an *exclusive* line, or if more than one cache line are in state *exclusive* (detected with transitions t_7, t_8 and t_9).

A.7 A reference counting model

We also look at a class of parameterized systems in which processes use shared counters like classical solutions to the readers/writers problem, and in reference counting schemes like those described in [19]. We consider (Fig 9) an abstract (but still infinite-state) model for the page manager (pmap) described in [19]. In this example we fix a generic process P and a physical page PP. We keep track, via an unbounded counter, of operations (e.g. allocation, deallocation, map and unmap) of the physical memory and the process' address space. The counter is modeled as a collection of processes with two states: *zero* and *one*. The current value of the counter is defined by the number of processes in state *one*. We use a shared variable *pmap* that is true iff PP is mapped in P's address space. We also use shared variables *check* and *test* to denote two special phases of the protocol: *check* is entered after each *unmapping* of a virtual page pointing to PP, whereas *test* is entered upon deallocation of the entire address space of P (i.e. when all references to PP are removed). The latter phase is implemented with a loop in

$States: \ Q = \{q_1, \ldots q_{13}\}$
$Transitions:$
$t_1: \ [q_1 \to q_2] \qquad t_2: \ [q_2 \to q_3]$
$t_3: \ [q_3 \to q_4] : \forall_{LR}\{q_1, q_2, q_3, q_4, q_7, q_8\}$
$t_4: \ [q_4 \to q_5]$
$t_5: \ [q_5 \to q_6] : \exists_{LR}\{q_3, q_4, q_{10}, q_{11}, q_{12}, q_{13}\}$
$t_6: \ [q_6 \to q_7]$
$t_7: \ [q_7 \to q_8] : \exists_{LR}\{q_{10}, q_{11}, q_{12}, q_{13}\}$
$t_8: \ [q_8 \to q_9]$
$t_9: \ [q_5 \to q_9] : \forall_{LR}\{q_1, q_2, q_5, q_6, q_7, q_8, q_9\}$
$t_{10}: \ [q_9 \to q_{10}]$
$t_{11}: \ [q_{10} \to q_{11}] : \forall_{LR}\{q_1, q_2, q_3, q_4, q_{10}, q_{11}, q_{12}, q_{13}\}$
$t_{12}: \ [q_{11} \to q_{12}] : \forall_{L}\{q_1, q_2, q_3, q_4, q_7, q_8\}$
$t_{13}: \ [q_{12} \to q_{13}] \qquad t_{14}: \ [q_{13} \to q_1]$
$Initial\ configurations: \ q_1^*$
$Bad\ configurations: \ \phi = (q_{12}q_{12}, Q^*)$

Fig. 6. Gribomont-Zenner Algorithm.

$States: \ Q = \{invalid, dirty, shared, valid\}$
$Transitions:$
$t_1: \begin{array}{l}[invalid \to shared]\\ [dirty \to shared]\end{array} \quad t_2: \ [invalid \to valid] : \forall\{invalid\}$
$t_3: \begin{array}{l}[invalid \to shared]\\ [valid \to shared]^*\end{array} : \exists\{shared, valid\}$
$t_4: \begin{array}{l}[shared \to dirty]\\ [shared \to invalid]^*\end{array} \quad t_5: \ [valid \to dirty]$
$t_6: \begin{array}{l}[invalid \to dirty]\\ [dirty \to invalid]^*\\ [shared \to invalid]^*\\ [valid \to invalid]^*\end{array} \quad t_7: \ [dirty] \to [invalid]$
$t_8: \ [shared \to invalid] \qquad t_9: \ [valid \to invalid]$
$Initial\ configurations: \ invalid^*$
$Bad\ configurations: \ \phi = (dirty\ dirty, Q^*), (dirty\ shared, Q^*),$
$(shared\ dirty, Q^*)$

Fig. 7. Illinois cache coherence protocol

$States: \ Q = \{invalid, exclusive, shared, dirty\}$
$Transitions:$
$t_1: \begin{array}{l}[invalid \to shared]\\ [dirty \to shared]\end{array}$
$t_2: \ [invalid \to exclusive] : \forall\{invalid\}$
$t_3: \begin{array}{l}[invalid \to shared]\\ [exclusive \to shared]^*\end{array} : \exists\{exclusive, shared\}$
$t_4: \ [exclusive \to dirty]$
$t_5: \ [invalid \to dirty] : \forall\{invalid\}$
$t_6: \ [shared \to exclusive] : \forall\{invalid, dirty, exclusive\}$
$Initial\ configurations: \ invalid^*$
$Bad\ configurations: \ \phi = (dirty\ shared, Q^*), (shared\ dirty, Q^*),$
$(dirty\ exclusive, Q^*), (exclusive\ dirty, Q^*),$
$(exclusive\ exclusive, Q^*)$

Fig. 8. DEC Firefly cache coherence protocol

$States:\ Q = \{zero, one\}$
$Shared\ variables:\ S = \{pmap, check, test \in Bool\}$
$Transitions:$

$p\_alloc/map:$	$[zero \to one]$	$: \neg check, \neg test, pmap'$
$p\_unmap:$	$[one \to zero]$	$: \neg check, \neg test, \neg pmap', check'$
$check_1:$	$[one \to one]$	$: check, \neg check', pmap'$
$check_2:$	$[q \to q]$	$: check, \forall\{zero\}, \neg check', \neg pmap' \quad for\ q \in Q$
$e\_dealloc:$	$[q \to q]$	$: \neg check, \neg test, \neg pmap', test' \quad for\ q \in Q$
$test_1:$	$[one \to zero]$	$: test$
$test_2:$	$[one \to zero]$	$: \forall\{zero\}, test, \neg test', \neg pmap'$

$Initial\ configurations:\ zero^*, \neg pmap, \neg check, \neg test$
$Bad\ configurations:\ \phi = (pmap, zero^*)$

Fig. 9. Abstract model of page reference counting.

$States:\ Q = \{idle, write, read\}$
$Shared\ variables:\ S = \{r, w, lockw \in Bool\}$
$Transitions:$
$read_1:\ [idle \to read]\ : lockw, \neg lockw', r'$

$write_1:\ [write \to write]$ —

$write_1:\ [idle \to write]\ : lockw, \neg lockw', w'$

$$read_2: \begin{matrix}[idle \to read]\\ [read \to read]\end{matrix} : r'$$

$$idle_1: \begin{matrix}[read \to idle]\\ [read \to read]\end{matrix}$$

$idle_2:\ [read \to idle]\ : \forall\{idle, write\}, lockw', \neg r'$

$idle_3:\ [write \to idle]\ : lockw', \neg w'$

$Initial\ configurations:\ idle^*, \neg r, \neg w, lockw$
$Bad\ configurations:\ \phi = (r\ w, Q^*)$

Fig. 10. Readers/writers with locks

which references to PP are removed one-by-one. We use a universal global condition (*no process with state one*) to model the zero-test on the counter. Our model is given in Fig. 9. The model is a generalization of the finite-state model extracted manually using skolemization and predicate abstraction in [19]. Bad configurations correspond to a situation in which the counter is inconsistent with the current state of the process (no references to PP while *pmap* is true).

A.8 Readers/writers

Readers/writers is a classical synchronization problem where an arbitrary number of reader and writer processes synchronize the access to a resource [16]. Readers only read the resource, while writers can also write to it. Several readers are allowed to simultaneously access the resource. However, if a writer and some other process (either a reader or a writer) access the resource simultaneously, the result becomes indeterminable. A solution to the readers/writers problem ensures such configurations do not occur. We consider in the following three pairs of formulations of such solutions, see e.g. [16]. Each pair represents the same solution described either with shared locks or with global tests. The first pair (Fig. 10, Fig. 11) represents the simplest solution. The second pair (Fig. 12, Fig. 13) represents a refined solution in which we introduce a lock for readers used during the checking of the entry condition. Both solutions give priority to readers in the sense that once a reader accesses the resource, a writer will have to wait even for readers that arrive after him but before the resource is released. In contrast, the third pair (Fig. 14, Fig. 15) gives priority to writers. In the former model we consider an additional lock *lockz* for the readers that is used in order to block readers in an intermediate stage when there are already waiting writers in the queue. These models are manually extracted by classical example of readers/writers control policies.

A.9 Light control scheme

In this algorithm (Fig. 16), an arbitrary number of persons may get in or out of an office. Each time a person gets in the office it turns the light on (t_1, t_2, t_3). The variables *empty*, *single* and *many* respectively keep track of whether there are none, one or more persons in the office. The light is turned on if there are several persons in the office (t_4). When persons go out of the office, the variables *single* and *many* are updated (t_5, t_6, t_7) according to their number. Bad configurations are those where some of the variables *empty*, *single* and *many* do

$States: \; Q = \{idle, write, read\}$
$Shared\ variables: \; S = \{r, w \in Bool\}$
$Transitions:$

$read_1 : [idle \rightarrow read] : \forall\{idle\}, r'$

$write_1 : [idle \rightarrow write] : \forall\{idle\}, w'$

$read_2 : \begin{array}{l}[idle \rightarrow read]\\ [read \rightarrow read]\end{array} : r'$

$idle_1 : \begin{array}{l}[read \rightarrow idle]\\ [read \rightarrow read]\end{array}$

$idle_2 : [read \rightarrow idle] : \forall\{idle, write\}, \neg r'$

$idle_3 : [write \rightarrow idle] : \neg w'$

$Initial\ configurations: \; idle^*, \neg r, \neg w$
$Bad\ configurations: \; \phi = (r\ w, Q^*)$

Fig. 11. Readers/writers without locks

$States: \; Q = \{idle, test1, sread, test2, write\}$
$Shared\ variables: \; S = \{r, w, lockr, lockw \in Bool\}$
$Transitions:$
$test_1 : [idle \rightarrow test1] : lockr, \neg lockr'$

$read_1 : [test1 \rightarrow sread] : \forall\{idle, test2, write\}, \neg lockr, lockw, lockr', \neg lockw', r'$

$read_2 : \begin{array}{l}[test1 \rightarrow sread]\\ [sread \rightarrow sread]\end{array} : \neg lockr, lockr', r'$

$test_2 : [sread \rightarrow test2] : lockr, \neg lockr'$

$idle_1 : [test2 \rightarrow idle] : \forall\{idle, test2, write\}, \neg lockr, lockr', \neg lockw, lockw', \neg r'$

$idle_2 : \begin{array}{l}[test2 \rightarrow idle]\\ [sread \rightarrow sread]\end{array} : \neg lockr, lockr', r'$

$write_1 : [idle \rightarrow write] : lockw, \neg lockw', w'$

$idle_3 : [write \rightarrow idle] : \neg lockw, lockw', \neg w'$

$Initial\ configurations: \; idle^*, \neg r, \neg w, lockr, lockw$
$Bad\ configurations: \; \phi = (r\ w, Q^*), (write\ write, Q^*)$

Fig. 12. Refined readers/writers with locks

$States: \; Q = \{idle, test1, sread, test2, write\}$
$Shared\ variables: \; S = \{r, w \in Bool\}$
$Transitions:$
$test_1 : [idle \rightarrow test1] : \forall\{idle, sread, write\}$

$read_1 : [test1 \rightarrow sread] : \forall\{idle, test2, write\}, r'$

$read_2 : \begin{array}{l}[test1 \rightarrow sread]\\ [sread \rightarrow sread]\end{array} : r'$

$test\ 2 : [sread \rightarrow test2] : \forall\{idle, sread, write\}$

$idle_1 : [test2 \rightarrow idle] : \neg r', \forall\{idle, test2, write\}$

$idle_2 : \begin{array}{l}[test2 \rightarrow idle]\\ [sread \rightarrow sread]\end{array} : r'$

$idle_3 : [write \rightarrow idle] : \neg w'$

$write_1 : [idle \rightarrow write] : w', \forall\{idle, test1\}$

$Initial\ configurations: \; idle^*, \neg r, \neg w$
$Bad\ configurations: \; \phi = (r\ w, Q^*), (write\ write, Q^*)$

Fig. 13. Refined readers/writers without using locks

$States:\ Q = \{idle, wait1, wait2, read, waitw, write, release\}$
$Shared\ variables:\ S = \{r, w, lockr, lockw, lockz \in Bool\}$
$Transitions:$

$wait_1 : [idle \rightarrow wait1] : lockz, \neg lockz'$

$wait_2 : [wait1 \rightarrow wait2] : lockr, \neg lockr'$

$read_1 : [wait2 \rightarrow read] : r', lockw, \neg lockw', \neg lockr, lockr', \neg lockz, lockz',$
$\forall\{idle, wait1, wait2, waitw, write, release\}$

$read_2 : \begin{matrix}[wait2 \rightarrow read]\\ [read \rightarrow read]\end{matrix} : r', \neg lockz, lockz', \neg lockr, lockr'$

$idle_1 : [read \rightarrow idle] : \neg r', \neg lockw, lockw', \forall\{idle, wait1, wait2, waitw, write, release\}$

$idle_2 : \begin{matrix}[read \rightarrow idle]\\ [read \rightarrow read]\end{matrix}$

$waitw_1 : [idle \rightarrow waitw] : lockr, \neg lockr', \forall\{idle, wait1, wait2, read\}$

$waitw_2 : \begin{matrix}[idle \rightarrow waitw]\\ [q \rightarrow q]\end{matrix}\ \ for\ q \in \{waitw, write, release\}$

$write : [waitw \rightarrow write] : lockw, \neg lockw, w'$

$idle_3 : [write \rightarrow release] : \neg lockw, lockw', \neg w'$

$idle_4 : [release \rightarrow idle] : \neg lockr, lockr', \forall\{idle, wait1, wait2, read\}$

$idle_5 : \begin{matrix}[release \rightarrow idle]\\ [q \rightarrow q]\end{matrix}\ \ for\ q \in \{waitw, write, release\}$

$Initial\ configurations:\ idle^*, lockr, lockw, lockz, \neg r, \neg w$
$Bad\ configurations:\ \phi = (r\ w, Q^*), (write\ write, Q^*)$

Fig. 14. Refined readers/writers with priority to writers and using locks.

$States:\ Q = \{idle, wait1, wait2, read, waitw, write, release\}$
$Shared\ variables:\ S = \{r, w \in Bool\}$
$Transitions:$

$wait_1 : [idle \rightarrow wait1] : \forall\{idle, read, waitw, write, release\}$

$wait_2 : [wait1 \rightarrow wait2] : \forall\{idle, wait1, read\}$

$read_1 : [wait2 \rightarrow read] : r', \forall\{idle, wait1, wait2, waitw, release\}$

$read_2 : \begin{matrix}[wait2 \rightarrow read]\\ [read \rightarrow read]\end{matrix} : r'$

$idle_1 : [read \rightarrow idle] : \neg r', \forall\{idle, wait1, wait2, waitw, write, release\}$

$idle_2 : \begin{matrix}[read \rightarrow idle]\\ [read \rightarrow read]\end{matrix}$

$waitw_1 : [idle \rightarrow waitw] : \forall\{idle, wait1, read\}$

$waitw_2 : \begin{matrix}[idle \rightarrow waitw]\\ [q \rightarrow q]\end{matrix}\ \ for\ q \in \{waitw, write, release\}$

$write : [waitw \rightarrow write] : w', \forall\{idle, wait1, wait2, waitw, release\}$

$idle_3 : [write \rightarrow release] : \neg w'$

$idle_4 : [release \rightarrow idle]$

$Initial\ configurations:\ idle^*, \neg r, \neg w$
$Bad\ configurations:\ \phi = (r\ w, Q^*), (write\ write, Q^*)$

Fig. 15. Refined readers/writers with priority to writers and no locks.

$States:\ Q = \{zero, one\}$
$Shared\ variables:\ S = \{empty, single, many, light \in Bool\}$
$Transitions:$

$t_1 : [zero \rightarrow one] : empty, \neg empty', single', light'$

$t_2 : [zero \rightarrow one] : single, \neg single', many', light'$

$t_3 : [zero \rightarrow one] : many, light'$

$t_4 : [q \rightarrow q] : many, light' \quad for\ q \in Q$

$$t_5 : \begin{array}{l}[one \rightarrow zero]\\ [one \rightarrow one]\\ [one \rightarrow one]\end{array} : many$$

$$t_6 : \begin{array}{l}[one \rightarrow zero]\\ [one \rightarrow one]\end{array} : many, \neg many', single', \forall\{zero\}$$

$t_7 : [one \rightarrow zero] : single, \neg single', \neg empty', \neg light'$

$Initial\ configurations:\ zero^*, empty, \neg single, \neg many, \neg light$
$Bad\ configurations:\ \phi = (many, zero^*), (single, zero^*),$
$(many, \neg light, Q^*), (single, \neg light, Q^*),$
$(empty, light, Q^*)$

Fig. 16. An office light control system.

not match the number of persons in the office (t_8, t_9), or those where there are persons in the office and the light is turned off (t_{10}, t_{11}), or there is no one in the office and the light is turned on (t_{12}).