Approximated Context-sensitive Analysis for Parameterized Verification

Parosh Aziz Abdulla[1] parosh@it.uu.se,
Giorgio Delzanno[2] giorgio@disi.unige.it, and
Ahmed Rezine[3] rezine.ahmed@liafa.jussieu.fr

[1] Uppsala University, Sweden
[2] Università di Genova, Italy
[3] University of Paris 7, France.

Abstract. We propose a verification method for parameterized systems with global conditions. The method is based on *context-sensitive constraints*, a symbolic representation of infinite sets of configurations defined on top of words over a finite alphabet. We first define context-sensitive constraints for an exact symbolic backward analysis of parameterized systems with global conditions. Since the model is Turing complete, such an analysis is not guaranteed to terminate. To turn the method into a verification algorithm, we introduce context-sensitive constraints that over-approximate the set of backward reachable states and show how to symbolically test entailment and compute predecessors. We apply the resulting algorithm to automatically verify parameterized models for which the exact analysis and other existing verification methods either diverge or return false positives.

1 Introduction

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 processes organized in a linear array. 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.

In [5] we have proposed a light-weight verification method for parameterized systems based on *monotonic abstraction* with the aim of avoiding the use of the full power of automata and regular languages (which require heavy manipulations like the use of transducers [21, 14, 7, 9]). The main idea of the method in [5] is to consider a transition relation that is an over-approximation of the one induced by the parameterized system. To do that, we modify the semantics of universal quantifiers by eliminating the processes that violate the given condition (*downward closed semantics*). The approximate transition system obtained in

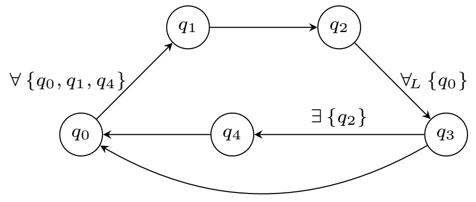

Fig. 1. State diagram of an individual process.

this manner is *monotonic* with respect to the subword relation on configurations (larger configurations are able to simulate smaller ones). Since the approximate transition relation is monotonic, it can be analyzed using a symbolic backward reachability algorithm based on a generic method introduced in [2]. The algorithm operates on sets of configurations that are upward closed with respect to the subword relation and uses symbolic operations that are much simpler than transducers and regular languages. The PFS tool [5] that implements this technique can thus be applied to verify safety properties for configurations with any number of processes. Monotonic abstraction has proven successful in verifying a wide range of parameterized, distributed, and heap manipulating systems [5, 4, 6, 3, 1]. However, it may return false positives due to a loss of precision in the representation of special witness processes. We give an example of a system where such a situation occurs.

An example in which monotonic abstraction may return false positives is the parameterized system where 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. Note that 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. Consider now the abstract transition system computed by applying monotonic abstraction. 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. $q_0q_4q_2q_0 \longrightarrow q_1q_4q_0 \longrightarrow q_2q_4q_0 \longrightarrow q_3q_4q_0 \longrightarrow q_4q_4q_0$. Notice that monotonic abstraction removes the guard (the process in state q_2) since it does not satisfy the global condition of the rule $q_0 \rightarrow q_1 : \forall\{q_0, q_1, q_4\}$. 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.

This kind of false positives arise typically in systems where correctness depends on the existence of a witness process. For this reason, it is relevant to study new approximations that can be used for more precise analysis than that provided by monotonic abstraction. The challenge here is to preserve the positive features of the latter approach such as the use of simple data structures and of a generic verification algorithm based on well-quasi orderings.

New Contribution We propose in this paper a new verification algorithm based on an approximated context sensitive analysis that improves the precision of monotonic abstraction. The method is always guaranteed to terminate, and is based on relatively simple symbolic data structures. We build the verification method in two steps.

We first define a symbolic representation, namely *context-sensitive constraints*, that are a natural generalization of the constraints used in the monotonic abstraction framework. In monotonic abstraction a word w of process states (referred to as the *basis*) is used as a symbolic representation of its upward closure computed with respect to word inclusion. This implies that any type of processes is allowed in between two consecutive states of the basis w (these allowed processes are referred to as *context*). Context-sensitive constraints generalize this idea by introducing constraints on the type of processes that are allowed to occur in each context. For each pair of consecutive states in the basis, constraints are expressed by using a subset R of states: only processes with states in R are allowed in this context. This kind of constraints can be used to exactly represent (one-step) predecessor configurations of a parameterized system with global conditions. An analysis based on this kind of constraints is not guaranteed to terminate in general. Furthermore, when testing in practice, even on simple examples the number of generated constraints often explodes after a few steps. Therefore, approximations are necessary to ensure both theoretical (e.g. using wqo theory) and practical termination (e.g. using more compact representations).

The approximated method we propose in this paper works on constraints of a special form, called *simple* context-sensitive constraints. In a simple context-sensitive constraint we use a single subset of states, called the *padding set*, to over-approximate the constraints on processes in each context. For this new symbolic representation, we have the following properties. The entailment ordering turns out to be a well-quasi ordering. The computation of predecessors is guaranteed to terminate and to return a finite representation of an over-approximation of the exact set of predecessor configurations. Our abstract predecessor operator incorporates accelerations in the computation of predecessors for ordered system that are similar in spirit to widening operators used in the unordered case (as those used in relation analysis for counter systems e.g. in [10, 27, 28]). Finally, the constraint operations are much simpler and more efficient than those used in the exact context-sensitive analysis. Since simple context-sensitive constraints can represent upward closed sets of configuration computed with respect to word inclusion, the resulting over-approximation is guaranteed to be at least as precise as monotonic abstraction. However, in several practical examples it gives more precise results (eliminates false positives).

As a first set of experiments, we have considered benchmark examples of parameterized systems taken from the literature [15, 7, 5, 8]. The performance of the new verification algorithm on this set of examples is comparable with that of the PFS tool based on monotonic abstraction [5]. We remark that in these examples exact analysis often diverges or suffers from the symbolic state explosion problem.

Furthermore, we also consider several new case-studies that include both ordered systems like formulations of Szymanski's algorithm with non-atomic updates (semi-automatically verified in [18, 22, 23]), and unordered concurrent systems like synchronization skeletons [10, 27, 28] and reference counting schemes for virtual memory [16]. For these examples the PFS tool based on monotonic abstraction often returns spurious error traces due to a loss of precision in the representation of *special processes* (as in Szymanski) or in the representation of *counters*. Our new verification algorithm eliminates all the false positives and verifies the new ordered/unordered case studies for any number of processes/unbounded value of counters. We are not aware of other tools that can automatically verify the same class of ordered/unordered parameterized models as our algorithm does.

2 Model

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 $q \to q'$, where the process changes local state from q to q' independently of the local states of the other processes. A global transition is of the form $q \to q' : \mathbb{Q}P$, 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.

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 $c = q_1 q_2 \cdots q_n$, we define $c^\bullet := \{q_1, \ldots, q_n\}$. For configurations $c = c_1 q c_2$, $c' = c_1 q' c_2$, 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 $q \to q'$.
- t is a global transition of the form $q \to q' : \mathbb{Q}P$, and one of the following conditions is satisfied:
  - either $\mathbb{Q}P = \exists_L P$ and ${c_1}^\bullet \cap P \neq \emptyset$, $\mathbb{Q}P = \exists_R P$ and ${c_2}^\bullet \cap P \neq \emptyset$, or $\mathbb{Q}P = \exists_{LR} P$ and $({c_2}^\bullet \cup {c_2}^\bullet) \cap P \neq \emptyset$.
  - either $\mathbb{Q}P = \forall_L P$ and ${c_1}^\bullet \subseteq P$, $\mathbb{Q}P = \forall_R P$ 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 $c \preceq c'$ implies $c' \in U$. For a configuration c, we use $\widehat{c}$ to denote the upward closure of c, i.e., the set $\{c' | \; c \preceq c'\}$. For sets of configurations $D, D' \subseteq C$ we use $D \longrightarrow D'$ to denote that there are $c \in D$ and $c' \in D'$ with $c \longrightarrow c'$. The *coverability problem* for parameterized systems is defined as follows:

PAR-COV
Instance
- A 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. [26]), 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). In Example 1 the set of bad configurations are those in $\widehat{q_4 q_4}$.

3 Exact Context-sensitive Symbolic 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. For the sake of clarity, we first present a simplified version of our constraints and then discuss extensions we use in our implementation. We 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. We use $p, q, \dots$ to denote states in Q, and $P, R, \dots$ 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 $(Rq_3R)^\bullet = \{q_1, q_2, q_3\}$.

Definition 1. *A* c*ontext-sensitive (*CC*-)constraint is a word in $\mathbb{A}^*$ of the form $R_0q_1R_1 \dots q_nR_n$, where $q_i \in Q$ for $i : 1 \le i \le n$ and $R_i \subseteq Q$ for $i : 0 \le i \le n$. The configuration $q_1 \dots q_n$ 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 $c_0q_1c_1 \dots q_nc_n$ where $c_i \in R_i^*$ for $i : 0 \le i \le n$.*

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 $R_0q_1R_1q_2R_2$ denotes all configurations of the form $c_0q_1c_1q_2c_2$ 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 $q_3q_1q_3q_2q_1$ and $q_3q_1q_3q_3q_2q_1$ belong to $[\![\phi]\!]$, whereas $q_1q_1q_2$ and $q_1q_3q_2q_2$ do not belong to $[\![\phi]\!]$. Notice that CC's of the form $Qq_1Q \dots q_nQ$ denote upward closed sets of states with respect to word inclusion (there are no constraints on the contexts). For instance, the set of bad states in Example 1 can be characterized by the CC Qq_4Qq_4Q where $Q = \{q_0, q_1, q_2, q_3, q_4\}$.

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

Entailment. For constraints $\phi = R_0q_1 \dots q_nR_n$ and $\phi' = R'_0q'_1 \dots q'_mR'_m$, 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 $i : 1 \le i \le n$ (the basis of ϕ is a subword of the basis of ϕ') and the following conditions hold:

- $(R'_0q'_1 \dots q'_{h(1)-1}R'_{h(1)-1})^\bullet \subseteq R_0^\bullet$
- $(R'_{h(i)}q'_{h(i)+1} \dots q'_{h(i+1)-1}R'_{h(i+1)-1})^\bullet \subseteq R_i^\bullet$ for $i : 1 \le i \le n-1$;
- $(R'_{h(n)}q'_{h(n)+1} \dots q'_mR'_m)^\bullet \subseteq R_n^\bullet$.

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 $[\![S']\!]$ is the set of configurations from which one can reach configurations in $[\![S]\!]$ using $\xrightarrow{*}$ (i.e., predecessors).

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 $\{\phi' \mid \phi \leadsto_t \phi'\}$ where $\leadsto_t$ is the minimal relation that satisfies one of the following conditions. Let $\phi = R_0 q_1 \dots q_n R_n$:

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

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 $\phi = R_0 q_1 \dots q_n R_n$. Consider a transition t of the form $q \to q' \ : \ \forall_{LR} P$ with $P \subseteq Q$. Then, $\leadsto_t$ is the minimal relation that satisfies one of 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 $\phi \ \leadsto_t \ (R_0 \cap P) q_1 \dots q_{i-1} (R_{i-1} \cap P) q (R_i \cap P) q_{i+1} \dots q_n (R_n \cap P)$.
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 $\phi \leadsto_t (R_0 \cap P) q_1 \dots q_i (R_i \cap P) q (R_i \cap P) q_{i+1} \dots q_n (R_n \cap P)$.

Local Let t be a local rule $q \to q'$, $\leadsto_t$ is the minimal relation that satisfies one of the following conditions:

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

Exist. Let t be the rule $q \to q' \ : \ \exists_L P$, $\leadsto_t$ is the minimal relation that satisfies one of the following conditions:

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

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

Symbolic Backward Reachability Context expressions can be used for an exact representation of predecessor configurations. Each application of Pre is effectively computable. Let Φ_0 be a set of CC's that represent an upward closed set of configurations (unsafe states). Starting from Φ_0, we compute the sequence of sets of CC's-constraints $\Phi_0, \ldots, \Phi_i, \ldots$ such that $\Phi_{i+1} = \Phi_i \cup \bigcup_{t \in \mathcal{T}, \phi \in \Phi_i} Pre_t(\phi)$. Each step of this sequence can be effectively computed. Furthermore, we can apply the entailment $\sqsubseteq$ to discharge CC's that do not add new information (i.e. stronger than an already computed constraint). If we reach a fixpoint at step k, then Φ_k gives us an exact representation of the predecessors of configurations in $[\![\Phi_0]\!]$. Thus, we can potentially use this fixpoint computation to solve PAR-COV, i.e., to verify/falsify safety properties for configurations of arbitrary size. However, since our model is Turing complete (e.g. we can encode two counter machines using universally quantified conditions) the resulting CC's-based symbolic backward reachability analysis is not guaranteed to terminate. Therefore, in order to obtain a terminating verification procedure, we need to introduce some approximation. We discuss this point in the next section.

4 Approximated Context-sensitive Symbolic Algorithm

In this section, we present an approximated representation of context-sensitive constraints that we use to turn the (possibly non-terminating) CC-based verification procedure into an approximated verification algorithm. For this purpose, we first define a special class of constraints.

Definition 2. *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 in included is the padding set has two consequences: it allows us to apply the theory of well-quasi ordering to ensure termination of the backward analysis (see Lemma 1); it gives us a natural way to define *accelerations* to speed up the symbolic computation of predecessors (see Section 4.1). 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 $\phi = R_0 q_0 \ldots q_n R_n$ can naturally be approximated by the following SCC:

$$\phi^\# = (q_0 \ldots q_n, \phi^\bullet)$$

Indeed, it is immediate to check that $[\![\phi]\!] \subseteq [\![\phi^\#]\!]$. 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, we can prove that $\sqsubseteq$ is a *Well Quasi-Ordering (WQO)* for SCC's, i.e., for any infinite sequence $\phi_0, \phi_1, \phi_2, \ldots$, of constraints, there are $i < j$ 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 $\phi_{i_0}, \phi_{i_1}, \phi_{i_2}, \ldots$ such that $R_{i_j} = R_{i_k}$ for all j, k. By Higman's lemma [20] (which implies that $\preceq$ is a WQO on Q^*), there are $j < k$ such that $c_{i_j} \preceq c_{i_k}$, and hence $\phi_{i_j} \sqsubseteq \phi_{i_k}$. This gives the following lemma which we use later to prove termination of our algorithm.

Lemma 1. *$\sqsubseteq$ is a WQO on the set of SCC's.*

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 states in Example 1 can be characterized by the SCC's $(q_4q_4, \{q_0, q_1, q_2, q_3, q_4\})$. Also, for the SCC $\phi = (pq, \{p, q, r, s\})$ and $\phi' = (qpprqp, \{p, q, r\})$, we have $\phi \sqsubseteq \phi'$.

4.1 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). As we discuss in Section 5, 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 $\{c_1qc_2 \mid c = c_1c_2\}$. The operation adds the singleton q in an arbitrary position inside c. We define $Pre^{\#}$ by means of a set of relations $\overset{t}{\leadsto}$ 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 $q \rightarrow q'$ then

- $(c_1q'c_2, R) \overset{t}{\leadsto} (c_1qc_2, R \cup \{q\})$.
- $(c, R) \overset{t}{\leadsto} (c_1, R \cup \{q\})$ 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 [15]). To illustrate this, consider the rule $p \rightarrow q$ 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 $q \rightarrow q' : \exists_L P$ then

- $(c_1 q' c_2, R) \overset{t}{\rightsquigarrow} (c_1 q c_2, R \cup \{q\})$ if $P \cap c_1{}^{\bullet} \neq \emptyset$.
- $(c_1 q' c_2, R) \overset{t}{\rightsquigarrow} (c_3 q c_2, R \cup \{q\})$ if $p \in P \cap R$, $p \notin c_1{}^{\bullet}$, $c_3 \in (c_1 \otimes p)$.
- $(c_1 p c_2, R) \overset{t}{\rightsquigarrow} (c_1 p c_3, R \cup \{q\})$ if $p \in P$, $q' \in R$, $q \notin R$, and $c_3 \in (c_2 \otimes q)$.
- $(c_1 c_2, R) \overset{t}{\rightsquigarrow} (c_1 p c_3, R \cup \{q\})$ if $p \in P$, $p \notin c_1{}^{\bullet}$, $q' \in R$, $q \notin R$, 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 process making the transition explicitly in the basis. In a similar manner to the local transition case, we add q to the padding to reflect the abstraction.
If t is a global transition of the form $q \rightarrow q' : \exists_R P$ or $q \rightarrow q' : \exists_{LR} P$, then analogous conditions to the previous case hold.

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

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

In the first case, a process in the basis moves from q to q'. The remaining processes in the basis must be in R. Furthermore, we restrict the padding set to those processes within R. 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 to reflect the abstraction.
If t is a global transition of the form $q \rightarrow q' : \forall_L P$, then

- $(c_1 q' c_2, R) \overset{t}{\rightsquigarrow} (c_1 q c_2, R \cup \{q\})$, if $c_1{}^{\bullet} \subseteq P$.
- $(c_1 c_2, R) \overset{t}{\rightsquigarrow} (c_1 q c_2, R \cup \{q\})$, 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'. Notice that in both cases, the state q is added to the padding to reflect the abstraction. 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 $q \rightarrow q' : \forall_R P$, then analogous conditions to the previous case hold.

Now let $\rightsquigarrow := \bigcup_{t \in T} \overset{t}{\rightsquigarrow}$ and define for a constraint ϕ the set $(\phi \rightsquigarrow) := \{\phi' | \ \phi \rightsquigarrow \phi'\}$.

Lemma 2. *For any constraint ϕ, we have $pre(\llbracket \phi \rrbracket) \subseteq \llbracket (\phi \rightsquigarrow) \rrbracket = \llbracket Pre^{\#}(\phi) \rrbracket$.*

Backward Reachability Algorithm We use the relation $\rightsquigarrow$ to define a symbolic backward reachability algorithm for approximating solutions to PAR-COV. We start with a finite set $\varPhi_F$ of SCC's denoting $\widehat{C_F}$ (notice that we can always define SCC's that describe an upward-closed set). We generate a sequence $\varPhi_0 \sqsupseteq \varPhi_1 \sqsupseteq \varPhi_2 \sqsupseteq \cdots$ of finite sets of constraints such that $\varPhi_0 = \varPhi_F$, and $\varPhi_{j+1} = \varPhi_j \cup (\varPhi_j \rightsquigarrow)$. Since $\llbracket \varPhi_0 \rrbracket \subseteq \llbracket \varPhi_1 \rrbracket \subseteq \llbracket \varPhi_2 \rrbracket \subseteq \cdots$, the procedure terminates when we reach a point j where $\varPhi_j \sqsubseteq \varPhi_{j+1}$. Thus, termination of the algorithm is guaranteed by Lemma 1. Notice that the termination condition implies that $\llbracket \varPhi_j \rrbracket = (\bigcup_{0 \le i \le j} \llbracket \varPhi_i \rrbracket)$. By Lemmas 2, $\varPhi_j$ denotes an over-approximation of the set of all predecessors of $\llbracket \varPhi_F \rrbracket$. This means that if $(Init \bigcap \llbracket \varPhi_j \rrbracket) = \emptyset$, then there exists no $c \in \llbracket \varPhi_F \rrbracket$ with $Init \xrightarrow{*} c$. Thus, the algorithm can be used as a semi-test for checking PAR-COV.

Extensions We discuss here possible extensions of the symbolic representation and of the model. The basic form of SCC's can be enriched in order to provide a more compact representation of sets of configurations. More specifically, as in [5], let us assume that individual processes have a state in Q and a set of local Boolean variables in V. Let $\mathbb{B}$ be the set of Boolean formulas with predicates in $Q \cup V$. We can work on CC-constraints of the form $R_0 b_0 R_1 \ldots b_n R_n$ ($(b_0, \ldots, b_n, R)$ for SCC-constraints) where b_i is a formula in $\mathbb{B}$ and R_i (R) is a subset of formulas in $\mathbb{B}$. Now the basis describes a finite set of configurations with n processes and each set R_i gives constraints either on the state or on the local variables for processes occurring in the context. Furthermore, we can extend the exact/approximated symbolic computation of predecessors to rules with other synchronization mechanisms like *broadcast* communication and read/write operations on globally shared variables either with range in a finite domain or in the natural numbers. Operations on the latter type of shared variables can be obtained by using synchronization with special processes with state *zero*/*one*: increment is modelled 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 modelled 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.

Tool	Method	Approximation	Precision	Termination
CC	backward reach.	none	exact	not guaranteed
SCC	backward reach.	abstraction of CC's	over-approx	always guaranteed
PFS	backward reach.	monotonic abst.	over-approx	always guaranteed

Table 1. Methods and tools listed in order of precision in the analysis.

5 Experimental Results

We have implemented the verification procedures based on CC and SCC (see Table 1) and compared them to PFS (monotonic abstraction). To this purpose, we used examples of cache coherence protocols, mutual exclusion algorithms, and counter based synchronization problems. In the following, we briefly discuss some of the case studies (we include a detailed description in the appendix) and introduce the behavior of the verification procedures on them.

The examples consist of the Illinois and the DEC Firefly cache coherence protocols from [15]; the Bakery and Burns mutual exclusion algorithms used in [5]; a compact model of Szymanski algorithm with atomicity conditions from [8, 25], a refinement of Szymanski algorithm from [23] (see Fig. 2), and the Gribomont-Zenner mutex from [18]. 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 [16], and solutions to the readers/writers problem from [27] with priorities to readers or writers. The results are summarized in Table 2. For each example, we give the number of iterations performed by the reachability algorithm, the number of constraints upon termination of the algorithm, and the time (in seconds or minutes). We use – in the appropriate fields to indicate that we had to stop the analysis after several hours.

Model	Method	# iter	# constr	ex-time	spurious trace	verified
Bakery [5]	PFS	2	2	0.01s		√
	CC	4	3	0.01s		√
	SCC	3	2	0.01s		√
Illinois [15]	PFS	5	33	0.02s		√
	CC	2	17	0.05s		√
	SCC	7	53	0.18s		√
Burns [5]	PFS	14	40	0.05s		√
	CC	–	–	–	–	–
	SCC	15	48	0.02s		√
DEC Firefly [15]	PFS	3	11	0.01s		√
	CC	–	–	–	–	–
	SCC	5	10	0.03s		√
Compact Szymanski [8, 25]	PFS	10	17	0.1s		√
	CC	–	–	–	–	–
	SCC	24	162	3.35s		√
Refined Szymanski [23]	PFS	24	658	1.5 s	√	
	CC	–	–	–	–	–
	SCC	34	641	1m		√
Gribomont-Zenner [18]	PFS	36	197	0.2 s	√	
	CC	–	–	–	–	–
	SCC	56	863	5m		√
Ref. counting [16]	PFS	7	15	0.02s	√	

	CC	--	--	--	--	--
	SCC	7	8	0.01s		√
Readers/writers[27] (locks:no locks)	PFS	(10:5)	(31:28)	(0.05s:0.02s)	(:√)	(√:)
	CC	--	--	--	--	--
	SCC	(7:7)	(12:8)	(0.02s:0.01s)		(√:√)
Readers/writers (locks:no locks) refined, priority to readers	PFS	(21:7)	(125:67)	(0.4s:0.6s)	(:√)	(√:)
	CC	--	--	--	--	--
	SCC	(25:12)	(128:34)	(1.7s:0.06s)		(√:√)
Readers/writers (locks:no locks) refined, priority to writers	PFS	(22:9)	(683:219)	(9.4s:0.3s)	(:√)	(√:)
	CC	--	--	--	--	--
	SCC	(27:9)	(646:19)	(17.2s:0.03s)		(√:√)
Light control [27]	PFS	13	96	0.06s	√	
	CC	--	--	--	--	--
	SCC	9	29	0.02s		√

Table 2: Experimental results.

In the simplest examples like the Bakery algorithm, each one of the three methods (PFS,SCC,CC) automatically verifies mutual exclusion. However, 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 be more than one dirty cache line for the same memory location) in the DEC Firefly model of [15]. A similar behavior was already observed with HyTech [19] (a tool manipulating polyhedra that can be used for unordered models) in [15]. In the more complicated examples like the algorithms of Burns and Szymanski exact analysis does not terminate.

```
var flag : array[N] of [0 − 4]
flag := (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 : $Q = \{s_0, s_1, \ldots, s_{11}\}$
Transitions :

instruction :	*transition*
1 :	$s_0 \to s_1$
2 :	$s_1 \to s_2$
3 :	$s_2 \to s_3 : \forall_{LR}\{s_0, s_1, s_2, s_3, s_7, s_8\}$
4 :	$s_3 \to s_4$
5 *then* :	$s_4 \to s_6 : \exists_{LR}\{s_2, s_3\}$
6 :	$s_6 \to s_7$
7 :	$s_7 \to s_8 : \exists_{LR}\{s_9, s_{10}, s_{11}\}$
8 :	$s_8 \to s_9$
5 *else* :	$s_4 \to s_5 : \forall_{LR}\neg\{s_2, s_3\}$
9 :	$s_5 \to s_9$
10 :	$s_9 \to s_{10} : \forall_{L}\{s_0, s_1, s_2, s_3\}$
11 :	$s_{10} \to s_{11} : \forall_{R}\neg\{s_4, s_5, s_6, s_7, s_8\}$
12 :	$s_{11} \to s_0$

Initial state : s_0
Bad states : $\phi = (s_{10}s_{10}, Q^*)$

Fig. 2. Algorithm of Szymanski [23] (left), and its parameterized model (right).

Monotonic abstraction proved to be precise for a wide range of parameterized systems [5, 4, 6, 3, 1]. However, it returned false positives for some of the protocols in Table 2. These are the fine grained formulations of Szymanski algorithm, the reference counting model, and particular versions of readers/writers. The main steps of the spurious error trace returned by PFS (monotonic abstraction) on the algorithm of Fig. 2 are described below.

$$(s_0, s_0, s_0) \to^* (s_1, s_1, s_1) \to^* (s_1, s_1, s_3) \to (s_2, s_1, s_3) \to (s_3, s_1, s_3) \to (s_3, s_1, s_4)$$
$$\to^* (s_5, s_2, s_4) \to (s_9, s_2, s_4) \to^* (s_9, s_2, s_7) \longrightarrow_0 (s_3, s_7) \to^* (s_{10}, s_{10})$$

The step indicated with $\longrightarrow_0$ 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 the denotation of the constraints manipulated by PFS always contain all processes (constraints are upward closed sets in monotonic abstraction). When applied to this model, the approximated SCC-based algorithm terminates without detecting error traces, i.e., mutual exclusion is verified for the refined model for any number of processes. Notice that the compact model studied in [8, 25], can be verified using both PFS and SCC.

6 Conclusions and Related Work

We have presented a new algorithm for parameterized verification based on special constraints, called SCC's, that retain approximated context-sensitive information on the type of processes executing in parallel with a finite set of completely specified individuals. We apply the new algorithm to several non-trivial examples in which other types of analysis fail (i.e. either diverge or give false positives). Furthermore, the new algorithm performs well on most of the examples that can be verified with existing parameterized verification techniques. In this paper, we consider protocols where variable updates are non-atomic. On the other hand, we consider in [6] models where global conditions are performed non-atomically. 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 on both variable updates and global conditions.

Related Work The constraints used for the exact analysis are similar to the APC regular expressions studied in [12]. The verification method proposed in [12] 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 [8, 24] and the *environment abstraction* method [13, 25]. In [25] environment abstraction is applied to a formulation of Szymanski with the same assumptions as the model in [8], called *compact Szymanski* in Table 2. This model can be verified using monotonic abstraction as discussed in Section 5. The refined model [23] 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 [8, 13].

The infinite-state reference counting example we consider in this paper is inspired by a finite-state abstraction studied in [16]: in contrast to the predicate-abstraction approach used in [16], 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 modelled with systems working on unbounded integer variables such as in ALV [27, 28] (based on the Omega library) and HyTech [19] (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 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.

References

1. P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziz, and A. Rezine. Monotonic abstraction for programs with dynamic memory heaps. In *Proc.* 20th *Int. Conf. on Computer Aided Verification*, 2008.
2. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96,* 11th *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.
3. P. A. Abdulla, G. Delzanno, F. Haziza, and A. Rezine. Parameterized tree systems. In *Proc. FORTE '08,* 28st *International Conference on Formal Techniques for Networked and Distributed Systems*, volume 5049 of *Lecture Notes in Computer Science*, pages 69–83. Springer Verlag, 2008.
4. 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.
5. 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.
6. 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.
7. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002,* 13th *Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.
8. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In Berry, Comon, and Finkel, editors, *Proc.* 13th *Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234, 2001.

9. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc. 15th Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.
10. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables. *ACM Trans. on Programming Languages and Systems*, 21(4):747–789, 1999.
11. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. CAV 2004: 372–386.
12. A. Bouajjani, A. Muscholl, and T. Touili. Permutation Rewriting and Algorithmic Verification. Inf. and Comp., 205(2): 199-224, 2007.
13. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. VMCAI '06, 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141, 2006.
14. D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, 2001.
15. G. Delzanno. Constraint-Based Verification of Parameterized Cache Coherence Protocols. FMSD 23(3): 257-301 (2003)
16. M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counted objects. To appear in TACAS 2009.
17. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! TCS 256(1-2):63–92, 2001.
18. E. Gribomont and G. Zenner. Automated verification of Szymanski's algorithm. In *Proc. TACAS '98, 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 424–438, 1998.
19. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. STTT 1:110-122, 1997.
20. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7):326–336, 1952.
21. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.
22. Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. chang, M. Colón, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STEP: the Stanford Temporal Prover. Draft Manuscript, June 1994.
23. Z. Manna and A. Pnueli. An exercise in the verification of multi – process programs. In W. Feijen, A. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business*, pages 289–301. Springer Verlag, 1990.
24. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. TACAS '01, 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 82–97, 2001.
25. M. Talupur. *Abstraction techniques for parameterized verification.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2006. Adviser-Edmund M. Clarke.
26. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86, 1st IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.
27. T. Yavuz-Kahveci, T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. STTT 5(1): 15-33, 2003.

28. T. Yavuz-Kahveci, T. Bultan. Verification of parameterized hierarchical state machines using action language verifier. MEMOCODE 2005: 79-88

$States:\ Q = \{q_1, q_2, q_3\}$
$Shared\ variables:\ S = \{bad \in Bool\}$
$Transitions:$
$t_1:\ [q_1 \to q_2] : \forall_R\{q_1\} \qquad t_2:\ [q_2 \to q_3] : \forall_L\{q_1\} \qquad t_3:\ [q_3 \to q_1]$
$t_4:\ \begin{matrix}[q_3 \to \cdot]\\ [q_3 \to \cdot]\end{matrix} : bad'$
$Initial\ state:\ q_1^*, \neg bad$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 3. Bakery algorithm

A Examples

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.

Mutual exclusion algorithms We briefly describe the mutual exclusion algorithms of Bakery, Burns, Szymanski, and Griboment-Zenner.

Bakery mutex In this algorithm, processes have states that range over $\{q_1, q_2, q_3\}$. A process gets a ticket with the highest number in the queue (transition t_1). A process accesses the critical section if it has a ticket with the lowest nulber 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.

Burns mutex In this algorithm, 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 return to q_1 (transitions $t_2, t5$), otherwise, it continues (transitions t_3, t_6). Once at t_6, all processes access the critical section (q_7) with priority to the right most processes. Mutual exclusion is violated in case more than one process is at state $(q_7, .)$.

Compact version of Szymanski mutex In this algorithm 5, processes have states that range over $\{q_0, \ldots q_7\}$. The state q_0 is inital, and q_7 is the critical section. Once processes take transition t_2, they are ensured to access the critical section. At transition t_5, processes go to state q_4 where they wait for processes at state q_1, q_2. Once a process is at state q_5, no other process can cross the doorstep (t_2), and all processes waiting at state q_4 can get to state q_5. Once all process that crossed the door step 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.

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

$t_1 : [q_1 \to q_2, \neg f]$	$t_2 : [q_2 \to q_1] : \exists_L\{f\}$	$t_3 : [q_2 \to q_1] : \forall_L\{\neg f\}$
$t_4 : [q_3 \to q_4, f]$	$t_5 : [q_4 \to q_1] : \exists_L\{f\}$	$t_6 : [q_4 \to q_5] : \forall_L\{\neg f\}$
$t_7 : [q_5 \to q_6] : \forall_R\{\neg f\}$	$t_8 : [q_6 \to q_7, \neg f]$	$t_9 : [q_7 \to q_1]$
$t_{10} : \begin{matrix}[q_7 \to .]\\ [q_7 \to .]\end{matrix} : bad'$		

$Initial\ state:\ (q_1, \neg f)^*, \neg bad$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 4. Burns algorithm.

$States:\ Q = \{q_0, \dots q_7\}$
$Shared\ variables:\ S = \{bad \in Bool\}$
$Transitions:$

$t_1 : [q_0 \to q_1]$	$t_2 : [q_1 \to q_2] : \forall_L\{q_0, q_1, q_2, q_4\}$	$t_3 : [q_2 \to q_3]$
$t_4 : [q_3 \to q_4] : \exists_{LR}\{q_1, q_2\}$	$t_5 : [q_3 \to q_5] : \forall_{LR}\{q_0, q_3, q_4, q_5, q_6, q_7\}$	
$t_6 : [q_4 \to q_5] : \exists_{LR}\{q_5, q_6, q_7\}$	$t_7 : [q_5 \to q_6] : \forall_{LR}\{q_0, q_1, q_2, q_5, q_6, q_7\}$	
$t_8 : [q_6 \to q_7] : \forall_L\{q_0, q_1, q_2\}$	$t_9 : [q_7 \to q_0]$	
$t_{10} : \begin{matrix}[q_7 \to \cdot]\\ [q_7 \to \cdot]\end{matrix} : bad'$		

$Initial\ state:\ q_0^*, \neg bad$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 5. Compact Szymanski Algorithm.

Griboment-Zenner Mutex This algorithm (Fig.6) could be seen as a version of Szymanski algorithm (Fig.5), with transitions that are finer grained in the sens 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 and q_{12} as state of process at the critical section. Configurations not satisfying mutual exclusion are those where at least two processes are at state q_{12}.

Cache coherence protocols In the following models of cache coherence protocols, we restrict ourselves to a single cache line. We aim at veriying cache coherence for any number of caches.

Illinois cache coherence In this protocol (Fig. 7), a cache 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 $t_7, t_8, t9$ 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 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*. Caches are not coherent if a cache in state dirty coexists with another cache in state dirty or shared.

DEC Firefly cache coherence In this proctocl (Fig. 8), caches range over states *invalid*, *exclusive*, *shared* and *dirty*. Transitions t_1, t_2 and t_3 correspond to read

$States: \ Q = \{q_1, \ldots q_{13}\}$
$Shared\ variables: \ S = \{bad \in Bool\}$
$Transitions:$
$t_1: \ [q_1 \rightarrow q_2] \qquad t_2: \ [q_2 \rightarrow q_3]$
$t_3: \ [q_3 \rightarrow q_4] : \forall_L \{q_1, q_2, q_3, q_4, q_7, q_8\}$
$t_4: \ [q_4 \rightarrow q_5]$
$t_5: \ [q_5 \rightarrow q_6] : \exists_{LR} \{q_3, q_4, q_{10}, q_{11}, q_{12}, q_{13}\}$
$t_6: \ [q_6 \rightarrow q_7]$
$t_7: \ [q_7 \rightarrow q_8] : \exists_{LR} \{q_{10}, q_{11}, q_{12}, q_{13}\}$
$t_8: \ [q_8 \rightarrow q_9]$
$t_9: \ [q_5 \rightarrow q_9] : \forall_L \{q_1, q_2, q_5, q_6, q_7, q_8, q_9\}$
$t_{10}: \ [q_9 \rightarrow q_{10}] \qquad t_{11}: \ [q_{10} \rightarrow q_{11}] : \forall_L \{q_1, q_2, q_3, q_4, q_7, q_8\}$
$t_{12}: \ [q_{12} \rightarrow q_{13}] \qquad t_{13}: \ [q_{13} \rightarrow q_1]$
$t_{14}: \ \begin{array}{l} [q_{12} \rightarrow \cdot] \\ [q_{12} \rightarrow \cdot] \end{array} : bad'$
$Initial\ state: \ q_1^*, \neg bad$
$Bad\ states: \ \phi = (bad, Q^*)$

Fig. 6. Gribomont-Zenner Algorithm.

misses, t_4, t_6 to write hits, and t_5 to a write miss. Caches are not coherent if a *dirty* cache coexists with a *shared* a *dirty* or an *exclusive* cache, or if more than one cache are in state *exclusive* (detected with transitions t_7, t_8 and t_9).

Counter based synchronization schemes 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 [16].

A reference counting model We consider (Fig 9) an abstract (but still infinite-state) model for the page manager (pmap) described in [16]. 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 modelled as a collection of processes with two states: *zero* and *one*. The current value of 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 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 [16]. Bad configurations are those where the variable *bad* is true. These 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).

$States:\ Q = \{invalid, dirty, shared, valid\}$
$Shared\ variables:\ S = \{bad \in Bool\}$
$Transitions:$

$$t_1 : \begin{array}{l}[invalid \rightarrow shared]\\ [dirty \rightarrow shared]\end{array} \quad t_2 : [invalid \rightarrow valid] : \forall\{invalid\}$$

$$t_3 : \begin{array}{l}[invalid \rightarrow shared]\\ [valid \rightarrow shared]^*\end{array} : \exists\{shared, valid\}$$

$$t_4 : \begin{array}{l}[shared \rightarrow dirty]\\ [shared \rightarrow invalid]^*\end{array}$$

$$t_5 : [valid \rightarrow dirty]$$

$$t_6 : \begin{array}{l}[invalid \rightarrow dirty]\\ [dirty \rightarrow invalid]^*\\ [shared \rightarrow invalid]^*\\ [valid \rightarrow invalid]^*\end{array}$$

$$t_7 : [dirty] \rightarrow [invalid] \quad t_8 : [shared \rightarrow invalid] \quad t_9 : [valid \rightarrow invalid]$$

$$t_{10} : \begin{array}{l}[dirty \rightarrow \cdot]\\ [dirty \rightarrow \cdot]\end{array} : bad' \quad t_{11} : \begin{array}{l}[shared \rightarrow \cdot]\\ [dirty \rightarrow \cdot]\end{array} : bad'$$

$Initial\ state:\ invalid^*, bad = false$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 7. Illinois cache coherence protocol

Readers/writers is a classical synchronization problem where an arbitrary number of reader and writer processes synchronize the access to a resource. 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. Each pair represent the same solution described either with shared locks or with global tests. The first pair (Fig. 10, Fig. 11) represent the simplest solution. The second pair (Fig. 12, Fig. 13) represent a refined solution. Both solutions give priority to readers in the sense that once a reader access 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.

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 wether there no, one or more persons are 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 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}).

$States:\ Q = \{invalid, exclusive, shared, dirty\}$
$Shared\ variables:\ S = \{bad \in Bool\}$
$Transitions:$

$$t_1: \begin{array}{l}[invalid \rightarrow shared]\\ [dirty \rightarrow shared]\end{array}$$

$$t_2: [invalid \rightarrow exclusive]\ : \forall\{invalid\}$$

$$t_3: \begin{array}{l}[invalid \rightarrow shared]\\ [exclusive \rightarrow shared]^*\end{array} : \exists\{exclusive, shared\}$$

$$t_4: [exclusive \rightarrow dirty]$$

$$t_5: \begin{array}{l}[invalid \rightarrow dirty]\\ [exclusive \rightarrow invalid]^*\\ [shared \rightarrow invalid]^*\\ [dirty \rightarrow invalid]^*\end{array}$$

$$t_6: [shared \rightarrow exclusive]\ : \forall\{invalid, dirty, exclusive\}$$

$$t_7: \begin{array}{l}[dirty \rightarrow .]\\ [shared \rightarrow .]\end{array} : bad' \quad t_8: \begin{array}{l}[dirty \rightarrow .]\\ [exclusive \rightarrow .]\end{array} : bad' \quad t_9: \begin{array}{l}[exclusive \rightarrow .]\\ [exclusive \rightarrow .]\end{array} : bad'$$

$Initial\ state:\ invalid^*, bad = false$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 8. DEC Firefly cache coherence protocol

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

$p\_alloc/map:$	$[zero \rightarrow one]$	$: \neg check, \neg test, pmap' = true$
$p\_unmap:$	$[one \rightarrow zero]$	$: \neg check, \neg test, pmap' = false, check' = true$
$check_1:$	$[one \rightarrow one]$	$: check, check' = false, pmap' = true$
$check_2:$	$[\cdot \rightarrow \cdot]$	$: check, \forall\{zero\}, check' = false, pmap' = false$
$e\_dealloc:$	$[\cdot \rightarrow \cdot]$	$: \neg check, \neg test, pmap' = false, test' = true$
$test_1:$	$[one \rightarrow zero]$	$: test$
$test_2:$	$[one \rightarrow zero]$	$: \forall\{zero\}, test, test' = false, pmap' = false$
$bad:$	$[\cdot \rightarrow \cdot]$	$: \forall\{zero\}, pmap, bad' = true$

$Initial\ state:\ zero^*, pmap = false, check = false, test = false$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 9. Abstract model of page reference counting.

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

$$read_1: [idle \rightarrow read]\ : lockw, \neg lockw', r'$$

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

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

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

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

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

$$bad_1: [\cdot \rightarrow \cdot]\ : r, w, bad'$$

$Initial\ state:\ idle^*, \neg bad$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 10. Readers/writers with locks

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

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

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

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

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

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

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

$bad_1 : [\cdot \rightarrow \cdot] : r, w, bad'$

$Initial\ state:\ idle^*, \neg bad$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 11. Readers/writers without locks

$States:\ Q = \{idle, test1, sread, test2, write\}$
$Shared\ variables:\ S = \{r, w, lockr, lockw, bad \in Bool\}$
$Transitions:$

$test_1 : [idle \rightarrow test1] : lockr, \neg lockr'$

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

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

$read_3 : \begin{matrix}[test1 \rightarrow sread]\\ [sread \rightarrow sread]\end{matrix} : \neg lockr, lockr', r'$

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

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

$idle_2 : \begin{matrix}[test2 \rightarrow idle]\\ [test1 \rightarrow test1]\end{matrix} : \neg lockr, lockr', r'$

$idle_3 : \begin{matrix}[test2 \rightarrow idle]\\ [sread \rightarrow sread]\end{matrix} : \neg lockr, lockr', r'$

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

$idle_4 : [write \rightarrow idle] : \neg lockw, lockw', \neg w'$

$bad\ 1 : [\cdot \rightarrow \cdot] : r, w, bad'$

$bad\ 2 : \begin{matrix}[write \rightarrow \cdot]\\ [write \rightarrow \cdot]\end{matrix} : bad'$

$Initial\ state:\ idle^*, \neg bad$
$Bad\ states:\ \phi = (bad, Q^*)$

Fig. 12. Refined readers/writers with locks

$States: \; Q = \{idle, test1, sread, test2, write\}$

$Shared \; variables: \; S = \{r, w, bad \in Bool\}$

$Transitions:$

$$test_1 : [idle \to test1] : \forall\{idle, sread, write\}$$

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

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

$$read_3 : \begin{array}{l}[test1 \to sread]\\ [sread \to sread]\end{array} : r'$$

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

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

$$idle_2 : \begin{array}{l}[test2 \to idle]\\ [test1 \to test1]\end{array} : r'$$

$$idle_3 : \begin{array}{l}[test2 \to idle]\\ [sread \to sread]\end{array} : r'$$

$$idle_4 : [write \to idle] : \neg w'$$

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

$$bad_1 : [\cdot \to \cdot] : r, w, bad'$$

$$bad_2 : \begin{array}{l}[write \to \cdot]\\ [write \to \cdot]\end{array} : bad'$$

$Initial \; state: \; idle^*, \neg bad$

$Bad \; states: \; \phi = (bad, Q^*)$

Fig. 13. Refined readers/writers without using locks

$States: \; Q = \{idle, wait1, wait2, read, waitw, write, release\}$

$Shared \; variables: \; S = \{r, w, lockr, lockw, lockz, bad \in Bool\}$

$Transitions:$

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

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

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

$$read_2 : \begin{array}{l}[wait2 \to read]\\ [read \to read]\end{array} : r', \neg lockz, lockz', \neg lockr, lockr'$$

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

$$idle_2 : \begin{array}{l}[read \to idle]\\ [read \to read]\end{array}$$

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

$$waitw_2 : \begin{array}{l}[idle \to waitw]\\ [q \to q]\end{array} \; for \; q \in \{waitw, write, release\}$$

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

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

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

$$idle_5 : \begin{array}{l}[release \to idle]\\ [q \to q]\end{array} : \neg lockr, lockr' \; for \; q \in \{waitw, write, release\}$$

$$bad_1 : [\cdot \to \cdot] : r, w, bad'$$

$$bad_2 : \begin{array}{l}[write \to \cdot]\\ [write \to \cdot]\end{array} : bad'$$

$Initial \; state: \; idle^*, lockr, lockw, lockz, \neg r, \neg w, \neg bad$

$Bad \; states: \; \phi = (bad, Q^*)$

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

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

$\mathit{wait}_1 : [\mathit{idle} \rightarrow \mathit{wait1}] : \forall\{\mathit{idle}, \mathit{read}, \mathit{waitw}, \mathit{write}, \mathit{release}\}$

$\mathit{wait}_2 : [\mathit{wait1} \rightarrow \mathit{wait2}] : \forall\{\mathit{idle}, \mathit{wait1}, \mathit{read}\}$

$\mathit{read}_1 : [\mathit{wait2} \rightarrow \mathit{read}] : r', \forall\{\mathit{idle}, \mathit{wait1}, \mathit{wait2}, \mathit{waitw}, \mathit{release}\}$

$\mathit{read}_2 : \begin{array}{l} [\mathit{wait2} \rightarrow \mathit{read}] \\ [\mathit{read} \rightarrow \mathit{read}] \end{array} : r'$

$\mathit{idle}_1 : [\mathit{read} \rightarrow \mathit{idle}] : \neg r', \forall\{\mathit{idle}, \mathit{wait1}, \mathit{wait2}, \mathit{waitw}, \mathit{write}, \mathit{release}\}$

$\mathit{idle}_2 : \begin{array}{l} [\mathit{read} \rightarrow \mathit{idle}] \\ [\mathit{read} \rightarrow \mathit{read}] \end{array}$

$\mathit{waitw}_1 : [\mathit{idle} \rightarrow \mathit{waitw}] : \forall\{\mathit{idle}, \mathit{read}\}$

$\mathit{waitw}_2 : \begin{array}{l} [\mathit{idle} \rightarrow \mathit{waitw}] \\ [q \rightarrow q] \end{array} \; \mathit{for}\ q \in \{\mathit{waitw}, \mathit{write}, \mathit{release}\}$

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

$\mathit{idle}_3 : [\mathit{write} \rightarrow \mathit{release}] : \neg w'$

$\mathit{idle}_4 : [\mathit{release} \rightarrow \mathit{idle}] : \forall\{\mathit{idle}, \mathit{wait1}, \mathit{wait2}, \mathit{read}\}$

$\mathit{idle5} : \begin{array}{l} [\mathit{release} \rightarrow \mathit{idle}] \\ [q \rightarrow q] \end{array} \; \mathit{for}\ q \in \{\mathit{waitw}, \mathit{release}, \mathit{write}\}$

$\mathit{bad}_1 : [\cdot \rightarrow \cdot] : r, w, \mathit{bad}'$

$\mathit{bad}_2 : \begin{array}{l} [\mathit{write} \rightarrow \cdot] \\ [\mathit{write} \rightarrow \cdot] \end{array} : \mathit{bad}'$

$\mathit{Initial\ state}: \; \mathit{idle}^*, \neg r, \neg w, \neg \mathit{bad}$
$\mathit{Bad\ states}: \; \phi = (\mathit{bad}, Q^*)$

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

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

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

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

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

$t_4 : [\cdot \rightarrow \cdot] : \mathit{many}, \mathit{light}'$

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

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

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

$t_8 : [\mathit{zero} \rightarrow \mathit{zero}] : \mathit{bad}', \mathit{many}, \forall\{\mathit{zero}\}$

$t_9 : [\mathit{zero} \rightarrow \mathit{zero}] : \mathit{bad}', \mathit{single}, \forall\{\mathit{zero}\}$

$t_{10} : [\cdot \rightarrow \cdot] : \mathit{bad}', \mathit{many}, \neg \mathit{light}$

$t_{11} : [\cdot \rightarrow \cdot] : \mathit{bad}', \mathit{single}, \neg \mathit{light}$

$t_{12} : [\cdot \rightarrow \cdot] : \mathit{bad}', \mathit{empty}, \mathit{light}$

$\mathit{Initial\ state}: \; \mathit{zero}^*, \mathit{empty}, \neg \mathit{single}, \neg \mathit{many}, \neg \mathit{light}, \neg \mathit{bad}$
$\mathit{Bad\ states}: \; \phi = (\mathit{bad}, Q^*)$

Fig. 16. An office light control system.