International Journal of Foundations of Computer Science
Vol. 20, No. 5 (2009) 779–801
© World Scientific Publishing Company

MONOTONIC ABSTRACTION (ON EFFICIENT VERIFICATION OF PARAMETERIZED SYSTEMS)

PAROSH AZIZ ABDULLA
Uppsala University, Sweden
parosh@it.uu.se

GIORGIO DELZANNO
Università di Genova, Italy
giorgio@disi.unige.it

NOOMENE BEN HENDA
Uppsala University, Sweden
Noomene.BenHenda@it.uu.se

AHMED REZINE
LIAFA, Paris 7, France
Ahmed.Rezine@liafa.jussieu.fr

Received 12 December 2008
Accepted 15 May 2009
Communicated by Vesa Halava and Igor Potapov

We introduce the simple and efficient method of *monotonic abstraction* to prove safety properties for parameterized systems with linear topologies. A process in the system is a finite-state automaton, where the transitions are guarded by both local and global conditions. Processes may communicate via broadcast, rendez-vous and shared variables over finite domains. The method of *monotonic abstraction* derives an over-approximation of the induced transition system that allows the use of a simple class of regular expressions as a symbolic representation. Compared to traditional regular model checking methods, the analysis does not require the manipulation of transducers, and hence its simplicity and efficiency. We have implemented a prototype that works well on several mutual exclusion algorithms and cache coherence protocols.

1. Introduction

In this paper, we consider analysis of safety properties for *parameterized systems.* Several parameterized systems consist of an arbitrary number of finite-state processes organized in a linear array. The task is to verify correctness of the system regardless of the number of processes inside it. Examples of parameterized systems include mutual exclusion algorithms, bus protocols, telecommunication protocols, and cache coherence protocols.

One important technique that has been used for verification of parameterized systems is that of *regular model checking* [24, 6, 10]. In regular model checking, states are represented by words, sets of states by regular expressions, and transitions by finite automata operating on pairs of states, so called *finite-state transducers.* Safety properties can be checked through reachability analysis, which amounts to applying the transducer relation iteratively to the set of initial states. The main problem with transducer-based techniques is that they are very heavy and usually rely on several layers of computationally expensive automata-theoretic constructions; in many cases severely limiting their applicability. In this paper, we propose an approach that uses a simple class of regular expressions as a symbolic representation. The new approach is then much more lightweight and efficient than general regular model checking. We describe the application of the approach in the context of parameterized systems.

In our framework, a process is modeled as a finite-state automaton that operates on a set of local variables ranging over finite domains. The transitions of the automaton are conditioned by the local state of the process, values of the local variables, and by *global conditions.* A global condition is either *universally* or *existentially* quantified. An example of a universal condition is that all processes to the left of a given process i should satisfy a property θ. Process i is allowed to perform the transition only in the case where all processes with indices $j < i$ satisfy θ. In an existential condition we require that *some* (rather than *all*) processes satisfy θ. In addition, processes may communicate through broadcast, rendez-vous, and shared variables over finite domains. Finally, processes may dynamically be created and deleted during the execution of the system.

The main idea of *monotonic abstraction* 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. For instance in the above universally quantified transition, process i is always allowed to take the transition. However, when performing the transition, we eliminate all processes that have indices $j < i$ and that violate the condition θ. The approximate transition system obtained in this manner is *monotonic* (hence the name for the method) with respect to the subword relation on configurations (larger configurations are able to simulate smaller ones). In fact, it turns out that universal quantification is the only operation that does not preserve monotonicity and hence it is the only source of approximation in the model. Since the approximate transition relation is monotonic, it can be analyzed using symbolic backward reachability algorithm based on a generic method introduced in [1]. An attractive feature of this algorithm is that it operates on sets of configurations that are upward closed with respect to the subword relation. In particular, reachability analysis can be performed by computing predecessors of upward closed sets, which is much simpler and more efficient than applying transducer relations on general regular languages. Also, as a side effect, the analysis of the approximate model is always guaranteed to terminate. This follows from the fact that the subword relation on configurations is a *well quasi-ordering.* The whole verification process is fully automatic since both the approximation and the reachability analysis are carried out without user intervention. Observe that if the approximate transition system satisfies a safety property then we can safely conclude that the original system satisfies the property, too.

To simplify the presentation, we introduce the class of systems we consider in a stepwise manner. First, we consider a basic model where we only allow Boolean local variables together with local and global conditions. We describe how to derive the approximate transition relation and how to analyze safety properties for the basic model. Then, we introduce the additional features one by one. This includes using general finite domains, shared variables, broadcast and rendez-vous communication, dynamic creation and deletion of processes, and counters. For each new feature, we describe how to extend the approximate transition relation and the reachability algorithm in a corresponding manner.

Based on the method, we have implemented a prototype that works well on several mutual exclusion algorithms and cache coherence protocols taken from the literature [12, 14, 15, 16, 24, 13, 25, 32, 33]. We describe two examples of such systems, namely the Java Meta-locking protocol [33] and German's directory-based cache coherence protocol [19].

Related work Several recent works have been devoted to develop regular model checking, e.g., [24, 13]; and in particular augmenting regular model checking with techniques such as widening [10, 35], abstraction [11], and acceleration [6]. All these works rely on computing the transitive closure of transducers or on iterating them on regular languages.
A technique of particular interest for parameterized systems is that of *counter abstraction.* The idea is to keep track of the number of processes that satisfy a certain property. In [20] the technique generates an abstract system that is essentially a Petri net. Counter abstracted models with *broadcast communication* are proved to be *well-structured* in [18]. In [14, 15] symbolic model checking based on real arithmetics is used to verify counter abstracted models of cache coherence protocols enriched with global conditions. The method works without guarantee of termination. The paper [32] refines the counter abstraction idea by truncating the counters at the value of 2, and thus obtains a finite-state abstract system. The method may require manual insertion of auxiliary program variables for programs that exploit knowledge of process identifiers. In general, counter abstraction is designed for systems with unstructured or clique architectures. Our method can cope with this kind of systems, since unstructured architectures can be viewed as a special case of linear arrays where the ordering of the processes is not relevant. In [23] and [34], the authors present a tool for the analysis and the verification of *linear parameterized hardware systems* using the *monadic second-order logic on strings.*

Other parameterized verification methods are based on reductions to finite-state models. Among these, the *invisible invariants* method [8, 31] exploits *cut-off* properties to check invariants for mutual exclusion protocols like the Bakery algorithm and German's protocol. The success of the method depends on the heuristic used in the generation of the candidate invariant. This method sometimes (e.g. for German's protocol) requires insertion of auxiliary program variables for completing the proof. In [9] finite-state abstractions for verification of systems specified in WS1S are computed on-the-fly by using the weakest precondition operator. The method requires the user to provide a set of predicates on which to compute the abstract model. Heuristics to discover *indexed predicates* are proposed in [25] and applied to German's protocol as well as to the Bakery algorithm. In contrast to these approaches, we provide a uniform approximation scheme that is independent of the analyzed system. *Environment abstraction* [12] combines predicate abstraction with the counter abstraction. The technique is applied to the Bakery and Szymanski algorithms. The model of [12] contains a more restricted form of global conditions than ours, and also does not include features such as broadcast communication, rendez-vous communication, and dynamic behavior. Other approaches tailored to snoopy cache protocols modeled with broadcast communication are presented in [17, 27]. In [16] German's directory-based protocol is verified via a manual transformation into a snoopy protocol. It is important to remark that frameworks for finite-state abstractions [12] and those based on cut-off properties [8, 31] can be applied to parameterized systems where each component itself contains counters and other unbounded data structures.

Finally, in [33] a parameterized version of the Java Meta-locking algorithm is verified by means of an induction-based proof technique that requires manual strengthening of the mutual exclusion invariant. The same example has been verified using supercompilation in the Refal functional language in [26].

In summary, our method provides a uniform simple abstraction which allows fully automatic verification of a wide class of systems. We have been able to verify all benchmarks available to us from the literature. We describe two programs from these benchmarks, namely the German protocol and the Java Meta-locking algorithm. The specifications of these case-studies use features (like unbounded counters) that make their verification particularly challenging. On the negative side, the current method only allows the verification of safety properties, while most regular model checking and abstraction-based techniques can also handle liveness properties.

Remarks This paper is an extended version with proofs and detailed descriptions of case-studies of the extended abstract in [4]. In [3, 5, 2] we have applied monotonic abstraction to other types of parameterized systems. More precisely, in [3] we have considered parameterized systems in which individual processes have local variables ranging over natural numbers; in [5] we have considered parameterized systems in which global conditions are evaluated non-atomically, finally, in [2] we have considered parameterized systems in which processes have a tree-like structure. Although defined on a common concept (monotonic abstraction of the transition system), the method presented in this paper is defined on a different class of parameterized systems than those studied in [3, 5, 2]. In particular, note that in general a non-atomic semantics of global conditions as that considered in [5] does not correspond to a refinement of the atomic semantics we consider in this paper. For instance, for mutual exclusion problems, when working without atomicity conditions, one has to consider different models and algorithms than those considered with atomicity conditions. This is reflected by the fact that the symbolic representations used in [5] are graphs and not words. Furthermore, for some of the examples we consider in the present paper (e.g. cache coherence protocols) it is not clear what is the meaning of a non-atomic evaluations of operations such as broadcast.

Outline In the next Section we give some preliminaries and define a basic model for parameterized systems. Section 3 describes the induced transition system and introduces the coverability (safety) problem. In Section 4 we define the over-approximated transition system on which we run our technique. Section 5 presents a generic scheme for deciding coverability. In Section 6 we instantiate the scheme on the approximate transition system. Section 7 explains how we extend the basic model to cover features such as shared variables, broadcast and binary communications, and dynamic creation and deletion of processes. In Sections 8 and 9 we respectively give detailed descriptions of the Java-metalock protocol and of the German directory-based cache coherence protocol. Finally, in Section 10, we give conclusions and directions for future work.

2. Preliminaries

In this section, we define a basic model of parameterized systems. This model will be enriched by additional features such as shared variables and dynamic instantiation of processes in Section 7.

For a natural number n, let $\overline{n}$ denote the set $\{1, \ldots, n\}$. We use $\mathcal{B}$ to denote the set $\{\mathit{true}, \mathit{false}\}$ of Boolean values. For a finite set A, we let $\mathbb{B}(A)$ denote the set of formulas that have members of A as atomic formulas, and that are closed under the Boolean connectives $\neg, \wedge, \vee$. A *quantifier* is either *universal* or *existential.* Universal and existential quantifiers are of the forms $\forall_\sim$ and $\exists_\sim$ respectively, where $\sim \in \{<, >, \neq\}$. The comparison operator $\sim$ stands implicitly for *Left* ($<$), *Right* ($>$), and *Left-Right* ($\neq$) respectively. A *global condition* over A is of the form $\Box\theta$ where $\Box$ is a quantifier and $\theta \in \mathbb{B}(A)$. A global condition is said to be *universal* (resp. *existential*) if its quantifier is *universal* (resp. *existential*). We use $\mathbb{G}(A)$ to denote the set of global conditions over A.

Parameterized Systems A parameterized system consists of an arbitrary (but finite) number of identical processes, arranged in a linear array. Each process is a finite-state automaton that operates on a finite number of Boolean local variables. The transitions of the automaton are conditioned by the values of the local variables and by *global* conditions in which the process checks, for instance, the local states and variables of all processes to its left or to its right. A transition may change the value of any local variable inside the process. A parameterized system induces an infinite family of finite-state systems, namely one for each size of the array. The aim is to verify correctness of the systems for the whole family (regardless of the number of processes inside the system).

A *parameterized system* $\mathcal{P}$ is a triple (Q, X, T), where Q is a set of *local states*, X is a set of *local variables*, and T is a set of *transition rules*. A transition rule t is of the form

$$t : \left[q \;\middle|\; grd \rhd stmt \;\middle|\; q' \right] \tag{1}$$

where $q, q' \in Q$ and $grd \rightarrow stmt$ is a *guarded command*. Below we give the definition of a guarded command. A *guard* is a formula $grd \in \mathbb{B}(X) \cup \mathbb{G}(X \cup Q)$. In other words, the guard grd constraints either the values of local variables inside the process (if $grd \in \mathbb{B}(X)$); or the local states and the values of local variables of other processes (if $grd \in \mathbb{G}(X \cup Q)$). A *statement* is a set of assignments of the form $x_1 = e_1; \ldots ; x_n = e_n$, where $x_i \in X$, $e_i \in \mathcal{B}$, and $x_i \neq x_j$ if $i \neq j$. A *guarded command* is of the form $grd \rightarrow stmt$, where grd is a guard and $stmt$ is a statement.

Remark 1. *We can extend the definition of the transition rule in* (1) *so that the grd is a conjunction of formulas in* $\mathbb{B}(X) \cup \mathbb{G}(X \cup Q)$. *All the definitions and algorithms that are later presented in this paper can easily be extended to the more general form. However, for simplicity of presentation, we only deal with the current form.*

In the rest of the presentation, we assume that $\sim$ ranges over the set $\{<, >, \neq\}$.

3. Transition System

In this section, we first describe the transition system induced by a parameterized system. Then we introduce the *coverability problem*.

Transition System A *transition system* $\mathcal{T}$ is a pair $(D, \Longrightarrow)$, where D is an (infinite) set of *configurations* and $\Longrightarrow$ is a binary relation on D. We use $\overset{*}{\Longrightarrow}$ to denote the reflexive transitive closure of $\Longrightarrow$. We will consider several transition systems in this paper.

First, a parameterized system $\mathcal{P} = (Q, X, T)$ induces a transition system $\mathcal{T}(\mathcal{P}) = (C, \longrightarrow)$ as follows. A configuration is defined by the local states of the processes, and by the values of the local variables. Formally, a *local variable state* v is a mapping from X to $\mathcal{B}$. For a local variable state v, and a formula $\theta \in \mathbb{B}(X)$, we evaluate $v \models \theta$ using the standard interpretation of the Boolean connectives. A *process state* u is a pair (q, v) where $q \in Q$ and v is a local variable state. Sometimes, abusing notation, we view a process state (q, v) as a mapping $u : X \cup Q \mapsto \mathcal{B}$, where $u(x) = v(x)$ for each $x \in X$, $u(q) = true$, and $u(q') = false$ for each $q' \in Q - \{q\}$. The process state thus agrees with v on the values of local variables, and maps all elements of

Q, except q, to *false*. For a formula $\theta \in \mathbb{B}(X \cup Q)$ and a process state u, the relation $u \models \theta$ is then well-defined. This is true in particular if $\theta \in \mathbb{B}(X)$.

A *configuration* $c \in C$ is a sequence $u_1 \cdots u_n$ of process states. Intuitively, the above configuration corresponds to an instance of the system with n processes. Each pair $u_i = (q_i, v_i)$ gives the local state and the values of local variables of process i. Notice that if c_1 and c_2 are configurations then their concatenation $c_1 \bullet c_2$ is also a configuration.

Next, we define the transition relation $\longrightarrow$ on the set of configurations as follows. For a statement *stmt* and a local variable state v, we use $stmt(v)$ to denote the local variable state v' such that $v'(x) = v(x)$ if x does not occur in *stmt*; and $v'(x) = e$ if $x = e$ occurs in *stmt*. Let t be a transition rule of the form of (1). Consider two configurations $c = u_1 \cdots u_n$ and $c' = u'_1 \cdots u'_n$. We write $c \xrightarrow{t} c'$ iff there is an $i : 1 \leq i \leq n$ such that u_i and u'_i are of the forms (q, v) and $(q', stmt(v'))$ respectively, $u_j = u'_j$ for all $j : 1 \leq j \neq i \leq n$, and one of the following conditions holds:

- $grd \in \mathbb{B}(X)$ and $v \models grd$, i.e. the local variables of the process in transition should satisfy *grd*.
- $grd = \Box\theta \in \mathbb{G}(X \cup Q)$ such that:
  - either $\Box = \forall_\sim$ and $\forall\, j : 1 \leq j \sim i \leq n \,.\, u_j \models \theta$,
  - or $\Box = \exists_\sim$ and $\exists\, j : 1 \leq j \sim i \leq n \,.\, u_j \models \theta$.

  In other words, if *grd* is a global condition then the other processes should satisfy θ (in a manner that depends on the type of the quantifier).

We use $c \longrightarrow c'$ to denote that $c \xrightarrow{t} c'$ for some $t \in T$.

Safety Properties In order to analyze safety properties, we study the *coverability problem* defined below. Given a parameterized system $\mathcal{P} = (Q, X, T)$, we assume that, prior to starting the execution of the system, each process is in an (identical) *initial* process state $u_{init} = (q_{init}, v_{init})$. In the induced transition system $\mathcal{T}(\mathcal{P}) = (C, \longrightarrow)$, we use *Init* to denote the set of *initial* configurations, i.e., configurations of the form $u_{init} \cdots u_{init}$ (all processes are in their initial states). Notice that this set is infinite.

We define an ordering on configurations as follows. Given two configurations, $c = u_1 \cdots u_m$ and $c' = u'_1 \cdots u'_n$, we write $c \preceq c'$ to denote the existence of a strictly monotonic[a] injection h from $\overline{m}$ to $\overline{n}$ such that $u_i = u'_{h(i)}$ for each $i : 1 \leq i \leq m$. A set of configurations $D \subseteq C$ is *upward closed* (with respect to $\preceq$) if $c \in D$ and $c \preceq c'$ implies $c' \in D$. 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:

[a] $h : \overline{m} \to \overline{n}$ strictly monotonic means: $i < j \Rightarrow h(i) < h(j)$ for all $i, j : 1 \leq i, j \leq m$.

PAR-COV
Instance

- A parameterized system $\mathcal{P} = (Q, X, T)$.
- An upward closed set C_F of configurations.

Question $Init \xrightarrow{*} C_F$?

It can be shown, using standard techniques (see e.g. [36, 21]), that checking safety properties (expressed as regular languages) can be translated into instances of the coverability problem. Therefore, checking safety properties amounts to solving PAR-COV(i.e., to the reachability of upward closed sets).

4. Approximation

In this section, we introduce an over-approximation of the transition relation of a parameterized system.

In Section 3, we mentioned that each parameterized system $\mathcal{P} = (Q, X, T)$ induces a transition system $\mathcal{T}(\mathcal{P}) = (C, \longrightarrow)$. A parameterized system $\mathcal{P}$ also induces an *approximate* transition system $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$, where the set C of configurations is identical to the one in $\mathcal{T}(\mathcal{P})$. We define $\rightsquigarrow = (\longrightarrow \cup \rightsquigarrow_1)$, where $\longrightarrow$ is the transition relation defined in Section 3, and $\rightsquigarrow_1$, which reflects the approximation of universal quantifiers, is defined as follows. Let t be a transition rule of the form of (1), such that $grd = \forall_{\sim}\theta$ is a universal global condition. Consider two configurations $c = u_1 \cdots u_n$ and $c' = u'_1 \cdots u'_m$. We write $c \overset{t}{\rightsquigarrow}_1 c'$ to denote that there is an $i : 1 \le i \le m$ and a monotonic injection $h : \overline{m} \to \overline{n}$ such that the following conditions hold:

(1) $u_{h(i)} = (q, v)$ and $u'_i = (q', stmt(v))$ for some variable state v. This means that the local variables of process i are updated according to *stmt*.
(2) $\forall j : 1 \le j \ne i \le m$, we have $u_{h(j)} = u'_j$. In other words the local states and local variables of the other processes are not changed.
(3) $\forall j : 1 \le j \ne h(i) \le n$, we have j is in the image of h iff either (a) $j \sim h(i)$ and $u_j \models \theta$, or (b) $j \not\sim h(i)$. That is we keep (in addition to the process taking the transition i) all the relevant processes who satisfy the formula θ and all the processes that are not in the range of the quantifier (with indices $j : j \not\sim h(i)$).

Intuitively, we derive c' from c by deleting all process states that do not satisfy θ.

We use $c \rightsquigarrow c'$ to denote that $c \overset{t}{\rightsquigarrow} c'$ for some $t \in T$. Observe that the resulting approximate transition relation $\rightsquigarrow$ is *monotonic* with respect to the ordering $\preceq$ defined in Section 3 (i.e., larger configurations are able to simulate smaller ones). We say that we perform a *monotonic abstraction* of the original transition system. We define the coverability problem for the approximate system as follows:

APRX-PAR-COV

Instance

- A parameterized system $\mathcal{P} = (Q, X, T)$.
- An upward closed set C_F of configurations.

Question $Init \overset{*}{\rightsquigarrow} C_F$?

Since $\longrightarrow \subseteq \rightsquigarrow$, a negative answer to APRX-PAR-COV implies a negative answer to PAR-COV.

5. Generic Backward Scheme

In this section, we recall a generic scheme from [1] for performing symbolic backward reachability analysis.

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 of $[\![\phi]\!]$ 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).

For a constraint ϕ, we let $Pre(\phi)$ be a finite set of constraints, such that $[\![Pre(\phi)]\!] = \{c |\ \exists c' \in [\![\phi]\!] \,.\, c \Longrightarrow c'\}$. In other words $Pre(\phi)$ characterizes the set of configurations from which we can reach a configuration in ϕ 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 $Init \overset{*}{\Longrightarrow} [\![\Phi_F]\!]$.

In the scheme, we perform a backward reachability analysis, generating a sequence $\Phi_0 \sqsupseteq \Phi_1 \sqsupseteq \Phi_2 \sqsupseteq \cdots$ of finite sets of constraints such that $\Phi_0 = \Phi_F$, and $\Phi_{j+1} = \Phi_j \cup Pre(\Phi_j)$. Since $[\![\Phi_0]\!] \subseteq [\![\Phi_1]\!] \subseteq [\![\Phi_2]\!] \subseteq \cdots$, the procedure terminates when we reach a point j where $\Phi_j \sqsubseteq \Phi_{j+1}$. Notice that the termination condition implies that $[\![\Phi_j]\!] = (\bigcup_{0 \le i \le j} [\![\Phi_i]\!])$. Consequently, Φ_j characterizes the set of all predecessors of $[\![\Phi_F]\!]$. This means that $Init \overset{*}{\Longrightarrow} [\![\Phi_F]\!]$ 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 [1], 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 $\phi_i \sqsubseteq \phi_j$.

6. Scheme Instantiation

In this section, we instantiate the scheme of Section 5 to derive an algorithm for solving APRX-PAR-COV. We do that by introducing an effective and well quasi-ordered constraint system.

Throughout this section, we assume a parameterized system $\mathcal{P} = (Q, X, T)$ and the induced approximate transition system $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$. We define a constraint to be a finite sequence $\theta_1 \cdots \theta_m$ where $\theta_i \in \mathbb{B}(X \cup Q)$. Observe that for any constraints ϕ_1 and ϕ_2, their concatenation $\phi_1 \bullet \phi_2$ is also a constraint. For a constraint $\phi = \theta_1 \cdots \theta_m$ and a configuration $c = u_1 \cdots u_n$, we write $c \models \phi$ to denote that there is a strictly monotonic injection h from $\overline{m}$ to $\overline{n}$ such that $u_{h(i)} \models \theta_i$ for each $i : 1 \leq i \leq m$. Given a constraint ϕ, we let $[\![\phi]\!] = \{c \in C |\ c \models \phi\}$. Notice that if $\phi = \theta_1 \cdots \theta_m$ and some θ_i is unsatisfiable then $[\![\phi]\!]$ is empty. Such a constraint can therefore be safely discarded in the algorithm.

An aspect of our constraint system is that each constraint characterizes a set of configurations that is upward closed with respect to $\preceq$. Conversely (by Higman's Lemma [22]), any upward closed set C_F of configurations can be characterized as $[\![\Phi_F]\!]$ where Φ_F is a finite set of constraints. In this manner, APRX-PAR-COV is reduced to checking the reachability of a finite set of constraints.

Below we show effectiveness and well quasi-ordering of our constraint system, meaning that we obtain an algorithm for solving APRX-PAR-COV.

Computing Predecessors For a constraint ϕ', we define $Pre(\phi') = \bigcup_{t \in T} Pre_t(\phi')$, i.e., we compute the set of predecessor constraints with respect to each transition rule $t \in T$. In the following, assume t to be a transition rule of the form (1). To compute $Pre_t(\phi')$, we define first the function $[t]$ on $X \cup Q$ as follows: for each $x \in X$, $[t](x) = stmt(x)$ if x occurs in *stmt* and $[t](x) = x$ otherwise. For each $q'' \in Q$, $[t](q'') = true$ if $q'' = q'$, and *false* otherwise. For $\theta \in \mathbb{B}(X \cup Q)$, we use $\theta[t]$ to denote the formula obtained from θ by substituting all occurrences of elements in θ by their corresponding $[t]$-images. Now, we define an operator $\odot$ on natural numbers. We use this operator to capture the effect of existential quantifiers when computing Pre. For natural numbers $k, j \geq 1$, we define $k \odot j$ to be k if $k < j$ and $k + 1$ if $k \geq j$. For example $2 \odot 4 = 2$ and $7 \odot 4 = 8$.

For a constraint $\phi' = \theta'_1 \cdots \theta'_n$ and a rule t of the form (1), we define $Pre_t(\phi')$ to be the set of all constraints $\phi = \theta_1 \cdots \theta_m$ for some $m \geq n$, such that one of the following conditions holds:

- $grd \in \mathbb{B}(X)$, this corresponds to a transition with a local condition. Two cases are possible depending on whether the process that took the transition is represented in the constraint ϕ' or not. If it is represented, i.e., if the transition is performed by a process represented by a formula θ'_i in ϕ' for some $i : 1 \leq i \leq n$, then we let $m = n$, $\theta_i = \theta'_i[t] \wedge grd \wedge q$, and $\theta_j = \theta'_j$ for each $j : 1 \leq j \neq i \leq n$. The local state and variables of process i are changed according to the transition t. This is reflected in the definition of θ_i, which is essentially the weakest precondition of θ'_i with respect to t. The local states and variables of the other processes are not changed and hence $\theta_j = \theta'_j$ for $j \neq i$. If the process is not represented, then none of the processes represented in ϕ' are changed, meaning that the obtained configuration is also in $[\![\phi']\!]$. This case does not add any information in the fixpoint calculation, and we do not need to compute the corresponding constraints.

- $grd = \forall_{\sim}\theta$, this corresponds to a transition with a universally quantified condition. Following a similar reasoning to above, we focus on the case where the process that performed the transition is represented by a formula θ'_i in ϕ', for some $i : 1 \leq i \leq n$. We let $m = n$, $\theta_i = \theta'_i[t] \wedge grd \wedge q$, and for each $j : 1 \leq j \neq i \leq n$:

$$\theta_j = \begin{cases} \theta'_j \wedge \theta & \text{if } j \sim i, \\ \theta'_j & \text{otherwise.} \end{cases}$$

  The local state of process i is changed back to q and the statement in t is performed backwards. The local state and variables of a process j different from i with $j \not\sim i$ are not changed. All the relevant processes (with index $j : j \sim i$) should satisfy the condition θ since otherwise such processes would have been eliminated according to the approximate transition relation. In the definition of ϕ, we capture that by taking $\theta_j = \theta'_j \wedge \theta$ for any j in the range of the quantifier ($j \sim i$).
- $grd = \exists_{\sim}\theta$, this corresponds to a transition with an existentially quantified condition. Again, we only consider cases where the process that performed the transition is represented by some formula θ'_i in ϕ' for some $i : 1 \leq i \leq n$. We consider two cases depending on whether the process that satisfies the formula θ is represented or not by some θ'_j in ϕ':
  - $m = n$, $\theta_i = \theta'_i[t] \wedge grd \wedge q$, and there is a $j : 1 \leq j \sim i \leq n$ such that
    * $\theta_j = \theta'_j \wedge \theta$, and
    * $\theta_k = \theta'_k$ for each $k : 1 \leq k \neq j \wedge k \neq i \leq n$

    The case is similar to that of universal quantifiers (see the previous item), except that we require that only one relevant process (rather than all) satisfies the condition ($\theta_j = \theta'_j \wedge \theta$). Observe that the process j is represented in ϕ'.
  - $m = n + 1$, and there is a $j : 1 \leq j \leq m$ such that $\theta_j = \theta$, $\theta_{i \odot j} = \theta'_i[t] \wedge grd \wedge q$, and for each $k : 1 \leq k \neq i \leq n$ we have

$$\theta_{k \odot j} = \begin{cases} \theta'_k \wedge \neg\theta & \text{if } k \sim i, \\ \theta'_k & \text{otherwise.} \end{cases}$$

    The difference compared to the previous case is that the process that satisfies the condition θ is not part of the representation of ϕ'. Therefore, we add a new process with index j to ϕ, and we require that it satisfies θ. All other processes in the range of the quantifier (with id $k \sim i$) do not satisfy the condition and therefore we constrain them with $\neg\theta$.

Remark 2. *Notice that in the second item of the case with an existentially quantified transition, the length of the resulting constraint is larger (by one) than the length of ϕ. This means that the lengths of the constraints that arise during the analysis are not a priori fixed. Nevertheless, termination is still guaranteed by the well quasi-ordering of the constraints.*

Entailment The following Lemma gives a syntactic characterization that allows computing of the entailment relation.

Lemma 3. *For constraints $\phi = \theta_1 \ldots \theta_m$ and $\phi' = \theta'_1 \ldots \theta'_n$, we have $\phi \sqsubseteq \phi'$ iff there exists a strictly monotonic injection $h : \overline{m} \to \overline{n}$ such that $\theta'_{h(i)} \Rightarrow \theta_i$ for each $i : 1 \leq i \leq m$.*

Proof. ($\Rightarrow$) Assume there is no such injection. We derive a configuration c such that $c \in [\![\phi']\!]$ and $c \notin [\![\phi]\!]$. To do that, we define the function g on $\overline{n}$ as follows: $g(1) = 1$, $g(i+1) = g(i)$ if $\theta'_i \not\Rightarrow \theta_{g(i)}$, and $g(i+1) = g(i)+1$ if $\theta'_i \Rightarrow \theta_{g(i)}$. Observe that, since the above mentioned injection does not exist, we have either $g(n) < m$, or $g(n) = m$ and $\theta'_n \not\Rightarrow \theta_m$. We choose $c = u_1 \cdots u_n$, where u_i is defined as follows: (i) if $\theta'_i \not\Rightarrow \theta_{g(i)}$ let u_i be any process state such that $u_i \models \neg\theta_{g(i)} \wedge \theta'_i$; and (ii) if $\theta'_i \Rightarrow \theta_{g(i)}$ let u_i be any process state such that $u_i \models \theta'_i$.

($\Leftarrow$) Assume there exists a strictly monotonic injection $h : \overline{m} \to \overline{n}$ such that $\theta'_{h(i)} \Rightarrow \theta_i$ for each $i : 1 \leq i \leq m$. Let $c = u_1 \ldots u_p$ be a configuration in $[\![\phi']\!]$. It follows that there exists a strictly monotonic injection $h' : \overline{n} \to \overline{p}$ such that $u_{h'(i)} \models \theta'_i$ for each $i : 1 \leq i \leq n$. By assumption, for each $j : 1 \leq j \leq m$, we have $\theta'_{h(j)} \Rightarrow \theta_j$. Therefore, for each $j : 1 \leq j \leq m$, $u_{h' \circ h(j)} \models \theta_j$. It is straightforward to check that $h' \circ h$ is a strictly monotonic injection from $\overline{m}$ to $\overline{p}$. It follows that $c \in [\![\phi]\!]$. □

Intersection with Initial States For a constraint $\phi = \theta_1 \ldots \theta_n$, we have $(Init \bigcap [\![\phi]\!]) = \emptyset$ iff $u_{init} \not\models \theta_i$ for some $i : 1 \leq i \leq n$.

Termination Termination follows from the following lemma.

Lemma 4. *The constraint system is* well quasi-ordered (WQO) *with respect to $\sqsubseteq$.*

Proof. $(A, \preceq)$ is obviously a *WQO* for any finite set A and any *quasi-order* $\preceq$ on A. Let A^* be the set of words over A, and $\preceq^*$ be the subword relation. Higman's Lemma [22] states that $(A^*, \preceq^*)$ is also a WQO. Take A to be the quotient sets of $\mathbb{B}(X \cup Q)$ under the equivalence relation. Let $\preceq$ be the implication relation on formulas in $\mathbb{B}(X \cup Q)$. By lemma 3, the relation $\sqsubseteq$ coincides with $\preceq^*$. We conclude that the constraint system is a WQO. □

7. Extensions

In this section, we add a number of features to the model of Section 2. For each additional feature, we show how to modify the constraint system of Section 6 in a corresponding manner.

Shared Variables We assume the presence of a finite set S of Boolean *shared variables* that can be read and written by all processes in the system. A guard may constraint the values of both the shared and the local variables, and a statement may assign values to the shared variables (together with the local variables). It is straightforward to extend the definitions of the induced transition system and the symbolic algorithm to deal with shared variables.

Variables over Finite Domains Instead of Boolean variables, we can use variables that range over arbitrary finite domains. Below we describe an example of such an extension. Let Y be a finite set of variables that range over $\{0, 1, \ldots, k\}$, for some natural number k. Let $\mathbb{N}(A)$ be the set of formulas of the form $x \simeq y$ where $\simeq \in \{<, \leq, =, \neq, >, \geq\}$ and $x, y \in Y \cup \{0, 1, \ldots, k\}$. A guard is a formula $grd \in \mathbb{B}(X \cup \mathbb{N}(Y)) \cup \mathbb{G}(X \cup Q \cup \mathbb{N}(Y))$. In other words, the guard grd may also constraint the values of the variables in Y. Similarly, a statement may assign values in $\{0, 1, \ldots, k\}$ to variables in Y. A local variable state is a mapping from $X \cup Y$ to $\mathcal{B} \cup \{0, 1, \ldots, k\}$ respecting the types of the variables. The definitions of configurations, the transition relation, and constraints are extended in the obvious manner. Well quasi-ordering of the constraint system follows in a similar manner to Section 6, using the fact that variables in Y range over finite domains.

Broadcast A broadcast transition is initiated by some process, called the *initiator*, and corresponds to an arbitrary number of processes changing states simultaneously. A *broadcast rule* is a sequence of transition rules of the following form

$$\left[q_0 \,\middle|\, grd_0 \rhd stmt_0 \,\middle|\, q_0' \right] \left[q_1 \,\middle|\, grd_1 \rhd stmt_1 \,\middle|\, q_1' \right]^* \cdots \left[q_m \,\middle|\, grd_m \rhd stmt_m \,\middle|\, q_m' \right]^* \tag{4}$$

where $grd_i \in \mathbb{B}(X)$ for each $i : 0 \leq i \leq m$. Below, we use t_i to refer to the i^{th} rule in the above sequence. We assume the broadcast rule to be deterministic in the sense that either $grd_i \wedge grd_j$ is not satisfiable or $q_i \neq q_j$ for each $i, j : 1 \leq i \neq j \leq m$. The *initiator* is represented by the leftmost transition rule t_0. This transition rule has the same interpretation as in Section 2. That is, in order for the broadcast transition to take place, the initiator should be in local state q_0 and its local variables should satisfy the guard grd_0. After the completion of the broadcast, the initiator has changed state to q_0' and updated its local variables according to $stmt_0$. Together with the initiator, an arbitrary number of processes, called the *receptors*, change state simultaneously. The receptors are modeled by the transition rules $t_1, \ldots, t_m$ (each rule being marked by a * to emphasize that an arbitrary number of receptors may execute that rule). More precisely, if the local state of a process is q_i and its local variables satisfy grd_i, then the process changes its local state to q'_i and updates its local variables according to $stmt_i$. Notice that since the broadcast rule is deterministic, a receptor satisfies the precondition of at most one of the transition rules. Processes that do not satisfy the precondition of any of the transition rules remain passive during the broadcast. We define a transition relation $\longrightarrow_B$ to reflect broadcast transitions. The definition of $\longrightarrow_B$ can be derived in a straightforward manner from the above informal description. We extend the transition relation $\longrightarrow$ defined in Section 3, by taking its union with $\longrightarrow_B$. In a similar manner, we extend the approximate transition relation $\rightsquigarrow$ (defined in Section 4) by taking its union with $\longrightarrow_B$. This means that the introduction of broadcast transitions are interpreted exactly, and thus they do not add any additional approximation to $\rightsquigarrow$.

We use the same constraint system as the one defined for systems without broadcast; consequently checking entailment, checking intersection with initial states, and proving termination are identical to Section 6. Below we show how to compute *Pre*. Consider a constraint $\phi' = \theta'_1 \cdots \theta'_n$ and a broadcast rule b of the above form. We define $Pre_b(\phi')$ to contain both following sets of constraints (corresponding to whether the initiator is represented or not by a formula in ϕ'):

- The initiator is represented by a formula θ'_i in ϕ' for some $i : 1 \leq i \leq n$. We include in $Pre_b(\phi')$ all constraints of the form $\theta_1 \cdots \theta_n$ such that the following properties are satisfied:
  - $\theta_i = \theta'_i[t_0] \wedge grd_0 \wedge q_0$. This represents the predecessor state of the initiator.
  - For each $j : 1 \leq j \neq i \leq n$, one of the following properties is satisfied:
    - $\theta_j = \theta'_j \wedge \neg((q_1 \wedge grd_1) \vee (q_2 \wedge grd_2) \vee \cdots \vee (q_m \wedge grd_m))$. This represents a passive process (a process other than the initiator may be passive if it does not satisfy the preconditions of any of the rules).
    - $\theta_j = \theta'_j[t_k] \wedge grd_k \wedge q_k$, for some $k : 1 \leq k \leq m$. This represents a receptor.
- The initiator is not represented in ϕ'. We include in $Pre_b(\phi')$ all constraints of the form $\theta_1 \cdots \theta_{n+1}$ such that there is $i : 1 \leq i \leq n+1$ and the following properties are satisfied:
  - $\theta_i = grd_0 \wedge q_0$. This represents the predecessor state of the initiator.
  - For each $j : 1 \leq j \leq n$, one of the following properties is satisfied:
    - $\theta_{j \odot i} = \theta'_j \wedge \neg((q_1 \wedge grd_1) \vee (q_2 \wedge grd_2) \vee \cdots \vee (q_m \wedge grd_m))$.
    - $\theta_{j \odot i} = \theta'_j[t_k] \wedge grd_k \wedge q_k$, for some $k : 1 \leq k \leq m$.

Binary Communication In *binary communication* two processes perform a *rendez-vous* changing states simultaneously. A rendez-vous rule consists of two transition rules of the from

$$\left[\, q_1 \;\middle|\, grd_1 \rhd stmt_1 \,\middle|\; q'_1 \,\right] \left[\, q_2 \;\middle|\, grd_2 \rhd stmt_2 \,\middle|\; q'_2 \,\right] \qquad (5)$$

where $grd_1, grd_2 \in \mathbb{B}(X)$. Binary communication can be treated in a similar manner to broadcast transitions (here there is exactly one receptor). The model definition and the symbolic algorithm can be extended in a corresponding way.

Dynamic Creation and Deletion We allow dynamic creation and deletion of processes. A *process creation* rule is of the form

$$\left[\, \cdot \,\middle|\, grd \rhd \cdot \,\middle|\, q' \,\right] \tag{6}$$

where $q' \in Q$ and $grd \in \mathbb{B}(X)$. The rule creates a new process whose local state is q' and whose local variables satisfy grd. The newly created processes may be placed anywhere inside the array of processes.

We define a transition relation $\longrightarrow_D$ to reflect process creation transitions as follows. For configurations c and c', and a process creation rule d of the form of (6), we define $c \xrightarrow{d}_D c'$ to denote that c' is of the form $c'_1 \bullet u' \bullet c'_2$ where $c = c'_1 \bullet c'_2$, $u' = (q', v')$ and $v' \models grd$. We use the same constraint system as the one defined for systems without process creation and deletion. We show how to compute *Pre*. Consider a constraint ϕ' and a creation rule d of the form of (6). We define $Pre_d(\phi')$ to be the set of all constraints ϕ such that ϕ' (resp. ϕ) is of the form $\phi'_1 \bullet \theta' \bullet \phi'_2$ (resp. $\phi'_1 \bullet \phi'_2$) and $\theta'[t] \wedge grd$ is satisfiable. Notice that $\theta'[t]$ does not change the values of the local variables in θ'.

A *process deletion* rule is of the form

$$\left[\, q \,\middle|\, grd \rhd \cdot \,\middle|\, \cdot \,\right] \tag{7}$$

where $q \in Q$ and $grd \in \mathbb{B}(X)$. The rule deletes a single process whose local state is q provided that the guard grd is satisfied. The definitions of the transition system and the symbolic algorithm can be extended in a similar manner to the case with process creation rules.

Counters Using deletion, creation, and universal conditions we can simulate counters, i.e., global unbounded variables that range over the natural numbers. For each counter c, we use a special local state q_c, such that the value of c is encoded by the number of occurrences of q_c in the configuration. Increment and decrement operations can be simulated using creation and deletion of processes in local state q_c. Zero-testing can be simulated through universal conditions. More precisely, $c = 0$ is equivalent to the condition that there is no process in state q_c. This gives a model that is as powerful as Petri nets with inhibitor arcs (or equivalently counter machines) [29, 30]. Observe that the approximation introduced by the universal condition means that we replace zero-testing (in the original model) by resetting the counter value to zero (in the approximate model). Thus, we are essentially approximating the counter machine by the corresponding lossy counter machine (see [28] for a description of lossy counter machines). In fact, we can equivalently add counters as a separate feature (without simulation through universal conditions), and approximate zero-testing by resetting as described above.

8. The Java Meta-locking Algorithm

The concurrent object-oriented programming language Java provides synchronization operations for the access to every object. *Synchronized methods* and *synchronized statements* are examples of such operations. In order to ensure fairness and efficiency, every object maintains some *synchronization data*, e.g., a FIFO queue of the threads requesting the object. The Java meta-locking algorithm [7] is the protocol that controls the access to the synchronization data of every object.

The Meta-locking protocol is a distributed algorithm observed by every object and thread. The algorithm ensures mutually exclusive access to the synchronization data of every object. The pattern followed by a synchronized method invoked by a thread is as follows:

- The thread gets the object *meta-lock* if no other thread is accessing the synchronization data (*fast path*), otherwise it waits for a *hand-off*
- The thread manipulates the synchronization data.
- The threads releases the meta-lock if no other thread is waiting (*fast path*), otherwise it hands off the meta-lock to a waiting thread.

In this paper we consider the parameterized model of the meta-locking algorithm defined in [33]. Our model is described in Table 1 in appendix. The model is defined for a single object in which synchronization data have been abstracted away. The model consists of the parallel composition of an object, a *hand-off* process, and an arbitrary number of threads.

Each thread has five possible states: *idle*, *owner* (i.e., possesses the meta-lock), *handin* (i.e., competes to acquire the meta-lock), *handout* (i.e., gets ready to hands off the meta-lock), and *waiting* (i.e., waits for acknowledgment to acquire the meta-lock). The object has one control variable *busy* and a data variable c (for count).

- The Boolean variable *busy* is true when there exists a thread that possesses the meta-lock, false otherwise.
- The variable c ranges over the natural numbers and keeps track of the number of threads waiting to acquire the meta-lock on the object. This variable is an abstraction of the FIFO queue contained in the object synchronization data.

The hand-off process models the races between acquiring and releasing threads via four possible states h_0, h_1, h_2 and h_3. This model can be specified in a direct way in our input language. The object is modeled via a global Boolean variable *busy* and a global unbounded variable c ranging over naturals. The hand-off process is modeled via a global variables h_off ranging over the interval [0..3].

The transitions are described below.

t_1**:** If a thread in state *idle* requires the meta-lock and the object is not busy, then the thread becomes owner and the busy flag is set to true.

t_2: If the object is busy, then the variable c is incremented and the thread moves to state *handin* (the thread races to acquire the meta-lock).

t_3: If a thread in state *owner* releases the meta-lock and no other threads are waiting, then it moves to state *idle* and the *busy* flag is set to false.

t_4: If some thread is waiting for the meta-lock, the releasing thread moves to state *handout* and c is decremented by one.

t_5: If a thread is in state *handin* and the hand-off process is in state h_0, then the thread moves to the state *waiting* in which it waits for an acknowledgment to acquire the meta-lock. The hand-off process moves to h_1 waiting to synchronize with a releasing thread.

t_6: If a thread is in state *handout* and the hand-off process is in state h_0, then the thread releases the meta-lock and moves to the state *idle*. The hand-off process moves to h_2 waiting to synchronize with an acquiring thread.

t_7: A similar transition occurs in state h_1; the hand-off process moves to state h_3 ready to send an acknowledgment to an acquiring thread.

t_8: A transition similar to t_5 occurs if the hand-off process is in state h_2. In this case it moves to state h_3 and it gets ready to send an acknowledgment to an acquiring thread.

t_9: In state h_3 the hand-off process sends an acknowledgment to a *waiting* thread. The thread then acquires the meta-lock.

Notice that during the hand-off phase the object busy flag remains set to true.

The violation of the mutual exclusion property corresponds to configurations with more than one *owner* thread, more than one *handout* thread, or with the simultaneous presence of *owner* and *handout* threads. We automatically proved that none of these three situations can occur. We used for that a prototype running on a Pentium 1.6 Ghz. At most, the analysis required 22 iterations, took 3.2 seconds, used less than five megabytes of memory and generated 376 constraints.

Observe that our approach permitted automatic verification despite the fact that this example has two infinite dimensions: the value of the c variable and the number of thread instances. In [33] the authors needed to manually strengthen the mutual exclusion invariant in order to apply their verification method based on induction proof techniques to the same infinite-state model.

9. German Cache Coherence Protocol

We describe a cache coherence protocol due to S.German [19]. This protocol is a challenge for parameterized verification [8, 31, 16]. Our model is described in Table 2 in appendix. In this protocol, a central controller denoted by *Home*, is used to manage the access of an arbitrary, but finite, number of *clients* $P_1, \ldots, P_N$ to a cache line. In the parameterized system model, each process models a client. The actions of Home are represented in each process while its bounded local variables are modeled as shared variables.

A process, i.e. client in [31], can be in one of the three states : *invalid*, *shared* or *exclusive*. A client is in state invalid if it does not have access to the cache line. A client is in state shared if it has been granted the access (by home) possibly with other clients (also in state shared). Home can also grant the access exclusively to a client (state exclusive).

Each client communicates with Home via three channels: $channel_1$, $channel_2$ and $channel_3$. Since the channels are considered to be of length one, each of them can be represented by a local variable ch_i for $channel_i$. In addition to channels, the central controller manipulates four data structures:

(i) a flag for whether exclusive access has been granted, modeled with a shared Boolean variable (*eGran*),
(ii) a pointer to the client that sent the request being served, modeled with a local Boolean variable (*cClt*),
(iii) a list of the processes having an access, either shared or exclusive, to the cache line, modeled with a local Boolean variable *sLst* for sharer list, and
(iv) a list of processes that have to be invalidated in order to serve the current request, modeled also with a local Boolean variable *iLst* for invalidate list.

Both Home and clients may perform actions. We start with clients. Depending on the channels content and the local state, a client may perform one of the following actions.

p_1: If in state invalid and $channel_1$ is empty, the client sends a request for a shared access via $channel_1$.
p_2: If in state invalid while $channel_1$ is empty, the client sends a request for exclusive access via $channel_1$.
p_3: If in state shared while $channel_1$ is empty, the client sends a request for exclusive access via $channel_1$.
p_4: If $channel_3$ is empty and the client receives an invalidation message through $channel_2$, then the client moves to state invalid, empties $channel_2$ and sends an invalidation acknowledgment to the central controller via $channel_3$.
p_5: If the client receives a grant for shared access via $channel_2$, it moves to state shared and empties $channel_2$.
p_6: If the client receives a grant for exclusive access via $channel_2$, it moves to state exclusive and empties $channel_2$.

Depending on the content of the channels and the values of the shared variables, Home may perform one of the following actions.

h_0: In case $channel_2$ is empty, the current command is a shared request and the exclusive access has not been granted, then home sends a grant for a shared access to the current client via $channel_2$, adds the client to the shared list and becomes idle (*cCm* empty).

h_1: In case $channel_2$ is empty, the current command is an exclusive request and the sharer list is empty, then Home sends a grant for exclusive access to the current client via $channel_2$, adds the client to the sharer list, sets the exclusive flag and becomes idle.

h_2: If Home is idle and receives a request via $channel_1$, then Home updates cCm with the received request, empties $channel_1$, selects the sender to be the current client and copies the content of the sharer list to the invalidation list. The client selection and the list copying are modeled with a broadcast.

h_3: If the current command is a shared request (while the exclusive flag is set), $channel_2$ is empty, then Home sends an invalidation message to every process through $channel_2$ and removes these processes from the invalidation list.

h_4: If the current command is either an exclusive request (while the exclusive flag is set), $channel_2$ is empty, then Home sends an invalidation message to every process through $channel_2$ and removes these processes from the invalidation list.

h_5: If the current command is a request for either a shared or an exclusive access and Home receives an invalidation acknowledgment from a client via $channel_3$, in this case Home removes a client from the sharer list, resets the exclusive flag and empties $channel_3$.

To simplify the presentation, we used assignment statement of the form $x = x'$ where x and x' are different variables (see h_2). In order to model this rigorously, we need n transitions where n is the size of the domain of x'.

The safety properties we checked are: (i) no two clients are simultaneously granted an exclusive access, and (ii) no client in state shared coexists with a client in state exclusive. We used our prototype to automatically prove that none of these two situations can occur. At most, the analysis required 34 iterations, took 232 seconds, used about 15 megabytes of memory and generated 10492 constraints.

10. Conclusion and Future Work

We have presented a method for verification of parameterized systems where the components are organized in a linear array. We derive an over-approximation of the transition relation that allows the use of symbolic reachability analysis defined on upward closed sets of configurations. Based on the method, we have implemented a prototype that performs favorably compared to existing tools on several protocols that implement cache coherence and mutual exclusion.

One direction for future research is to apply the method to other types of topologies than linear arrays. For instance, in the cache coherence protocols we consider, the actual ordering on the processes inside the protocol has no relevance. These protocols fall therefore into a special case of our model where the system can be viewed as set of processes (without structure) rather than as a linear array. This indicates that the verification algorithm can be optimized even further for such systems.

Furthermore, since our algorithm relies on a small set of properties of words, which are shared by other data structures, we believe that our approach can be lifted to a more general setting. In particular we work on developing similar algorithms for systems whose behaviors are captured by relations on trees and on general forms of graphs.

11. Appendix

Table 1: Java Meta-locking Algorithm.

Instance

Q: $q_{idle}, q_{owner}, q_{handin}, q_{handout}, q_{waiting}$

X: $\{\}$

S: $busy \in \mathcal{B}, h\_off \in [0..3], c \in \mathcal{N}$

T:

$t_1 : \left[q_{idle} \,\middle|\, \neg busy \rhd busy \,\middle|\, q_{owner} \right]$

$t_2 : \left[q_{idle} \,\middle|\, busy \rhd c = c + 1 \,\middle|\, q_{handin} \right]$

$t_3 : \left[q_{owner} \,\middle|\, busy \wedge c = 0 \rhd \neg busy \,\middle|\, q_{idle} \right]$

$t_4 : \left[q_{owner} \,\middle|\, busy \wedge c \geq 1 \rhd c = c - 1 \,\middle|\, q_{handout} \right]$

$t_5 : \left[q_{handin} \,\middle|\, h\_off = 0 \rhd h\_off = 1 \,\middle|\, q_{waiting} \right]$

$t_6 : \left[q_{handout} \,\middle|\, h\_off = 0 \rhd h\_off = 2 \,\middle|\, q_{idle} \right]$

$t_7 : \left[q_{handout} \,\middle|\, h\_off = 1 \rhd h\_off = 3 \,\middle|\, q_{idle} \right]$

$t_8 : \left[q_{handin} \,\middle|\, h\_off = 2 \rhd h\_off = 3 \,\middle|\, q_{waiting} \right]$

$t_9 : \left[q_{waiting} \,\middle|\, h\_off = 3 \rhd h\_off = 0 \,\middle|\, q_{owner} \right]$

Initial State of Shared Vars:

$\neg busy, h\_off = 0, c = 0$

Initial Process State:

u_{init}: q_{idle}

Final Constraints:

Φ_F: $q_{owner}q_{onwer}$, $q_{onwer}q_{handout}$, $q_{handout}q_{owner}$, $q_{handout}q_{handout}$

Table 2: German Protocol.

Instance

Q: q_{inv}, q_{sh}, q_{exc}; q_{all} is any state in Q.

X: $cClt, sLst, iLst \in \mathcal{B}$, $ch_1 \in \{\epsilon, rSh, rExc\}$, $ch_2 \in \{\epsilon, gSh, gExc, inval\}$, $ch_3 \in \{\epsilon, iAck\}$

S: $eGran \in \mathcal{B}$, $cCm \in \{\epsilon, rSh, rExc\}$

T:

$$h_0 : \left[q_{all} \;\middle|\; \begin{array}{c} cCm = rSh, \\ \neg eGran, \\ ch_2 = \epsilon, cClt \end{array} \rhd \begin{array}{c} cCm = \epsilon, \\ sLst, \\ ch_2 = gSh \end{array} \;\middle|\; q_{all} \right]$$

$$h_1 : \left[q_{all} \;\middle|\; \begin{array}{c} cCm = rExc, \\ ch_2 = \epsilon, \\ \forall_{\neq} \neg sLst, \\ \neg sLst \end{array} \rhd \begin{array}{c} cCm = \epsilon, \\ sLst, \\ eGran, \\ ch_2 = gExc \end{array} \;\middle|\; q_{all} \right]$$

$$h_2 : \left[q_{all} \;\middle|\; \begin{array}{c} cCm = \epsilon, \\ ch_1 \neq \epsilon \end{array} \rhd \begin{array}{c} cCm = ch_1, \\ ch_1 = \epsilon, \\ iLst = sLst, \\ cClt \end{array} \;\middle|\; q_{all} \right] \left[q_{all} \;\middle|\; \mathbf{tt} \rhd \begin{array}{c} iLst = sLst, \\ \neg cClt \end{array} \;\middle|\; q_{all} \right]^*$$

$$h_3 : \left[q_{all} \;\middle|\; \begin{array}{c} cCm = rSh, \\ eGran, iLst, \\ ch_2 = \epsilon \end{array} \rhd \begin{array}{c} \neg iLst, \\ ch_2 = inval \end{array} \;\middle|\; q_{all} \right]$$

$$h_4 : \left[q_{all} \;\middle|\; \begin{array}{c} cCm = rExc, \\ iLst, ch_2 = \epsilon \end{array} \rhd \begin{array}{c} \neg iLst, \\ ch_2 = inval \end{array} \;\middle|\; q_{all} \right]$$

$$h_5 : \left[q_{all} \;\middle|\; \begin{array}{c} cCm \neq \epsilon, \\ ch_3 = iAck \end{array} \rhd \begin{array}{c} \neg sLst, \neg eGran, \\ ch_3 = \epsilon \end{array} \;\middle|\; q_{all} \right]$$

$$p_1 : \left[q_{inv} \;\middle|\; ch_1 = \epsilon \rhd ch_1 = rSh \;\middle|\; q_{inv} \right]$$

$$p_2 : \left[q_{inv} \;\middle|\; ch_1 = \epsilon \rhd ch_1 = rExc \;\middle|\; q_{inv} \right]$$

$$p_3 : \left[q_{sh} \;\middle|\; ch_1 = \epsilon \rhd ch_1 = rExc \;\middle|\; q_{sh} \right]$$

$$p_4 : \left[q_{all} \;\middle|\; \begin{array}{c} h_2 = inval, \\ ch_3 = \epsilon \end{array} \rhd \begin{array}{c} h_2 = \epsilon, \\ ch_3 = iAck \end{array} \;\middle|\; q_{inv} \right]$$

$$p_5 : \left[q_{all} \;\middle|\; ch_2 = gSh \rhd ch_2 = \epsilon \;\middle|\; q_{sh} \right]$$

$$p_6 : \left[q_{all} \;\middle|\; ch_2 = gExc \rhd ch_2 = \epsilon \;\middle|\; q_{exc} \right]$$

Initial State of Shared Vars:

$eGran \mapsto \mathbf{ff}$, $cCm \mapsto \epsilon$

Initial Process State:

u_{init}: q_{inv}, $(ch_1, ch_2, ch_3, cClt, sLst, iLst) \mapsto (\epsilon, \epsilon, \epsilon, \mathbf{ff}, \mathbf{ff}, \mathbf{ff})$

Final Constraints:

Φ_F: $q_{exc}q_{exc}$, $q_{sh}q_{exc}$, $q_{exc}q_{sh}$

References

[1] 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.

[2] 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.

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

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

[5] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Proc. VMCAI '08, 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, 2008.

[6] P. A. Abdulla, 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.

[7] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *OOPSLA 1999*, pages 207–222, 1999.

[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] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *Proc. VMCAI 2002*, pages 317–330, 2002.

[10] 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.

[11] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV04*, Lecture Notes in Computer Science, pages 372–386, Boston, July 2004. Springer Verlag.

[12] 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.

[13] 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.

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

[15] G. Delzanno. Verification of consistency protocols via infinite-state symbolic model checking. In *Proc. FORTE/PSTV 2000*, pages 171–186, 2000.

[16] E. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME 2003*, pages 247–262, 2003.

[17] E. Emerson and V. Kahlon. Model checking guarded protocols. In *Proc. LICS '03, 19th IEEE Int. Symp. on Logic in Computer Science*, Lecture Notes in Computer Science, 2003.

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

[19] S. German. Personal communication, 2007.

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

[21] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

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

[23] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. MOSEL: A flexible toolset for monadic second-order logic. In *Proc. TACAS '97, 3th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217, pages 183–202. Lecture Notes in Computer Science, 1997.

[24] 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.

[25] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV 2004*, pages 135–147, 2004.

[26] A. Lisitsa, A. P. Nemytykh, Reachability analysis in verification via supercompilation, Int. J. Found. Comput. Sci. 19 (4) (2008) 953–969.

[27] M. Maidl. A unifying model checking approach for safety properties of parameterized systems. In Berry, Comon, and Finkel, editors, *Proc. 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336, 2001.

[28] R. Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297:347–354, 2003.

[29] M. L. Minsky, Computation: finite and infinite machines, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[30] K. Reinhardt, Reachability in petri nets with inhibitor arcs, Electron. Notes Theor. Comput. Sci. 223 (2008) 239–264.

[31] 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.

[32] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *Proc. 14th Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[33] A. Roychoudhury and I. Ramakrishnan. Automated inductive verification of parameterized protocols. In *Proc. 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 25–37, 2001.

[34] C. Topnik, E. Wilhelm, T. Margaria, and B. Steffen. jMosel: A Stand-Alone Tool and jABC Plugin for M2L(Str). In *Model Checking Software: 13th International SPIN Workshop*, volume 3925 of *Lecture Notes in Computer Science*, pages 293–298, 2006.

[35] T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. Workshop on Verification of Parametrized Systems (VEPAS'01), Crete, July, 2001.

[36] 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.