Regular Model Checking without Transducers

(On Efficient Verification of Parameterized Systems)

Parosh Aziz Abdulla[1] parosh@it.uu.se,
Giorgio Delzanno[2] giorgio@disi.unige.it,
Noomene Ben Henda[1] Noomene.BenHenda@it.uu.se, and
Ahmed Rezine[1] Rezine.Ahmed@it.uu.se

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

Abstract. We give a simple and efficient method 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. The method derives an over-approximation of the induced transition system, which 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 which 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*. Typically, a parameterized system consists 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 the system. Examples of parameterized systems include mutual exclusion algorithms, bus protocols, telecommunication protocols, and cache coherence protocols.

One important technique which has been used for verification of parameterized systems is that of *regular model checking* [23, 2, 7]. 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 performing reachability analysis, which amounts to applying the transducer relation iteratively to the set of initial states. The main problem with transducer-based techniques is 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 a much more light-weight and efficient approach to regular model checking, and describe its application in the context of parameterized systems.

In our framework, a process is modeled as a finite-state automaton which 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. Finally, processes may dynamically be created and deleted during the execution of the system.

The main idea of our method is to consider a transition relation which 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 which violate the given condition. For instance in the above example, process i is always allowed to take the transition. However, when performing the transition, we eliminate all processes which have indices $j < i$ and which violate the condition θ. The approximate transition system obtained in this manner is *monotonic* 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 which 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 which 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 which works well on several mutual exclusion algorithms and cache coherence protocols, such as the Bakery and Szymanski algorithms, the Java Meta-locking protocol, the Futurebus+ protocol, German's directory-based protocol, etc.

Related work Several recent works have been devoted to develop regular model checking, e.g., [23, 10]; and in particular augmenting regular model checking with techniques such as widening [7, 33], abstraction [9], and acceleration [2]. 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 which satisfy a certain property. In [18] the technique generates an abstract system which is essentially a Petri net. Counter abstracted models with *broadcast communication* are introduced in [15] and proved to be *well-structured* in [17]. In [11, 12] 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 [30] 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 (examples of such programs are the mutual exclusion protocols we consider in this paper). 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 [22] and [32], 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 [4, 29] 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 [5, 6] 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 which is independent on the analyzed system. *Environment abstraction* [13] combines predicate abstraction with the counter abstraction. The technique is applied to the Bakery and Szymanski algorithms. The model of [13] 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 behaviour. Other approaches tailored to snoopy cache protocols modeled with broadcast communication are presented in [14, 26]. 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 [13] and those based on cutoff properties [4, 29] can be applied to parameterized systems where each component itself contains counters and other unbounded data structures. This allows for instance to deal with a model of the Bakery algorithm which is more concrete (precise) than ours.

Finally, in [31] a parameterized version of the Java Meta-locking algorithm is verified by means of an induction-based proof technique which requires manual strengthening of the mutual exclusion invariant.

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 (with the exception of the Bakery protocol, where we can only model an abstraction of the protocol). The benchmarks include some programs, e.g. the German protocol and Java Meta-locking algorithm, which (to our knowledge) have previously not been possible to verify without user interaction or specialized heuristics. 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.

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 Section 8 we report the results of our prototype on a number of mutual exclusion and cache coherence examples. Finally, in Section 9, we give conclusions and directions for future work. The appendix gives a detailed description of the case studies.

2 Preliminaries

In this section, we define a basic model of parameterized systems. This model will be enriched by additional features 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 $\{true, false\}$ of Boolean values. For a finite set A, we let $\mathbb{B}(A)$ denote the set of formulas which have members of A as atomic formulas, and which are closed under the Boolean connectives $\neg, \wedge, \vee$. A *quantifier* is either *universal* or *existential*. A universal quantifier is of one of the forms $\forall_{LR}$, $\forall_L$, $\forall_R$. An existential quantifier is of one of the forms $\exists_L$, $\exists_R$, or $\exists_{LR}$. The subscripts L, R, and LR stand for *Left*, *Right*, and *Left-Right* 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 which 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

$$t : \begin{bmatrix} q \\ grd \rightarrow stmt \\ q' \end{bmatrix} \tag{1}$$

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 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 which 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.

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 $x \in X$, $u(q) = \mathit{true}$, and $u(q') = \mathit{false}$ 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. We will define the semantics of global conditions in terms of two quantifiers $\forall$ and $\exists$. For a configuration $c = u_1 \cdots u_n$ and a formula $\theta \in \mathbb{B}(X \cup Q)$, we write $c \models \forall\theta$ if $u_i \models \theta$ for each $i : 1 \leq i \leq n$; and write $c \models \exists\theta$ if $u_i \models \theta$ for some $i : 1 \leq i \leq n$. For a statement *stmt* and a local variable state v, we use $\mathit{stmt}(v)$ 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 $c = c_1 \bullet u \bullet c_2$ and $c' = c_1 \bullet u' \bullet c_2$, where $u = (q, v)$ and $u' = (q', v')$. We write $c \xrightarrow{t} c'$ to denote that the following three conditions are satisfied:

1. If $\mathit{grd} \in \mathbb{B}(X)$ then $v \models \mathit{grd}$, i.e., the local variables of the process in transition should satisfy *grd*.
2. If $\mathit{grd} = \Box\theta \in \mathbb{G}(X \cup Q)$ then one of the following conditions is satisfied:
   - $\Box = \forall_L$ and $c_1 \models \forall\theta$.
   - $\Box = \forall_R$ and $c_2 \models \forall\theta$.
   - $\Box = \forall_{LR}$ and $c_1 \models \forall\theta$ and $c_2 \models \forall\theta$.
   - $\Box = \exists_L$ and $c_1 \models \exists\theta$.
   - $\Box = \exists_R$ and $c_2 \models \exists\theta$.
   - $\Box = \exists_{LR}$ and either $c_1 \models \exists\theta$ or $c_2 \models \exists\theta$.

   In other words, if *grd* is a global condition then the rest of the processes should satisfy θ (in a manner which depends on the type of the quantifier).
3. $v' = \mathit{stmt}(v)$.

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 that c is a subword of c', i.e., there is a strictly monotonic[3] injection h from the set $\overline{m}$ to the set $\overline{n}$ 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 the order $\preceq$) 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:

<u>PAR-COV</u>
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. [34, 19]), 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. For a configuration c, and a formula $\theta \in \mathbb{B}(X \cup Q)$, we use $c \ominus \theta$ to denote the maximal configuration c' (with respect to $\preceq$) such that $c' \preceq c$ and $c' \models \forall\theta$. In other words, we derive c' from c by deleting all process states which do not satisfy θ. Consider two configurations $c = c_1 \bullet u \bullet c_2$ and $c' = c'_1 \bullet u' \bullet c'_2$, where $u = (q, v)$ and $u' = (q', v')$. Let t be a transition rule of the form of (1), such that $grd = \Box\theta$ is a universal global condition. We write $c \overset{t}{\rightsquigarrow}_1 c'$ to denote that the following conditions are satisfied:

1. if $\Box = \forall_L$, then $c'_1 = c_1 \ominus \theta$ and $c'_2 = c_2$.
2. if $\Box = \forall_R$, then $c'_1 = c_1$ and $c'_2 = c_2 \ominus \theta$.
3. if $\Box = \forall_{LR}$, then $c'_1 = c_1 \ominus \theta$ and $c'_2 = c_2 \ominus \theta$.
4. $v' = stmt(v)$.

We use $c \rightsquigarrow c'$ to denote that $c \overset{t}{\rightsquigarrow} c'$ for some $t \in T$. We define the coverability problem for the approximate system as follows:

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

<u>APRX-PAR-COV</u> **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 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 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 $[\![\Phi_j]\!] = (\bigcup_{0 \leq i \leq j} [\![\Phi_i]\!])$. 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 Algorithm

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 the set $\overline{m}$ to the set $\overline{n}$ 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 if it arises in the algorithm.

An aspect of our constraint system is that each constraint characterizes a set of configurations which is upward closed with respect to $\preceq$. Conversely (by Higman's Lemma [21]), 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.

Pre 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 two operators, $\otimes$ and $\oplus$, which we use to capture the effects of universal and existential quantifiers when computing *Pre*. We use $\otimes$ to handle universal quantifiers. For a constraint $\phi = \theta_1 \cdots \theta_m$ and a $\theta \in \mathbb{B}(X \cup Q)$, we define $\phi \otimes \theta$ to be the constraint $(\theta_1 \wedge \theta) \cdots (\theta_m \wedge \theta)$. We use $\oplus$ to deal with existential quantifiers. For a constraint $\phi = \theta_1 \cdots \theta_m$ and a $\theta \in \mathbb{B}(X \cup Q)$, we define $\phi \oplus \theta$ to be the set of constraints which are of one of the following forms:

– $\theta_1 \cdots \theta_{i-1}(\theta_i \wedge \theta)\theta_{i+1} \cdots \theta_m$ where $1 \leq i \leq m$; or
– $(\theta_1 \wedge \neg\theta) \cdots (\theta_i \wedge \neg\theta)\theta(\theta_{i+1} \wedge \neg\theta) \cdots (\theta_m \wedge \neg\theta)$ where $0 \leq i \leq m + 1$.

In the first case, the constraint implies that there is at least one process in the configuration satisfying θ. In the the second case, the constraint does not imply the existence of such a process, and therefore the formula θ is added explicitly to the representation of the constraint. Notice that in the second case the length of the resulting constraint is larger (by one) than the length of ϕ. This means that the lengths of the constraints which arise during the analysis are not *a priori* fixed. Nevertheless, termination is still guaranteed by well quasi-ordering of the constraints.

For a constraint ϕ' and a rule t of the form (1), we define $Pre_t(\phi')$ 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. $\phi'_1 \bullet \theta' \bullet \phi'_2$) and the following conditions are satisfied:

– If $grd \in \mathbb{B}(X)$ (i.e. grd is a local condition), then $\theta = \theta'[t] \wedge grd \wedge q$, $\phi_1 = \phi'_1$ and $\phi_2 = \phi'_2$;
– If $grd = \Box grd'$, where $grd' \in \mathbb{B}(X \cup Q)$, then $\theta = \theta'[t] \wedge q$ and depending on $\Box$ the following conditions hold:
  - If $\Box = \forall_L$ then $\phi_1 = \phi'_1 \otimes grd'$ and $\phi_2 = \phi'_2$.
  - If $\Box = \forall_R$ then $\phi_1 = \phi'_1$ and $\phi_2 = \phi'_2 \otimes grd'$.
  - If $\Box = \forall_{LR}$ then $\phi_1 = \phi'_1 \otimes grd'$ and $\phi_2 = \phi'_2 \otimes grd'$.
  - If $\Box = \exists_L$ then $\phi_1 \in \phi'_1 \oplus grd'$ and $\phi_2 = \phi'_2$.
  - If $\Box = \exists_R$ then $\phi_1 = \phi'_1$ and $\phi_2 \in \phi'_2 \oplus grd'$.
  - If $\Box = \exists_{LR}$ then either $\phi_1 \in \phi'_1 \oplus grd'$ and $\phi_2 = \phi'_2$; or $\phi_1 = \phi'_1$ and $\phi_2 \in \phi'_2 \oplus grd'$.

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

Lemma 1. *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 $i \in \overline{m}$.*

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 $\theta'_i \not\Rightarrow \theta_{g(i)}$ then take u_i to be any process state such that $u_i \models \neg\theta_{g(i)} \wedge \theta'_i$; and (ii) if $\theta'_i \Rightarrow \theta_{g(i)}$ then take u_i to 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 $i \in \overline{m}$. 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 $i \in \overline{n}$. By assumption, for each $j \in \overline{m}$, we have $\theta'_{h(j)} \Rightarrow \theta_j$. Therefore, for each $j \in \overline{m}$, $u_{h' \circ h(j)} \models \theta_j$. 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 $(\mathit{Init} \bigcap [\![\phi]\!]) = \emptyset$ iff $u_{init} \not\models \theta_i$ for some $i \in \overline{n}$.

Termination We show that the constraint system is *well quasi-ordered (WQO)* with respect to $\sqsubseteq$. $(A, \preceq)$ 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 [21] 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 1, 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 which range over arbitrary finite domains. Below we describe an example of such an extension. Other extensions can be carried out in a similar manner. Let Y be a finite set of variables which range over $\{0, 1, \ldots, k\}$, for some natural number k. Let $\mathbb{N}(A)$ be the set of formulas of the form $x \sim y$ where $\sim \in \{<, \leq, =, \neq, >, \geq\}$ 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. A statement may assign values in $\{0, 1, \ldots, k\}$ to variables in Y (together with assigning values in $\mathcal{B}$ to the Boolean variables). 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 In a broadcast transition, an arbitrary number of processes change states simultaneously. A *broadcast rule* is a sequence of transition rules of the following form

$$\begin{bmatrix} q_0 \\ grd_0 \to stmt_0 \\ q_0' \end{bmatrix} \begin{bmatrix} q_1 \\ grd_1 \to stmt_1 \\ q_1' \end{bmatrix}^* \begin{bmatrix} q_2 \\ grd_2 \to stmt_2 \\ q_2' \end{bmatrix}^* \cdots \begin{bmatrix} q_m \\ grd_m \to stmt_m \\ q_m' \end{bmatrix}^* \quad (2)$$

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. The broadcast rule is 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 broadcast is initiated by a process, called the *initiator*, which is represented by t_0 (i.e., the leftmost transition rule). 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 which 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 be the set of all constraints of the form $\theta_1 \cdots \theta_n$ such that there is $i : 1 \leq i \leq n$ and 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, is allowed to 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.

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

$$\begin{bmatrix} q_1 \\ grd_1 \rightarrow stmt_1 \\ q'_1 \end{bmatrix} \begin{bmatrix} q_2 \\ grd_2 \rightarrow stmt_2 \\ q'_2 \end{bmatrix} \tag{3}$$

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

$$\begin{bmatrix} \cdot \\ grd \rightarrow \cdot \\ q' \end{bmatrix} \tag{4}$$

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 (4), 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 (4). 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

$$\begin{bmatrix} q \\ grd \rightarrow \cdot \\ \cdot \end{bmatrix} \tag{5}$$

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 similarly to the case with process creation rules. We omit the details here due to shortage of space.

Counters Using deletion, creation, and universal conditions we can simulate counters, i.e., global unbounded variables which 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 which is as powerful as Petri nets with inhibitor arcs (or equivalently counter machines). 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 [27] 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 Experimental Results

Based on our method, we have implemented a prototype tool and run it on a collection of mutual exclusion and cache coherence protocols. The results, using a Pentium M 1.6 Ghz with 1G of memory, are summarized in Tables 1 and 2. For each of the mutual exclusion protocols, we consider two variants; namely one with dynamic creation and deletion of processes (marked with a * in Table 1), and one without. Full details of the examples can be found in the appendix. For each example, we give the number of iterations performed by the reachability algorithm, the largest number of constraints maintained at any point during the execution of the algorithm, and the time (in milliseconds). The computation for each example required less than 15MB of memory.

	# iter	# constr	t(ms)
Bakery	2	2	4
Bakery*	2	2	4
Burns	14	71	230
Burns*	9	21	32
Java M-lock	5	24	30
Java M-lock*	5	17	30
Dijkstra	13	150	1700
Dijkstra*	8	57	168
Szymanski	17	334	3880
Szymanski*	17	334	4080

Table 1. Mutual exclusion algorithms

	# iter	# constr	t(ms)
Synapse	3	3	4
Berkeley	2	6	8
Mesi	3	8	8
Moesi	1	12	12
Dec Firefly	3	11	16
Xerox P.D	3	20	52
Illinois	5	33	80
Futurebus	7	153	300
German	44	14475	3h45mn

Table 2. Cache coherence protocols

9 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 which allows the use of symbolic reachability analysis defined on upward closed sets of configurations. Based on the method, we have implemented a prototype which performs favorably compared to existing tools on several protocols which 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 aim to develop similar algorithms for systems whose behaviours are captured by relations on trees and on general forms of graphs. This would allow us to extend our method in order to verify systems such as those in [8, 24].

References

1. P. A. Abdulla, K. Čerāns, B. Jonsson, and T. Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc. LICS' 96* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.
2. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002,* 13^{th} *Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.
3. 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.

4. 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. 13^{th} Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234, 2001.
5. K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized networks. *Journal of Universal Computer Science*, 7(2), 2001.
6. 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.
7. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.
8. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *Proc. 13^{th} Int. Symp. on Static Analysis*, 2006.
9. 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.
10. 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.
11. G. Delzanno. Automatic verification of cache coherence protocols. In Emerson and Sistla, editors, *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Verlag, 2000.
12. G. Delzanno. Verification of consistency protocols via infinite-state symbolic model checking. In *Proc. FORTE/PSTV 2000*, pages 171–186, 2000.
13. H. V. E. Clarke, M. Talupur. Environment abstraction for parameterized verification. In *Proc. VMCAI '06, 7^{th} Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141, 2006.
14. E. Emerson and V. Kahlon. Model checking guarded protocols. In *Proc. LICS' 03 19^{th} IEEE Int. Symp. on Logic in Computer Science*, Lecture Notes in Computer Science, 2003.
15. E. Emerson and K. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. LICS' 98 13^{th} IEEE Int. Symp. on Logic in Computer Science*, pages 70–80, 1998.
16. E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME 2003*, pages 247–262, 2003.
17. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS' 99 14^{th} IEEE Int. Symp. on Logic in Computer Science*, 1999.
18. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.
19. 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.
20. J. Handy. *The Cache Memory Book.* Academic Press, 1993.
21. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, 2:326–336, 1952.
22. P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. MOSEL: A flexible toolset for monadic second-order logic. In E. Brinksma, editor, *Proc. TACAS '97, 3^{th}*

Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, volume 1217, pages 183–202, Enschede, The Netherlands, 1997. Lecture Notes in Computer Science.
23. 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.
24. B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. TACAS '06,* 12^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 39200 of *Lecture Notes in Computer Science*, 2006.
25. S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV 2004*, pages 135–147, 2004.
26. M. Maidl. A unifying model checking approach for safety properties of parameterized systems. In Berry, Comon, and Finkel, editors, *Proc.* 13^{th} *Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336, 2001.
27. R. Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297:347–354, 2003.
28. M. Nilsson. Regular model checking. Technical report, Dept. of Computer Systems, Uppsala University, Sweden, 2000. Licentiate thesis.
29. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. TACAS '01,* 7^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 82–97, 2001.
30. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.
31. A. Roychoudhury and I. Ramakrishnan. Automated inductive verification of parameterized protocols. In *Proc.* 13^{th} *Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 25–37, 2001.
32. 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, Vienna (Austria)*, volume 3925 of *Lecture Notes in Computer Science*, pages 293–298. Springer-Verlag, 2006.
33. 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.
34. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

A Appendix – Details of the Case Studies

In the following, we give detailed description of the case studies. For each example, we will describe how we instantiate the algorithm. We do this by specifying (i) the parameterized system model (Q, X, T), (ii) the initial process state u_{init}, and (iii) the constraint set Φ_F. For clarity, we refer to *true* and *false* by $\mathbf{tt}$ and $\mathbf{ff}$ respectively.

A.1 Mutual Exclusion Algorithms

We derived the parameterized system models from the versions found in [28].

Burn's Algorithm Burn's mutual exclusion algorithm can be modeled by a parameterized system where each process has a local Boolean variable f (for $flag$) . The local state ranges over $\{q_1, \ldots, q_7\}$ where q_6 represents the critical section.

<u>Burn's Algorithm</u>
Instance
Q: $q_1, \ldots, q_7$
X: $f \in \mathcal{B}$
T:

$$t_1 : \begin{bmatrix} q_1 \\ \mathbf{tt} \to f = \mathbf{ff} \\ q_2 \end{bmatrix} \qquad t_2 : \begin{bmatrix} q_2 \\ \exists_L f \to \{\} \\ q_1 \end{bmatrix} \qquad t_3 : \begin{bmatrix} q_2 \\ \forall_L \neg f \to \{\} \\ q_3 \end{bmatrix}$$

$$t_4 : \begin{bmatrix} q_3 \\ \mathbf{tt} \to f = \mathbf{tt} \\ q_4 \end{bmatrix} \qquad t_5 : \begin{bmatrix} q_4 \\ \exists_L f \to \{\} \\ q_1 \end{bmatrix} \qquad t_6 : \begin{bmatrix} q_4 \\ \forall_L \neg f \to \{\} \\ q_5 \end{bmatrix}$$

$$t_7 : \begin{bmatrix} q_5 \\ \forall_R \neg f \to \{\} \\ q_6 \end{bmatrix} \qquad t_8 : \begin{bmatrix} q_6 \\ \mathbf{tt} \to f = \mathbf{ff} \\ q_7 \end{bmatrix} \qquad t_9 : \begin{bmatrix} q_7 \\ \mathbf{tt} \to \{\} \\ q_1 \end{bmatrix}$$

Initial Process State u_{init}: $q_1, f \mapsto \mathbf{ff}$
Final Constraints Φ_F: $q_6 q_6$

In this model, any process at state q_1, q_3 q_6 or q_7 can always fire t_1, t_4, t_8 and t_9 respectively since the guard is always satisfied. Transitions t_3 and t_5 are enabled iff there are no other processes to the left with f equal to $\mathbf{tt}$. This guarantees that in any configuration, among all processes at state q_5 (with $f = \mathbf{tt}$), the rightmost process must have been the first to reach that location. Call that process i. Observe that t_7 will eventually be enabled first for i. In fact, the remaining processes cannot fire t_7 since the guard requires that all processes to the right should have f set to *false*.

Dijkstra's Algorithm A process has seven states $q_1, \ldots, q_7$ where q_6 represents the critical section. The local variables are the Boolean p for *pointer* and the number $f \in \{0, 1, 2\}$ for $flag$.

Dijkstra's Algorithm

Instance

Q: $q_1, \ldots, q_7$

X: $p \in \mathcal{B}, f \in [0..2]$

T:

$$t_1 : \begin{bmatrix} q_1 \\ \mathbf{tt} \to f = 1 \\ q_2 \end{bmatrix} \qquad t_2 : \begin{bmatrix} q_2 \\ \forall_{LR}(f = 0 \vee \neg p) \to \{\} \\ q_3 \end{bmatrix} \qquad t_4 : \begin{bmatrix} q_4 \\ \mathbf{tt} \to f = 2 \\ q_5 \end{bmatrix}$$

$$t_3 : \begin{bmatrix} q_3 \\ \mathbf{tt} \to p = \mathbf{tt} \\ q_4 \end{bmatrix} \begin{bmatrix} q_1 \\ \mathbf{tt} \to p = \mathbf{ff} \\ q_1 \end{bmatrix}^* \ldots \begin{bmatrix} q_7 \\ \mathbf{tt} \to p = \mathbf{ff} \\ q_7 \end{bmatrix}^* \qquad t_5 : \begin{bmatrix} q_5 \\ \forall_{LR} f \neq 2 \to \{\} \\ q_6 \end{bmatrix}$$

$$t_6 : \begin{bmatrix} q_5 \\ \exists_{LR} f = 2 \to \{\} \\ q_1 \end{bmatrix} \qquad t_7 : \begin{bmatrix} q_6 \\ \mathbf{tt} \to f = 0 \\ q_7 \end{bmatrix} \qquad t_8 : \begin{bmatrix} q_7 \\ \mathbf{tt} \to \{\} \\ q_1 \end{bmatrix}$$

Initial Process State u_{init}: $q_1, f \mapsto 0, p \mapsto \mathbf{ff}$

Final Constraints Φ_F: $q_6 q_6$

In the algorithm, a pointer, i.e., a variable ranging over process indices is used. We model this by the local Boolean variable p. A process has the variable p equal to $\mathbf{tt}$ iff it is pointed by the pointer. In order to simulate how the pointer changes we add the broadcast transition t_3. By firing t_3, a process changes state from q_3 to q_4, sets its local variable p to $\mathbf{tt}$ and simultaneously sets p to $\mathbf{ff}$ in all other processes. We can also model these pointer changes by a rendez-vous rule as described below. A process i in q_3 can synchronize with any other process j with p equal to $\mathbf{tt}$ such that i sets p to $\mathbf{tt}$ and moves to q_4 while j sets p to $\mathbf{ff}$ and remains in the same state. Observe that since there are no restrictions on the state of j, we need a rendez vous rule for each possible state of j:

$$\begin{bmatrix} q_3 \\ \mathbf{tt} \to p = \mathbf{tt} \\ q_4 \end{bmatrix} \begin{bmatrix} q_i \\ p \to p = \mathbf{ff} \\ q_i \end{bmatrix} : \ 1 \leq i \leq 7.$$

Given a set of processes in their initial states. Assume that these processes are "racing" to the critical section. We notice first that any number of processes can fire t_1. Afterwards, there are 2 scenarios:

- In case only one process (call it i) fires t_2 followed by t_3 before any other process "makes a move", then the remaining processes will be blocked at q_2. In fact, the values of the variables of i ($p = \mathbf{tt}$ and $f = 1$) violates the guard of t_2 for any other process at q_2. Process i can then fire the rest of the transitions thus accessing the critical section. The transition t_2 will eventually be enabled for other process once i fires t_7.
- In case some processes fire t_2, then t_3 and t_4 ending in q_5. Among these processes, the last (say i) that had fired t_3 must be the only process with p equal to $\mathbf{tt}$. Observe now that all processes at q_5 have the variable f equal to 2. Therefore, among these processes, only the "laziest" process (call it j)

can fire t_5 since the rest of processes would by then have fired t_6 and moved back to q_1. In case i is one of these processes, then it is the only process (after j) able to reach q_6 in the next "round".

Bakery's Algorithm In the Bakery algorithm, a ticket is given to each process that requires access the critical section. The last received ticket is the maximum of all outstanding tickets plus one. The process with the minimum ticket gets into the critical section and drops the ticket when leaving.

In the model, we consider processes to be owners of tickets. Each owner can be in state q_{idle}, q_{wait} or $q_{critical}$. The exact values of the tickets are irrelevant in our model. Nevertheless, we assume that the processes (owner of the tickets) are ordered from left to right with increasing tickets.

Bakery's Algorithm

Instance

Q: $q_{idle}, q_{wait}, q_{critical}$

X:

T:

$$t_1 : \begin{bmatrix} q_{idle} \\ \forall_R q_{idle} \to \{\} \\ q_{waiting} \end{bmatrix} \qquad t_2 : \begin{bmatrix} q_{waiting} \\ \forall_L q_{idle} \to \{\} \\ q_{critical} \end{bmatrix} \qquad t_3 : \begin{bmatrix} q_{critical} \\ \mathbf{tt} \to \{\} \\ q_{idle} \end{bmatrix}$$

Initial Process State u_{init}: q_{idle}

Final Constraints Φ_F: $q_{critical} q_{critical}$

Given a number of processes where each of them is in the initial state, the first process firing t_1 is the first process to access the critical section ($q_{critical}$). If it happens that many processes fire t_1, then among the processes in state $q_{waiting}$ the leftmost process must have been the first to "ask for permission". Observe that this is consistent with our assumption that tickets are ordered with the smallest to the left. Finally, due to the guard in t_2, only the leftmost process in $q_{waiting}$ is able to move to $q_{critical}$.

Szymanski's Algorithm Szymanski's mutual exclusion algorithm induces a parameterized system with states in $\{q_1, \ldots, q_7\}$ and two Boolean variables s and w. A process is in the critical section whenever its local state is q_7.

Given a set of processes in their initial states, there are two cases capturing the behavior of the system:

- Assume that a process i fires t_1 and t_2 while the other processes remain in the initial state. Since these processes are in q_1, t_3 is enabled and i can fire it. Once process i is in state q_5, t_1 will remain disabled for all other process since i has its variable s set to $\mathbf{tt}$. Afterwards, i can fire t_6 then t_7 accessing the critical section (q_7). Finally, by firing t_8, i resets s to $\mathbf{ff}$ (thus enabling t_1 for other processes) and moves back to q_1.
- Assume that some processes fire t_2 then t_3, while the rest remain in q_1. Consider i to be one of the moving processes. If process i fires t_3, then t_3 is

disabled for all other processes in q_3. In the same time i will be blocked at q_5 since the remaining processes, if they are not in q_1, have w set to $\mathbf{tt}$ thus disabling t_6. However, for processes in q_3 both t_4 and t_5 are enabled because of i (in q_5 with $s = \mathbf{tt}$ and $w = \mathbf{ff}$). Observe that i will remain blocked until these processes reach q_5. In such case, all processes (including i) can move to q_6. Now, among all processes in state q_6 (assuming the rest in q_1), only the leftmost process can fire t_7.

<u>Szymanski's Algorithm</u>
Instance
Q: $q_1, \ldots, q_7$
X: $s, w \in \mathcal{B}$
T:

$$t_1 : \begin{bmatrix} q_1 \\ \forall_{LR} \neg s \to \{\} \\ q_2 \end{bmatrix} \quad t_2 : \begin{bmatrix} q_2 \\ \mathbf{tt} \to s, w = \mathbf{tt}, \mathbf{tt} \\ q_3 \end{bmatrix} \quad t_3 : \begin{bmatrix} q_3 \\ \forall_{LR}(q_1 \vee w) \to w = \mathbf{ff} \\ q_5 \end{bmatrix}$$

$$t_4 : \begin{bmatrix} q_3 \\ \exists_{LR}(\neg q_1 \wedge \neg w) \to s = \mathbf{ff} \\ q_4 \end{bmatrix} \quad t_5 : \begin{bmatrix} q_4 \\ \exists_{LR}(s \wedge \neg w) \to s, w = \mathbf{tt}, \mathbf{ff} \\ q_5 \end{bmatrix}$$

$$t_6 : \begin{bmatrix} q_5 \\ \forall_{LR} \neg w \to \{\} \\ q_6 \end{bmatrix} \quad t_7 : \begin{bmatrix} q_6 \\ \forall_{L} \neg s \to \{\} \\ q_7 \end{bmatrix} \quad t_8 : \begin{bmatrix} q_7 \\ \mathbf{tt} \to \neg s \\ q_1 \end{bmatrix}$$

Initial Process State u_{init}: $q_1, s \mapsto \mathbf{ff}, w \mapsto \mathbf{ff}$
Final Constraints Φ_F: $q_7 q_7$

The Java Meta-locking Algorithm The concurrent object-oriented programming language Java provides synchronization operation for the access to every object like *synchronized methods* and *synchronized statements*. 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 [3] is the protocol that controls the access to the synchronization data of every object. The Meta-locking protocol is a distributed algorithm which is 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 [31]. 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* (it possesses the meta-lock), *handin* (it competes to acquire the meta-lock), *handout* (it gets ready to hands off the meta-lock), and *waiting* (it waits for acknowledgment to acquire the meta-lock). The object has one control variable *busy* and a data variable *count*.

– The boolean variable *busy* is true when there exists a thread which possesses the meta-lock, false otherwise.
– The variable *count* 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.

Java Meta-locking Algorithm

Instance

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

X: $\{\}$

S: $object\_busy \in \mathcal{B}, hand\_off \in [0..3], count \in \mathcal{N}$

T:

$$t_1 : \left[\begin{array}{c} q_{idle} \\ \neg object\_busy \rightarrow object\_busy \\ q_{owner} \end{array}\right] \; t_2 : \left[\begin{array}{c} q_{idle} \\ object\_busy \rightarrow count = count + 1 \\ q_{handin} \end{array}\right]$$

$$t_3 : \left[\begin{array}{c} q_{owner} \\ object\_busy \wedge count = 0 \rightarrow \neg object\_busy \\ q_{idle} \end{array}\right]$$

$$t_4 : \left[\begin{array}{c} q_{owner} \\ object\_busy \wedge count \geq 1 \rightarrow count = count - 1 \\ q_{handout} \end{array}\right]$$

$$t_5 : \left[\begin{array}{c} q_{handin} \\ hand\_off = 0 \rightarrow hand\_off = 1 \\ q_{waiting} \end{array}\right] \; t_6 : \left[\begin{array}{c} q_{handout} \\ hand\_off = 0 \rightarrow hand\_off = 2 \\ q_{idle} \end{array}\right]$$

$$t_7 : \left[\begin{array}{c} q_{handout} \\ hand\_off = 1 \rightarrow hand\_off = 3 \\ q_{idle} \end{array}\right] \; t_8 : \left[\begin{array}{c} q_{handin} \\ hand\_off = 2 \rightarrow hand\_off = 3 \\ q_{waiting} \end{array}\right]$$

$$t_9 : \left[\begin{array}{c} q_{waiting} \\ hand\_off = 3 \rightarrow hand\_off = 0 \\ q_{owner} \end{array}\right]$$

Initial State of Shared Vars $\neg object\_busy, handoff = 0$

Initial Process State u_{init}: q_{idle}

Final Constraints Φ_F: $q_{owner}(q_{onwer} \vee q_{handout})$, $(q_{onwer} \vee q_{handout})q_{owner}$, $q_{handout}q_{handout}$

The hand-off process models the races between acquiring and releasing threads via four possible states h_0, h_1, h_2, h_3. This model can be specified in a direct way in our input language. The object is modeled via a global Boolean variable $object_busy$ and a global unbounded variable *count* ranging over naturals. The hand-off process is modeled via a global variables $hand\_off$ 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 *count* is incremented and the thread moves to state *handin* (it 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 *count* 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.

Observe that we automatically proved mutual exclusion despite the fact that this example has two infinite dimensions: the value of the *count* variable and the number of thread instances. In [31] 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.

A.2 Cache Coherence Protocols

We consider here parameterized system models of different cache coherence protocols. Most of the models (except for German and Futurebus+ protocols) are inspired from the descriptions found in [11]. In this article, each cache is described by a *Deterministic Finite State Automaton (DFSA)* (with alphabet $\{R, \overline{R}, W, \overline{W}\}$) specifying the possible states of a cache and the corresponding transitions in case of *read hit*(R), *read miss*$(\overline{R})$, *write hit*(W), or *write miss*$(\overline{W})$.

The behavior of a system (with several caches) under that protocol is captured by an *Extended Finite State Machine (EFSM)* where each "rule" describes the *Coherence Actions* (possibly simultaneous changes of cache states) required in case of read and write operations. The states of the EFSM are vectors of counters where each one represents the number of caches in a certain state. The transitions are defined by means of integer arithmetics formulas defined over the counters and over their primed version.

For each protocol, we considered the set of states to be the same as in the corresponding DFSA, while the transitions are derived from the EFSM. Once we derived the model, the set of final constraints is chosen according to the set of *unsafe configurations* that are usually of the form: (i) cache in state q_x coexists with a cache in state q_y, (ii) there is more than one cache in state q_z. This gives respectively the constraints (i) $q_x q_y$ and $q_y q_x$, (ii) $q_z q_z$.

Illinois In this protocol [20], the cache states are: $q_{invalid}$, q_{dirty}, q_{shared} and q_{valid}. Initially all caches are in state $q_{invalid}$. The remaining states are described below.

– $q_{invalid}$: The cache content is not up-to-date.
– q_{dirty}: The cache contains a modified copy of the data, i.e the data in main memory is obsolete and the other caches are invalid.
– q_{shared}: The cache has a copy of the data that is consistent with the memory and with other caches.
– q_{valid}: The cache contains an exclusive copy of the data that is consistent with the memory.

The coherence actions are as follows. In case of a **read hit**, no coherence actions are taken. The other cases are described bellow.

read miss: There are four cases:
  – There is another cache in state q_{dirty}, in which case the latter supplies the data, writes it back to the memory and both caches move simultaneously to state q_{shared} (t_1).
  – If there is a cache in state q_{shared} or q_{valid}, then the latter supplies the data, and all caches at q_{valid} move simultaneously to q_{shared} (t_5).
  – If there is no cached copy, then the current cache gets a copy from memory and moves to state q_{valid} (t_2).
  – Transitions t_4, t_7, and t_8 all model a cache line replacement.

write hit: Two cases are possible:
  – If the cache is in state q_{valid}, then it moves to q_{dirty} (t_9).
  – If the cache is in state q_{shared}, then it moves to q_{dirty} and all other copies of the data are "invalidated" (t_6).

write miss: Like for **read miss**, all other copies are invalidated and the current cache moves to q_{dirty} (t_3).

Illinois Protocol

Instance

Q: $q_{invalid}, q_{dirty}, q_{shared}, q_{valid}$

X:

T:

$$t_1 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shared} \end{bmatrix} \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shared} \end{bmatrix} \quad t_2 : \begin{bmatrix} q_{invalid} \\ \forall_{LR} \neg (q_{shared} \vee q_{dirty} \vee q_{valid}) \rightarrow \{\} \\ q_{valid} \end{bmatrix}$$

$$t_3 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{dirty} \end{bmatrix} \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{valid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \quad t_4 : \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}$$

$$t_5 : \begin{bmatrix} q_{invalid} \\ \exists_{LR} q_{valid} \vee q_{shared} \rightarrow \{\} \\ q_{shared} \end{bmatrix} \begin{bmatrix} q_{valid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shared} \end{bmatrix}^* \quad t_6 : \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{dirty} \end{bmatrix} \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^*$$

$$t_7 : \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix} \quad t_8 : \begin{bmatrix} q_{valid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix} \quad t_9 : \begin{bmatrix} q_{valid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{dirty} \end{bmatrix}$$

Initial Process State u_{init}: $q_{invalid}$

Final Constraints Φ_F: $q_{dirty}q_{shared}, q_{shared}q_{dirty}, q_{dirty}q_{dirty}$

In the Illinois protocol, unsafe configurations are those where: caches in state q_{dirty} and q_{shared} coexist; or, there is more than 1 cache at state q_{dirty}. Therefore, the choice of the final constraints $q_{dirty}q_{shared}$, $q_{shared}q_{dirty}$ and $q_{dirty}q_{dirty}$ follows.

Mesi In the Mesi cache coherence protocol [20], a cache can be in state: $q_{modified}$, $q_{exclusive}$, q_{shared} or $q_{invalid}$. Initially, all caches are in state $q_{invalid}$. The remaining states are described below.

- A cache is in state $q_{exclusive}$ if the data has been written once by the CPU, and the data is current with the memory.
- A cache is in $q_{modified}$ if the data has been written more than once and the only current copy of the data resides in the cache.
- A cache is in state q_{shared} if there may be other caches holding the same copy of the data.
- The state $q_{exclusive}$ must be entered before the state $q_{modified}$.

Excluding the **read hit** (no actions required), the coherence actions are as follows. In case of a **read miss**, the cache moves from $q_{invalid}$ to q_{shared}. Simultaneously, the other caches in $q_{exclusive}$ or $q_{modified}$ move to q_{shared} (t_2). In case of a **write miss**, the cache goes from $q_{invalid}$ to $q_{exclusive}$ and makes "invalid" the copies of the data in all other caches (t_4). In case of a **write hit**, there are two cases:

- If the cache is in state $q_{exclusive}$ then it moves to $q_{modified}$ (t_1).

– If the cache is in state q_{shared}, then it goes to $q_{exclusive}$ and forces all other caches to move to $q_{invalid}$ (t_3).

<u>Mesi Protocol</u>
Instance
Q: $q_{invalid}, q_{exclusive}, q_{shared} q_{modified}$
X:
T:

$$t_1 : \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \to \{\} \\ q_{modified} \end{bmatrix} \quad t_2 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \to \{\} \\ q_{shared} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \to \{\} \\ q_{shared} \end{bmatrix}^* \begin{bmatrix} q_{modified} \\ \mathbf{tt} \to \{\} \\ q_{shared} \end{bmatrix}^*$$

$$t_3 : \begin{bmatrix} q_{shared} \\ \mathbf{tt} \to \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{shared} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{modified} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^*$$

$$t_4 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \to \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{shared} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{modified} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^*$$

Initial Process State u_{init}: $q_{invalid}$
Final Constraints Φ_F: $q_{modified} q_{shared}, q_{shared} q_{modified}, q_{modified} q_{modified}$

The set of constraints Φ_F denotes in fact the set of unsafe configurations in the Mesi protocol; i.e., configurations where caches in states $q_{modified}$ and q_{shared} coexist, or configurations where there is more than one cache in $q_{modified}$.

Moesi In the Moesi cache coherence protocol [20], the process model of a cache has five states: $q_{modified}$, q_{owned}, $q_{exclusive}$, q_{shared} and $q_{invalid}$. Initially all caches are invalid. Then, depending on the "quality" of the data copy, a cache may be in state:

– q_{shared}: if it contains a copy (of the data) of a cache in state q_{owned}. The data may or may not be up-to-date with the memory.
– $q_{exclusive}$: if the data has been written once and there are no other caches sharing the same data (no other caches in state q_{shared}).
– $q_{modified}$: if the data has been written more than once and there are no other caches sharing it.
– q_{owned}: if the data has been written more than once and other caches are sharing it.

In case of a **read hit**, no coherence actions are required. The remainder scenarios are defined below. In case of a **read miss**, the owner or the memory supplies the data and the cache goes to shared. If the owner is in a state $q_{modified}$ then it moves to q_{owned} and forces other caches possibly in $q_{exclusive}$ to move to q_{shared} (t_2).

Moesi Protocol

Instance

Q: $q_{modified}, q_{owned}, q_{exclusive}, q_{shared}, q_{invalid}$

X:

T:

$$t_1 : \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{modified} \end{bmatrix} \qquad t_2 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shared} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shared} \end{bmatrix}^* \begin{bmatrix} q_{modified} \\ strue \rightarrow \{\} \\ q_{owned} \end{bmatrix}^*$$

$$t_3 : \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{modified} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{owned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^*$$

$$t_4 : \begin{bmatrix} q_{owned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{modified} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{owned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^*$$

$$t_5 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{shared} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{modified} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{owned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^*$$

Initial Process State u_{init}: $q_{invalid}$

Final Constraints Φ_F: $q_{modified}(q_{shared} \vee q_{exclusive} \vee q_{owned}), (q_{shared} \vee q_{exclusive} \vee q_{owned})q_{modified}, q_{exclusive}(q_{shared} \vee q_{owned}), (q_{shared} \vee q_{owned})q_{exclusive}, q_{modified}q_{modified}, q_{exclusive}q_{exclusive}$

In case of a **write hit**, there are two cases:

- If the cache is exclusive, it goes to state $q_{modified}$ (t_1).
- If the cache is in state shared or owned, it goes to state $q_{exclusive}$ and "invalidates" all other caches, i.e, forces all other caches to move to $q_{invalid}$ (t_3 and t_4).

In case of a **write miss**, the cache moves to state $q_{exclusive}$ and invalidates all other caches (t_5).

In this protocol, the set of unsafe configurations consists of those where one of the following holds:

1. A modified cache coexists with a cache in state q_{shared}, $q_{exclusive}$ or q_{owned}. This set is denoted by the constraints $q_{modified}(q_{shared} \vee q_{exclusive} \vee q_{owned})$ and $(q_{shared} \vee q_{exclusive} \vee q_{owned})q_{modified}$.
2. An exclusive cache coexists with a shared or an owned cache. In order to denote this set, we use the constraints $q_{exclusive}(q_{shared} \vee q_{owned})$ and $(q_{shared} \vee q_{owned})q_{exclusive}$.
3. There is more than one cache in state $q_{exclusive}$. These configurations are in the set $[\![q_{exclusive}q_{exclusive}]\!]$.
4. There is more than one cache in state $q_{modified}$. This set of configurations is equal to $[\![q_{modified}q_{modified}]\!]$.

Berkeley Berkeley cache coherence protocol [20] induces a parameterized system where the states are in $\{q_{unowned}, q_{non-exclusive}, q_{exclusive}, q_{invalid}\}$. Initially, all caches are in state $q_{invalid}$. The rest of the states is defined according to the "status" of the data copies as follows:

- $q_{non-exclusive}$: the memory is not coherent with the possibly multiple cached copies of the owner's data.
- $q_{unowned}$: there may be other caches sharing the data.
- $q_{exclusive}$: the only current data resides in this cache.

Berkeley Protocol

Instance

Q: $q_{unowned}, q_{non-exclusive}, q_{exclusive}, q_{invalid}$

X:

T:

$$t_1 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{unowned} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{non-exclusive} \end{bmatrix}^*$$

$$t_2 : \begin{bmatrix} q_{unowned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{non-exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{unowned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^*$$

$$t_3 : \begin{bmatrix} q_{non-exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{unowned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{non-exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^*$$

$$t_4 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclusive} \end{bmatrix} \begin{bmatrix} q_{non-exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{unowned} \\ \mathbf{tt} \rightarrow \{\} \\ q_{invalid} \end{bmatrix}^*$$

Initial Process State u_{init}: $q_{invalid}$

Final Constraints Φ_F: $q_{exclusive}(q_{unowned} \vee q_{non-exclusive}), (q_{unowned} \vee q_{non-exclusive})q_{exclusive}, q_{exclusive}q_{exclusive}$

Coherence actions are not required in case of a **read hit**. In case of a **read miss**, the cache updates the data from the owner and goes to state $q_{unowned}$. Simultaneously, all cache in $q_{exclusive}$ moves to $q_{non-exclusively}$ (t_1). In case of a **write miss**, the cache moves to $q_{exclusively}$. The other caches are invalidated (t_4). In the case of a **write hit**, then if the cache is in state $q_{unowned}$ or $q_{non-exclusively}$, it moves to $q_{exclusive}$ while invalidating the caches in states $q_{non-exclusive}$ and $q_{unowned}$ (t_2 and t_3).

In the Berkeley protocol, unsafe configurations are those where a cache in state $q_{exclusive}$ coexists with a cache in $q_{unowned}$ or $q_{non-exclusive}$, or where there is more than one cache in state $q_{exclusive}$.

Synapse For Synapse cache coherence protocol [20], the model states are q_{valid}, q_{dirty} and $q_{invalid}$. Initially, caches are in state $q_{invalid}$.

The coherence actions are as follows.

read miss: The cache moves to state q_{valid}. All caches in a state q_{dirty} move to $q_{invalid}$ (t_1).
write hit: If the cache is valid, it moves to state q_{dirty} invalidating all other caches (t_2).
write miss: All caches are invalidated (t_3).

Synapse Protocol
Instance
Q: $q_{valid}, q_{dirty}, q_{invalid}$
X:
T:

$$t_1 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \to \{\} \\ q_{valid} \end{bmatrix} \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \quad t_2 : \begin{bmatrix} q_{valid} \\ \mathbf{tt} \to \{\} \\ q_{dirty} \end{bmatrix} \begin{bmatrix} q_{valid} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^*$$

$$t_3 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \to \{\} \\ q_{dirty} \end{bmatrix} \begin{bmatrix} q_{valid} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^*$$

Initial Process State u_{init}: $q_{invalid}$
Final Constraints Φ_F: $q_{dirty}q_{dirty}, q_{valid}q_{dirty}, q_{dirty}q_{valid}$

In this protocol unsafe configurations are those where there is more than one cache in q_{dirty} (denoted by the constraint $q_{dirty}q_{dirty}$), or where caches in q_{dirty} and in q_{valid} coexist (corresponding constraints are $q_{dirty}q_{valid}$ and $q_{valid}q_{dirty}$).

DEC Firefly In the parameterized system model derived from the DEC Firefly cache coherence protocol [20], each process has four states: q_{dirty}, $q_{exclusive}$, q_{shared} and $q_{invalid}$. A cache is in state $q_{exclusive}$ if it has the only copy of the data in the memory. While, a cache is in state q_{shared} if there are other copies of the data. Finally, a cache is "dirty" (in q_{dirty}) if it has a copy that is not consistent with the memory. Initially all processes are at state $q_{invalid}$.

The coherence actions are as follows.

read miss: There are three cases:
- If there is another cache in q_{dirty}, the latter supplies the data, writes it back to the memory and both caches move to q_{shared} (t_1).
- If there is a cache in state shared or exclusive, then the latter supplies the data, and all caches in $q_{exclusive}$ go to q_{shared} (t_3).
- If there is no cached copy , (i.e., all other caches are in $q_{invalid}$), then the current cache gets a copy from memory and moves to state $q_{exclusive}$ (t_2).

write hit: One of the following coherence actions is required:
- If the cache is in state $q_{exclusive}$, then it moves to q_{dirty} (t_4).
- If the cache is the only one in q_{shared}, then it moves to $q_{exclusive}$ (t_6).

write miss: The cache moves to q_{dirty} and simultaneously all other copies are invalidated, i.e., all other caches go to q_{dirty} (t_5).

Observe that no coherence actions are needed for a **read hit**.

DEC Firefly Protocol

Instance

Q: $q_{valid}, q_{exclusive}, q_{shared}, q_{invalid}$.

X:

T:

$$t_1 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \to \{\} \\ q_{shared} \end{bmatrix} \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \to \{\} \\ q_{shared} \end{bmatrix} \qquad t_2 : \begin{bmatrix} q_{invalid} \\ \forall_{LR}\ q_{invalid} \to \{\} \\ q_{exclusive} \end{bmatrix}$$

$$t_3 : \begin{bmatrix} q_{invalid} \\ \exists_{LR}(q_{shared} \vee q_{exclusive}) \to \{\} \\ q_{shared} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \to \{\} \\ q_{shared} \end{bmatrix}^* \qquad t_4 : \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \to \{\} \\ q_{dirty} \end{bmatrix}$$

$$t_5 : \begin{bmatrix} q_{invalid} \\ \mathbf{tt} \to \{\} \\ q_{dirty} \end{bmatrix} \begin{bmatrix} q_{shared} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \to \{\} \\ q_{invalid} \end{bmatrix}^* \qquad t_6 : \begin{bmatrix} q_{shared} \\ \forall_{LR} \neg q_{shared} \to \{\} \\ q_{exclusive} \end{bmatrix}$$

Initial Process State u_{init}: $q_{invalid}$

Final Constraints Φ_F: $q_{dirty}(q_{shared} \vee q_{exclusive}), (q_{shared} \vee q_{exclusive})q_{dirty}, q_{exclusive}q_{exclusive}, q_{dirty}q_{dirty}$

In the DEC Firefly protocol, a configuration is unsafe if it satisfies one of these conditions.

- A cache in state q_{dirty} coexists with a cache in q_{shared} or $q_{exclusive}$, thus the choice of the constraints $q_{dirty}(q_{shared} \vee q_{exclusive})$ and $(q_{shared} \vee q_{exclusive})q_{dirty}$.
- Two or more caches in q_{dirty} coexist ($q_{dirty}q_{dirty}$).
- There is more than one cache is in state $q_{exclusive}$ ($q_{exclusive}q_{exclusive}$).

Xerox Parc Dragon From the Xerox Parc Dragon cache coherence protocol [20], we derived a parameterized system where states are: $q_{invalid}$, q_{sclean} (for shared clean), q_{sdirty} (for shared dirty), $q_{exclusive}$ and q_{dirty}. For instance, a cache is in q_{sclean} if other caches are sharing a clean (up-to-date) copy of the same data. It is in state q_{sdirty} if there are caches sharing a dirty (not up-to-date) copy of the same data. A cache is exclusive if it holds exclusively a clean copy of the data. Finally, the state q_{dirty} implies that the cache contains an exclusive dirty copy. The state $q_{invalid}$ is added to model the initial state.

The coherence actions are described below.

read hit: There are no coherence actions.

read miss: Two cases are possible.

- If there are no other caches sharing copies of the data, then the cache moves to $q_{exclusive}$ (t_2).
- if there are caches holding copies of the data, then the cache moves to q_{sclean}, while other caches at states $q_{exclusive}$ and q_{dirty} move respectively to q_{sclean} and q_{sdirty} (t_8).

write hit: There are three cases.

- If the cache is in $q_{exclusive}$, then it moves to q_{dirty} (t_6).
- If the cache is in state q_{sclean} or q_{sdirty} and there are no other caches sharing copies of the data, then it moves to state dirty (t_1 and t_3).
- If the cache is shared clean or shared dirty and there are other caches having copies of the data, then it moves to state $q_{shared-dirty}$ and forces all other caches in q_{sdirty} to go to q_{sclean} (t_5 and t_7).

write miss: Two cases are possible.

- If there are no other cached copies of the data, then the cache moves to q_{dirty} (t_4).
- If there are other copies, then the cache moves to state q_{sdirty}, while simultaneously all other caches holding copies move to q_{sclean} (t_9).

Xerox Parc Dragon Protocol

Instance

Q: $q_{invalid}, q_{sclean}, q_{sdirty}, q_{exclusive}, q_{dirty}$

X:

T:

$$t_1 : \begin{bmatrix} q_{sdirty} \\ \forall_{LR}\neg(q_{sclean} \vee q_{sdirty}) \rightarrow \{\} \\ q_{dirty} \end{bmatrix} \quad t_2 : \begin{bmatrix} q_{invalid} \\ \forall_{LR}\ q_{invalid} \rightarrow \{\} \\ q_{exclusive} \end{bmatrix}$$

$$t_3 : \begin{bmatrix} q_{sclean} \\ \forall_{LR}\neg(q_{sclean} \vee q_{sdirty}) \rightarrow \{\} \\ q_{dirty} \end{bmatrix} \quad t_4 : \begin{bmatrix} q_{invalid} \\ \forall_{LR}\ q_{invalid} \rightarrow \{\} \\ q_{dirty} \end{bmatrix}$$

$$t_5 : \begin{bmatrix} q_{sdirty} \\ \exists_{LR}(q_{sclean} \vee q_{sdirty}) \rightarrow \{\} \\ q_{sdirty} \end{bmatrix} \begin{bmatrix} q_{sdirty} \\ \mathbf{tt} \rightarrow \{\} \\ q_{sclean} \end{bmatrix}^* \quad t_6 : \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{dirty} \end{bmatrix}$$

$$t_7 : \begin{bmatrix} q_{sclean} \\ \exists_{LR}(q_{sclean} \vee q_{sdirty}) \rightarrow \{\} \\ q_{sdirty} \end{bmatrix} \begin{bmatrix} q_{sdirty} \\ \mathbf{tt} \rightarrow \{\} \\ q_{sclean} \end{bmatrix}^*$$

$$t_8 : \begin{bmatrix} q_{invalid} \\ \exists_{LR}\neg q_{invalid} \rightarrow \{\} \\ q_{sclean} \end{bmatrix} \begin{bmatrix} q_{dirty} \\ \mathbf{tt} \rightarrow \{\} \\ q_{sdirty} \end{bmatrix}^* \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{sclean} \end{bmatrix}^*$$

$$t_9 : \begin{bmatrix} q_{invalid} \\ \exists_{LR}\neg q_{invalid} \rightarrow \{\} \\ q_{sdirty} \end{bmatrix} \begin{bmatrix} q_{exclusive} \\ \mathbf{tt} \rightarrow \{\} \\ q_{sclean} \end{bmatrix}^* \begin{bmatrix} q_{sdirty} \\ \mathbf{tt} \rightarrow \{\} \\ q_{sclean} \end{bmatrix}^*$$

Initial Process State u_{init}: $q_{invalid}$

Final Constraints Φ_F: $q_{dirty}(q_{sdirty} \vee q_{sclean} \vee q_{exclusive}), (q_{sdirty} \vee q_{sclean} \vee q_{exclusive})q_{dirty}, q_{exclusive}(q_{sclean} \vee q_{sdirty}), (q_{sclean} \vee q_{sdirty})q_{exclusive}, q_{exclusive}q_{exclusive}, q_{dirty}q_{dirty}$

For this protocol we check the following invariants.

1. No cache in q_{sdirty} coexists with a cache in state q_{sclean}, q_{sdirty} or $q_{exclusive}$. The constraints that denote the set of configurations violating it are $q_{dirty}(q_{sdirty} \vee q_{sclean} \vee q_{exclusive})$ and $(q_{sdirty} \vee q_{sclean} \vee q_{exclusive})q_{dirty}$.

2. No cache in state $q_{exclusive}$ coexists with a shared clean or shared dirty cache. The complement of this set is denoted by the constraints $q_{exclusive}(q_{sclean} \vee q_{sdirty})$ $(q_{sclean} \vee q_{sdirty})q_{exclusive}$.
3. There is at most one cache in state $q_{exclusive}$. We verify that by checking reachability of the constraint $q_{exclusive}q_{exclusive}$.
4. There is at most one cache in state q_{dirty}. The corresponding constraint is $q_{dirty}q_{dirty}$.

Futurebus+ The Futurebus+ coherence protocol is described in [20]. The parameterized model we derive is from [12]. This protocol uses a signal denoted by tf^* for *transaction flag.* The state of this flag depends on the transaction taking effect. By monitoring a tf* signal, the requesting cache has the option of taking a line immediately into an *exclusive* state, if that line is copied in no other cache.

Our parameterized model of this protocol is inspired from the state machines described in [12]. Read cycles are described by means of the following local states of cache lines: q_{inv} (invalid), q_{shU} (shared unmodified), q_{exclU} (exclusive unmodified), q_{exclM} (exclusive modified), q_{pendR} (pending read command), q_{pendE} (a cache in exclusive modified state is ready to respond with the data to a read command); q_{pendSU} (caches in q_{shU} or in q_{exclU} state are ready to assert the tf* signal). Write cycles are described via two additional local states: q_{pendW} indicates caches that have issued a *read modified* command and wait for a data acknowledgment; q_{pendEW} indicates that a cache in state q_{exclM} is preparing to issue the data acknowledgment. In our model we assume that the signal tf* is up only if there exists a cache in state q_{pendE} or q_{pendSU}, i.e., we model tf* $= \mathbf{tt}$ iff the guard $\exists_{LR}(q_{pendE} \vee q_{pendSU})$ is satisfied. Furthermore, to avoid trivial inconsistencies we assume that a cache in state q_{inv} can issue a *read shared* or a *read modified* command only if there are no pending write commands ($\forall_{LR}\neg q_{pendW}$).

1. Read hits do not cause state transitions.
2. When a cache issues a *read shared* command and goes to q_{pendR}, all caches in state q_{shU} or q_{exclU} go to q_{pendSU} and prepare to issue the signal tf* A cache in state q_{exclM} goes to q_{pendE} and prepares to send a data acknowledgment (ready signal) (t_1).
3. If the data acknowledgment (together with tf*) is sent by a cache in state q_{pendE} or by main memory and tf* has been asserted then the caches with pending read go to q_{shU} (t_2 and t_3).
4. If tf* has not been asserted and more than one cache has pending read requests than all caches move to q_{shU} (t_4).
5. If tf* has not been asserted and only one cache has a pending read than its state changes to q_{exclU} (t_5).
6. When a cache issues a *read modified* command and goes to q_{pendW}, the cache in state q_{exclM} goes to q_{pendEW}, while all other caches go to q_{inv} (t_6).
7. When a data acknowledgment is issued than the cache goes to *exclM* (t_7 and t_8).
8. Write hits is in state q_{exclU} and q_{exclM} need no consistency actions (t_9 and t_10).

9. When the cache is in state q_{shU} all other caches in state q_{shU} are forced to the state *invalid* (t_{11}).

Futurebus+ Protocol

Instance

Q: $q_{pendR}, q_{shU}, q_{pendSU}, q_{pendE}, q_{exclU}, q_{exclM}, q_{pendEW}, q_{pendW}, q_{inv}$

X:

T:

$$t_1 : \begin{bmatrix} q_{inv} \\ \forall_{LR} \neg q_{pendW} \rightarrow \{\} \\ q_{pendR} \end{bmatrix} \begin{bmatrix} q_{exclM} \\ \mathbf{tt} \rightarrow \{\} \\ q_{pendE} \end{bmatrix}^* \begin{bmatrix} q_{shU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{pendSU} \end{bmatrix}^* \begin{bmatrix} q_{exclU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{pendSU} \end{bmatrix}^*$$

$$t_2 : \begin{bmatrix} q_{pendE} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shU} \end{bmatrix} \begin{bmatrix} q_{pendR} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shU} \end{bmatrix}^* \quad t_3 : \begin{bmatrix} q_{pendSU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shU} \end{bmatrix} \begin{bmatrix} q_{pendR} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shU} \end{bmatrix}^* \begin{bmatrix} q_{pendSU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{shU} \end{bmatrix}^*$$

$$t_4 : \begin{bmatrix} q_{pendR} \\ (\forall_{LR} \neg (q_{pendSU} \vee q_{pendE}) \wedge (\exists_{LR} q_{pendR}) \rightarrow \{\} \\ q_{shU} \end{bmatrix} \begin{bmatrix} q_{pendR} \\ \rightarrow \{\} \\ q_{shU} \end{bmatrix}^*$$

$$t_5 : \begin{bmatrix} q_{pendR} \\ \forall_{LR} \neg (q_{pendSU} \vee q_{pendE} \vee q_{pendR}) \rightarrow \{\} \\ q_{exclU} \end{bmatrix}$$

$$t_6 : \begin{bmatrix} q_{inv} \\ \forall_{LR} \neg q_{pendW} \rightarrow \{\} \\ q_{pendW} \end{bmatrix} \begin{bmatrix} q_{pendE} \\ \mathbf{tt} \rightarrow \{\} \\ q_{inv} \end{bmatrix}^* \begin{bmatrix} q_{shU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{inv} \end{bmatrix}^* \begin{bmatrix} q_{exclU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{inv} \end{bmatrix}^*$$
$$\begin{bmatrix} q_{pendSU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{inv} \end{bmatrix}^* \begin{bmatrix} q_{pendR} \\ \mathbf{tt} \rightarrow \{\} \\ q_{inv} \end{bmatrix}^* \begin{bmatrix} q_{pendEW} \\ \mathbf{tt} \rightarrow \{\} \\ q_{inv} \end{bmatrix}^* \begin{bmatrix} q_{exclM} \\ \mathbf{tt} \rightarrow \{\} \\ q_{pendEW} \end{bmatrix}^*$$

$$t_7 : \begin{bmatrix} q_{pendEW} \\ \mathbf{tt} \rightarrow \{\} \\ q_{inv} \end{bmatrix} \begin{bmatrix} q_{pendW} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclM} \end{bmatrix}^* \quad t_8 : \begin{bmatrix} q_{pendW} \\ \forall_{LR} \neg q_{pendEW} \rightarrow \{\} \\ q_{exclM} \end{bmatrix} \begin{bmatrix} q_{pendW} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclM} \end{bmatrix}^*$$

$$t_9 : \begin{bmatrix} q_{exclU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclM} \end{bmatrix} \quad t_1 0 : \begin{bmatrix} q_{exclM} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclM} \end{bmatrix} \quad t_{11} : \begin{bmatrix} q_{shU} \\ \mathbf{tt} \rightarrow \{\} \\ q_{exclM} \end{bmatrix} \begin{bmatrix} q_{shU} \\ \rightarrow \{\} \\ q_{inv} \end{bmatrix}^*$$

Initial Process State u_{init}: q_{inv}

Final Constraints Φ_F: $q_{shU}(q_{exclM} \vee q_{exclU}), (q_{exclM} \vee q_{exclU}) q_{shU}, q_{exclU}(q_{exclU} \vee q_{exclM}), q_{exclM} q_{exclU}, q_{pendW}(q_{pendW} \vee q_{pendR}), q_{pendR} q_{pendW}$

For this model, we were able to automatically verify that no more than one cache is in an exclusive state, that different caches cannot be simultaneously in exclusive and shared states, that there cannot be simultaneously caches that emitted write and read commands or more than one write command.

German Cache Coherence Protocol [29] 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 [29], 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 to remember whether the exclusive access has been granted, modeled with a shared Boolean variable ($excGran$),

(ii) a pointer to the client that sent the request being served, modeled with a local Boolean variable ($curClt$),

(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 which have to be invalidated in order to serve the current request, modeled also with a local Boolean variable $iLst$.

Depending on the channels content and the local state, a client may perform one of the following actions.

1. If in state invalid and $channel_1$ is empty, then the client sends a request for a shared access via $channel_1$ (p_1).
2. If in state invalid or shared while $channel_1$ is empty, the client sends a request for the exclusive access via $channel_1$ (p_2 and p_3).
3. If the client receives a grant for shared access via $channel_2$, then it moves to state shared and empties $channel_2$ (p_5).
4. If it receives a grant for exclusive access via $channel_2$, then the client moves to state exclusive and empties $channel_2$ (p_6).
5. 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_4).

German Protocol

Instance

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

X: $curClt, sLst, iLst \in \mathcal{B}$, $ch_1 \in \{\epsilon, reqSh, reqExc\}$,
$ch_2 \in \{\epsilon, grantSh, grantExc, inval\}$, $ch_3 \in \{\epsilon, invAck\}$

S: $excGran \in \mathcal{B}, curCm \in \{\epsilon, reqSh, reqExc\}$

T:

$$h_0 : \left[\begin{array}{c} q_{all} \\ curCm = reqSh, \neg excGran, ch_2 = \epsilon, curClt \\ \to curCm = \epsilon, sLst, ch_2 = grantSh \\ q_{all} \end{array}\right]$$

$$h_1 : \left[\begin{array}{c} q_{all} \\ curCm = reqExc, ch_2 = \epsilon, \forall_{LR} \neg sLst, \neg sLst \\ \to curCm = \epsilon, sLst, excGran, ch_2 = grantExc \\ q_{all} \end{array}\right]$$

$$h_2 : \left[\begin{array}{c} q_{all} \\ curCm = \epsilon, ch_1 \neq \epsilon \\ \to curCm = ch_1, ch_1 = \epsilon, iLst = sLst, curClt \\ q_{all} \end{array}\right] \left[\begin{array}{c} q_{all} \\ \mathbf{tt} \\ \to iLst = sLst, \neg curClt \\ q_{all} \end{array}\right]^*$$

$$h_3 : \left[\begin{array}{c} q_{all} \\ curCm = reqSh, excGran, iLst, ch_2 = \epsilon \\ \to \neg iLst, ch_2 = inval \\ q_{all} \end{array}\right]$$

$$h_4 : \left[\begin{array}{c} q_{all} \\ curCm = reqExc, iLst, ch_2 = \epsilon \\ \to \neg iLst, ch_2 = inval \\ q_{all} \end{array}\right] h_5 : \left[\begin{array}{c} q_{all} \\ curCm \neq \epsilon, ch_3 = invAck \\ \to \neg sLst, \neg excGran, ch_3 = \epsilon \\ q_{all} \end{array}\right]$$

$$p_1 : \left[\begin{array}{c} q_{inv} \\ ch_1 = \epsilon \\ \to ch_1 = reqSh \\ q_{inv} \end{array}\right] p_2 : \left[\begin{array}{c} q_{inv} \\ ch_1 = \epsilon \\ \to ch_1 = reqExc \\ q_{inv} \end{array}\right] p_3 : \left[\begin{array}{c} q_{sh} \\ ch_1 = \epsilon \\ \to ch_1 = reqExc \\ q_{sh} \end{array}\right]$$

$$p_4 : \left[\begin{array}{c} q_{all} \\ ch_2 = inval, ch_3 = \epsilon \\ \to ch_2 = \epsilon, ch_3 = invAck \\ q_{inv} \end{array}\right] p_5 : \left[\begin{array}{c} q_{all} \\ ch_2 = grantSh \\ \to ch_2 = \epsilon \\ q_{sh} \end{array}\right] p_6 : \left[\begin{array}{c} q_{all} \\ ch_2 = grantExc \\ \to ch_2 = \epsilon \\ q_{exc} \end{array}\right]$$

Initial State of Shared Vars $excGran \mapsto \mathbf{ff}, curCm \mapsto \epsilon$

Initial Process State u_{init}:

$q_{inv}, (ch_1, ch_2, ch_3, curClt, sLst, iLst) \mapsto (\epsilon, \epsilon, \epsilon, \mathbf{ff}, \mathbf{ff}, \mathbf{ff})$

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

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

1. If it is idle ($curCm$ empty) and receives a request via $channel_1$, then Home updates $curCm$ 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_2).

2. 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 $channel_2$ and adds the client to the shared list and becomes idle (h_0).
3. 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 the exclusive access to the current client via $channel_2$, adds the client to the sharer list, sets the exclusive flag and becomes idle (h_1).
4. If the current command is either a shared request (while the exclusive flag is set) or an exclusive request, $channel_2$ is empty, then Home sends an invalidation message every process through $channel_2$ and removes these processes from the invalidation list (h_3 and h_4).
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 $channel_3$ (h_5).

To simplify the presentation, we used assignment statement of the form $x = x'$ where x and x' are different variables (like in 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, (ii) no client in state shared coexists with a client in state exclusive.