Formal Methods in System Design, 23, 257–301, 2003

Constraint-Based Verification of Parameterized Cache Coherence Protocols

GIORGIO DELZANNO giorgio@disi.unige.it
Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, via Dodecaneso 35, I-16146 Genova, Italy

Received September 22, 2000; Revised November 19, 2002; Accepted November 20, 2002

Abstract. We propose a new method for the parameterized verification of formal specifications of *cache coherence protocols*. The goal of parameterized verification is to establish system properties for an arbitrary number of caches. In order to achieve this purpose we define abstractions that allow us to reduce the original parameterized verification problem to a *control state reachability problem* for a system with integer data variables. Specifically, the methodology we propose consists of the following steps. We first define an abstraction in which we only keep track of the number of caches in a given state during the execution of a protocol. Then, we use linear arithmetic constraints to symbolically represent infinite sets of global states of the resulting abstract protocol. For reasons of efficiency, we relax the constraint operations by interpreting constraints over real numbers. Finally, we check parameterized safety properties of abstract protocols using symbolic backward reachability, a strategy that allows us to obtain sufficient conditions for termination for an interesting class of protocols. The latter problem can be solved by using the infinite-state model checker HYTECH: Henzinger, Ho, and Wong-Toi, "A model checker for hybrid systems," *Proc. of the 9th International Conference on Computer Aided Verification (CAV'97)*, Lecture Notes in Computer Science, Springer, Haifa, Israel, 1997, Vol. 1254, pp. 460–463. HYTECH handles linear arithmetic constraints using the *polyhedra library* of Halbwachs and Proy, "Verification of real-time systems using linear relation analysis," *Formal Methods in System Design*, Vol. 11, No. 2, pp. 157–185, 1997. By using this methodology, we have automatically validated parameterized versions of widely implemented *write-invalidate* and *write-update* cache coherence protocols like Synapse, MESI, MOESI, Berkeley, Illinois, Firefly and Dragon (Handy, *The Cache Memory Book*, Academic Press, 1993). With this application, we have shown that symbolic model checking tools like HYTECH, originally designed for the verification of hybrid systems, can be applied successfully to new classes of *infinite-state* systems of practical interest.

Keywords: cache coherence protocols, abstractions, constraints, symbolic model checking

1. Introduction

In a shared-memory multiprocessor system *local caches* are used to reduce memory access latency and network traffic. Each processor is connected to a fast memory backed up by a large (and slower) main memory. This configuration enables processors to work on local copies of main memory blocks, greatly reducing the number of memory accesses that the processor must perform during program execution. Although local caches improve system performance, they introduce the *cache coherence problem*: multiple cached copies of the same block of memory must be consistent at any time during a run of the system. A *cache coherence protocol* ensures the *data consistency* of the system: *the value returned by a read must always be the last value written to that location* (cf. [5, 27]). Coherence policies can be

described as *finite state machines* that specify the way a single cache reacts to read and write requests. As an example, let us consider a CC-UMA (Uniform Memory Access with local Caches) multiprocessor system, i.e., a system in which all processors have a local cache connected to the main memory via a shared bus. In *write-invalidate* protocols, whenever a processor modifies its cache block a *bus invalidation signal* is sent to all other caches in order to invalidate their data. By contrast, in *write-update* protocols a copy of the new data is sent to all caches that share the old data.

Due to the increasing complexity of hardware architectures, the development of *automatic* verification techniques is becoming a major goal to discover errors at an early stage of protocol design (see e.g. [14, 37]). One of the main challenges in this area is to develop techniques for validating protocols for *any number* of processors and caches.

Drawing inspiration from recent works like [21, 20, 23], in this paper we propose a new method for the verification of *parameterized* cache coherence protocols at the *behavior* level. As mentioned before, a multiprocessor system can be modelled as a collection of many *identical* finite-state machines. As a first step, we apply the following *abstraction*: we keep track of the *number* of caches in every possible protocol state, only. The resulting abstract protocol can be represented as a transition system with integer data paths. Abstract protocols can be formally described as Extended Finite State Machines (EFSMs) [12]. Via this abstraction, we represent all *symmetric* global states (collection of individual cache states) using a single EFSM-state.

As a second step, we use arithmetic constraints to implicitly represent potentially infinite sets of EFSM-states (tuples of non-negative integers). This way, we express *data consistency* properties independently from the number of processors. Furthermore, the verification problem for parameterized cache coherence protocols can be reduced to a *reachability problem* for EFSMs. The latter problem can be attacked using *general purpose*, *infinite-state* symbolic model checking methods working over integer [8, 11] or real arithmetics [18, 26, 28].

Following the general methodology proposed in [18], we apply here efficient tools based on *real* arithmetics. Specifically, we apply a *relaxation* from integers to reals for the symbolic operations used in backward reachability, namely existential quantification, satisfiability, and entailment over linear constraints. The resulting procedure can be applied to *automatically* check safety properties like *data-consistency* and *exclusivity* of special states for *snoopy*, *write-invalidate* and *write-update* cache coherence protocols for CC-UMA multiprocessors [5, 27, 43].

1.1. Summary of contributions

The conceptual and technical contributions of this paper can be summarized as follows.

Abstractions. We first show that parameterized versions of a large class of cache coherence protocols can be formulated in terms of EFSMs. The class of EFSMs we consider here is an extension of the operational model underlying the Broadcast Protocols of Emerson and Namjoshi [20]. Specifically, in our model transitions for a single cache can depend, in a restricted way, on the global state of the system and thus on the state of other caches. This feature is necessary to model coherence policies such as the Illinois protocol

without abstracting away properties that are crucial for their validation (see discussion in Section 9).

Backward reachability. We define a general method for validating protocols. The method is based on the use of the *backward reachability* procedure introduced in [2]. This procedure is used to check invariant properties of the EFSMs resulting from the application of the counting abstraction. In contrast to *forward* reachability, the procedure of [2] is guaranteed to terminate for the subclass of EFSMs associated with Broadcast Protocols and for set of unsafe states that are upward-closed [21]. Interestingly, the set of possible *violations* of safety properties can often be modelled as upward-closed sets [3].

Constraints. We choose a symbolic representation of (potentially infinite) sets of states based on *linear arithmetic constraints*. The constraint operations of *variable elimination*, *satisfiability* and *entailment test* can be used to implement a symbolic version for the procedure of [2]. Following [18], in order to obtain an *efficient* procedure we interpret the above mentioned constraint operations over *reals* instead that over *integers*. This *relaxation* technique is widely-used in integer programming and program analysis.

As for other methods handling *global conditions* in parameterized systems (e.g. [1]) and other methods for infinite-state systems (e.g. [8, 11]), our symbolic backward reachability procedure is a *semi-algorithm* (it may not terminate) that must be evaluated on practical examples. In the paper we isolate, however, a special class of verification problems for which our symbolic procedure (working over reals) is guaranteed to solve the *control reachability problem*. The decidable class is such that the input EFSM is a Broadcast Protocol [20], and the unsafe states are represented via a special class of constraints that denote *upward-closed* sets of markings. This result seems to be a new application of the general theory of well-structured infinite-state systems [2, 22].

For practical experiments, we have used the symbolic model checker HYTECH [28] that implements the symbolic backward reachability procedure described above. HYTECH is defined on top of the *polyhedra manipulation library* of [38].

Using our methodology and the infinite-state model checker HYTECH, we have verified several safety properties for the Synapse N + 1, MESI, MOESI, University of Illinois, Berkeley RISC, DEC Firefly and Xerox PARC Dragon protocols [5, 27, 42]. Although the termination of our method is guaranteed only for Broadcast Protocols, the experimental results show that it performs well in practice. To our knowledge, this is the first time that *general purpose* symbolic model checkers for infinite-state systems working over arithmetic domains are used for verification of parameterized cache coherence protocols.

With this application, we have shown that techniques developed in the last years for *hybrid* [28] and *concurrent systems* with local data ranging over integer values [11, 18] can also be applied to a new class of infinite-state systems of practical interest.

Plan of the paper. After discussing related works (Section 2), in Section 3 we will introduce the cache coherence problem and the terminology used in the corresponding protocols. In Section 4 we will introduce a finite model for individual caches. The model is based on global synchronization labels and on global guards. In Section 5 we will introduce

the counting abstraction that reduce a family of finite state machine into a single extended finite state machine. In Section 6 we will introduce a symbolic representation of sets of states of extended finite state machines via constraints. In Section 7 we will introduce the symbolic backward reachability procedure we propose as verification method. In Section 8 we will present a gallery of case-studies and related verification problems. In Section 9 we will discuss experimental results obtained using the tool HYTECHIn Section 10 we will address future directions of research.

Note. A short version of this paper appeared in [15].

2. Related work

In [20] Emerson and Namjoshi present a generalization of the coverability tree for Petri Nets to be used for more general parameterized systems. Several forward algorithms used for infinite state systems like [30, 42] are an instance of this general verification framework. Among other examples, in [20] Emerson and Namjoshi apply their method to the analysis of extensions of Petri Nets, called *Broadcast Protocols*. Broadcast Protocols provide primitives for the definition of internal actions, rendez-vous and broadcast communication. Thus, they can be used to model abstractions of cache coherence policies like MESI [27]. However, Broadcast Protocols are not general enough to describe the *global* conditions required by the protocols we have validated in our experiments.

The framework proposed in [20] makes us of acceleration operators based on a simulation preorder defined over global system states. For Broadcast Protocols, the adopted simulation preorder is the component-wise ordering of Petri Nets markings enriched with the ω tag, i.e., tuples of fixed size defined over non-negative integers and ω, where $\omega > k$ for any integer k. For instance, let us consider the chain of markings having the form $\langle 1, 0, k\rangle$, where k is a non-negative integer. Then, every element of the chain is strictly less than the tuple $\langle 1, 0, \omega\rangle$. The tuple $\langle 1, 0, \omega\rangle$ is indeed the least upper bound of the chain. During the construction of the reachability graph, chains of markings (generated using the post-image operator) are replaced by their least upper bound. This way, infinite paths in the resulting approximation of the reachability graph are compressed and replaced by their "limit". As proved by the counterexample given by Esparza, Finkel and Mayr in [21], this verification technique may not terminate when applied to Broadcast Protocols.

An alternative verification procedure for infinite-state systems has been introduced by Abdulla et al. in [2]. This procedure is based on backward reachability combined with a symbolic representation of sets of states based on *constraints*. The *entailment* relation (containment of constraint denotations), extended in natural way to sets of constraints, is used here as termination test. The resulting procedure is guaranteed to terminate whenever the following conditions are satisfied:

- there exists an algorithm (*pre-image* operator) that, given a set of constraints, computes a representation of the predecessors of the states denoted by the constraints;
- the class of constraints under consideration is closed under application of the *pre-image* operator;
- entailment is a *well-quasi-ordering* [2].

In [21] Esparza, Finkel and Mayr have shown that, for Broadcast Protocols, backward reachability always terminates when the target of the analysis is an *upward-closed set* of markings. An upward-closed set of markings S is such that for every marking $\mathbf{m} \in S$: all markings that are greater equal to $\mathbf{m} \in S$ in the component-wise ordering are contained in S. Upward closed set of markings can always be represented via a unique and finite set of *minimal markings*. The well-quasi ordering here is the component-wise ordering of markings of Petri Nets.

Upward closed sets of markings can be symbolically represented via linear arithmetic constraints. For instance, the infinite set of markings with at least 2 tokens in a Petri Net with two places can be represented as the constraint $x + y \geq 2$ where x and y are variables associated with the two places. Efficient data structures for representing integer constraints for the verification of Broadcast Protocols have been studied in [17]. Graph-based data structures for verification of Unbounded Petri Nets have been studied in [19].

The formalism of Extended Finite State Machines (EFSMs) introduced by Cheng and Krishnakumar in [12] gives us a convenient way to represent the operational model underlying Broadcast Protocols. EFSMs have been designed as a high-level formalisms for hardware circuits. Basically, an EFSM can be viewed as an extension of finite-state machines in which expressions defined over a finite set of variables ranging over integer values can be used as guards and actions of a transition. Local variables can be used to compactly represent the values stored in registers. EFSM transitions have the following form $s \rightarrow t : (f, i)/(u, o)$. A transition depends on an input signal i and on a guard f defined over the local variables (a high-level definition of conditions over numerical data). When executed they give rise to an output signal o and to an update of the current values of the local variables u (a high-level specification of operations of numerical data). This formalism can be used to symbolically explore the state space of complex circuits. The symbolic representation of sets of states is based on *regions* defined over the data variables.

In the paper we will use a restricted form of EFSMs without input and output signals and with a single location. Specifically, EFSMs will be used to represent the evolution of the counters associated to the local state of caches. The counters will be modelled via EFSM data variables. More in general, the finite-state component of an EFSM can be used to model a single monitor cooperating with several processes as in [16]. In [16] the evolution of the state of the monitor is modelled via transitions over a finite sets of states, annotated with guards and actions over counters that describe the evolution of client processes. For instance, the following rule models a synchronization step between the monitor and one of the processes in state $wait$

$$idle \rightarrow busy : x_{wait} \geq 1 / x'_{wait} = x_{wait} - 1, x'_{req} = x_{req} + 1$$

The state of the monitor changes from *idle* to *busy*, whereas the selected process moves from *wait* to *req*.

Let us go back now to the problem of protocol verification. The forward reachability technique of [20] generalizes specialized symbolic state exploration techniques for the analysis of *parameterized* coherence protocols like the one proposed by Pong and Dubois in [42]. In [42] the *symbolic state model* (SSM) is used for the representation of the state space and the verification of protocols. Specifically, they apply an *abstraction* and represent sets of

global states via *repetition* operators to indicate 0,1, or multiple caches in a particular state. The SSM verification method is based on a *forward* exploration with *ad hoc* expansion and aggregation rules. In [39] Norris Ip and Dill have incorporated these technique in Murφ. Murφ automatically checks the soundness of the abstraction based on Pong and Dubois repetition operator, verifies the correctness of an abstract state graph of a fixed size using *on_the_fly state enumeration*, and, finally, tries to generalize the result for systems with larger (unbounded) size.

Differently from [39, 42], our method is based on general purpose techniques (backward reachability and constraints to represent sets of states) that have been successfully applied to the verification of timed, hybrid and concurrent systems (see e.g. [11, 18, 28]). Furthermore, in our model based on EFSMs we keep track of the *exact* number of caches in each state by associating a counter x_s with each local cache state s. The counter keeps track of the number of caches in state s during a protocol execution. The repetition abstraction can be easily obtained by focusing on a symbolic representation of sets of states via linear constraints that, for each counter x_s, have one of the following forms: $x_s = 0$, $x_s = 1$, and $x_s \geq 2$.

Although we are not aware of other applications of infinite-state symbolic model checkers based on arithmetic domains to verification of *parameterized cache coherence protocols*, several approaches have been proposed to attack verification problems of parameterized *concurrent systems*.

Abstractions based on the idea of counting the number of processes in a given state have been applied to asynchronous systems, e.g., in [23, 33]. In [23] German and Sistla propose an automated method for the verification of parameterized systems where processes are modelled in CCS. In [33] Lubachevski verifies properties of collections of identical processes for the *NYU-Ultracomputer* architecture. Here processes work on shared variables that range over a finite domain. *Fetch& Add* operations are used to coordinate processes. Methods that handle *global conditions* like ours (e.g. [1, 31]) are often semi-algorithms, i.e., they do not guarantee the termination of the analysis.

Methods based on the representation of arbitrary collections of processes via *regular languages* have been proposed in [1, 7, 13, 31]. In [1, 9] regular expressions are used to represent parameterized system configurations. *Acceleration* operators working on transducers on words are used as heuristic to achieve the termination of the forward symbolic reachability analysis. In [13] context free grammar of automata are used to represent linear ad hierarchal configurations. In approaches like [7, 31, 41] second order monadic logic is used to represent networks of processes. Combined with abstractions and accelerations, decision procedures for the logic are used to verify properties of skeleton of parameterized concurrent programs. In [7] the authors use this method to verify properties of the Illinois protocol.

Among semi-automatic methods that require the construction of abstractions and invariants we would like to mention [6, 10, 13, 24, 29, 36, 40]. For instance, in [29] Henzinger, Qaader and Rajamani consider more general processor models (they do not apply any counting argument) and apply an inductive scheme based on *merge-invariants* to verify *sequential consistency*. Though their approach is more general than ours, it is only partially automated. Specifically, the model checker MOCHA is used to prove finite-state lemmas

obtained during the manual application of the inductive scheme. Techniques for the automated generation of process invariants have been studied in [13, 32]. Automated generation of abstract transition graphs for infinite-state systems has been studied in [25]. Symmetry reductions for parameterized systems have been considered, e.g., in [36].

3. Cache coherence protocols

In this paper we will focus our attention on CC-UMA multiprocessor systems. As illustrated in figure 1, these are systems in which all processors have a local cache connected to the main memory via a shared bus. Cache coherence protocols are used to maintain the consistency of data stored in local caches and main memory. Cache coherence protocols are defined on the basis of several possible coherence policies such as write-invalidation, write-update, and write-allocation. In *write-invalidate* protocols whenever a processor modifies its cache block a *bus invalidation signal* is sent to all other caches in order to invalidate their content. In *write-update* protocols a copy of the new data is sent to all caches that share the old data. *Copy-back* coherence protocols often support policies like *direct data intervention*, *write-allocation* and *write snarf*. *Direct data intervention* indicates that on a read miss data can be read from another cache. *Write-allocation* indicates that in a write cycle a valid copy of data is first read and then merged with the (part of) line to be written. *Write snarf* indicates that when a processor is attempting to read data and the data is not available the cache is capable of listening to other transactions in order to grab copies of data.

In this paper we will focus our attention on some of the write-invalidate coherence protocols described by Jim Handy in "The Cache Book" [27]. Following [42], we will make the following assumptions:

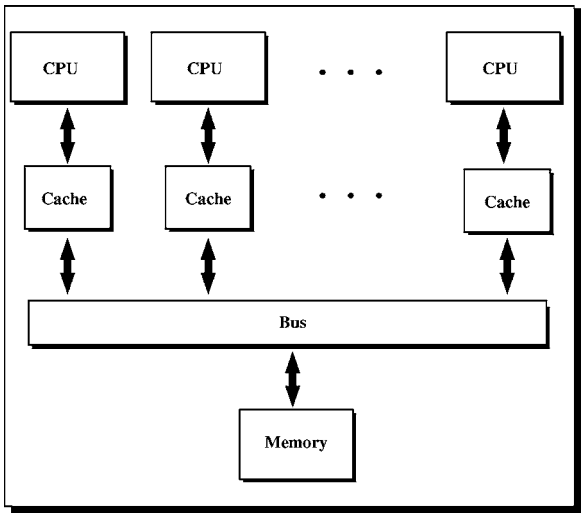

Figure 1. The structure of single bus CC-UMA multiprocessor systems.

– we will consider a single-bus multiprocessor system where each CPU has a local cache connected on the system bus;
– we will consider caches with a single line (i.e. cache state = state of the cache line);
– we will abstract from the data read from and written into cache and main memory.

Furthermore, following [14] we hide all handshaking necessary to issue a command, i.e., we model a command as an atomic action and we use non-determinism to simplify the bus-protocol introducing arbitrary delays in response to read and write commands. Finally, we will restrict our attention to data consistency properties that can be studied by only looking at the bit states of the local caches. In the next section we will introduce other terminology peculiar of these protocols like the notion of *write(read) hit(miss)* with the help of a simple example.

3.1. An example: The Synapse N+1 protocol

Cache coherence protocols are often described informally by the means of tables like the one in Table 1. These specifications describe the behavior of caches during read and write operations in local caches and main memory. The state of a cache depends on the hardware (usually supporting a *valid* and a *dirty* bit) and denotes (shared or exclusive) ownership of (clean or modified) data. The state changes according to write and read commands issued by the corresponding CPU or coming from the system bus. Table 1 illustrates the Synapse N + 1 protocol, a simple example taken from [27]. This *write-allocation* protocol was developed by Synapse for the N + 1 computer. Here a cache can be in one of three possible states: *invalid*, the cache has no valid data; *valid*, the cache has a potentially shared copy of the data; *dirty*, the cache has a modified copy of the data. *dirty* is an exclusive state, only one cache can have a dirty line. The transitions of Table 1 are given from the perspective of an individual cache. Their meaning is as follows.

Table 1. CPU and Bus Cycles for Synapse N + 1.

Current state	Command	To CPU	To BUS	New state
Invalid	READ		Public read	Valid
	WRITE		Private read	Dirty
Valid	READ	Supply data		Valid
	WRITE		Private read	Dirty
	Public read			Valid
	Private read			Invalid
Dirty	READ	Supply data		Dirty
	WRITE			Dirty
	Public read			Invalid
	Private read			Invalid

Read Hit: if a processor issues a read command for a memory block already stored in the local cache, the cache controller simply returns the data; no coherence action is needed here.

Read Miss: if a processor issues a read command for a memory block that is not in the cache, the cache updates the line from main memory using the Public Read command (sent to the bus), and goes to the *valid* state. When a cache in state *dirty* receives the Public Read command from the bus, it writes its contents to main memory and then it goes to *invalid*. No state change is required for a cache in state *valid* that receives the Public Read command from the bus.

Write Hit: if a processor issues a write command for a memory block that is already in the cache, then: if the cache is in state *dirty* no state change is needed; if the cache is in state *valid*, it issues a Private Read command to invalidate the line of all other valid caches, and then goes to *dirty*.

Write Miss: if a processor issues a write command for a memory block that is not in the cache, then: the cache updates the line from main memory using the Private Read command, writes to the cache line, and then goes to *dirty*. All caches that receive the Private Read command invalidate their line (a dirty cache is written back to main memory before invalidation).

4. The finite state machine model

Based on the assumption that all caches behave identically, we can define a *template* to describe the protocol from the perspective of one of the caches in the system. The template describes the finite set of states Q of an individual cache, the set Σ of commands coming from a CPU and broadcasted on the bus, and the set $\bar{\Sigma}$ of commands coming from the bus. Intuitively, $\bar{\sigma} \in \bar{\Sigma}$ should be viewed as the synchronous reaction of every single cache controller to a command $\sigma \in \Sigma$ issued by a given processor on its cache controller. The template can be instantiated to any number n of caches yielding what we will call a *global machine* of dimension n.

Let us first define the notion of *protocol*.

Definition 1. A *protocol* $\mathcal{P}$ is a tuple $\langle Q, \Sigma, \bar{\Sigma}, \tau \rangle$, where

- Q is a finite set of *states*;
- Σ is a finite set of *actions* (CPU commands);
- $\bar{\Sigma} = \{\bar{\sigma} \mid \sigma \in \Sigma\}$ is the set of *reactions* (bus commands);
- $\tau : Q \times \{\Sigma \cup \bar{\Sigma}\} \times Q$ is the transition relation where

$$\forall \sigma \in \Sigma \; \forall q \in Q \; \exists! r \in Q \text{ such that } \langle q, \bar{\sigma}, r \rangle \in \tau.$$

The last condition in the previous definition ensures that every cache always *reacts* to a given action σ. We introduce now a special kind of actions we will use later in the paper.

Definition 2. An *internal action* σ is an action such that $\langle q, \bar{\sigma}, q \rangle$ for all $q \in Q$.

Internal actions have a local effect. In fact, when a cache executes one of them the state of all other caches remains unchanged. We introduce now the notions of global machine and protocol execution.

Definition 3. Given a protocol $\mathcal{P} = \langle Q, \Sigma, \bar{\Sigma}, \tau \rangle$, the *global machine* $\mathcal{M}_G$ of dimension n is the tuple $\langle Q_G, \Sigma_G, \tau_G \rangle$, where $Q_G = Q^n$; $\Sigma_G = \Sigma$; and $\tau_G : Q_G \times \Sigma_G \times Q_G$ is such that $\langle \langle s_1, \ldots, s_n \rangle, \sigma, \langle s'_1, \ldots, s'_n \rangle \rangle \in \tau_G$ if and only if $\exists i$ s.t. $\langle s_i, \sigma, s'_i \rangle \in \tau$, and $\forall j \neq i \, \langle s_j, \bar{\sigma}, s'_j \rangle \in \tau$.

Let us call $G \in Q_G$ a *global* state. In the following we the notation $G \xrightarrow{\sigma} G'$ will indicate a tuple $\langle G, \sigma, G' \rangle \in \tau_G$.

Definition 4. A *run* of a global machine $\mathcal{M}_G$ is a possibly infinite sequence $G_1 \ldots G_i \ldots$ of global states of $\mathcal{M}_G$ such that there exists $\sigma_i \in \Sigma_G$ s.t. $G_i \xrightarrow{\sigma_i} G_{i+1}$ for all $i \geq 0$.

Example 1. To specify the Synapse N + 1 protocol described in Section 3.1, we define Q as the set of states $\{invalid, valid, dirty\}$, and Σ as the set of actions $\{Rh, Rm, W\}$. The previous labels have the following meaning: Rh is an *internal action* that denotes a read hit, Rm denotes a read miss, W denotes a write. The transition relation τ is given in figure 2. For simplicity, from here on we will always omit the reactions of internal actions: they always correspond to self-loops like $\langle invalid, \overline{Rh}, invalid \rangle$. Let us consider now the global machine of dimension three. The initial state is the tuple $\langle invalid, invalid, invalid \rangle$. The following run leads the first cache to the dirty state after a read and a write command issued by the first processor:

$$\begin{aligned} \langle invalid, invalid, invalid \rangle &\xrightarrow{Rm} \langle valid, invalid, invalid \rangle \\ &\xrightarrow{W} \langle dirty, invalid, invalid \rangle. \end{aligned}$$

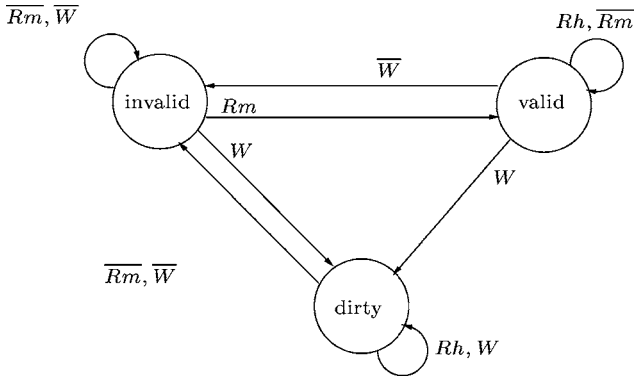

Figure 2. The Synapse N + 1 protocol from the perspective of cache C_i.

If the second processor now issues a read command (a *read miss* occurs), then the state of the first cache is invalidated:

$$\langle dirty, invalid, invalid\rangle \xrightarrow{Rm} \langle invalid, valid, invalid\rangle.$$

As a last example, if the third processor issues a read command, then no invalidation is needed (a *read hit* occurs); the second and the third cache will share the data:

$$\langle invalid, valid, invalid\rangle \xrightarrow{Rh} \langle invalid, valid, valid\rangle.$$

4.1. Global conditions

Many real protocols cannot be specified using the model of Section 4. As an example, several protocols admit caches in *exclusive*, *clean* state. In this class of protocols the cache that receives the read request from the CPU is always informed through a special signal (e.g., the *transaction flag* specified in the Futurebus+ protocol [27]) of the presence of other caches in *dirty* or *clean* states. The cache moves to the exclusive state if and only if it receives no signals. In order to model this behavior, in this section we will enrich our model by introducing *conditional* actions. A conditional action has the following form

$$P \to \sigma$$

where P is a predicate defined over global states, and σ is an action in Σ. Since we are interested in conditions that can be expressed independently from the number of processors in the system, we introduce a special language for the global conditions.

Definition 5. Given a set Q of states, a *predicate* is a formula generated by the following grammar

$$P ::= P \wedge P \mid P \vee P \mid \#q = c \mid \#q \geq c \mid true$$

where $q \in Q$ and $c \in \mathbb{N}$.

Intuitively, the expression $\#q$ denotes the number of caches in state q in the current global state. As an example, the predicate

$$\#dirty = 0 \wedge \#shared = 0 \wedge \#valid = 0$$

is valid in all global states with no caches in state *dirty*, *shared*, and *valid*. We are now in a position to extend the definition of *protocols* (Definition 1, Section 4).

Definition 6. A *protocol $\mathcal{P}$ with global conditions* is defined as a tuple $\langle Q, \Sigma, \bar{\Sigma}, C, \tau\rangle$, where

– Q is a finite set of *states*;

- C is a finite set of predicates over Q;
- Σ is a finite set of *actions* (CPU commands);
- $\bar{\Sigma} = \{\bar{\sigma} \mid \sigma \in \Sigma\}$ is the set of *reactions*;
- $C \times \Sigma$ is the set of *conditional actions*, $p \rightarrow \sigma$ denotes an element of $C \times \Sigma$. For the sake of simplicity, we will use σ as an abbreviation for *true* $\rightarrow \sigma$;
- $\tau : Q \times \{(C \times \Sigma) \cup \bar{\Sigma}\} \times Q$ is such that

$$\forall (p \rightarrow \sigma) \in (C \times \Sigma)\, \forall q \in Q\, \exists! q' \in Q \text{ such that } \langle q, \bar{\sigma}, q' \rangle \in \tau.$$

In order to define the semantics of conditional actions, we need to define what *satisfiability of a predicate in a global state* means.

Let $Q = \{q_1, \ldots, q_k\}$ and let G be a global state of dimension n. We first introduce the mapping $\lceil \cdot \rceil : Q_G \rightarrow \mathbb{N}^k$ from global states to tuples of non-negative integers defined as follows

$$\lceil G \rceil = \langle c_1, \ldots, c_k \rangle, c_i \text{being the number of occurrences of } q_i \in Q \text{ in } G$$

Given $\lceil G \rceil = \langle c_1, \ldots, c_k \rangle$, we say that the predicate p *is satisfied in* G if the mapping $\#q_i \mapsto c_i$ for $i : 1, \ldots, k$ makes p true.

Definition 7. Given a protocol with global conditions defined as $\mathcal{P} = \langle Q, \Sigma, \bar{\Sigma}, C, \tau \rangle$, the *global machine* $\mathcal{M}_G$ of dimension n is the tuple $\langle Q_G, \Sigma_G, \tau_G \rangle$, where

- $Q_G = Q^n$ and $\Sigma_G = \Sigma$;
- $\tau_G : Q_G \times \Sigma_G \times Q_G$ where $\langle s_1, \ldots, s_n \rangle \xrightarrow{\sigma} \langle s'_1, \ldots, s'_n \rangle$ is in τ_G if and only if $\exists i$ s.t. $\langle s_i, P \rightarrow \sigma, s'_i \rangle \in \tau$, P is satisfied in $\langle s_1, \ldots, s_n \rangle$, and $\forall j \neq i \ \langle s_j, \bar{\sigma}, s'_j \rangle \in \tau$.

A run is a sequence of global states computed via the transition relation τ_G.

Example 2. The University of Illinois protocol is a *snoopy* cache, *write-invalidate*, *write-in* coherence policy. This protocol was originally proposed in [34] and it is also described in [5, 42]. The special feature is that caches can have exclusive copies of data. Bus invalidation signals are sent only for writes to shared data. The memory copy is updated using a write-back policy (*replacement*). Formally, in addition to *invalid*, caches assume one of the following states: *valid-exclusive*, the cache has an exclusive copy of the data that is consistent with the memory such that a modification of its content requires no bus invalidation signal; *shared*, the cache has a copy of the data consistent with the memory and other caches may have copies of the data; *dirty*, the cache has a modified copy of the data, i.e., the data in main memory are obsolete and the content of the other caches is not valid. The behavior of cache C_i is described as follows.

Read Hit: no coherence action needs to be taken.
Read Miss: if there exists a cache C_j $j \neq i$ whose state is *dirty*, C_j supplies the missing block to C_i and updates the main memory. Both C_i and C_j end up in state *shared*. If

there are *shared* or *valid-exclusive* copies in other caches, C_i gets the missing block from one of the caches and all caches with a copy end up in state *shared*. If there is no cached copy, C_i receives a *valid-exclusive* copy from main memory.

Write Hit: if C_i is in state *dirty*, no action is taken. If C_i is in state *valid-exclusive*, its state changes to *dirty* (note: no invalidation signal is needed). If C_i is in state *shared*, its state changes to *dirty* and all remote copies must be invalidated.

Write Miss: similar to *Read Miss*, except that all remote copies are invalidated and the state of C_i changes to *dirty*.

Replace: if C_i is in state *dirty*, the data are written back to memory.

To formalize the protocol, we introduce the internal actions Rh, We (write in exclusive state), and Wd (write in dirty state), and the actions Rm (read miss), WI (write and invalidate), and Rep (replacement with a new memory line). Furthermore, we introduce a predicate P that is used by a cache to decide whether or not to move from *invalid* to *valid-exclusive*. Formally, P is defined as follows.

$$P \equiv \#dirty = 0 \wedge \#shared = 0 \wedge \#valid\text{-}exclusive = 0$$

Furthermore, we define its dual $\neg P$ as follows.

$$\neg P \equiv \#dirty \geq 1 \vee \#shared \geq 1 \vee \#valid\text{-}exclusive \geq 1$$

The protocol is formally specified in figure 3.

As in Example 1, let us consider the global machine of dimension three. The initial state is again the tuple $G_0 = \langle invalid, invalid, invalid \rangle$. The following run leads the first cache

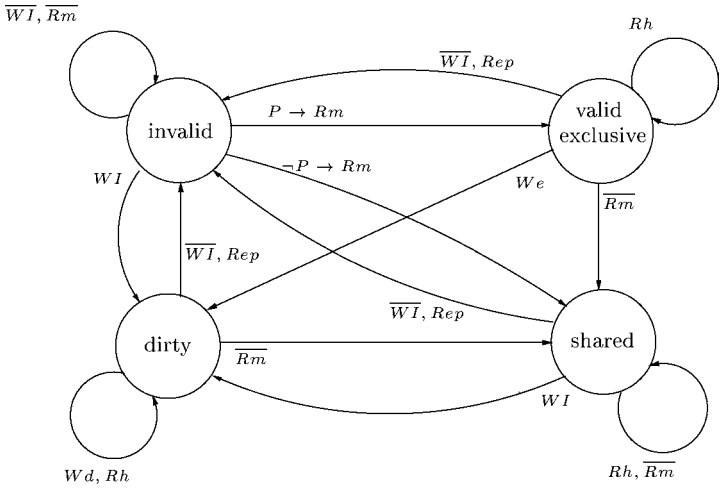

Figure 3. Formal specification of the Illinois protocol.

to the dirty state after a read and a write command issued by the first processor:

$$\langle \mathit{invalid}, \mathit{invalid}, \mathit{invalid}\rangle \xrightarrow{Rm} \langle \mathit{valid\text{-}exclusive}, \mathit{invalid}, \mathit{invalid}\rangle \xrightarrow{WI} \langle \mathit{dirty}, \mathit{invalid}, \mathit{invalid}\rangle.$$

(note that the predicate P is satisfied in G_0). If the second processor now issues a read command, the predicate $\neg P$ is satisfied and the first and the second cache update their state to *shared*:

$$\langle \mathit{dirty}, \mathit{invalid}, \mathit{invalid}\rangle \xrightarrow{Rm} \langle \mathit{shared}, \mathit{shared}, \mathit{invalid}\rangle.$$

As a last example, if the second processor issues a write command, then the first cache is invalidated:

$$\langle \mathit{valid}, \mathit{valid}, \mathit{invalid}\rangle \xrightarrow{WI} \langle \mathit{invalid}, \mathit{dirty}, \mathit{invalid}\rangle.$$

4.2. Safety properties

In this paper we will restrict ourselves to consider the *safety* property of data consistency [43]. For instance, let us consider the Synapse protocol of Section 3.1. In this protocol the state *valid* indicates that a cache has a clean copy consistent with the memory and other caches copies, whereas the state *dirty* indicates that it has the latest and sole copy. Thus, if we restrict ourselves to the behavior of the caches, there are two possible sources of data inconsistency for Synapse:

UNS_1: a *dirty* cache co-exists with one or more caches in state *valid*;
UNS_2: more that one cache is in state *dirty*.

All global states that satisfy conditions UNS_1 or UNS_2 are *unsafe* with respect to data consistency.

Our general goal is to prove protocols like Synapse correct (with respect to the given set of safety properties) for every possible number of caches. For a fixed number of processors k and a given protocol $\mathcal{P}$, let $\mathcal{I}(k)$ be the set of initial global states and $\mathcal{U}(k)$ be the set of unsafe global states. The *parameterized reachability problem* is defined as follows.

$$\exists k \geq 1.\ \exists G_1 \in \mathcal{I}(k).\ \exists G_n \in \mathcal{U}(k) \text{ such that } G_1 G_2 \ldots G_n \text{ is a run?}$$

If the previous statement is true for a given k', then the protocol is not correct, i.e., an unsafe state may be reached for a system configuration with k' caches.

5. The counting abstraction

In order to check parameterized safety properties we apply the following *counting abstraction* from global states to tuples of non-negative integers. Let Q be the set of cache states

$q_1, \ldots, q_k$ and G be a global state, then

we count the number of occurrences *of every* $q \in Q$ *in* G.

Via this abstraction, a global state G with n components (where n is the number of caches and processors in the system) is mapped to a tuple of *non-negative integers* with k components (where k is the number of cache states). This way, all *symmetric* global states are clustered together into a single representation. The abstraction cannot be used to prove properties of individual caches like *cache i and cache j cannot be in state* dirty *simultaneously*, however it preserves all properties that do not depend on the number of processors, like *two different caches cannot be in state* dirty *simultaneously*. Actually, this is the kind of property we are interested in to prove that the protocol is correct with respect to the intended semantics of cache states.

The behavior of an *arbitrary* number of caches can be described *finitely* as a set of linear transformations describing the effect of the actions on the *counters* associated with the states in Q. For this purpose, we model the abstract protocol as a single *Extended Finite State Machine* (EFSM) [12], i.e., a finite automaton with integer data paths.

Formally, let $\mathcal{M}_G$ be the global machine $\langle Q_G, \Sigma_G, \delta_G \rangle$ associated with a protocol $\mathcal{P}$, and let $Q_G = \{q_1, \ldots, q_k\}$. We model $\mathcal{M}_G$ as an EFSM $\mathcal{E}$ having the following features:

- $\mathcal{E}$ has *only one location* (for simplicity, we will always omit the location);
- $\mathcal{E}$ has k data variables $\langle x_1, \ldots, x_k \rangle$ ranging over *non-negative integers*, each one associated with a state in Q;
- the EFSM-states are tuples of non-negative integers $\mathbf{c} = \langle c_1, \ldots, c_k \rangle$, where c_i denotes the number of caches with state $q_i \in Q$ during a run of $\mathcal{M}_G$.
- The transitions are represented via a collection of *guarded linear transformations* defined over the variables $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$ and $\mathbf{x}' = \langle x'_1, \ldots, x'_k \rangle$ and having the following form

  $$G(\mathbf{x}) \;\rightarrow\; T(\mathbf{x}, \mathbf{x}'),$$

  where x_i and x'_i denote the number of caches in state q_i respectively before and after the occurrence of the transition.

Intuitively, a run of the EFSM $\mathcal{E}$ is a (possibly infinite) sequence of EFSM-states $\mathbf{c_1}, \ldots, \mathbf{c_i} \ldots$ where $G_i(\mathbf{c_i}) \wedge T_i(\mathbf{c_i}, \mathbf{c_{i+1}}) = \mathit{true}$ for some transitions $G_i \rightarrow T_i$ in $\mathcal{E}$ for $i \geq 0$. In the following section we will formalize these ideas by presenting a translation of protocols into EFSMs.

5.1. *From protocols to EFSMs*

Let $\mathcal{P}$ be a protocol with set of states $Q = \{q_1, \ldots, q_k\}$. In the translation we associate a variable x_i to each state q_i. The variable x_i maintains the count of the number of caches in state q_i during a run of a global machine of $\mathcal{P}$. According to this abstraction, the transition relation is mapped into a set of linear transformations defined over $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$.

Definition 8. Let R be the transition $\langle q_i, \sigma, q_j \rangle \in \tau$ for $\sigma \in \Sigma$. The EFSM-transition $\hat{R}$ (with components G_R and T_R) associated with $R = \langle q_i, \sigma, q_j \rangle$, is defined as the guarded linear transformation

$$G_R(\mathbf{x}) \equiv x_i \geq 1,$$
$$T_R(\mathbf{x}, \mathbf{x}') \equiv \exists \mathbf{y}.\ \exists \mathbf{z}. E_1(\mathbf{x}, \mathbf{y}) \ \wedge \ E_2(\mathbf{y}, \mathbf{z}) \ \wedge \ E_3(\mathbf{z}, \mathbf{x}').$$

and E_1, E_2 and E_3 are defined as follows:

$$E_1(\mathbf{x}, \mathbf{y}) \equiv (y_i = x_i - 1) \wedge \bigwedge_{s \neq i} y_s = x_s$$

$$E_2(\mathbf{y}, \mathbf{z}) \equiv \bigwedge_{t=1}^{k} z_t = S(\mathbf{y}, t)$$

$$E_3(\mathbf{z}, \mathbf{t}) \equiv (t_j = z_j + 1) \wedge \bigwedge_{l \neq j} t_l = z_l$$

where

– $P_\sigma(t) = \{s \mid \langle q_s, \bar{\sigma}, q_t \rangle \in \tau\}$,
– and
  - $S(\mathbf{y}, t) = \Sigma_{s \in P_\sigma(t)}\ y_s$ if $P_\sigma(t) \neq \emptyset$;
  - $S(\mathbf{y}, t) = 0$, otherwise.

It is easy to check that for any R, $\hat{R}$ is a linear transformation such that $G_R(\mathbf{x}) \equiv x_i \geq 1$ and $T_R(\mathbf{x}, \mathbf{x}') \equiv \mathbf{x}' = M \cdot \mathbf{x} + \mathbf{d}$ where M is a $k \times k$-matrix with unit vectors as columns, and $\mathbf{d}$ is a vector of constants.

As we will show in the next example, it is often possible to simplify the resulting EFSM, e.g., by merging together rules sharing the same right hand side. Note, in fact, that the two rules $G_1 \rightarrow T$ and $G_2 \rightarrow T$ are logically equivalent to the rule $(G_1 \vee G_2) \rightarrow T$. Furthermore, via equivalences like $x \geq 1 \vee y \geq 1 \ \equiv x + y \geq 1$, it is often possible to express the resulting guards as a single linear arithmetic constraint.

As a consequence of the previous observations, in the rest of the paper we will consider guards having the form $A \cdot \mathbf{x} \geq \mathbf{c}$, where A is matrix with unit vectors as columns and $\mathbf{c}$ is a vector of non-negative integers. Since the number of caches does not change during a protocol execution, the transformation also satisfies the condition $x'_1 + \cdots + x'_n = x_1 + \cdots + x_n$.

Example 3. The EFSM obtained as abstraction of the Synapse N + 1 protocol (after eliminating all existentially quantified variables) is shown in figure 4: here we use x_i, x_v and x_d to denote the integer counters for the states *invalid*, *valid*, *dirty*, respectively. Furthermore, we omit the location and all equalities of the form $x'_i = x_i$ and, we use the notation $G \rightarrow$ *skip* to denote that all counters remain unchanged.

$$
\begin{array}{lll}
(rh) & x_d \geq 1 & \rightarrow \quad skip. \\
(rh) & x_v \geq 1 & \rightarrow \quad skip. \\
(rm) & x_i \geq 1 & \rightarrow \quad x'_d = 0, x'_v = x_v + 1, x'_i = x_i + x_d - 1. \\
(wh_1) & x_d \geq 1 & \rightarrow \quad skip. \\
(wh_2) & x_v \geq 1 & \rightarrow \quad x'_v = 0, x'_d = 1, x'_i = x_i + x_d + x_v - 1. \\
(wm) & x_i \geq 1 & \rightarrow \quad x'_v = 0, x'_d = 1, x'_i = x_i + x_d + x_v - 1.
\end{array}
$$

Figure 4. EFSM for the Synapse N+1 Protocol: x_i is the counter for *invalid*, etc.

The two (rh) rules denote read hits (no state changes). As an example of EFSM simplification, note that the two rh rules can be merged into the single rule $x_d + x_v \geq 1 \longrightarrow skip$. Rule (rm) denotes a read miss. Rule (wh_1) and (wh_2) denote write hits in state *dirty* (no state change) and *valid* (with bus invalidation), respectively. Rule (wm) denotes a write miss with bus invalidation. Rule (rh) denotes read hits (no state changes).

To illustrate in more details how the translation works, let us consider the transition $\langle invalid, Rm, valid\rangle$ and the corresponding reactions $\langle valid, \overline{Rm}, invalid\rangle$, $\langle valid, \overline{Rm}, valid\rangle$, and $\langle dirty, \overline{Rm}, invalid\rangle$. Let $\mathbf{x} = \langle x_i, x_v, x_d\rangle$, $\mathbf{y} = \langle y_i, y_v, y_d\rangle$, $\mathbf{z} = \langle z_i, z_v, z_d\rangle$, and $\mathbf{t} = \langle t_i, t_v, t_d\rangle$ be the variables associated with *invalid*, *valid* and *dirty*, respectively. In the first step, we get then the equalities $y_i = x_i - 1, y_v = x_v, y_d = x_d$. Since $P_{Rm}(invalid) = \{invalid, dirty\}$, $P_{Rm}(valid) = \{valid\}$ and $P_{Rm}(dirty) = \emptyset$, in the second step we get the formula $z_i = y_i + y_d, z_v = y_v, z_d = 0$. In the third step we get the formula $t_i = z_i, t_v = z_v + 1, t_d = z_d$. After eliminating the variables in $\mathbf{y}$ and $\mathbf{z}$ from the conjunction of the three systems and by renaming $\mathbf{t}$ into $\mathbf{x}'$, we finally get the EFSM rule $x_i \geq 1 \longrightarrow x'_i = x_i + x_d - 1, x'_v = x_v + 1, x'_d = 0$.

5.2. *Encoding global conditions*

The encoding of a protocol $\mathcal{P}$ with global conditions into an EFSM $\mathcal{E}$ is defined as follows. We first define a protocol $\mathcal{P}'$ in which all conditions in $\mathcal{P}$ are replaced by *true*. Then, we compute the EFSM $\mathcal{E}'$ associated with the protocol $\mathcal{P}'$ following the construction we have given in Section 5.1. Now, let R' be an action in $\mathcal{P}'$ and $\widehat{R}'$ be a rule in $\mathcal{E}'$. If the action R in the original protocol $\mathcal{P}$ has a condition P, we transform the condition P (for simplicity, in disjunctive normal form) into the disjunction (set) Φ_P of constraints over the variables $\mathbf{x}$ of $\mathcal{E}'$ defined as follows:

$$\Phi_P \equiv P[\#q_1 \mapsto x_1, \ldots, \#q_k \mapsto x_k]$$

Let $\Phi_P = \varphi_1 \vee \cdots \vee \varphi_m$. The *set* of EFSM-transitions associated with R in $\mathcal{E}$ is defined then as

$$\varphi_1 \wedge \hat{R}', \ldots, \varphi_m \wedge \hat{R}'.$$

It is easy to check that any transformation in $\mathcal{E}$ is such that:

$$G_R(\mathbf{x}) \equiv (A \cdot \mathbf{x} \geq \mathbf{c}) \wedge (B \cdot \mathbf{x} = \mathbf{c}')$$

$$
\begin{array}{lll}
(r1) & x_d + x_s + x_e \geq 1 & \rightarrow \quad skip. \\
(r2) & x_i \geq 1, x_d = 0, x_s = 0, x_e = 0 & \rightarrow \quad x_i' = x_i - 1, x_e' = x_e + 1. \\
(r3) & x_i \geq 1, x_s + x_e + x_d \geq 1 & \rightarrow \\
 & & \quad x_i' = x_i - 1, x_e' = 0, x_d' = 0, x_s' = x_s + x_e + x_d + 1. \\
(w1) & x_d \geq 1 & \rightarrow \quad skip. \\
(w2) & x_e \geq 1 & \rightarrow \quad x_e' = x_e - 1, x_d' = x_d + 1. \\
(w3) & x_s + x_i \geq 1 & \rightarrow \\
 & & \quad x_i' = x_i + x_e + x_d + x_s - 1, x_e' = 0, x_s' = 0, x_d' = 1. \\
(w4) & x_d \geq 1 & \rightarrow \quad x_i' = x_i + 1, x_d' = x_d - 1. \\
(w5) & x_s \geq 1 & \rightarrow \quad x_i' = x_i + 1, x_s' = x_s - 1. \\
(w6) & x_e \geq 1 & \rightarrow \quad x_e' = x_e - 1, x_i' = x_i + 1.
\end{array}
$$

Figure 5. EFSM for the Illinois protocol.

for two matrices A and B with unit vectors as columns, and two vectors of constants $\mathbf{c}$ and $\mathbf{c}'$;

$$T_R(\mathbf{x}, \mathbf{x}') \equiv (\mathbf{x}' = M \cdot \mathbf{x} + \mathbf{d})$$

where M is a $k \times k$-matrix with unit vectors as columns, and $\mathbf{d}$ is a vector of constants.

Example 4. The EFSM for the Illinois protocol is shown in figure 5. Here x_i, x_d, x_s, and x_e are the counters associated with the states *invalid*, *dirty*, *shared*, *exclusive*, respectively. Rule $r1$ of figure 5 represents *read hit* events: since no coherence action is needed, the only precondition is that there exists at least one cache in a valid state, i.e., $x_d + x_s + x_e \geq 1$. Rules $r2$-$r3$ correspond to *read miss* events where the global predicate P introduced in figure 3 is expressed via guards containing tests for zero. Specifically, rule $r2$ represents a read miss such that $P = true$, i.e., one cache can move to *valid-exclusive*. Rule $r3$ applies whenever the block is copied from a cache in *dirty*, *shared* or *valid-exclusive* state. Rules $w1 - w3$ model *write hits*. Specifically, rule $w1$ models a write in state *dirty* (no action is taken). Rule $w2$ models a write in state *valid-exclusive* where the state changes to *dirty* without bus invalidation signal. Rule $w3$ models a write in state either *shared* or *invalid* where the copies in all other caches are invalidated. Finally, rules $w4$–$w6$ model *replacement* events. If the cache is in one of the states *dirty*, *shared* or *exclusive* its state changes to *invalid*.

5.3. *Properties of the encoding*

In this section we will study the properties of the encoding. We first note that the mapping

$$\ulcorner . \urcorner : Q_G \rightarrow \mathbb{N}^k$$

defined in Section 4.1 maps global states into EFSM-states. Then, we have the following propositions.

Lemma 1. *Let R be an action that leads from the global state G_1 to the global state G_2, where $\ulcorner G_1 \urcorner = \langle c_1, \ldots, c_k \rangle$ and $\ulcorner G_2 \urcorner = \langle c'_1, \ldots, c'_k \rangle$. Then, the tuple $\langle c_1, \ldots, c_k, c'_1, \ldots, c'_k \rangle$ is a solution for the formula $\hat{R}(\mathbf{x}, \mathbf{x}')$ defined as $G_R(\mathbf{x}) \wedge T_R(\mathbf{x}, \mathbf{x}')$.*

Proof: Let G_1 be $\langle s_1, \ldots, s_n \rangle$ and $G_2 = \langle s'_1, \ldots, s'_n \rangle$. Under the hypothesis of the lemma, there exists an index i s.t. $R = \langle s_i, P \rightarrow \sigma, s'_i \rangle$, and $\langle s_j, \bar{\sigma}, s'_j \rangle \in \tau$ for all $j \neq i$. The formula $G_R(\mathbf{x})$ contains the guard $x_i \geq 1$ and the encoding of P, both satisfied by the assignment $\ulcorner G_1 \urcorner$. In order to analyze the effect of the action σ, we split the transition from G_1 to G_2 via R in the following three steps.

- We first replace s_i with a new label $\bullet$ ($\bullet \notin Q$) in G_1, and call G' the resulting global state. For instance, if the invalid process in $G_1 = \langle \mathit{invalid}, \mathit{valid}, \mathit{valid} \rangle$ moves via σ, then $G' = \langle \bullet, \mathit{valid}, \mathit{valid} \rangle$.
- Then, we apply the $\bar{\sigma}$ operation to G' to obtain the global state G''. This step has the effect of reallocating all states in G' but the label $\bullet$ that remains at position i. For instance, if the valid processes in G' are invalidated by σ, then $G'' = \langle \bullet, \mathit{invalid}, \mathit{invalid} \rangle$.
- Finally, we replace the label $\bullet$ with s_j in G''. This way we get G_2. For instance, if the target state of σ is *dirty*, then we have to replace the label $\bullet$ with *dirty* to obtain $G_2 = \langle \mathit{dirty}, \mathit{invalid}, \mathit{invalid} \rangle$.

These three steps correspond to the three systems of equations E_1, E_2, and E_3 defined in the translation:

- $E_1(\mathbf{x}, \mathbf{y}) \equiv y_i = x_i - 1 \wedge \bigwedge_{s \neq i} y_s = x_s$,
- $E_2(\mathbf{y}, \mathbf{z}) \equiv \bigwedge_{t=1}^{k} z_t = S(\mathbf{y}, t)$,
- $E_3(\mathbf{z}, \mathbf{t}) \equiv t_j = z_j + 1 \wedge \bigwedge_{l \neq j} t_l = z_l$

We recall that $P_\sigma(t) = \{s \mid \langle q_s, \bar{\sigma}, q_t \rangle \in \tau\}$, and $S(\mathbf{y}, t) = \Sigma_{s \in P_\sigma(t)}\ y_s$ if $P_\sigma(t) \neq \emptyset$; $S(\mathbf{y}, t) = 0$, otherwise.

To prove the claim, we simply have to show the existence of two vectors $\mathbf{d}$ and $\mathbf{e}$ such that

$$\exists \mathbf{y}.\ \exists \mathbf{z}.\ E_1(\ulcorner G_1 \urcorner, \mathbf{y}) \ \wedge\ E_2(\mathbf{y}, \mathbf{z}) \ \wedge\ E_3(\mathbf{z}, \ulcorner G_2 \urcorner)$$

is satisfiable. Let us define the vector $\mathbf{d}$ as $d_i = c_i - 1$ and $d_j = c_j$ for all $j \neq i$.

- Then, by definition of E_1, it follows that $E_1(\ulcorner G \urcorner, \mathbf{d}) \equiv \mathit{true}$. Furthermore, we have that $\mathbf{d} = \ulcorner G' \urcorner$.
- The execution of the broadcast $\bar{\sigma}$ on G' has the effect of reallocating the local states so that all states in $P_\sigma(s_r)$ move to s_r for $r : 1, \ldots, k$ ($\bar{\sigma}$ has no effect on the label $\bullet$). Thus, $\ulcorner G'' \urcorner = \mathbf{e}$, where $e_r = \Sigma_{\{l \mid s_l \in P_\sigma(s_r)\}}\ d_l$ if $P_\sigma(s_r) \neq \emptyset$, $e_r = 0$ otherwise, for $r : 1, \ldots, k$. Clearly, $\mathbf{e}$ is such that $E_2(\mathbf{d}, \mathbf{e}) \equiv \mathit{true}$.
- As a last step, we simply have to check that $E_3(\mathbf{e}, \ulcorner G_2 \urcorner) \equiv \mathit{true}$. But this follows immediately by noting that G_2 is obtained from G'' by replacing the label $\bullet$ with s_j, i.e., by incrementing e_j by one. Thus, the tuple $\mathbf{m}$ obtained by joining $\ulcorner G_1 \urcorner$ and $\ulcorner G_2 \urcorner$ satisfies $\hat{R}$.

□

Lemma 2. *Let $\hat{R}$ be a transition enabled in the EFSM-state* $\mathbf{c}$, *and leading to the state* $\mathbf{c}'$. *Then, there exist two global states G_1 and G_2 such that $\lceil G_1 \rceil = \mathbf{c}$, $\lceil G_2 \rceil = \mathbf{c}'$, and G_1 moves to G_2 via R.*

Proof: The proof is similar to the previous lemma. □

The elimination of existentially quantified variables that occur in $\hat{R}$ returns a formula that can be still decomposed into a guard $G_R(\mathbf{x})$ and an assignment $T_R(\mathbf{x}, \mathbf{x}')$ having the form described above. Furthermore, the following results hold.

Theorem 1. *Let $\mathcal{P}$ be a protocol and $\mathcal{E}$ be the EFSM obtained via the encoding of each action $R \in \mathcal{P}$ into the EFSM rule $\hat{R}$. Then, for every run $G_1 \ldots G_r$ in $\mathcal{P}$ there exists a run $\lceil G_1 \rceil \ldots \lceil G_r \rceil$ in $\mathcal{E}$. Vice versa, for every run $\mathbf{c}_1 \ldots \mathbf{c}_s$ in $\mathcal{E}$, there exists a run $G_1 \ldots G_s$ in $\mathcal{P}$ such that $\lceil G_i \rceil = \mathbf{c}_i$ for any $i : 0, \ldots, s$.*

Proof: We prove the first part of the theorem by induction on the length of a run in $\mathcal{P}$. The base case trivially holds. If the length ℓ is greater or equal than one, we first assume that the thesis holds for ℓ, i.e., for every run $G_1 \ldots G_\ell$ in $\mathcal{P}$, $\lceil G_1 \rceil \ldots \lceil G_\ell \rceil$ is a run in $\mathcal{E}$.

Now let R be an action that can be performed in the global state G_ℓ and leads to $G_{\ell+1}$. By Lemma 1, the vector $\mathbf{m}$ obtained by concatenating $\lceil G_\ell \rceil$ and $\lceil G_{\ell+1} \rceil$ is a solution for $\hat{R}$. Thus, $\lceil G_1 \rceil \ldots \lceil G_\ell \rceil \lceil G_{\ell+1} \rceil$ is an admissible run of $\mathcal{E}$. The proof of the second implication uses Lemma 2 in a similar way. It is omitted for brevity. □

Based on the previous lemmas, we have the following result.

Theorem 2. *Let $\mathcal{P}$ a protocol and $\mathcal{E}$ be the EFSM obtained via the encoding presented above. Then, for every run $G_1 \ldots G_r$ in $\mathcal{P}$ there exists a run $\lceil G_1 \rceil \ldots \lceil G_r \rceil$ in $\mathcal{E}$. Vice versa, for every run $\mathbf{c}_1 \ldots \mathbf{c}_s$ in $\mathcal{E}$, there exists a run $G_1 \ldots G_s$ in $\mathcal{P}$ such that $\lceil G_i \rceil = \mathbf{c}_i$ for $i : 0, \ldots, s$.*

5.4. EFSM pre-image operator

In this paper we will define a verification procedure based on the *pre-image* operator associated with an EFSM. The EFSM *pre-image* operator defines a transformation over infinite collections of EFSM-states, i.e., via infinite sets of symmetric global states. Its definition is as follows.

Definition 9. Given an EFSM $\mathcal{E}$, the predecessor operator pre is defined over *sets* of EFSM-states as follows:

$$pre(S) = \{\mathbf{c} \mid \mathbf{c} \rightarrow \mathbf{c}',\ \mathbf{c}' \in S\}.$$

Here $\rightarrow$ indicates the application of an EFSM-transition.

Let $\mathcal{E}$ be the EFSM resulting from the encoding of a protocol $\mathcal{P}$. By Theorem 2, if S is a set of EFSM-states representing the infinite sets of global states $\mathbf{I}$, then $pre(S)$ characterizes the set of predecessors state of $\mathbf{I}$ via the transition relation of the protocol $\mathcal{P}$. Thus, the pre-image operator *pre* only generates the representative elements of collection of symmetric states. As an example, consider again the EFSM of figure 4. Let $\langle c_1, c_2, c_3\rangle$ denote an EFSM-state in which $x_i = c_1, x_v = c_2$, and $x_d = c_3$, respectively. Then, $\langle k', 0, 0\rangle \in pre(\{\langle k, 1, 0\rangle \mid k \geq 0\})$ for any k'. The tuple $\langle k', 0, 0\rangle$ represents all global states with k' caches in state invalid.

In the following section we will define a symbolic representation of collections of EFSM-states based on the notion of *constraints*.

6. Constraints

The counting abstraction allows us to reduce the *parameterized reachability* problem of cache coherence protocols to a *reachability* problem for EFSMs (Theorem 2). Our approach to attack the second problem is based on symbolic search for EFSMs. In order to represent concisely (possibly infinite) sets of global states independently from the number of caches in the system, we use linear arithmetic constraints as a symbolic representation of sets of EFSM-states.

Definition 10. A *linear arithmetic constraint* over $x_1, \ldots, x_n$ is a formula $a_1 \wedge \cdots \wedge a_m$, where a_i is an *atomic* constraint. Atomic constraints have the form $c_1 \cdot x_{i_1} + \cdots + c_k \cdot x_{i_k} \odot d$, where c_i is an integer constant for $i : 1, \ldots, k$, and $\odot$ is one of $=, \leq, \geq, >, <$.

This class of constraints is powerful enough to express *initial* and *unsafe* sets of states for the verification problems we are interested in.

Example 5. The set of unsafe states of the Illinois protocol where at least one cache is in state *shared* and *at least* one cache is in state *dirty* can be represented finitely as the constraint $x_s \geq 1 \wedge x_d \geq 1$.

In the rest of the paper we will use the lower-case letters $\varphi, \psi, \ldots$ to denote constraints and the upper-case letters $\Psi, \Phi, \ldots$ to denote *sets* (disjunctions) of constraints. Furthermore, we will use the tuple $\langle t_1, \ldots, t_n\rangle$ to denote the assignment $x_1 \mapsto t_1, \ldots, x_n \mapsto t_n$ of values to the variables $x_1, \ldots, x_n$. In our setting the variables will range over *non negative integer* values. An assignment $\langle t_1, \ldots, t_n\rangle \in \mathbb{N}^n$ *satisfies* a constraint φ defined over $x_1, \ldots, x_n$ if and only if $\varphi[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ evaluates to *true* (according to the usual interpretations of the operators in φ). The constraint *true* is an abbreviation for a constraint that is always satisfiable. The *denotation* of a constraint and the *entailment relation* are defined below.

Definition 11. The *denotation* of a constraint φ with free variables $x_1, \ldots, x_n$ is defined as follows

$$[\![\varphi]\!] = \{\mathbf{t} \mid \mathbf{t} = \langle t_1, \ldots, t_n\rangle \in \mathbb{N}^n \text{ satisfies } \varphi\}.$$

Note that $[\![true]\!] = \mathbb{N}^n$. The definition is extended to sets as follows

$$[\![\Phi]\!] = \bigcup_{\varphi \in \Phi} [\![\varphi]\!].$$

Definition 12. We say that a constraint ψ *entails* a constraint φ, written $\varphi \sqsubseteq \psi$, if and only if $[\![\psi]\!] \subseteq [\![\varphi]\!]$.

Constraints give a very concise representation of *sets of states* of a protocol. A constraint represents symbolically a sets of EFSM-states, and each EFSM-state represents a set of *symmetric* states of the family of global machines associated with the protocol.

Example 6. Let us consider again the Illinois protocol. Let $\langle c_1, c_2, c_3, c_4 \rangle$ be the EFSM-state where $x_i = c_1$, $x_e = c_2$, $x_s = c_3$, and $x_d = c_4$. Then, the constraint φ defined as $x_d \geq 1 \wedge x_s \geq 1$ represents the following set of EFSM-states.

$$[\![\varphi]\!] = \{\langle 0, 0, 1, 1\rangle, \langle 0, 0, 1, 2\rangle, \ldots, \langle 1, 0, 2, 2\rangle, \ldots\}$$

In turn, each tuple in $[\![\varphi]\!]$ represents a collection of symmetric global states. For instance,

$$\langle 0, 0, 1, 1\rangle \quad \text{denotes} \quad \{\langle \mathit{dirty}, \mathit{shared}\rangle, \langle \mathit{shared}, \mathit{dirty}\rangle\}.$$

In order to define a *symbolic* reachability procedure for EFSMs, we need to formulate the *predecessor* operator of Definition 9 in terms of an operator working over constraints. For this purpose, we introduce existentially quantified constraints having the form $\exists x.\varphi$. The denotation of such an existentially quantified constraint is defined as follows:

$$[\![\exists x.\varphi]\!] = \bigcup_{t \in \mathbb{N}} [\![\varphi[x \mapsto t]]\!]$$

As an example, $[\![\exists x. y \leq x \wedge x \leq 2]\!] = [\![y \leq 2]\!]$, whereas we have that $[\![\exists x. y < x \wedge x < y + 1 = \emptyset]\!]$. Note that the variables that occur free in a constraint are all *implicitly* existentially quantified.

Remark 1. Following from [44], given an existentially quantified constraint φ, there exists an algorithm to compute a linear arithmetic constraint ψ such that $[\![\varphi]\!] = [\![\psi]\!]$.

6.1. Symbolic predecessor operator

Based on the previous definitions, we can define a *symbolic version* **sym_pre** of the predecessor operator *pre* of Definition 9, Section 5. Formally, given an EFSM $\mathcal{E}$ and a constraint $\varphi(\mathbf{x}')$ with variables over $\mathbf{x}'$, **sym_pre** is defined as follows

$$\mathbf{sym\_pre}(\varphi(\mathbf{x}')) = \bigcup_{R \in \mathcal{E}} \{\exists\, \mathbf{x}'.\ R(\mathbf{x}, \mathbf{x}')\ \wedge\ \varphi(\mathbf{x}')\}$$

where $\mathbf{x}$ and $\mathbf{x}'$ range over $\mathbb{N}$, and $R(\mathbf{x}, \mathbf{x}') = G(\mathbf{x}) \rightarrow T(\mathbf{x}, \mathbf{x}')$ is an EFSM-transition. The operator **sym_pre** satisfies the following properties.

Proposition 1. *For any linear arithmetic constraint φ, there exists a* set *of linear arithmetic constraints Ψ such that $[\![\Psi]\!] = [\![\mathbf{sym\_pre}(\varphi)]\!]$.*

Proof: By definition of **sym_pre** and Remark 1. □

Proposition 2. *For any linear constraint φ, $[\![\mathbf{sym\_pre}(\varphi)]\!] = pre([\![\varphi]\!])$.*

Proof: Let us prove that for a fixed constraint φ and a fixed EFSM-transition $R(\mathbf{x}, \mathbf{x}')$, $[\![\mathbf{sym\_pre}(\varphi)]\!] = pre([\![\varphi]\!])$. The left-hand side of the equation can be rewritten as

$$\begin{aligned}[\![\mathbf{sym\_pre}(\varphi)]\!] &= [\![\exists\, \mathbf{x}'.\ R(\mathbf{x}, \mathbf{x}') \wedge \varphi(\mathbf{x}')]\!] \\ &= \bigcup_{\mathbf{t}' \in [\![\varphi(\mathbf{x}')]\!]} [\![R(\mathbf{x}, \mathbf{t}')]\!]\end{aligned}$$

And, by definition of *denotations* and *pre*, we have the following

$$\begin{aligned}[\![\mathbf{sym\_pre}(\varphi)]\!] &= \{\mathbf{t} \mid R(\mathbf{t}, \mathbf{t}') \equiv \mathit{true} \text{ and } \mathbf{t}' \in [\![\varphi(\mathbf{x}')]\!]\} \\ &= pre([\![\varphi]\!]).\end{aligned}$$

□

7. Backward reachability

Via the counting abstraction, we can reduce parameterized verification problems to reachability problem for EFSMs. The symbolic representation of infinite sets of EFSM-states via linear arithmetic constraints can be used then to explore efficiently the state space of the original protocol. In our approach we will use a variation of the *backward reachability* procedure of [2], where all operations on sets of states are lifted to the constraint-level. The resulting procedure, called Symb-Reach, is shown in figure 6. The procedure works as follows. Starting from a set of linear constraints Φ_f that denotes *unsafe* protocol states, the procedure repeatedly applies the operator **sym_pre** until the set of symbolic representations of the predecessors of Φ_f is saturated. The termination test is implemented using the entailment relation between linear constraints (i.e. the logical implication of the corresponding formulas). If the procedure terminates, we still have to check whether the initial states (represented as the set of constraints Φ_o) are in the denotation of the resulting set of constraints. We can check this property via the test *sat* defined as follows:

$$sat(\Phi \wedge \Psi) = \bigvee_{\varphi \in \Phi, \psi \in \Psi} \varphi \wedge \psi \text{ is satisfiable.}$$

The reason why we adopt *backward reachability* is based on the following considerations. Let us first introduce the notion of upward closed set of states.

```
Proc Symb-Reach(E : EFSM, Φ_o, Φ_f : sets of constraints)
  Φ := Φ_f;
  Ψ := ∅;
  while Φ ≠ ∅ do
    select φ ∈ Φ and set Φ := Φ \ {φ};
    if there are no ψ ∈ Ψ such that ψ ⊑ φ
       then
         Ψ := Ψ ∪ {φ};
         Φ := Φ ∪ sym_pre(φ);
  endwhile;
  if sat(Φ_o ∧ Ψ)
    then return "Φ_f is reachable from Φ''_o;
    else return "Φ_f is not reachable from Φ''_o.
end.
```

Figure 6. Symbolic reachability.

Definition 13. A set $S \subseteq \mathbb{N}^k$ of EFSM-states is upward-closed whenever for all tuples $\mathbf{t} = \langle t_1, \ldots, t_k \rangle \in S$: if $\mathbf{t}' = \langle t'_1, \ldots, t'_k \rangle$ is greater equal than $\mathbf{t}$ wrt. the component-wise ordering of tuples (i.e. $t_i \leq t'_i$ for $i : 1, \ldots, k$), then $\mathbf{t}'$ is also in S.

An upward closed set of tuples of non-negative integers is always generated by a finite number of minimal points.

Example 7. The denotation of the constraint $x_s \geq 1 \wedge x_d \geq 1$ (the variables x_i and x_e are implicitly ≥ 0) is an upward-closed set over $\mathbb{N}^4$. The tuple $\langle 0, 0, 1, 1 \rangle$ is the mininal point that generates the set.

In contrast to *forward reachability*, the procedure of [2] always terminates whenever all the guards of the input EFSM (resulting from the translation from global machines) have no equalities, and the set of unsafe states is upward-closed. In the following we will call this class of systems *EFSMs without global conditions*. The application of the counting abstraction to the Broadcast Protocols of [20] returns an EFSM without global conditions [17].

Formally, the operational model underlying a Broadcast Protocol is defined as follows.

Definition 14 (Broadcast Protocols). In the EFSM with variables in $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$ associated with a Broadcast Protocol the transitions are such that

- guards have at most two constraints of the form $x_i \geq 1$;
- transitions have the form $\mathbf{x}' = A \cdot \mathbf{x} + \mathbf{b}$ where A is a matrix with unit vectors as columns, and $\mathbf{b}$ is either the null vector or a vector with two non zero elements having the values 1 and -1.

The form of the guards is induced by the type of communication allowed in a Broadcast Protocol. The Broadcast Protocol proposed in [20] allows in fact internal actions, rendez-vous

synchronization, and broadcast communication, but no global conditions. Clearly, Broadcast Protocols are a subclass of EFSMs without global conditions.

In order to study the properties of the latter class, we first introduce the notion of *additive* constraint.

Definition 15. An *additive constraint* over $x_1, \ldots, x_n$ is a linear arithmetic constraint where the *atomic constraints* have the following form $x_{i_1} + \cdots + x_{i_k} \geq c$ where $x_{i_1}, \ldots, x_{i_k}$ are distinct variables.

Additive constraints can be used to symbolically represent upward-closed set of EFSM-states.

Example 8. The additive constraint $x + y \geq 2, z \geq 3$ denotes the upward closed set generated by the minimal points $\langle 2, 0, 3\rangle$, $\langle 1, 1, 3\rangle$, and $\langle 0, 2, 3\rangle$.

Furthermore, we have the following property.

Lemma 3. *Additive constraints are closed under applications of the operator* **sym_pre** *associated with an EFSM without global conditions.*

Proof: Let φ be an additive constraint and $G(\mathbf{x}) \rightarrow T(\mathbf{x}, \mathbf{x}')$ be a rule of $\mathcal{E}$. By hypothesis $G(\mathbf{x})$ has no equalities. Thus, from the definition of the conditions of Definition 5, Section 4.1, it is easy to verify that $G(\mathbf{x})$ is an additive constraint.

Since conjunctions of additive constraints are still additive constraints, it remains to show that $\exists \mathbf{x}'.T(\mathbf{x}, \mathbf{x}') \wedge \varphi(\mathbf{x}')$ is an additive constraint. By definition, $T(\mathbf{x}, \mathbf{x}')$ is a linear transformation having form $\mathbf{x}' = M \cdot \mathbf{x} + \mathbf{c}$ where M is a matrix with unit vectors as columns, and $\mathbf{c}$ is a vector of integers. Furthermore, $\varphi(\mathbf{x}')$ can be expressed as a system of inequalities $N \cdot \mathbf{x}' \geq \mathbf{d}$, where N is a matrix with 0, 1 coefficients. Thus, if we replace $\mathbf{x}'$ in φ with the right-hand side of $T(\mathbf{x}, \mathbf{x}')$ we obtain:

$$N \cdot (M \cdot \mathbf{x} + \mathbf{c}) \geq \mathbf{d} = (N \cdot M) \cdot \mathbf{x} \geq \mathbf{d} - N \cdot \mathbf{c}.$$

The matrix product $N \cdot M$ yields a matrix with 0, 1 coefficients. Thus, the resulting expression can still be written as an additive constraint. □

Then, the following lemma holds.

Lemma 4. *The class of additive constraints equipped with the entailment relation $\sqsubseteq$ is a* well-quasi ordering.

Proof: The proof is given in [4]. □

We recall that a pre-order $\langle D, <\rangle$ is a *well-quasi ordering* if for every infinite sequence of elements in D $d_1 d_2 \ldots$ there exists indices i, j such that $i < j$ and $d_i < d_j$.

Thus, Lemma 4 guarantees that there cannot be infinite sequences $\varphi_1 \ldots \varphi_i$ of additive constraints such that for all j and for all $i < j$, $\varphi_i \not\sqsubseteq \varphi_j$. Based on this result, we obtain the following theorem.

Theorem 3. *The procedure Symb-Reach of figure* 6 *always terminates whenever the argument $\mathcal{E}$ is an EFSM without global conditions, and Φ_f is a set of* additive *constraints.*

Proof: From Lemmas 3 and 4. □

As a corollary, we obtain the termination for the symbolic analysis for EFSMs associated with Broadcast Protocols.

7.1. Complexity issues

Although termination is guaranteed only for subclasses of EFSMs, in practical cases (see Section 9) few iterations of the main loop of the procedure Symb-Reach are needed to achieve a 'fixpoint'. For this reason, it is interesting to study the complexity of the operations involved in each single iteration of the procedure.

Let $\mathcal{E}$ be an EFSM, and $\varphi(\mathbf{x}')$ be a linear constraint. For each rule R in $\mathcal{E}$, we first have to replace all primed variables in φ with the right-hand side of the transition T_R. Then, we have to check the satisfiability of the resulting constraint conjoined with the guard G_R. While the first step is polynomial in the size of φ and T_R, the second step may be exponential for an arbitrary φ. In fact, checking satisfiability of a *integer linear problem* has been proved to be NP-complete [44]. Note that the test is trivial if φ is an additive constraint. Let us consider now the entailment test. As shown in [17], checking that $\varphi \sqsubseteq \psi$ holds is co-NP hard. Thus, the test may require exponential time in the size of the input constraints. The satisfiability test between Φ_o and Ψ may have exponential cost in the size of the constraints in Φ_o and Ψ.

As a final remark, we mention that non elementary bounds on the number of iterations needed for the termination of the decidable subclass of EFSMs have been studied in [35].

7.2. Relaxed constraint operations

In order to reduce the complexity of the manipulation of linear arithmetic constraints, we can resort to the relaxation technique used in integer programming and program analysis. Specifically, every time we need to solve a system of inequalities

$$A \cdot \mathbf{x} \geq \mathbf{b}$$

we relax the condition that $\mathbf{x}$ must be a vector of non-negative *integers* and look for a *real* solution of the corresponding linear problem. We apply the *relaxation* to the operations over constraints, i.e., we interpret the *satisfiability* test, *variable elimination*, and *entailment* test (needed to implement the symbolic backward reachability procedure) over the domain of *reals*. The relaxation allows us to exploit efficient (polynomial) operations over the reals in

contrast to potentially exponential operations over the integers. The drawback is a potential loss of precision of the analysis.

The relaxation is defined formally as follows. An assignment $\mathbf{r} = \langle r_1, \ldots, r_n \rangle \in \mathbb{R}_+^n$ *satisfies* a linear arithmetic constraint φ defined over $x_1, \ldots, x_n$ if and only if $\varphi[x_1 \mapsto r_1, \ldots, x_n \mapsto r_n]$ evaluates to *true* (according to the usual interpretations of the operators in φ).

Definition 16. The denotation over $\mathbb{R}_+$ of a linear arithmetic constraint φ is defined as

$$[\![\varphi]\!]_{\mathbb{R}} = \{\mathbf{r} \in \mathbb{R}_+ \mid \mathbf{r} \text{ satisfies } \varphi\}.$$

The definition extends to sets in the natural way.

Definition 17. The entailment relation over $\mathbb{R}_+$ is defined as $\varphi \sqsubseteq_{\mathbb{R}} \psi$ if and only if $[\![\psi]\!]_{\mathbb{R}} \subseteq [\![\varphi]\!]_{\mathbb{R}}$.

Definition 18. Given an existential linear constraint $\exists x.\varphi$,

$$[\![\exists x.\varphi]\!]_{\mathbb{R}} = \bigcup_{r \in \mathbb{R}_+} [\![\varphi[x \mapsto r]]\!]_{\mathbb{R}}.$$

Following [44], given an existential linear constraint φ, there are algorithms (e.g. Fourier-Motzkin variable elimination) to compute a linear arithmetic constraint ψ such that $[\![\varphi]\!]_{\mathbb{R}} = [\![\psi]\!]_{\mathbb{R}}$. Finally, if we apply the relaxation to the symbolic predecessor operator, we obtain the new operator $\mathbf{sym\_pre}_{\mathbb{R}}$ defined as follows

$$\mathbf{sym\_pre}_{\mathbb{R}}(\varphi(\mathbf{x}')) = \bigcup_{i \in I} \{\exists \mathbf{x}'.G_i(\mathbf{x}) \wedge\ T_i(\mathbf{x}, \mathbf{x}') \wedge \varphi(\mathbf{x}')\},$$

where $\mathbf{x}$ and $\mathbf{x}'$ range now over *positive real numbers*.

7.3. Relaxed backward reachability

The symbolic reachability procedure obtained by applying the relaxed constraint operators introduced in the previous section is shown in figure 7. This is the procedure for backward reachability implemented in existing symbolic model checkers for hybrid and concurrent systems like HYTECH [28].

If the procedure Symb-Reach-over-$\mathbb{R}$ of figure 7 terminates computing a set Ψ of constraints, it returns an over-approximation, namely $[\![\Psi]\!]$ of the backward reachable states of the EFSM taken into consideration. Thus, the procedure of figure 7 can still be used to *verify* safety properties. If the initial states are not contained in $[\![\Psi]\!]_{\mathbb{R}}$, then no such state is in $[\![\Psi]\!]$.

The procedure $\mathbf{sym\_pre}_{\mathbb{R}}$ can be implemented using the satisfiability test over $\mathbb{R}$, and a variable elimination algorithm (for a fixed number of variables, polynomial in the size of the input constraints). Furthermore, $\varphi \sqsubseteq_{\mathbb{R}} \psi$ can be tested in polynomial time: $\phi \sqsubseteq_{\mathbb{R}} (\psi_1 \wedge \psi_2)$

```
Proc Symb-Reach-over-ℝ(ℰ : EFSM, Φ_o, Φ_f : sets of constraints)
  Φ := Φ_f;
  Ψ := ∅;
  while Φ ≠ ∅ do
    select φ ∈ Φ and set Φ := Φ \ {φ};
    if there are no ψ ∈ Ψ such that ψ ⊑_ℝ φ
       then
         Ψ := Ψ ∪ {φ};
         Φ := Φ ∪ sym_pre_ℝ(φ);
  endwhile;
  if sat_ℝ(Φ_o ∧ Ψ)
    then return 'Φ_f is reachable from Φ'_o;
    else return 'Φ_f is not reachable from Φ'_o.
end.
```

Figure 7. Symbolic reachability over $\mathbb{R}$.

holds if and only if $\phi \wedge \neg\psi_1$ and $\phi \wedge \neg\psi_2$ are not satisfiable. Thus, each step of the procedure Symb-Reach-over-$\mathbb{R}$ involves only polynomial time operations.

The procedure of figure 7 has the following interesting properties when executed on EFSMs without global conditions.

Lemma 5. *Additive constraints are closed under applications of the operator* $\mathbf{sym\_pre}_{\mathbb{R}}$ *associated with an EFSM without global conditions.*

Proof: We first note that the elimination of the primed variables in the body of the definition of $\mathbf{sym\_pre}_{\mathbb{R}}$ corresponds to a replacement by equals both over integers and over reals.

Furthermore, additive constraints are always satisfiable both over reals and over integers. The thesis follow then from Proposition 3. □

As a consequence, given an EFSM without global conditions if φ is an additive constraint, then

$$[\![\mathbf{sym\_pre}_{\mathbb{R}}(\varphi)]\!] = [\![\mathbf{sym\_pre}(\varphi)]\!] = \mathit{pre}([\![\varphi]\!])$$

Here $[\![\cdot]\!]$ denotes *integer* solutions.

In order to find sufficient conditions for termination of the procedure working over reals we need to consider the following restricted class of additive constraints introduced in [17].

Definition 19. An *additive constraint with distinct variables* is such that the sets of variables occurring in atomic constraints are mutually disjoint.

Additive constraints can always be expressed in terms of disjunctions of additive constraints with distinct variables.

Example 9. The constraint

$$x + y + z \geq 2, x \geq 1$$

is equivalent to the disjunction

$$x \geq 2, y + z \geq 0 \quad \vee \quad x \geq 1, y + z \geq 1$$

In general this reduction may return an exponential number of constraints. However, there exists a *polynomial algorithm* that maintains constraints in additive form with distinct variables under applications of **sym_pre** when the input EFSM is a Broadcast protocol [17]. This property is due to the special form of guards of Broadcast Protocols (at most two constraints of the form $x \geq 1$) that reduces the number of possible decompositions of additive constraints. Note, in fact, that the application of $T(\mathbf{x}, \mathbf{x}')$ to an additive constraint $\varphi(\mathbf{x}')$ with distinct variables returns a new additive constraint ψ with distinct variables. In order to maintain this special form in the computation of $\mathbf{sym\_pre}_{\mathbb{R}}$, we just have to simplify each one of two guards $x \geq 1$ with the constraint ψ. As a consequence, we only need to consider a linear number of possible decompositions of the atomic constraint in ψ in which x occurs.

The following property then holds.

Lemma 6. *Let $x_1, \ldots, x_n$ be a fixed set of variables. The class of additive constraints with distinct variables equipped with the entailment relation $\sqsubseteq_{\mathbb{R}}$ forms a* well-quasi ordering.

Proof: Per absurdum, let $\{\varphi_i\}_{i \geq 0}$ be an infinite sequence of additive constraints with distinct variables such that, for any i, $\varphi_q \not\sqsubseteq \varphi_i$ for all $q < i$. Since there are finitely many possible sums $S_k = x_{k1} + \cdots + x_{kr}$ over the variables $x_1, \ldots, x_n$, there exists a subsequence $\{\psi_j\}_{j \geq 0}$ such that:

- for any j $\psi_q \not\sqsubseteq \psi_j$ for all $q < j$;
- all constraints in the subsequence are defined over the same sums $\psi_j = \wedge_{k=1}^{m} S_k \geq c_{jk}$

Furthermore, we have the following properties.

- Since atomic constraints have distinct variables each other, $S_1 \geq c_{i1}, \ldots, S_r \geq c_{ir} \sqsubseteq_{\mathbb{R}} S_1 \geq c_{j1} \ldots, S_r \geq c_{jr}$ holds if and only if $S_k \geq c_{ik} \sqsubseteq_{\mathbb{R}} S_k \geq c_{jk}$ holds for $k : 1, \ldots, r$.
- $S_k \geq d \sqsubseteq_{\mathbb{R}} S_k \geq c$ can be checked equivalently over the reals and over the integers. In fact, $S_k \geq d \sqsubseteq_{\mathbb{R}} S_k \geq c$ holds if and only if $c \leq d$.

Let us now rename every sum S_k with a new variable u_k in every constraint ψ_j occurring in the subsequence. We obtain a new infinite sequence of constraints of the form

$$\gamma_j \; = \; u_1 \geq c_{j1}, \ldots u_r \geq c_{jr}$$

Furthermore, by the previous observations, we have that $\gamma_q \sqsubseteq_{\mathbb{R}} \gamma_j$ if and only if $\psi_q \sqsubseteq_{\mathbb{R}} \psi_j$. To conclude, we note that the constraints defined over the variables $u_1, \ldots, u_k$ are a subclass

of additive constraints. The existence of an infinite sequence of such constraints would violate Lemma 4. □

Based on the previous results, we obtain the following theorem.

Theorem 4. *The procedure Symb-Reach-over-*$\mathbb{R}$ *of figure 7 always terminates whenever the argument* $\mathcal{E}$ *is an EFSM without global conditions,* Φ_f *is a set of* additive *constraint, and intermediate constraints are maintained in* additive form with distinct variables.

Proof: From Lemma 5 and Lemma 6. □

As a corollary, we obtain that the procedure Symb-Reach-over-$\mathbb{R}$ of figure 7 always terminates for Broadcast Protocols if constraints are maintained in additive form with distinct variables. To our knowledge, this result was not considered in previous works on well structured system [2, 22].

In practice the procedure Symb-Reach-over-$\mathbb{R}$ behaves well on a larger class of examples (e.g. where it is not possible to maintain constraints in additive form). In the rest of the paper we will discuss experimental results on several cache coherence protocols taken from the literature.

8. A gallery of verification problems

In this section we will present a gallery of verification problems for the MESI, Berkeley RISC, Illinois, Xerox PARC Dragon and DEC Firefly protocols [27]. In Section 9 we will discuss the experimental results obtained running HYTECH on these problems.

Let us first summarize the verification problems for the two case-studies we introduced in the previous sections.

8.1. Synapse N + 1 protocol

The protocol in specified in Example 1, figure 2. The EFSM obtained by compiling the finite-state model is shown in figure 4. We recall that the possible sources of data inconsistency are:

UNS_1, a *dirty* cache co-exists with one or more caches in state *valid*;
UNS_2, there is more than one *dirty* cache.

The parameterized initial configuration is expressed as $\Phi_o = \mathit{invalid} \geq 1, \mathit{dirty} = 0, \mathit{valid} = 0$. The potentially *unsafe* states UNS_1 are represented as the constraint $\Phi_1 = \mathit{dirty} \geq 1, \mathit{valid} \geq 1$, and the *unsafe* states w.r.t. the invariant UNS_2 is the constraint $\Phi_2 = \mathit{dirty} \geq 2$.

8.2. Illinois protocol

We described the protocol in Example 2, figure 3. The EFSM obtained by compiling the finite-state model is shown in figure 5. The possible sources of data inconsistency are:

UNS_1, a *dirty* cache co-exists with caches either in state *shared* or em valid-exclusive;
UNS_2, there is more than one *dirty* cache.

The other possible violations of the exclusivity of state *valid-exclusive* are:

UNS_3: there is more than one *valid-exclusive* cache;
UNS_4: a *shared* cache co-exists with a cache in state em valid-exclusive.

The parameterized initial configuration is expressed as $\Phi_o = x_i \geq 1, x_e = 0, x_d = 0, x_s = 0$. Similarly, we can represent the potentially *unsafe* states as follows.

UNS_1 $x_d \geq 1,\ x_s + x_e \geq 1$
UNS_2 $x_d \geq 2$
UNS_3 $x_e \geq 2$
UNS_4 $x_e \geq 1,\ x_s \geq 1$

8.3. MESI protocol

The MESI protocol is a write-once cache-based protocol [27]. The acronym MESI denotes the four states of the protocol, named *modified*, *exclusive*, *shared* and *invalid*. The *exclusive* state identifies cache lines that have been written *once* by the corresponding CPU; the main memory is current with the contents of the cache. The *modified* state identifies cache lines that have been written *more than once* by the corresponding CPU; the only current version of the modified block resides in the cache. If one cache has an *exclusive* or *modified* state, all matching lines in other caches are marked *invalid*. Shared is the only state which allows another (valid) copy of the same memory block to be stored in other caches. The *exclusive* state must be entered before going to the *modified* state.

Read Hit: no coherence action is needed.
Read Miss: the cache goes from *invalid* to *shared*. The other caches with *exclusive* or *modified* state go to *shared*.
Write Miss: the cache goes from *invalid* to *exclusive* after invalidating the contents of all other caches.
Write Hit: if the cache is in the *shared* state, it will go to *exclusive* after invalidating the contents of all the other caches. The transition from *exclusive* to *modified* does not require any bus transaction.

The protocol is given in figure 8. where we used the internal actions *Rh*, *Wh*, and *We*, and the broadcasts Rm, WI. The EFSM associated with the MESI protocol is shown in figure 9. The counters x_m, x_e, x_s, x_i are associated with the states *modified*, *exclusive*, *shared*, and *invalid*,

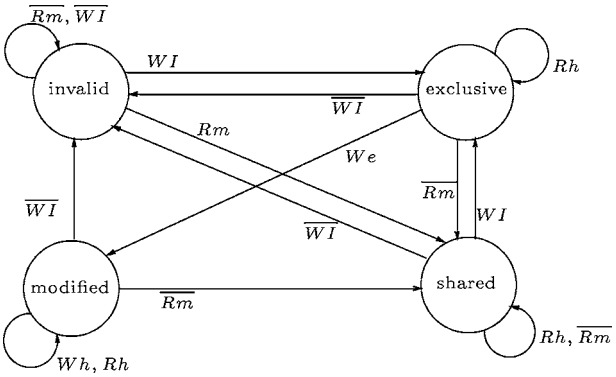

Figure 8. The MESI write-once protocol from the perspective of cache C_i.

$$
\begin{array}{ll}
(rh) & x_m + x_s + x_e \geq 1 \ \rightarrow \ skip. \\
(rm) & x_i \geq 1 \ \rightarrow \\
 & \quad x_i' = x_i - 1, x_e' = 0, x_m' = 0, x_s' = x_s + x_e + x_m + 1. \\
(wh_1) & x_m \geq 1 \ \rightarrow \ skip. \\
(wh_2) & x_e \geq 1 \ \rightarrow \ x_e' = x_e - 1, x_m' = x_m + 1. \\
(wh_3) & x_s \geq 1 \ \rightarrow \\
 & \quad x_s' = 0, x_e' = 1, x_m' = 0, x_i' = x_i + x_m + x_e + x_s - 1. \\
(wm) & x_i \geq 1 \ \rightarrow \\
 & \quad x_e' = 1, x_s' = 0, x_m' = 0, x_i' = x_i + x_e + x_m + x_s - 1.
\end{array}
$$

Figure 9. EFSM for the MESI protocol.

respectively. Rule (rh) denotes read hits. Rule (rm) denotes a read miss. Rule (wh_1), (wh_2) and (wh_3) denote write hits in state *modified*, *exclusive* and *shared*, respectively. Rule (wm) denotes a write miss.

In the MESI protocol there are two possible sources of data inconsistency:

UNS_1, a *modified* cache co-exists with caches either in state *shared* or in state *exclusive*;
UNS_2, there is more than one *modified* cache.

The other possible violations of the exclusivity of state *exclusive* are:

UNS_3, an *exclusive* cache co-exists with caches either in state *shared* or in state *exclusive*;
UNS_4, there is more than one *exclusive* cache.

The parameterized initial configuration is expressed as $\Phi_o = x_i \geq 1,\ x_e = 0,\ x_s = 0,\ x_m = 0$. The potentially *unsafe* states UNS_1 are represented as follows.

UNS_1 $x_m \geq 1,\ x_s + x_e \geq 1$
UNS_2 $x_m \geq 2$
UNS_3 $x_e \geq 1,\ x_s \geq 1$
UNS_4 $x_e \geq 2$

8.4. MOESI protocol

The MOESI is a protocol with direct-data intervention and ownership. The acronym stands for states with the same name as the MESI protocol (but different meaning), with the addition of the Owned state. If a cache is in state Shared then the cache line contains a copy of the owner's data that may or may not be current with main memory (Note: in the MESI protocol, Shared means that the line is current with main memory). Exclusive means that the cache line has been written once and it has not been snooped by other caches. Modified means that the cache line has been written more than once and it has not been snooped. Finally, Owned means that the cache line has been written more than once and it has been snooped, i.e., other caches in state Shared may coexist.

Read Hit: no coherence action is needed.
Read Miss: the cache updates the line (either from main memory or from the owner) and goes to *shared*. If the owner is in the *modified* state it goes to *owned*. If another cache is in state *exclusive* it moves to *shared*.
Write Hit: if the cache is in state *exclusive* or *modified* the cache ends up in state *modified* without bus interaction. If the cache is either in state *shared* or in state *owned* then it writes the line, writes to the bus (again to invalidate the other caches) and moves to *exclusive*; all other caches invalidate their contents.
Write Miss: the cache writes the line and writes to the bus for invalidating the other caches, and then goes to *exclusive*; all other caches invalidate their contents.

Note that on a write hit an *owned* line is downgraded to *exclusive*, whereas all its copies are invalidated. After this bus cycle main memory will be current with the *exclusive* line. Thus, if another write occurs it is not necessary to update main memory (as if the cache would have remained in state *owned*).

The protocol is given in figure 10; the internal actions are Wh, Rh, We. The EFSM for the MOESI protocol is shown in figure 11. The counters x_m, x_o, x_e, x_s, x_i are associated with the states *modified*, *owner*, *exclusive*, *shared*, and *invalid*, respectively. Rule (rh) denotes read hits. Rule (rm) denotes a read miss. Rule (wh_1) and (wh_2) denote write hits in state *modified* and *exclusive*; no bus activity is needed. Rule (wh_3) denotes a write hit either in state *shared* or *owned* with bus invalidation. Finally, rule (wm) denotes a write miss. The contents of all other caches is invalidated.

In the MOESI protocol there are several possible sources of data inconsistency or possible exclusivity violations:

UNS_1, a *modified* cache co-exists with one or more caches either in state *shared*, *exclusive* or *owned*;
UNS_2, an *exclusive* cache co-exists with one or more caches either in state *shared*, or *owned*;
UNS_3, there is more than one *modified* cache;
UNS_4, there is more than one *exclusive* cache.

The parameterized initial configuration is expressed as $\Phi_o = invalid \geq 1,\ x_e = 0,\ x_o = 0,\ x_s = 0,\ x_m = 0$. The potentially *unsafe* states UNS_1 are represented as follows.

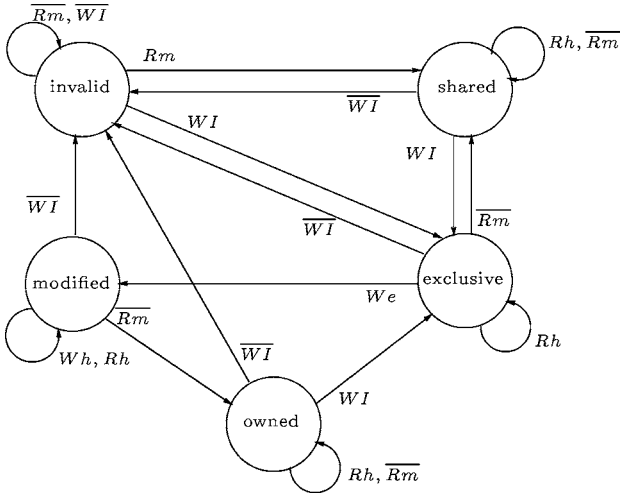

Figure 10. The MOESI write-once protocol from the perspective of cache C_i.

$$
\begin{array}{ll}
(rh) & x_m + x_o + x_s + x_e \geq 1 \ \rightarrow \ skip. \\
(rm) & x_i \geq 1 \ \rightarrow \\
 & \quad x_i' = x_i - 1, x_e' = 0, x_m' = 0, \\
 & \quad x_s' = x_s + x_e + 1, x_o' = x_o + x_m. \\
(wh_1) & x_m \geq 1 \ \rightarrow \ skip. \\
(wh_2) & x_e \geq 1 \ \rightarrow \ x_e' = x_e - 1, x_m' = x_m + 1. \\
(wh_3) & x_s + x_o \geq 1 \ \rightarrow \\
 & \quad x_e' = 1, x_s' = 0, x_m' = 0, x_o' = 0, \\
 & \quad x_i' = x_i + x_e + x_m + x_o + x_s - 1. \\
(wm) & x_i \geq 1 \ \rightarrow \\
 & \quad x_e' = 1, x_s' = 0, x_m' = 0, x_o' = 0, \\
 & \quad x_i' = x_i + x_e + x_m + x_o + x_s - 1.
\end{array}
$$

Figure 11. EFSM for the MOESI protocol.

UNS_1 $x_e + x_s + x_o \geq 1,\ x_m \geq 1$
UNS_2 $x_e \geq 1,\ x_s + x_o \geq 1$
UNS_3 $x_m \geq 2$
UNS_4 $x_e \geq 2$.

8.5. *Berkeley protocol*

The Berkeley protocol is a variation of MESI with write-allocation and with a *shared modified* state, named *owned non-exclusively*. In this state the main memory is not coherent with the possible multiple, cached copies of the owner data. The other three states are *invalid*, *unowned* (similar to the MESI *shared* state), and *owned exclusively* (similar to the

MESI *modified* state). On a read (write) miss, the owner provides the data during special bus cycles named Conventional Read, and Read-for-Ownership. The latter is used to take the ownership.

Read Hit: no coherence action is needed.

Read Miss: the cache updates the line from the 'owner' using a Conventional Read cycle, and goes to Unowned. In the Conventional Read cycle a cache in state *owned non-exclusively* outputs the data without changing state, whereas a cache in state *owned exclusively*, after providing the data, updates its state to *owned non-exclusively*.

Write Miss: the cache updates the line from the owner starting a Read-for-Ownership cycle (write-allocation), merges the line with the new data, and then goes to *owned exclusively* (i.e., it becomes the owner of the valid data). During the Read-for-Ownership cycle, the current owner outputs the data and invalidates its line. The caches in state *unowned* invalidate their lines.

Write Hit: if the cache is in state *unowned*, it writes the line and then starts a Write-for-Invalidation cycle during which all the caches either in state *unowned* or *owned non-exclusively* invalidate their lines. Then, it updates its state to *owned exclusively*. If the cache is in state *owned non-exclusively*, it writes the line, starts a Write-for-Invalidation cycle and then goes to *owned exclusively*. Finally, if the cache is in state *owned exclusively*, it writes the line without state changes and without bus invalidation.

The protocol is given in figure 12; the only internal actions are Wh and Rh. The EFSM associated with the Berkeley protocol is shown in figure 13. The counters x_n, x_e, x_u, x_i are associated with the states *owned non-exclusively*, *unowned*, *owned exclusively*, and *invalid*, respectively. Rule (rh) denotes *read hits*. Rule (rm) denotes a *read miss*. Rule (wm) denotes a *write miss* together with the corresponding Read-for-Ownership cycle (denoted by WO in figure 12). Rule (wh_1) denotes a *write hit* either in state *owner non-exclusive* or in state *unowned* with the corresponding Invalidate cycle (denoted by WI in figure 12). Rule (wh_2) denotes a *write hit* in state *owner exclusive* (denoted by W in figure 12).

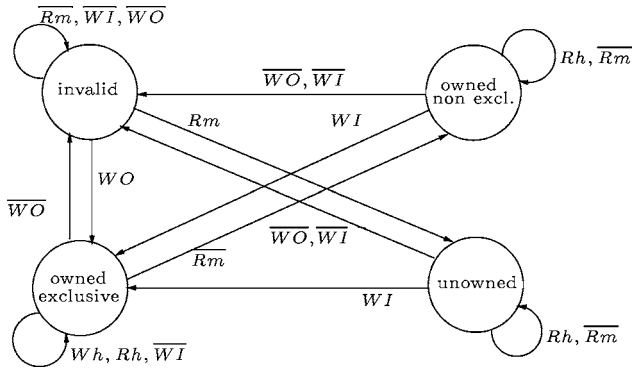

Figure 12. The Berkeley protocol from the perspective of cache C_i.

$$
\begin{array}{ll}
(rm) & x_i \geq 1 \ \rightarrow \ x_i' = x_i - 1, x_u' = x_u + 1, x_e' = 0, x_n' = x_n + x_e. \\
(rh) & x_e + x_n + x_u \geq 1 \ \rightarrow \ skip. \\
(wm) & x_i \geq 1 \ \rightarrow \ x_n' = 0, x_u' = 0, x_e' = 1, x_i' = x_i + x_e + x_n + x_u - 1. \\
(wh_1) & x_n + x_u \geq 1 \ \rightarrow \ x_n' = 0, x_u' = 0, x_e' = x_e + 1, x_i' = x_i + x_n + x_u - 1. \\
(wh_2) & x_e \geq 1 \ \rightarrow \ skip.
\end{array}
$$

Figure 13. EFSM for the Berkeley protocol.

In the Berkeley protocol we have the following sources of data inconsistency:

UNS_1, a *owned exclusively* cache co-exists with one or more caches either in state *owned non-exclusively*, or *unowned*;
UNS_2, there is more than one *owned exclusively* cache;

The parameterized initial configuration is expressed as $\Phi_o = x_i \geq 1,\ x_n = 0,\ x_u = 0,\ x_e = 0$. The potentially *unsafe* states are represented as follows.

UNS_1 $x_e \geq 1, x_u + x_n \geq 1$
UNS_2 $x_e \geq 2$

8.6. DEC Firefly protocol

The DEC Firefly is a write-allocation protocol. As in the Illinois and Futurebus+ protocols, a special signal is used to indicate that a snoop hit is occurred in another processor's cache. In [27], the formulation of the protocol is given without *invalid* state. Intuitively, the caches are supposed to be valid after a sort of *cold start* in which all invalid caches are disabled until their lines have been replaced. The protocol has three states: *valid exclusive*, *shared* and *dirty*. To simplify the presentation, we add a dummy *invalid* state that we use to initialize the system.

Read Hit: no coherence action is needed.
Read and Write Miss: as in the Illinois protocol.
Write Hit: if the cache is in state *dirty*, no action is taken. If the cache is in state *valid exclusive*, its state changes to *dirty* (note: no invalidation signal is needed). If the cache is in state *shared* and no other cache is in state *shared*, then its state changes to *valid exclusive*. If the cache is in state *shared* and there are caches in state *shared*, then it broadcasts the updated line to all other valid caches without changing state.

The protocol is given in figure 14. The only internal actions are *Rh*, *Wh* and *We*. The predicate Q is used by a cache to decide whether or not to move from *shared* to *valid exclusive*, and it is defined as follows.

$$Q \equiv \#shared = 1$$

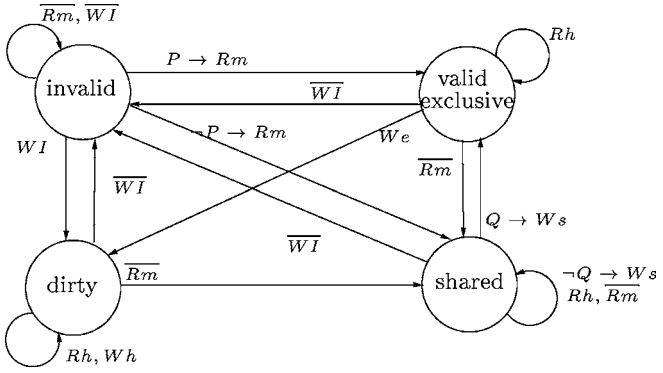

Figure 14. Firefly protocol for 2 caches viewed from the perspective of cache C_i.

Furthermore, its dual predicate $\neg Q$ is defined as

$$\neg Q \equiv \#shared \geq 2$$

On a write miss, we invalidate all valid and dirty caches. In order to handle correctly read miss, we use also the predicate P of the Illinois protocol. The EFSM associated with the Firefly protocol is shown in figure 15. The counters x_d, x_e, x_s, x_i are associated with the states *dirty*, *valid exclusive*, *shared*, and *invalid*, respectively. Rule (rh) denotes *read hits*. Rules (rm_1) and (rm_2) denote a *read miss* (in presence or not of the special signal). Rule (wh_1) through (wh_4) denote write hits. Rule (wh_3) models the condition $g_i = false$ (only one cache is in the *shared* state, and it goes to *exclusive*; rule (wh_3) models the condition $g_i = true$ (at least two caches in state *shared*, no state change is needed). Finally, rule (wm) denotes a write miss.

In the Firefly protocol we have the following possible sources of data inconsistency and of possible exclusivity violations:

$$\begin{aligned}
&(rh) \quad x_d + x_s + x_e \geq 1 \quad \rightarrow \quad skip.\\
&(rm_1) \quad x_i \geq 1, x_d = 0, x_s = 0, x_e = 0 \quad \rightarrow \quad x_i' = x_i - 1, x_e' = x_e + 1.\\
&(rm_2) \quad x_i \geq 1, x_d + x_s + x_e \geq 1 \quad \rightarrow \\
&\qquad\qquad x_i' = x_i - 1, x_s' = x_s + x_d + x_e + 1, x_e' = 0, x_d' = 0.\\
&(wh_1) \quad x_d \geq 1 \quad \rightarrow \quad skip.\\
&(wh_2) \quad x_e \geq 1 \quad \rightarrow \quad x_e' = x_e - 1, x_d' = x_d + 1.\\
&(wh_3) \quad x_s = 1 \quad \rightarrow \quad x_s' = x_s - 1, x_e' = x_e + 1.\\
&(wh_4) \quad x_s \geq 2 \quad \rightarrow \quad skip.\\
&(wm) \quad x_i \geq 1 \quad \rightarrow \quad x_i' = x_i + x_e + x_d + x_s - 1, x_e' = 0, x_s' = 0, x_d' = 1.
\end{aligned}$$

Figure 15. EFSM for the Firefly protocol.

UNS_1, a *dirty* cache co-exists with one or more caches either in state *shared*, or *exclusive*;
UNS_2, there is more than one *exclusive* caches;
UNS_3, there is more than one *dirty* caches;
UNS_4, an *exclusive* cache co-exists with *shared* caches.

The parameterized initial configuration is expressed as $\Phi_o = x_i \geq 1,\ x_e = 0,\ x_s = 0,\ x_d = 0$. The potentially *unsafe* states are represented as follows.

UNS_1 $x_d \geq 1, x_s + x_e \geq 1$
UNS_2 $x_e \geq 2$
UNS_3 $x_d \geq 2$
UNS_4 $x_e \geq 1, x_s \geq 1$

8.7. Xerox PARC Dragon protocol

Dragon is a write-allocation protocol that uses a signal to indicate snoop hits on the bus. The protocol has four states *shared clean* (multiple clean copy may coexist), *shared dirty* (multiple dirty copies may coexist), *shared valid exclusive* (the cache has an exclusive clean copy), and *dirty* (the cache has an exclusive dirty copy). The possible transitions from the perspective of cache C_i are as follows.

Read Hit: no coherence action is needed.
Read Miss: if the read miss does not generate snoop hits in other caches (the signal has not been raised) the line is loaded into invalid cache as *valid exclusive*; otherwise, the line is loaded as *shared clean*. The responding caches take their line from *valid exclusive* to *shared clean*, or from *dirty* to *shared dirty*; *shared dirty* and *shared clean* responding caches do not change state.
Write Miss: since the protocol uses write allocation, before writing the line the cache updates the data as in a read miss cycle. During the preliminary read cycle, *valid exclusive* and *dirty* caches update their state to *shared clean* and *shared dirty*, respectively. If the read miss does not generate snoop hit signals the cache goes to *dirty*; otherwise, it broadcasts the data via a Cache Write cycle to the other caches, and then goes to *shared dirty*. In response to the Cache Write cycle, any other *shared dirty* cache will update the line and go to *share clean*. Note that all caches that originally were either in state *valid exclusive* or *dirty* end up in state *shared clean* as result of the read and Cache write cycles.
Write Hit: if the cache is either in state *dirty* or *valid exclusive* no coherence action is needed and the cache will end up in state *dirty*. If the cache is in state *shared clean* or *shared dirty* the line is broadcast to all other caches via a Cache Write cycle. If some other cache responds to the Cache Write cycle with the snoop hit signal, the line status will be *shared dirty*, otherwise it will be *dirty*.

Note that in the description of a Write cycle nothing is said about the behavior of lines in state other than *shared dirty* and *shared clean*, we will model the protocol in accord to this assumption.

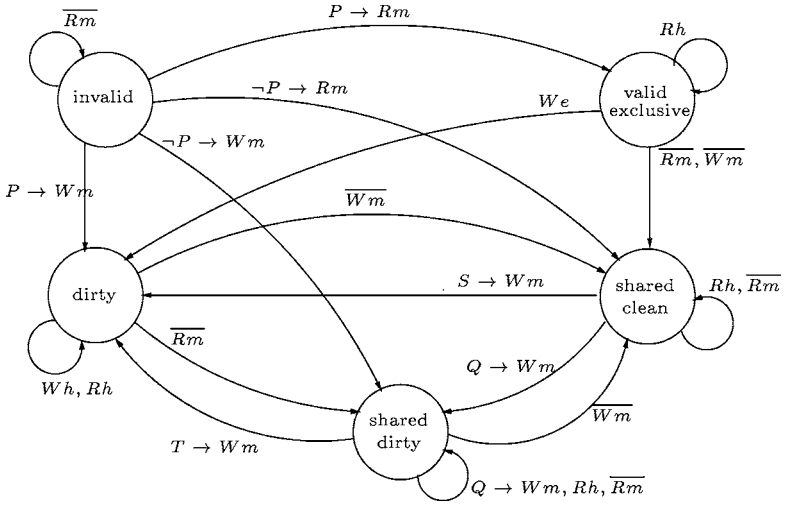

Figure 16. Dragon protocol for 2 caches viewed from the perspective of cache C_i.

The protocol is given in figure 16. The internal actions are Rh, Wh, and We. In order to model the special signals we use the following predicates:

– the predicate P is used to decide if a cache can go from invalid to an exclusive state

$$P \equiv \#exclusive = 0 \ \wedge \ \#dirty = 0 \\ \wedge \#shared\_dirty = 0 \ \wedge \#shared\_clean = 0$$

– the predicate $\neg P$ is the dual of P

$$\neg P \equiv \#exclusive \geq 1 \ \vee \ \#dirty \geq 1 \\ \vee \#shared\_dirty \geq 1 \ \vee \ \#shared\_clean \geq 1$$

– the predicate Q is used to decide if there are other caches in a shared state

$$Q \ \equiv \ \#shared\_clean + \#shared\_dirty \geq 2$$

– the predicate S is the dual of Q from the perspective of a cache in state *shared_clean*

$$S \ \equiv \ \#shared\_clean = 1 \wedge \#shared\_dirty = 0$$

– the predicate T is the dual of Q from the perspective of a cache in state *shared_dirty*

$$T \ \equiv \ \#shared\_clean = 0 \wedge \#shared\_dirty = 1$$

$$
\begin{array}{ll}
(rh) & x_d + x_{sc} + x_e + x_{sd} \geq 1 \ \rightarrow \ skip. \\
(rm_1) & x_i \geq 1, x_d = 0, x_{sc} = 0, x_{sd} = 0, x_e = 0 \ \rightarrow \\
 & \quad x_i' = x_i - 1, x_e' = x_e + 1. \\
(rm_2) & x_i \geq 1, x_{sc} + x_{sd} + x_e + x_d \geq 1 \ \rightarrow \\
 & \quad x_i' = x_i - 1, x_d' = 0, x_e' = 0, x_{sd}' = x_{sd} + x_d, \\
 & \quad x_{sc}' = x_{sc} + x_e + 1. \\
(wm_1) & x_i \geq 1, x_d = 0, x_{sc} = 0, x_e = 0, x_{sd} = 0 \ \rightarrow \\
 & \quad x_i' = x_i - 1, x_d' = x_d + 1. \\
(wm_2) & x_i \geq 1, x_d + x_{sc} + x_e + x_{sd} \geq 1 \ \rightarrow \\
 & \quad x_i' = x_i - 1, x_{sc}' = x_{sc} + x_e + x_d + x_{sd}, \\
 & \quad x_e' = 0, x_d' = 0, x_{sd}' = 1. \\
(wh_1) & x_d \geq 1 \ \rightarrow \ skip. \\
(wh_2) & x_e \geq 1 \ \rightarrow \ x_e' = x_e - 1, x_d' = x_d + 1. \\
(wh_3) & x_{sd} = 1, x_{sc} = 0 \ \rightarrow \ x_{sd}' = 0, x_d' = x_d + 1. \\
(wh_4) & x_{sd} = 0, x_{sc} = 1 \ \rightarrow \ x_{sc}' = 0, x_d' = x_d + 1. \\
(wh_5) & x_{sd} + x_{sc} \geq 2 \ \rightarrow \ x_{sc}' = x_{sc} + x_{sd} - 1, x_{sd}' = 1.
\end{array}
$$

Figure 17. EFSM for the Dragon protocol.

The EFSM associated with the Dragon protocol is shown in figure 17. The counters x_d, x_{sd}, x_{sc}, x_e, and x_i are associated with the states *dirty*, *shared dirty*, *shared clean*, *exclusive*, and *invalid*, respectively. Rule (rh) denotes *read hits*. Rule (rm_1), and (rm_2) denote a *read miss* in presence or not of the snoop hit signal. Rules (wm_1) and (wm_2) denote write miss in presence or not of the snoop hit signal. Finally, rules (wh_1) through (wh_5) denote write hits.

In the Dragon protocol there are several possible sources of data inconsistency:

UNS$_1$, a *dirty* cache co-exists with one or more caches either in state *shared_dirty*, *shared_clean* or *valid_exclusive*;
UNS$_2$, an *valid_exclusive* cache co-exists with one or more caches either in state *shared_clean*, or *shared_dirty*;
UNS$_3$, there is more than one *dirty* cache;
UNS$_4$, there is more than one *valid_exclusive* cache.

The parameterized initial configuration is expressed as $\Phi_o = x_i \geq 1,\ x_e = 0,\ x_{sc} = 0,\ x_{sd} = 0,\ x_d = 0$. The potentially *unsafe* states are represented as follows.

UNS$_1$ $x_e + x_{sc} + x_{sd} \geq 1,\ x_d \geq 1$
UNS$_2$ $x_e \geq 1,\ x_{sc} + x_{sd} \geq 1$
UNS$_3$ $x_d \geq 2$
UNS$_4$ $x_e \geq 2$

9. Experimental results

In order to automatically solve the verification problems described in the previous sections, we implemented our methodology using the tool HyTech [28]. HyTech is an infinite-state

model checker based on the *polyhedra manipulation library* of [38]. In our experiments we employed the HYTECH command *reach backward* to compute the predecessors of a set of violations represented via linear constraints. HYTECH also provides operations to check if the initial states are contained in the resulting set of linear arithmetic constraints. Finally, it

Table 2. Experimental results.

Protocol	Unsafe states	Time	Steps	Verified
Synapse	1	0.44 s	1	yes
	2	0.45 s	1	yes
	1–2	0.55 s	1	yes
MESI	1	0.79 s	2	yes
	2	0.91 s	3	yes
	3	0.63 s	1	yes
	4	0.57 s	1	yes
	1–4	1.30 s	2	yes
MOESI	1	0.96 s	2	yes
	2	0.63 s	1	yes
	3	0.92 s	3	yes
	4	0.57 s	1	yes
	1–4	1.58 s	2	yes
Berkeley	1	0.53 s	1	yes
	2	0.68 s	2	yes
	1–2	0.65 s	1	yes
Illinois	1	0.80 s	3	yes
	2	1.87 s	4	yes
	3	0.54 s	1	yes
	4	0.63	1	yes
	1–4	1.25 s	2	yes
Firefly	1	0.90 s	2	yes
	2	↗	↗	yes
	3	↗	↗	yes
	4	0.65 s	1	yes
	1–4	1.56 s	2	yes
Dragon	1	1.08 s	2	yes
	2	0.68 s	1	yes
	3	↗	↗	yes
	4	0.62 s	1	yes
	1–4	1.93 s	2	yes

Time indicates the CPU execution time on a Sun-SPARCstation-5 OS 5.6.
Steps indicates the number of iterations before a fixpoint is reached.
↗ indicates that symbolic backward search does not terminate.

provides operations to extract *error-traces* when a violation is detected. This feature turned out to be very important during the modelling phase.

In Table 2 we list the experimental results obtained with HYTECH on the verification problems described in Section 8. In Table 2 for every protocol of Section 8 we use the following terminology: property i denotes the set of unsafe states UNS_i, i.e., the complement of an invariant property; property 1-n denotes the disjunction (union) of the constraints representing unsafe states $\text{UNS}_1 \vee \cdots \vee \text{UNS}_n$, i.e., the complement of the conjunction of the corresponding invariants.

The analysis performed by HYTECH does not terminate in three problems (all outside the decidable fragment) out of the 31 taken into considerations. However, HYTECH always terminates by taking as unsafe states the *disjunction* of the constraints representing the negation of all invariants (last line of the experiments associated with each protocol in Table 2). By weakening the set of unsafe states, we perform a sort of static *invariant strengthening*. To illustrate this idea, let U_1 and U_2 be the set of constraints that represent the violations of the safety properties P_1 and P_2, respectively. Starting from $U_1 \cup U_2$, if backward analysis terminates and the initial states are not in the resulting set of states, then both invariants hold. Using this technique, we managed to automatically verify *all* the safety properties listed in Section 8 for the considered cache coherence protocols.

We have also tried the alternative verification strategies described below, without obtaining however interesting results:

- *Forward* analysis using *parameter* variables (to represent the initial configurations) does not terminate in several examples.
- Approximations based on the *convex hull* (using the built-in *hull* operator applied to intermediate collections of states) returned no interesting results.
- We have also experimented other type of abstractions. Specifically, we have analyzed the EFSMs obtained by weakening the guards of the original descriptions (e.g. turning tests for zero into inequalities) so as to obtain EFSMs for which our procedure always terminates. Unfortunately, with this abstraction we found *errors* that were not present in the original protocol (the resulting reachable states are a super-set of those of the Illinois protocol).

In [15] we studies a subset of the verification problem listed in figure 9 using both HYTECH and the DMC model checker implemented in CLP(Q,R) [18]. Although less efficient, DMC implements domain-specific widening operators that can be used to enforce termination on some of the problems for which HYTECH diverges (see [15]).

10. Conclusions

In this paper we have proposed a new method for the automated parameterized verification of coherence protocols. We have applied our methods to successfully verify safety properties of several protocols taken from the literature [5, 27]. This result is obtained using technology originally developed for the verification of hybrid and concurrent systems.

In our approach we propose the following *abstractions*. We count the number of caches in every possible protocol state. The abstract protocol can be described as infinite-state system

with integer data variables. We relax the constraint operations needed to implement the symbolic backward reachability procedure for the resulting integer system. The abstraction based on the relaxation often gives *accurate* results and allows us to prove all the properties of the examples we were interested in.

As shown in [20], previous approaches based on abstractions like the "0, 1, many" of [42] can be viewed as a specialization of more general approaches based on forward analysis (namely, the covering tree procedure for Petri Nets). In this work we made a further step showing that by using a symbolic representation via linear constraints and by using backward analysis instead of forward, we obtain a natural way of dealing with systems with an arbitrary number of processes (i.e. in several practical cases we do not need ad hoc accelerations as the one used in the [42] approach). Furthermore, we have shown that we can apply a tool like HYTECH to verify several interesting properties of cache coherence protocols without need of any specialized abstraction to be performed using the analysis. As shown by other recent works inspired by this connection (e.g. [16, 19]), we believe that this is a valuable contribution to move forward the research in the automated verification of infinite-state systems.

Acknowledgments

The author would like to thank Parosh Aziz Abdulla, Javier Esparza, Kedar Namjoshi, Howard Wong-Toi, and James Larus for encouragement and fruitful discussions, Lenore Zuck, Edmund Clarke, and the anonymous reviewers for comments and suggestions that helped improving the presentation of this work.

References

1. P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson, "Handling global conditions in parameterized system verification," in N. Halbwachs and D. Peled (Eds.), *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, *Lecture Notes in Computer Science*, Trento, Italy, 1999, Vol. 1633, pp. 134–145.
2. P.A. Abdulla, K. Cerãns, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *Proceedings of the 11th Annual International Symposium on Logic in Computer Science (LICS '96)*, New Brunswick, New Jersey, 1996, pp. 313–321.
3. P.A. Abdulla and B. Jonsson, "Ensuring completeness of symbolic verification methods for infinite-state systems," *Theoretical Computer Science*, Vol. 256, Nos. 1/2, pp. 145–167, 2001.
4. P.A. Abdulla and A. Nylén, "Better is better than well: On efficient verification of infinite-state systems," in *Proceedings of the 15th Annual International Symposium on Logic in Computer Science (LICS '00)*, Santa Barbara, California, 2000, pp. 132–140.
5. P.A. Archibald and J. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems*, Vol. 4, No. 4, pp. 273–298, 1986.
6. T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L.D. Zuck, "Parameterized verification with automatically computed inductive assertions," in G. Berry, H. Comon, and A. Finkel (Eds.), *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, *Lecture Notes in Computer Science*, Paris, France, 2001, Vol. 2102, pp. 221–234.
7. J.-P. Bodeveix and M. Filali, "FMona: A tool for expressing validation techniques over infinite state systems," in *Proceedings of the 6th Int. Con. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, *Lecture Notes in Computer Science*, Berlin, Germany, 2000, Vol. 1785, pp. 204–218.

8. B. Boigelot and P. Wolper, "Verifying systems with infinite but regular state space," in *Proceedings of the 10th Conf. on Computer Aided Verification (CAV '98)*, *Lecture Notes in Computer Science*, 1998, Vol. 1427, pp. 88–97.
9. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, "Regular model checking," in E.A. Emerson and A.P. Sistla (Eds.), *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, *Lecture Notes in Computer Science*, Chicago, Illinois, 2000, Vol. 1855, pp. 403–418.
10. M.C. Browne, E.M. Clarke, and O. Grumberg, "Reasoning about networks with many identical finite state processes," *Information and Computation*, Vol. 81, No. 1, pp. 13–31, 1989.
11. T. Bultan, R. Gerber, and W. Pugh, "Symbolic model checking of infinite state systems using presburger arithmetics," in O.Grumberg (Ed.), *Proceedings 9th International Conference on Computer Aided Verification (CAV '97)*, *Lecture Notes in Computer Science*, Haifa, Israel, 1997, Vol. 1254, pp. 400–411.
12. K.-T. Cheng and A.S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 1, No. 1, pp. 57–79, 1996.
13. E. Clarke, O. Grumberg, and S. Jha, "Verifying parameterized networks," *TOPLAS*, Vol. 19, No. 5, pp. 726–750, 1997.
14. E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness, "Verification of the futurebus + cache coherence protocol," in L.J.M.C.D. Agnew and R. Camposano (Eds.), *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications (CHDL '93), sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC*, Ottawa, Ontario, Canada, 1993, pp. 15–30.
15. G. Delzanno, "Automatic verification of parameterized cache coherence protocols," in E.A. Emerson and A.P. Sistla (Eds.), *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, *Lecture Notes in Computer Science*, Chicago, Illinois, Vol. 1855, 2000.
16. G. Delzanno and T. Bultan, "Constraint-based verification of client-server protocols," in T. Walsh (Ed.), *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP '01)*, *Lecture Notes in Computer Science*, Paphos, Cyprus, 2001, Vol. 2239, pp. 286–301.
17. G. Delzanno, J. Esparza, and A. Podelski, "Constraint-based analysis of broadcast protocols," in J. Flum and M. Rodríguez-Artalejo (Eds.), *Proceedings of the 1999 Annual Conference of the European Association for Computer Science Logic (CSL '99)*, *Lecture Notes in Computer Science*, Madrid, Spain, pp. 50–66, 1999, Vol. 1683.
18. G. Delzanno and A. Podelski, "Model checking in CLP," in R. Cleaveland (Ed.), *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*, *Lecture Notes in Computer Science*, Amsterdam, The Netherlands, 1999, Vol. 1579, pp. 223–239.
19. G. Delzanno, J.-F. Raskin, and L.V. Begin, "Attacking symbolic state explosion," in G. Berry, H. Comon, and A. Finkel (Eds.), *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, *Lecture Notes in Computer Science*, Paris, France, 2001, Vol. 2102, pp. 298–310.
20. E. Emerson and K. Namjoshi, "On model checking for non-deterministic infinite-state systems," in *Proceedings of the 13th International Symposium on Logic in Computer Science (LICS '98)*, Indianapolis, Indiana, 1998, pp. 70–80.
21. J. Esparza, A. Finkel, and R. Mayr, "On the verification of broadcast protocols," in *Proceedings 14th International Symposium on Logic in Computer Science (LICS '99)*, Trento, Italy, 1999, pp. 352–359.
22. A. Finkel and P. Schnoebelen, "Well-structured transition systems everywhere," *Theoretical Computer Science*, Vol. 256, Nos. 1/2, pp. 63–92, 2001.
23. S.M. German and A.P. Sistla, "Reasoning about systems with many processes," *Journal of the ACM*, Vol. 39, No. 3, pp. 675–735, 1992.
24. S. Graf, "Characterization of a sequentially consistent memory and verification of a cache memory by abstraction," *Distributed Computing*, Vol. 12, Nos. 2/3, pp. 75–90, 1999.
25. S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in O. Grumberg (Ed.), *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, *Lecture Notes in Computer Science*, Haifa, Israel, 1997, Vol. 1254, pp. 72–83.

26. N. Halbwachs, "Delay analysis in synchronous programs," in C. Courcoubetis (Ed.), *Proceedings of the 5th Conference on Computer-Aided Verification (CAV'93)*, *Lecture Notes in Computer Science*, Elounda, Greece, 1993, Vol. 697, pp. 333–346.
27. J. Handy, *The Cache Memory Book*, Academic Press, 1993.
28. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," in O. Grumberg (Ed.), *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, *Lecture Notes in Computer Science*, Springer, Haifa, Israel, 1997, Vol. 1254, pp. 460–463.
29. T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Verifying sequential consistency on shared-memory multiprocessor systems," in N. Halbwachs and D. Peled (Eds.), *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, *Lecture Notes in Computer Science*. Trento, Italy: Springer, 1999, Vol. 1633, pp. 301–315.
30. R.M. Karp and R.E. Miller, "Parallel program schemata," *Journal of Computer and System Sciences*, Vol. 3, No. 2, pp. 147–195, 1969.
31. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar, "Symbolic model checking with rich assertional languages," in O. Grumberg (Ed.), *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, *Lecture Notes in Computer Science*, Haifa, Israel, 1997, Vol. 1254, pp. 424–435.
32. D. Lesens, N. Halbwachs, and P. Raymond, "Automatic verification of parameterized linear networks of processes," in *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, 1997, pp. 346–357.
33. B.D. Lubachevsky, "An approach to automating the verification of compact parallel coordination programs," *Acta Informatica*, Vol. 21, pp. 125–169, 1984.
34. M.S. Papamarcos, J.H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of 11th Annual Symposium on Computer Architecture (ISCA'84)*, Ann Arbor, USA, 1984, pp. 348–354.
35. K. McAloon, "Petri nets and large finite sets," *Theoretical Computer Science*, Vol. 32, pp. 173–183, 1984.
36. K. McMillan, "Verification of infinite state systems by compositional model checking," in L. Pierre and T. Kropf (Eds.), *Proceedings of 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, 1999, Vol. 1703, pp. 219–234,
37. K. McMillan and J. Schwalbe, "Verification of infinite state systems by compositional model checking," in *Proc. Int. Symp. on Shared Memory Multiprocessors*, pp. 242–251, 1991.
38. N. Halbwachs, Y.-E. Proy, P. Raymond, "Verification of real-time systems using linear relation analysis," *Formal Methods in System Design*, Vol. 11, No. 2, pp. 157–185, 1997.
39. C. Norris Ip and D.L. Dill, "Verifying systems with replicated components in murphi," *Formal Methods in System Design*, Vol. 14, No. 3, pp. 273–310, 1999.
40. A. Pnueli, S. Ruah, and L.D. Zuck, "Automatic deductive verification with invisible invariants," in T. Margaria and W. Yi (Eds.), *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '01)*, *Lecture Notes in Computer Science*, Genova, Italy, 2001, Vol. 2031, pp. 82–97.
41. A. Pnueli and E. Shahar, "Liveness and acceleration in parameterized verification," in E.A. Emerson and A.P. Sistla (Eds.), *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, *Lecture Notes in Computer Science*, Chicago, Illinois, 2000, Vol. 1855, pp. 328–343.
42. F. Pong and M. Dubois, "A new approach for the verification of cache coherence protocols," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 8, 1995.
43. F. Pong and M. Dubois, "Verification techniques for cache coherence protocols," *ACM Computing Surveys*, Vol. 29, No. 1, pp. 82–126, 1997.
44. A. Schrijver, *Theory of Linear and Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley and Sons, 1998.