VERIFICATION OF CONSISTENCY PROTOCOLS VIA INFINITE-STATE SYMBOLIC MODEL CHECKING

A Case Study: the IEEE Futurebus+ Protocol

Giorgio Delzanno
DISI - University of Genova
via Dodecaneso 35, 16146 Italy
giorgio@disi.unige.it

Abstract We apply infinite-state model checking to verify safety properties of a *parameterized* formulation of the IEEE Futurebus+ coherence protocol modeled at the behavior level in a system with split transaction. This case-study shows that verification techniques previously applied to hybrid and real-time systems can be used as tools for validating parameterized protocols. This technology transfer is achieved by combining abstraction techniques, symbolic representation via constraints, efficient operations based on real arithmetics, and reachability algorithms. To our knowledge this is the first time that safety properties for a parameterized version of the Futurebus+ protocol has been automatically verified.

Keywords: Parameterized Consistency Protocols, Symbolic Model Checking

1. INTRODUCTION

In a multiprocessor system with local caches, coherence protocols ensure the consistency of the data stored in main memory and in the caches during the execution of a program [17]. For a fixed number of caches, these protocols can be described as finite-state systems obtained by composing the specifications of individual caches [23]. 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 in accord with write and read commands issued by the corresponding CPU or coming from the system bus. Automatic methods based on finite-state symbolic model checking with BDDs have been successfully applied to the verification (and debugging) of consis-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35533-7_26

Tommaso Bolognesi and Diego Latella (Eds.), *Formal Methods for Distributed System Development.*
© 2000 IFIP International Federation for Information Processing.
Published by Kluwer Academic Publishers. All rights reserved.

tency protocols for systems with a fixed number of processors [6, 22]. Unfortunately, proving the correctness of the protocol for few special cases does not guarantee its correctness for *any* number of processors.

In recent years new techniques that extend the applicability of symbolic model checking to infinite-state systems have been used to attack the problem of verification of *parameterized* formulations of concurrent systems [1], and communication protocols [15, 12, 13, 11]. Based on these ideas, in this paper we apply infinite-state symbolic model checking methods working over real arithmetics to verify a parameterized (where the parameter=number of processors) version of the IEEE Futurebus+ cache coherence protocol specified at the *behavior* level for a single-bus multiprocessor system with split-transaction.

Our work is based on the description of the Futurebus+ coherence protocol given by Jim Handy in *The Cache Book* [17]. Following [23], we consider caches with a single line, and we abstract from the data read from and written into memories and buffers. Despite these limitations, we still model the special features of the protocol like the conditions based on the *transaction flag* used to give special meaning to read operations. The behavior level of the caches will be described via the four states *invalid*, *shared unmodified*, *exclusive modified*, and *exclusive modified*, and we will model split-transaction via non-atomic synchronization actions. For this purpose, we need to introduce auxiliary states that represent the behavior of caches when pending read and write commands occur.

Our verification methodology is based on the use of *abstraction*, *constraints*, and *backward* reachability analysis. Let us discuss all these aspects more in detail. The state of a cache ranges over a *finite* (and usually small) set of possible local states. Then, we can apply the following *abstraction* from global states (global state=collection of local cache states) to tuples of natural numbers: we only keep track of the *number* of caches in every possible local state. This abstraction allows us to cluster together all *symmetric* states. The resulting abstract protocol can be represented as a transition system with data variables ranging over positive integers formally described as an Extended Finite State Machine (EFSM) [5]. EFSM-transitions are linear transformations where global and local conditions are expressed as guards (=linear constraints). Note that in the abstract protocol the only fixed dimension is the number of local states (not the number of processors), i.e., the EFSM-model gives us a parameterized formulation of the original protocol. Parameterized initial and unsafe configurations can be represented as linear constraints as well. This way, we reduce the problem of checking safety properties for parameterized protocols to a *reachability* problem for a

system with integer data variables. To solve the reachability problem, we adopt the backward reachability algorithm of [2]. Starting from the set of (abstract) *unsafe* states we compute the closure of the *predecessor* operator associated to the abstract protocol (an EFSM). To make this algorithm effective, we use a symbolic representation of potentially infinite sets of EFSM-states (i.e. tuples of natural numbers) via linear constraints. This way, we represent compactly sets of global states of the original (family of) protocols. As a last ingredient of our recipe, in order to obtain efficient constraint operations we apply a relaxation from integers to reals when testing satisfiability, checking entailment, and eliminating existentially quantified variables (the operations used in the symbolic reachability algorithm). This technique is widely used in program analysis and verification [8, 19]. The relaxation introduces a further abstraction that, however, is applied only during the execution of the backward reachability analysis. The combination of all of these ingredients leads us to the following *technology transfer*: tools devised for the automatic verification of hybrid, concurrent and real-time systems like HyTech [18], Polka [19], and DMC [10], can now be applied to check *temporal properties* of consistency protocols. In order to enhance the underlying verification algorithms based on symbolic fixpoint computations, some of these tools implement further approximation techniques like widening and acceleration operators [19, 10].

As practical results, we have automatically verified that several safety properties for the Futurebus+ protocol hold for any number of processors. An example of safety property we have considered is *different caches can not be simultaneously in exclusive states.* In our experiments we have used the infinite-state symbolic model checkers DMC [10] (based on a constraint solver over reals) and HyTech [18] (based on Halbwachs' polyhedra library). To our knowledge, this is the first time that a parameterized version of (an abstraction of) the Futurebus+ protocol has been automatically verified. Furthermore, the use of infinite-state symbolic model checking tools working over real arithmetics seems to be a novel approach to attack the verification of consistency protocols with split-transaction bus structure. We have used the same methodology to validate many other cache coherence protocols taken from the literature [9].

Related Works. Finite-state symbolic model checking has been successfully applied to the verification of consistency protocols for multiprocessor systems with a relatively small number of processors. In [6], Clarke et al. found several design errors in the IEEE specification of the Futurebus+ for a hierarchical bus system with up to 8 processors.

As in our approach, they consider single cache lines, one action as an abstraction of a complete transaction, non-determinism to simplify the management of read and write requests, no exception handling, and no *write-* and *read-invalid* commands. However, being in the finite-state case they can also distinguish processes. It would be interesting to investigate in the combination of BDDs and constraint-based symbolic model checking in order to handle more detailed parameterized models. In [23], Pong and Dubois apply a technique called Symbolic State Model to the verification of several coherence protocols formulated at the behavior level. However, they do not consider the Futurebus+ protocol. In [20], Norris-Ip and Dill have incorporated the abstractions (based on repetition operators) used in [23] in Murφ.

In [9], we have applied our methodology to analyze other existing protocols taken from the literature (see e.g. [17]). In none of the examples in [9], we have considered a split-transaction bus structure (we considered indeed only atomic synchronization actions). Our method is inspired to [15, 12, 13, 11] where the authors present decidability results for proper subclasses of the EFSMs we consider in this paper and relate them to verification problems of parameterized systems. Semi-automatic methods for parameterized system have been investigated in [4, 7, 22], whereas abstraction techniques for parameterized systems have been investigated in [16, 20, 21, 23]. In [2, 3, 10], constraints are used as symbolic representation of (potentially infinite) sets of states. Finally, relaxation techniques for handling linear constraints in program analysis and automatic verification have been investigated in [8, 10, 21, 19].

Plan of the paper. In Section 2, we give a formal model of the behavior level of the IEEE Futurebus+ Protocol in terms of the finite-state machines of Pong and Dubois [23]. In Section 3, we describe the abstract protocol obtained applying the abstraction discussed before. In Section 4, we apply our verification algorithm to abstract protocol. Finally, in Section 5 we address future directions of research.

2. THE IEEE FUTUREBUS+ PROTOCOL

In this paper we focus our attention on the open standard IEEE Futurebus+ coherence protocol as described by Jim Handy in *The Cache Book* [17]. Following [23], we make the following assumptions: we consider a single-bus multiprocessor system where each CPU has a local cache connected on the system bus, we consider caches with a single line (i.e. cache state=state of the cache line), and we abstract from the data read from and written into (cache) memory. Furthermore, following [6] we hide all handshaking necessary to issue a command, i.e., we model a

Current State	Message	To CPU	To BUS	Cond.	New State
invalid	READ		Read Shared	tf*	shared-U
				not tf*	exclusive-U
	WRITE		Read Modified		exclusive-U
sharedU exclusiveU exclusiveM	READ	Data			(same state)
sharedU	WRITE		Invalidate		exclusive-M
exclusiveU exclusiveM	WRITE				exclusive-M

Figure 1 CPU Cycles for Futurebus+

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. Futurebus+ uses a *split transaction* synchronous bus with *reflection*. In a split-transaction, a processor issues a command and then removes itself from the bus until the memory or a I/O device issues a response. In the sequel of the paper, we will introduce non-atomic actions to take into account split-transactions. When using reflection, memory is allowed to snoop cache-to-cache data transfers on the system bus. This way, it always keeps an updated copy of data. Furthermore, Futurebus+ specifies a *copy-back* coherency protocol that supports *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 on other transactions in order to grab copies of data. Snoop hit acknowledgments are broadcast from the snooped cached onto the bus via a signal called tf* (transaction flag), a bus signal whose definition depends on the type of transaction which is occurring on the bus. By monitoring a tf* signal, the requesting cache has the option of taking a line immediately into an *exclusive* state, if that line is copied in no other cache. The behavior level of the protocol is illustrated in the tables of Fig. 1 (CPU cycles) and Fig. 2 (Bus cycles) taken from [17]. Following [6], we don't consider here the *read-* and *write-invalid* commands used in DMA transfers to and from memory. A cache can be in one of the following states: *invalid*, the cache has no valid data; *exclusive unmodified* (exclusiveU), the cache has an exclusive copy of the data; *shared unmodified* (sharedU), the cache has a potentially shared copy of the data; *exclusive modified* (exclusiveM),

Current State	Message	To CPU	To BUS	New State
invalid	*any*			invalid
sharedU exclusiveU	Read Shared		assert tf*	shared-U
exclusiveM		Data	assert tf*	shared-U
sharedU exclusiveU	Read Modified			invalid
exclusiveM		Data		invalid
sharedU exclusiveU exclusiveM	Invalidate			invalid

Figure 2 Bus Cycles.

the cache has a modified copy of the data. Let us describe the protocol more in detail starting from the *read cycles.* On a CPU read miss (the cache line is in the *invalid* state), a Read Shared command is placed on the bus. If the snooped cache is in state *shared unmodified* or *exclusive unmodified* then it asserts the signal tf* and its state changes to *shared unmodified*; the data are supplied by main memory. If the snooped cache is in state *exclusive modified* then main memory is disabled, the modified (dirty) line is put on the bus, the snooped cache asserts tf* and its state changes to *shared unmodified.* Reflection ensures that main memory has a consistent copy of data at the end of each Read Shared cycle. If tf* has been asserted, the requesting cache changes its state to *shared unmodified*; otherwise, its state changes to *exclusive unmodified.* When the cache is in one of *shared*, *exclusive unmodified* or *exclusive modified*, read commands do not require any bus activity. On a CPU write miss, according to the write-allocation policy, a valid copy of the data must be loaded first into the cache and then merged with the new data. Modified lines are first copied back to main memory. The Read Modified command is then placed on the bus; the owner supplies the data and goes to *invalid*, while the copy of all other caches are invalidated. The requesting cache goes to *exclusive modified.* Write hits on *exclusive modified* or *unmodified* line do not require bus interaction: the data are written into the cache line, and the cache changes its state to *exclusive modified.* On write hits on *shared unmodified* lines, the other copies are invalidated through the Invalidate command. The requesting cache loads the data and changes its state to *exclusive modified.*

Safety properties. In this paper we limit ourselves to consider *safety* properties that are related to the *semantics* of the cache states. Specifically, we would like to prove that the protocol is free from the

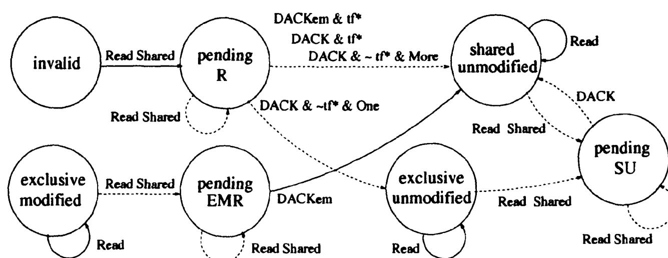

Figure 3 Read Cycles.

following sources of data inconsistency: (a) more than one cache is in an exclusive state; (b) different caches are simultaneously in an exclusive state and in the shared state. In the following section we will show that for a fixed number of caches these properties can be checked using reachability analysis over a finite-state system describing the global behavior of the protocol.

A Finite-state Model with Non-atomic Actions

In this section we use the finite-state model of Pong and Dubois [23] to formally specify the protocol. In the model of [23], the behavior of each cache is specified as a finite-state machine where additional global conditions (i.e. involving the state of other caches) are used as guards for transitions. The behavior of the global system with n caches is obtained composing the individual machines. In the global machine, transitions may require a synchronization of all caches on a given event (e.g. a bus invalidate broadcast message). To simplify the presentation of the formal model for our case-study we discuss separately read and write cycles (plus additional global conditions we will introduce later in this section). The complete model is obtained merging the transitions described at each step.

Read Cycles. In order to describe the read cycles of Futurebus+ taking into account split-transaction, in addition to the basic states *invalid*, *sharedU*, *exclusiveU*, and *exclusiveM*, we introduce the following new states: *pendingR* indicates that the cache has issued a Read Shared command and it is waiting for the data acknowledgement; *pendingEMR* indicates that a cache in *exclusive modified* state is ready to respond with the data to the Read Shared command; *pendingSU* indicates that caches either in *shared* or in *exclusive unmodified* state are ready to assert the tf* signal. The finite-state machine from the perspective of an

individual cache is depicted in Fig. 3. Solid edges labeled by an action ℓ from state s_1 to state s_2 indicate that one cache in state s_1 can perform the action ℓ and move to s_2. Dashed edges labeled with ℓ denote the actions of all other caches when the event ℓ occurs. Read hits do not cause state transitions as indicated by the loops in state *sharedU*, *exclusiveU*, and *exclusiveM*. When a cache issues the Read Shared command and goes to *pendingR*, all caches in state *sharedU* or *exclusiveU* go to *pendingSU* and prepare to issue the signal tf*; alternatively, the cache in state *exclusiveM* goes to *pendingEMR* and prepares to send a data acknowledgement (ready signal) (DACK_{em}). If the data acknowledgement (together with tf*) is sent by a cache in state *pendingEMR* (DACK_{em}) or by main memory (DACK) and tf* has been asserted then the caches with pending read go to *sharedU*. if tf* has not been asserted and only one cache has a pending read (this condition is indicated as *One* in Fig.3) than its state changes to *exclusiveU*. If tf* has not been asserted and more than one cache has pending read requests (this condition is indicated as *More* in Fig.3) than all caches move to *sharedU* (note in fact that if one cache first moves to *exclusiveU* then it will immediately move to *sharedU* as soon as the other cache will receive an acknowledgment and move to *sharedU*). We model the test for tf* as the following global conditions:

tf* $\equiv$ there exists a cache in state *pendingEMR* or *pendingSU*.

Write Cycles. In order to model the write miss cycle we introduce other two states. *pendingW* indicates caches that have issued a Read Modified command and wait for a data acknowledgment. *pendingEMW* indicates that a cache in state *exclusiveM* is preparing to issue the data acknowledgement. The Read Modified cycle from the perspective of an individual cache is depicted in Fig. 4. When the cache issues a Read Modified command and goes to *pendingW*, the cache in state *exclusiveM* goes to *pendingEMW*, while all other caches go to *invalid*. When a data acknowledgment is issued than the cache goes to *exclusiveM*.

Let us consider for a moment the *global* machine obtained fixing the number n of caches and merging the read and write miss cycles described above. In the previous model we allow both multiple pending read and write commands. However, it is easy to check that this model gives rise immediately to data inconsistencies where, e.g., different caches are in state *exclusiveM* and *sharedU*. For instance, let $n = 3$ and let $G_0 = \langle invalid, invalid, invalid \rangle$ be the initial state. If the first cache issues a Read Modified before the second one issues a Read Shared the system ends up in the global state $G_1 = \langle pendingW, pendingR, invalid \rangle$. An acknowledgment from memory will force the two caches simultaneously

Figure 4 Write cycles.

to the unsafe state $G_2 = \langle exclusiveM, exclusiveU, invalid\rangle$. To avoid this situation we add the following global condition on the transitions Read Shared and Read Modified in state *invalid.*

> A cache can issue a Read Shared or a Read Modified command only if there are no pending writes.

Finally, as shown in Fig. 4 write hits is in state *exclusiveU* and *exclusiveM* need no consistency actions. On the other hand, when the cache is in state *sharedU* all other caches in state *sharedU* are forced to the state *invalid* via the Invalidate signal.

The finite-state model obtained fixing the number of processors can be used to test the safety properties described in Section 2 via finite-state symbolic model checking [6]. However, we would like to check the properties for any number of caches. We attack the problem in the following section.

3. FROM PROTOCOLS TO EFSMs

In order to check parameterized safety properties we apply the following *abstraction.* Let Q be the finite set of states of a cache. Then,

> we keep track only of the *number* of caches in every state $q \in Q$.

Via this abstraction, a global state G with n components (n=number of processes) is mapped to a tuple of *positive integers* with k components where (k=number of cache states). Consider for instance our case-study with three caches. The states $\langle invalid, invalid, exclusiveU\rangle$, $\langle invalid, exclusiveU, invalid\rangle$, and $\langle exclusiveU, invalid, invalid\rangle$ are represented by the abstract state consisting of a tuple with 9 counters; the counter for *invalid* is $= 2$, that for *exclusiveU* is $= 1$ and all the others

are $= 0$. Using this abstraction, we cannot prove properties of individual caches like 'cache i and cache j cannot be simultaneously in a *dirty* state'. However, we can still try to prove global properties like 'two different caches cannot be simultaneously in a *dirty* state'. This is the kind of properties we are interested in to prove that the protocol will not give inconsistent (wrt. the semantics of states) results.

Abstract Protocol. 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 to the states in Q. For this purpose, we (automatically) *compile* the finite-state model into an 'abstract protocol' defined as an *Extended Finite State Machine* (EFSM) [5], i.e., a finite automaton with data variables (ranging over integers) associated to the locations and with guarded linear transformations associated to the transitions. Formally, let $Q = \{s_1, \ldots, s_n\}$. The abstract protocol is an EFSM with one location (omitted for simplicity) and n data variables $\langle x_1, \ldots, x_n \rangle$ (denoted as $\mathbf{x}$) ranging over *positive integers.* An EFSM-state is a tuple of natural numbers $\langle c_1, \ldots, c_n \rangle$ (=$\mathbf{c}$) where c_i denotes the number of caches in state $s_i \in Q$. Transitions are represented via a collection of *guarded linear transformations* defined over the vector of variables $\langle x_1, \ldots, x_n \rangle$ and $\langle x'_1, \ldots, x'_n \rangle$ where x_i and x'_i denote the *number of caches* in state s_i, respectively, before and after the occurrence of an event. More precisely, transitions have the following form $G(\mathbf{x}) \rightarrow T(\mathbf{x}, \mathbf{x}')$, where $G(\mathbf{x})$ is the *guard* and $T(\mathbf{x}, \mathbf{x}')$ is the *transformation.* The guard $G(\mathbf{x})$ is an arbitrary *linear constraint.* A linear constraint is a conjunction (written $\varphi_1, \ldots, \varphi_n$) of atomic predicates φ_i built over the relation symbols $=, <, >, \leq, \geq$ and on linear expressions over $\mathbf{x}$. For instance, the global condition *all caches are in state 'invalid'* can be expressed as the guard $x_1 \geq 1, x_2 = 0, \ldots, x_n = 0$ if x_1 is the counter for *invalid.* The transformation $T(\mathbf{x}, \mathbf{x}')$ is defined as $\mathbf{x}' = M \cdot \mathbf{x} + \mathbf{c}$ where M is an $n \times n$-matrix with unit vectors as columns. This way, we can represent the changes of states of the caches in the system (including the invalidation signals). Since the number of caches is an invariant of the system, we require the transformation to satisfy the condition $x'_1 + \ldots + x'_n = x_1 + \ldots + x_n$. A run of an EFSM is a (possibly infinite) sequence of EFSM-states $\mathbf{c_1}, \ldots, \mathbf{c_i} \ldots$ where $G_r(\mathbf{c_i}) \wedge T_r(\mathbf{c_i}, \mathbf{c_{i+1}}) = true$ for some transition $G_r \rightarrow T_r$; $\mathbf{c} \rightarrow^* \mathbf{c}'$ denotes the existence of a run starting from $\mathbf{c}$ and passing through $\mathbf{c}'$. The *predecessor* operator *pre* defined over a *set* S of EFSM-states as $pre(S) = \{\mathbf{c} \mid \mathbf{c} \rightarrow \mathbf{c}',\ \mathbf{c}' \in S\}$, where $\rightarrow$ indicates a one-step EFSM state transition.

The EFSM for the Futurebus+ Protocol. In this section *invalid, sharedU, exclusiveU, exclusiveM, pendingR, pendingW, pending-EMR, pendingEMW*, and *pendingSU* will denote the counters (integer variables) associated to the cache states. Furthermore, we will omit the equalities of the form $x = x'$ from $T(\mathbf{x}, \mathbf{x}')$ ($T(\mathbf{x}, \mathbf{x}') = _$ will indicate a transformation where all variables remain unchanged). Finally, we use $x_1 = x_2 = \ldots = x_n = c$ as an abbreviation of the conjunction $x_1 = c, x_2 = c, \ldots, x_n = c$.

The read cycle of Fig. 3 is described by the EFSM-transitions of Fig. 5. Rule r_1 represents read hits (no state change). Rule r_2 occurs when a cache issues a Read Shared command (i.e. a cache moves from *invalid* to *pendingR*) with the proviso that there are no pending writes (according to the global condition we discussed in the previous section). The remaining caches change state simultaneously as indicated, e.g., by $pendingSU' = pendingSU + sharedU + exclusiveU$ (all caches in *exclusive* and *shared unmodified* move to *pendingSU*). Rule r_3 occurs when the data are supplied by another cache (in state *pendingEMR*). Rule r_4 occurs when the data are supplied by main memory and tf* has been asserted: all caches with pending read go to *sharedU*. Rule r_5 and r_6 occur when the data are supplied by main memory and tf* has not been asserted. In case there is only one pending read the cache goes to *exclusiveU* r_6. The write cycles of Fig. 4 are described by the EFSM-transitions of Fig. 6. Rule w_1 occurs when a cache issues a Read Modified command. The Invalidate command is modeled enforcing all caches to *invalid*. As for Read Shared we require that there are no pending writes (alternatively, existing write cycles could be aborted). Rule w_2 occurs when the data are supplied by a cache in state *pendingEMW*. Rule w_3 occurs when the data are supplied by main memory. Rule w_4 occurs when a write is issued on a cache in state *exclusiveM* (no state change). Rule w_5 occurs when a write is issued on a cache in state *exclusiveU*. Rule w_6 occurs when a write is issued on a cache in state *sharedU*.

4. VERIFICATION AS REACHABILITY

For every run of the global machine associated to a protocol there exists a run in the EFSM obtained applying our abstraction. Vice versa, every run of the EFSM corresponds to a set of possible runs of the original protocol. This property (proved in [9]) allows us to reduce a family of reachability problem for a given protocol (indexed on the number of processors) to a reachability problem for the corresponding EFSM. We note that we can use *linear constraints* to represent concisely (possibly

(r_1) $sharedU + exclusiveU + exclusiveM \geq 1 \longrightarrow _.$

(r_2) $invalid \geq 1, pendingW = 0 \rightarrow$
$invalid' = invalid - 1,\ pendingR' = pendingR + 1,$
$pendingEMR' = pendingEMR + exclusiveM,$
$pendingSU' = pendingSU + sharedU + exclusiveU,$
$sharedU' = exclusiveU' = exclusiveM' = 0.$

(r_3) $pendingEMR \geq 1 \longrightarrow$
$pendingEMR' = pendingEMR - 1,\ pendingR' = 0,$
$sharedU' = sharedU + pendingR + 1.$

(r_4) $pendingSU \geq 1 \longrightarrow$
$sharedU' = sharedU + pendingR + pendingSU,$
$pendingR' = pendingSU' = 0.$

(r_5) $pendingR \geq 2, pendingSU = 0,\ pendingEMR = 0 \longrightarrow$
$sharedU' = sharedU + pendingR,\ pendingR' = 0.$

(r_6) $pendingR = 1, pendingSU = 0, pendingEMR = 0 \longrightarrow$
$pendingR' = pendingR - 1,\ exclusiveU' = exclusiveU + 1.$

Figure 5 Read Cycles.

infinite) sets of EFSM-states (hence, indirectly, sets of global states independently from the number of caches in the system). Furthermore, this class of constraints is powerful enough to express *initial* and *target* sets of states for the verification problems we are interested in.

Symbolic Representation via Constraints. In this section we will use the lower-case letters $\varphi, \psi, \ldots$ to denote linear constraints and the upper-case letter $\Psi, \Phi, \ldots$ to denote *sets* (disjunctions) of constraints. The *denotation* of a constraint φ is defined as $[\![\varphi]\!]$ is the set of *solutions* (tuples of positive integers) that satisfy φ. The denotation of a set of constraints is the union of the denotation of its components. Furthermore, we say that a constraint ψ *entails* a constraint φ, written $\varphi \sqsubseteq \psi$, iff $[\![\psi]\!] \subseteq [\![\varphi]\!]$. In order to reduce the complexity of the manipulation of arithmetic constraints, we follow techniques used in program analysis [8]. Specifically, we interpret the *satisfiability* test, *variable elimination*, and *entailment* test needed to implement the symbolic reachability algorithm of [2] over the domain of *reals*. The relaxation allows us to exploit efficient (polynomial) operations over the reals in contrast to (worst-case) exponential operations over the integers. Formally, given a constraint φ, we define $[\![\varphi]\!]_{\mathbb{R}}$ as the set of *real* solutions of φ. The entailment relation over $\mathbb{R}_+$ is defined then as $\varphi \sqsubseteq_{\mathbb{R}} \psi$ if and only if $[\![\psi]\!]_{\mathbb{R}} \subseteq [\![\varphi]\!]_{\mathbb{R}}$.

(w_1) $invalid \geq 1, pendingW = 0 \longrightarrow$
$pendingW' = pendingW + 1,$
$pendingEMW' = pendingEMW + exclusiveM,$
$sharedU' = exclusiveU' = exclusiveM' = 0,$
$pendingR' = pendingEMR' = pendingSU' = 0,$
$invalid' = invalid + exclusiveU + sharedU + pendingSU +$
$+ pendingR + pendingEMR - 1.$

(w_2) $pendingEMW \geq 1 \longrightarrow$
$pendingEMW' = pendingEMW - 1, \; invalid' = invalid + 1,$
$exclusiveM' = exclusiveM + pendingW, pendingW' = 0.$

(w_3) $pendingEMW = 0 \longrightarrow$
$exclusiveM' = exclusiveM + pendingW, \; pendingW' = 0.$

(w_4) $exclusiveM \geq 1 \longrightarrow _.$

(w_5) $exclusiveU \geq 1 \longrightarrow$
$exclusiveU' = exclusiveU - 1, \; exclusiveM' = exclusiveM + 1.$

(w_6) $sharedU \geq 1 \longrightarrow$
$invalid' = invalid + sharedU - 1,$
$exclusiveM' = exclusiveM + 1, \; sharedU' = 0.$

Figure 6 Write Cycles.

We apply the above relaxation to define a *symbolic predecessor operator*, **sym_pre**$_{\mathbb{R}}$ defined as follow. Let $\varphi(\mathbf{x}')$ be a constraint with variables over $\mathbf{x}'$, $\mathbf{sym_pre}_{\mathbb{R}}(\varphi(\mathbf{x}')) = \bigvee_{i \in I} \exists \mathbf{x}' \in \mathbb{R}_+ . \, G_i(\mathbf{x}) \wedge T_i(\mathbf{x}, \mathbf{x}') \wedge \varphi(\mathbf{x}')$, where $G_i(\mathbf{x}) \to T_i(\mathbf{x}, \mathbf{x}')$ is an EFSM transition rule for $i \in I$ (I=index set).

Algorithm for Symbolic Reachability. The symbolic reachability algorithm [2] works as follows. Let Φ_f be a set of constraint representing the *unsafe* states, and let $\Phi_0 = \Phi_f$. At step i, we apply the operator **sym_pre**$_{\mathbb{R}}$ to every constraint in Φ_i. A newly computed constraint ψ is added to Φ_i if and only if there exist no $\varphi \in \Phi_i$ such that $\varphi \sqsubseteq_{\mathbb{R}} \psi$. If we cannot add new constraints, the algorithm terminates. If the algorithm terminates at step k, the solutions of the constraints in Φ_k represent a *superset* of the set of states that are backward reachable from Φ_f. In fact, using the relaxation integer-reals we may compute an over-approximations of the integer solutions of a linear constraint. As a consequence, if the initial set of states is not in $[\![\Phi_k]\!]$, then the original protocol is *safe for any number of caches.* Otherwise, we must return a *don't know* answer. To check the last condition, we simply conjunct the constraint representing the initial states with the constraints in Φ_k.

The previous algorithm is implemented in the backward reachability procedures of existing symbolic model checkers for hybrid and concurrent systems like DMC [10] and HyTech [18]. As for other classes of infinite-state systems (see e.g. [1, 3, 18, 19]), the method is not guaranteed to terminate for all systems and properties. The theoretical termination is guaranteed whenever the abstract protocol is well-structured and the property can be expressed as an upward-closed set of states [11]. Unfortunately, our case study does not satisfy the conditions in [11]. Thus, we must evaluate the method on practical experiments.

Analysis of the Futurebus+ Protocol. The EFSM associated to the Futurebus+ Protocol consists of the set of rules of Fig. 5 and Fig. 6. In accord with the semantics of the states of the protocol, we would like to show that starting from the initial global state where all caches have *invalid* state, it is not possible to reach one of the global states denoting data inconsistency listed at the end of Section 2. The initial state Φ_{init} of the abstract protocol is expressed then as the constraint where $invalid \geq 1$ and all other variables are $= 0$. The unsafe states can be formulated as the set (disjunction) $\Phi_f = \{\varphi_1, \varphi_2\}$, where *more than one cache is in an exclusive state* is expressed as $\varphi_1 \equiv exclusiveU + exclusiveM \geq 2$, and *different caches are simultaneously in exclusive and shared states* is expressed as $\varphi_2 \equiv sharedU \geq 1, exclusiveU + exclusiveM \geq 1$. Using DMC [10], we have automatically checked that none of the unsafe states in Φ_f can be reached from the initial parameterized state Φ_{init}. DMC required 8 steps and 196.1 seconds (on a Pentium) to reach a fixpoint. The fixpoint consists of 47 constraints. This implies that our model of the Futurebus+ is safe (wrt. the data consistency properties we discussed before) for *any number* of processes in the system. We have obtained the same result using HyTech [18]. The description of the EFSM and of the above mentioned analysis together with the experiments described in [9] are available on the web at the address `http://www.disi.unige.it/person/DelzannoG/protocol.html`.

5. CONCLUSIONS

Parameterized protocols for systems with many identical processes can be represented (via suitable abstractions) as infinite-state systems with integer variables [15, 12, 13, 11]. Following this idea, in this paper we have modeled a parameterized version of the Futurebus+ cache coherence protocol at the behavior level taking into consideration a split transaction bus structure. The combination of backward reachability and symbolic representation via constraints is the key technology we used to verify automatically several safety properties (for any number

of caches) of an abstraction of the protocol. As future works, it would be interesting to model protocols at a finer level of abstraction (e.g. introducing explicit delays in the actions), and to study other important properties of cache-based systems like *sequential consistency*.

References

[1] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized System Verification. In *Proc. CAV '99*, LNCS 1633, pages 134–145, 1999.

[2] P. A. Abdulla, K. Cerāns, B. Jonsson and Y.-K. Tsay. General Decidability Theorems for Infinite-State Systems. In *Proc. LICS '96*, pages 313–321, 1996.

[3] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite-state Systems using Presburger Arithmetics. In *Proc. CAV '97*, LNCS 1254, pages 400–411, 1997.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about Networks with Many Identical Finite State Processes. Information and Computation 81(1): 13–31, 1989.

[5] 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 1(1):57–79, 1996.

[6] 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 *Proc. 11th Int. Symp. on Computer Hardware Description Languages and their Applications*, 1993.

[7] E. Clarke, O. Grumberg, and S. Jha. Verifying Parameterized Networks. TOPLAS 19(5): 726–750 (1997).

[8] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proc. POPL 78*, pages 84-96, 1978.

[9] G. Delzanno. Automated Verification of Parameterized Cache Coherence Protocols. To appear in *Proc. CAV '00*, July 2000.

[10] G. Delzanno and A. Podelski, Constraint-based Deductive Model Checking in CLP. To appear in *Software Tools for Technology Transfer*.

[11] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. LICS '99*, pages. 352–359, 1999.

[12] E. A. Emerson and K. S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proc. CAV '96*, LNCS 1102, pages 87–98, 1996.

[13] E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-deterministic Infinite-state Systems. In *Proc. LICS '98*, pages 70–80, 1998.

[14] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! Technical Report LSV-98-4, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan, April 1998.

[15] S. M. German and A. P. Sistla. Reasoning about Systems with Many Processes. JACM 39(3): 675–735 (1992)

[16] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proc. CAV '97*, LNCS 1254, pages 72-83, 1997.

[17] J. Handy. The Cache Memory Book. Academic Press, 1993.

[18] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In *Proc. CAV '97*, LNCS 1254, pages 460–463, 1997.

[19] N. Halbwachs, Y-E. Proy, and P. Roumanoff. Verification of Real-time Systems using Linear Relation Analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[20] C. Norris-Ip and D. L. Dill. Verifying Systems with Replicated Components in Murphi. *Formal Methods in System Design*, 14(3): 273–310, 1999.

[21] D. Lesens, N. Halbwachs, and P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. In *Proc. POPL '97*, 1997.

[22] K. L. McMillan and J. Schwalbe. Formal Verification of the Gigamax Cache Consistency Protocol. In *Proc. Int. Symp. on Shared Memory Multiprocessors*, pp. 242-51, 1991.

[23] F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. IEEE Transactions on Parallel and Distributed Systems 6(8), August 1995.