Electronic Notes in Theoretical Computer Science 50 No. 4 (2001) – Proc. VEPAS 2001
URL: http://www.elsevier.nl/locate/entcs/volume50.html 14 pages

Verifying a Network Invariant for All Configurations of the Futurebus+ Cache Coherence Protocol

Marcel Kyas

Institute for Computer Science and Applied Mathematics, Christian-Albrechts-Universität, D-24105 Kiel, Germany,
mky@informatik.uni-kiel.de

Abstract

In this paper we describe a network invariant for all configurations of the Futurebus+ Cache Coherence Protocol. The network invariant was computed with PAX and verified by a model checker. Using this invariant we are able to prove a specification of cache coherence correct for an arbitrary number of components on a single bus of the system. This specification includes a progress property not proven yet. We show how the result for the single bus system can be extended to tree-shaped systems. This is, as far as we know, the first uniform proof of the protocol with multiple data-buses.

1 Introduction

The automatic or semi-automatic verification of *parameterised networks*, i.e., of the *family* of systems $\mathcal{P} = \{P_i \mid i \in \omega\}$, where each P_i is a network consisting of i processes, is an interesting but difficult task: In [2] it is shown, that the verification problem for parameterised networks is undecidable in general. Nevertheless, automatic and semi-automatic methods for the verification of restricted classes of parameterised networks have been developed, e.g., [21,29,16,23,1,7,5]. These methods have not been applied to a real-world example, yet. We demonstrate, that the method of *abstraction of WS1S transition systems* [3,4,5] can be applied to the Futurebus+ Cache Coherence Protocol [19].

Considerable effort has been invested into the verification of the IEEE Futurebus+ protocol [11,6,20,12]. Either these efforts have not presented a *uniform* proof of correctness, i.e., a proof which establishes correctness for *all* instances of the parameterised network, or they have only verified a subset of the cache coherence specification.

We base our proof method on the method described in [3,4,5]. It consists of building a model of the parameterised network in the theory of weak second order logic of one successor (WS1S) [8,15,28]. The dynamic behaviour is described with a WS1S transition system. An abstraction relation is formulated in WS1S. Both serve as input to the tools PAX, which computes an finite-state abstract system from the model using MONA [18]. The resulting abstract system is suitable as an input to a symbolic model checker [26,10,12], which validates the specification. Our specification is given in *linear time temporal logic* (LTL) [25]. It can be checked using the model checker NuSMV [9] is used.

In [3,4,5] the parallel composition is asynchronous, i.e., based on an interleaving semantics. To handle architecture of the Futurebus+ protocol we extend the method with synchronous parallel composition.

Using this method we prove the protocol correct with respect to a specification of cache coherence, which includes a progress property, which has not yet been proven in literature. Moreover, by using compositional reasoning, we prove the protocol correct for tree structures.

Road Map

In the next section we introduce the notation and definitions used in this article. We recall the definition of WS1S transition systems. Section 3 describes our extension of the proof method in [3,4,5] to synchronous parallel composition. A short description of the Futurebus+ Cache Coherence Protocol is given in Sect. 4. Section 5 recalls the definition of L-simulation and network invariants, summarises the results of [22] and describes the construction of the network invariant for the general case. We close this article with a short conclusion and comparison to related work.

2 Preliminaries

In this section we briefly recall the basic definitions of WS1S [8,15,28] and boolean transition systems [3,4].

Terms of WS1S are build from the constant 0, first-order variables, and the successor function $\lambda t.t + 1$ applied to a term t. *Second-order terms* are either second-order variables, the constant $\emptyset$ denoting the empty set, or terms of the form $X \cup X'$, $X \cap X'$, or $X \setminus X'$, where X and X' denote second order terms. *Atomic formulae* are of the form b, $t = t'$, $t < t'$, or $t \in X$, where b is a *propositional variable*, t and t' are both terms, and X is a second-order term. *WS1S-formulae* are build from atomic formulae by applying the boolean connectives as well as quantification over first and second-order variables. *First-order monadic formulae* are WS1S-formulae in which no second-order variables occur. WS1S-formulae are interpreted over models that assign finite subsets of ω (the natural numbers) to second-order variables and elements of

ω to first-order variables. The interpretation is defined in the usual way [28]. Given a WS1S formula f, $[\![f]\!]$ denotes the set of models of f. The set of free variables is denoted by $\mathrm{FV}(f)$. By $\mathrm{FV}_2(f)$ we denote the free second-order variables, by $\mathrm{FV}_1(f)$ we denote the set of free first-order variables. One says that f is (first-order) *closed*, if $\mathrm{FV}(f) = \emptyset$ ($\mathrm{FV}_1(f) = \emptyset$). WS1S is a decidable formalism. Given a WS1S formula f we can construct a finite automaton, which is enumerating $[\![f]\!]$. However, in [27] it is shown that the space needed to compute the set of models of a WS1S formula is bounded from below by a stack of powers of two the size of the length of the formula. Despite this discouraging fact, a decision procedure has been implemented in the MONA tool [18].

We define components of a WS1S transition systems in a sub-language of WS1S called $\mathrm{AF}(n)$. For a first-order variable n let $\mathrm{AF}(n)$ be the set of formulae produced by the grammar:

$$f ::= x \in B \mid \neg f \mid f \wedge f \mid f \vee f \mid f \to f \mid \forall_n x.f \mid \exists_n x.f \quad ,$$

where B is a *second-order variable* and x is a *position variable*. If $m \in \omega$ then Σ_m denotes the set of evaluations σ such that $\sigma(n) = m$, $0 \leq \sigma(x) < m$, and $\sigma(B) \subseteq \{0, \ldots, m-1\}$. $\forall_n x.f$ abbreviates the formula $\forall x.(0 \leq x \wedge x < n) \to f$, and $\exists_n x.f$ abbreviates the formula $\exists x.(0 \leq x \wedge x < n) \wedge f$. Similarly for second-order variables B we abbreviate the formula $\forall i.i \in B \to f$ by $\forall_B i.f$ and $\exists i.i \in B \wedge f$ by $\exists_B i.f$. The formulae in $\mathrm{AF}(n)$ are interpreted over evaluations $\sigma \in \bigcup_{m \in \omega} \Sigma_m$ in the usual way.

Definition 2.1 *A* boolean transition system *(BTS) $S(i,n) = (\Theta, V, \mathcal{T})$ parameterised by i and n consists of*

- *a finite set V of boolean array variables. A* state *of a BTS is a valuation σ of the variables of V. The set of all states is denoted by Σ.*
- *a satisfiable assertion $\Theta \in \mathrm{AF}(n)$ with $\mathrm{FV}(\Theta) \subseteq V \cup \{i\}$ characterising the set of initial states.*
- *a set $\mathcal{T}$ of* transitions. *Any transition $\tau \in T$ can be represented by a predicate $\rho_\tau(V, V') \in \mathrm{AF}(n)$ with $\mathrm{FV}(\rho_\tau) \subseteq V \cup V' \cup \{i\}$, relating the unprimed variables in V to their primed successors in V'. We require an* idling *transition $I \in \mathcal{T}$ with $\rho_I(V, V') = \bigwedge_{v \in V}(v \leftrightarrow v')$.*

If $(s, s') \models \rho_\tau(V, V')$, then s' is a *τ-successor* of s, and we will write $s \xrightarrow{\tau} s'$. A *computation* of a BTS $S(i,n) = (\Theta, V, \mathcal{T})$ is an infinite sequence such that:

Initiality: $\pi(0) \models \Theta$.

Consecution: For all $i \in \omega$ we have $\pi(i) \xrightarrow{\tau}_S \pi(i+1)$ for some $\tau \in \mathcal{T}$.

3 Synchronous Monadic Parameterised Systems

In [3,4] a proof method is described for the abstraction of parameterised networks. This proof method is based on an asynchronous, i.e., interleaving

based, parallel composition operation. In this section we extend thos proof method to synchronous parallel composition. To this end, we define synchronous monadic parameterised systems.

Definition 3.1 *The* synchronous monadic parameterised system *(SMPS), built from a BTS $S(i,n)$, is the family $\mathcal{P} = \{\|_{l=0}^{m-1} S(i,n)[i/l, n/m] \mid m \in \omega_{>0}\}$, where $[\cdot/\cdot]$ is the* substitution *operation and $\|$ is the synchronous parallel composition.*

A SMPS can be represented as a WS1S transition system (WS1S-TS).

Definition 3.2 *A* WS1S transition system $\mathcal{S} = (\Theta, V, \mathcal{T})$ *is defined by:*

- *V is a finite set of second order variables, where each variable is interpreted as a finite set of natural numbers.*
- *The initial condition is given by a satisfiable assertion Θ with $\mathrm{FV}(\Theta) \subseteq V$.*
- *$\mathcal{T}$ is a finite set of transitions, where each τ is represented by a WS1S formula $\rho_\tau(V, V')$ with $\mathrm{FV}(\rho_\tau) \subseteq V \cup V'$.*

The computations of $\mathcal{S}$ are sequences of states and defined as usual. A SMPS is translated into a bisimilar WS1S-TS by the following translation procedure: Let $P = \{1, \ldots, n\}$ be a set of process indices from 1 to n. We use a translation function μ which replaces all occurrences of n by $\max(P) + 1$. The predicate

$$\mathrm{part}(P, \mathcal{B}) = (\bigwedge_{B,B' \in \mathcal{B} \wedge B \neq B'} B \cap B' = \emptyset) \wedge \bigcup_{B \in \mathcal{B}} B = P$$

determines whether the set system $\mathcal{B}$ is a partition of the set P. The translation procedure is:

Definition 3.3 *Consider an SMPS $\mathcal{P}$ built from $S(i,n) = (\Theta, V, \mathcal{T})$. Then the WS1S-TS $(\tilde{\Theta}, \tilde{V}, \tilde{\mathcal{T}})$ is defined by:*

- *$\tilde{V} = V \cup \{P\}$, where $P \notin V$.*
- *Let $\tilde{\Theta} = \exists n.P = \{1, \ldots, n\} \wedge (\bigwedge_{B \in V} B \subseteq P) \wedge (\forall_P m.\Theta\mu)$.*
- *Let $\tilde{\mathcal{T}} = \{\tilde{\rho}_\tau\}$, where $\tilde{\rho}_\tau = \exists_{\tau \in \mathcal{T}} Y_\tau.\mathrm{part}(P, \{Y_\tau \mid \tau \in \mathcal{T}\}) \wedge (\bigwedge_{B \in V} B \subseteq P) \wedge (P = P') \wedge (\forall_{\tau \in \mathcal{T}} i_\tau. \bigwedge_{\tau \in \mathcal{T}} (i_\tau \in Y_\tau \to \rho_\tau \mu[i/i_\tau \in X_p]))$ and $\lambda_{i \in \{1,\ldots,n\}} x_i$ abbreviates $\lambda x_1.\lambda x_2.\cdots\lambda x_n.$ for $\lambda \in \{\forall, \exists\}$.*

The transition relation is given by just one formula. However, the space needed to compute the set of models of a WS1S formula is bounded from below by a stack of powers of two the size of the length of the formula [27]. In our case study this resulted in a formula which is to large to be handled with our resources. This makes it necessary to keep the formula of the transition relation small. In [22] a method to remedy this situation is presented.

4 The Futurebus+ Cache Coherence Protocol

The intention of this section is to give an overview of how the IEEE Futurebus+ Protocol [19] works. It defines a cache to be a number of so-called *cache*

```
next(cmd) := case
  state=invalid: {none,read-shared,read-modified};
  state=shared-unmodified: {none,invalidate,read-shared,
                            read-modified};
  state=exclusive-unmodified: {none,copyback};
  state=exclusive-modified: {none,copyback};
esac;
```

Fig. 1. SMV Code of a Cache-Line's Command Part

lines. Part of this standard is a cache-coherence protocol designed to work in a hierarchically-structured multiple-bus system. The protocol maintains coherence on individual buses by having the caches *snoop*, or observe, all bus transactions. Coherence across buses are maintained using *bus-bridges*. Special agents, called *cache* and *memory agents*, at the ends of bridges represent remote caches and memories. In order to increase performance, the protocol uses so-called *split-transactions*. When a transaction is split, its completion is delayed and the bus is freed; at some later time, an explicit response is issued to complete the transaction. This facility makes it possible to service local requests while remote requests are being processed.

The state of a cache line is characterised by the four attributes *invalid*, *shared-unmodified*, *exclusive-modified*, and *exclusive-unmodified*. Figure 1 displays the command part of the system in NuSMV syntax [9]. The following paragraphs give an informal overview over all transitions.

A *transaction* is a sequence of events on the bus. Basically, it is a bus command, which is describing the nature of the transaction. The exact semantics of a bus command may be modified by one of the bus attributes *IV, SR, TF* or *WT*. Each processor has a local copy of those attributes, called *iv, sr, tf* or *wt* respectively. Usually, if a the local copy of such an attribute is set, it is also set on the bus by an or ($\vee$) operation.

Initially, any cache line is *invalid*. An *invalid* cache line must issue a command or snarf a copy of the desired data to change its state (*snarfing* means reading the data from a transaction issued by another component). It is possible for a cache line to transition to any other state from the *invalid* state. A module that initiates a *read-shared* transaction and completes it successfully transitions to *shared-unmodified* state if *TF* is asserted and to *exclusive-unmodified* state, otherwise. It may assert *tf* during a *read-shared*, *read-invalid* or *copy-back* transaction to change its state to *shared-unmodified*. A module initiating a *read-modified* transaction that completes successfully will cause its cache line to change state to *exclusive-modified*.

If a cache line is in *shared-unmodified* state, it may change its state to *invalid* state without a bus transaction at any time. It is required to do so if it snoops a *read-modified*, *write-invalid*, or *invalidate* transaction. If it snoops a *read-invalid* or *read-shared* transaction and does not assert *tf* to snarf the data, it changes state to *invalid*, too. A module that initiates a *invalidate*

transaction that completes successfully changes state to *exclusive-modified*. It does not change state otherwise.

Without using any bus transaction a module in *exclusive-unmodified* state may change its state to any of the three others at any time. If it snoops a *read-shared* or *read-invalid* transaction during which it asserts *tf* it changes state to *shared-unmodified* and to *invalid* otherwise. It is required to do so if it snoops a *read-modified*, *write-invalid* or *invalidate* transaction.

A processor may change the state of a cache line in *exclusive-modified* state to *invalid* or *shared-unmodified* by initiating a *copy-back* transaction. It must assert *tf* to change state to *shared-unmodified*. If this module snoops a *read-shared*, *read-invalid* or *read-modified* transaction, it must assert *iv* to intervene and supply the data in place of the memory. It may keep a *shared-unmodified* copy if it asserts *tf* during a *read-shared* or *read-invalid* transaction. It always changes the state of a cache line to *invalid* if it snoops a *write-invalid* transaction.

For any command with *WT* asserted, a cache line shall not change its state.

Furthermore, modules may *split* transactions if access time of a module is slow compared to bus access time. Cache and memory modules determine if they need to split a transaction by decoding the address and command for each cache-coherent read transaction that they snoop. If the module is responsible for that address and cannot respond immediately, it asserts *sr*.

If a module splits a *read-shared* or *read-invalid* transaction, it must eventually respond with a *shared-response* transaction. This transaction may be snarfed by any cache. The memory must snarf data from this transaction. If no other cache signals that it has the cache line or that it is snarfing the data by asserting *tf* on a *shared-response* transaction, the line may be tagged *exclusive-unmodified* by the requester. Otherwise, it is *shared*.

If a module splits a *read-modified* transaction, it must eventually respond with a *modified-response* transaction. This transaction may not be snarfed. The module that originally generated a *read-modified* or *invalidate* transaction and subsequently received a *modified-response* changes its cache line state to *exclusive-modified*.

Modules that split a transaction are required to assert *wt* if any other module initiates a request transaction to the same cache line. This forces a limit of a single outstanding transaction per cache line. When a module initiating a transaction receives wait status, it has to wait until a *shared-response* or *modified-response* is observed by the cache line, and then it may repeat its request. In case of a *shared-response* transaction it may snarf the data and satisfy its request without a bus transaction.

Specifying Cache Coherence.

We have based our specification of cache coherence on [11,12]. One specification was replaced by a progress property with an fairness assumption. To do this, the specification was rewritten in LTL [25].

In [19] illegal attribute combinations are specified. Whenever such a combination is observed, the system sets a *bus-error*. Even if the observed transaction is legal, it may indicate an *error*. The absence of these error conditions is specified as (1).

Next we want that at most one processor, indexed i, has got a writeable cache line; this is expressed by (2).

If any cache lines i and j have got a cache line in readable state, we require that they agree on the data. If a cache line i is holding a readable copy, and the memory line is known to be unmodified, then both should agree on the data (see (3) and (4)).

The *fairness assumption* Φ states that each member on the bus will be master of the bus infinitely often, a command is issued on the bus infinitely often, and that the wait attribute is not persistent, as expressed by (5). Under this assumption, we require that if a processor is waiting for a request, it will eventually receive an answer. Also, if a component is obligated to respond to a request, it will eventually fulfil its obligation, as expressed by (6) and (7).

$$\Box\,\neg(\mathit{bus\text{-}error} \vee \mathit{error}) \tag{1}$$

$$\forall i.\forall j.i \neq j \rightarrow (\Box\, p_i.\mathit{writeable} \rightarrow \neg(p_j.\mathit{readable} \vee p_j.\mathit{writeable})) \tag{2}$$

$$\forall i.\forall j.i \neq j \rightarrow (\Box\, p_i.\mathit{readable} \wedge p_j.\mathit{readable} \rightarrow p_i.\mathit{data} = p_j.\mathit{data}) \tag{3}$$

$$\forall i.\Box\, p_i.\mathit{readable} \wedge \neg m.\mathit{memory\text{-}line\text{-}modified} \rightarrow p_i.\mathit{data} = m.\mathit{data} \tag{4}$$

$$\Phi = (\forall d.d \in D \rightarrow \Box \Diamond d.\mathit{master}) \wedge (\Box \Diamond \mathit{CMD} \neq \mathit{none}) \wedge (\Box \Diamond \neg \mathit{WT}) \tag{5}$$

$$\forall i.\Phi \rightarrow \Box\,(p_i.\mathit{requester} \neq \mathit{none} \rightarrow \Diamond\,(p_i.\mathit{requester} = \mathit{none})) \tag{6}$$

$$\forall i.\Phi \rightarrow \Box\,(p_i.\mathit{responder} \neq \mathit{none} \rightarrow \Diamond\,(p_i.\mathit{responder} = \mathit{none})) \tag{7}$$

5 A Network Invariant for the Futurebus+

No one has yet presented a correct network invariant for the Futurebus+ Cache Coherence Protocol yet. Let $\sqsubseteq_{\mathrm{L}}^{\alpha}$ denote that α is a L-simulation [14] and $\sqsubseteq_{\mathrm{L}}$ that there exist an L-simulation. Then a network invariant is formally defined by:

Definition 5.1 *Let $(\mathcal{P}; \sqsubseteq_{\mathrm{L}})$ be a partially ordered set of processes, and $\|$* monotone *with respect to $\sqsubseteq_{\mathrm{L}}$, i.e., for all $P, Q \in \mathcal{P}$ with $P \sqsubseteq_{\mathrm{L}} Q$ and for all $R \in \mathcal{P}$ we have $P \parallel R \sqsubseteq_{\mathrm{L}} Q \parallel R$. Then we call a process $I \in \mathcal{P}$ a* network invariant*, if it satisfies $P \sqsubseteq_{\mathrm{L}} I$ and $P \parallel I \sqsubseteq_{\mathrm{L}} I$ for all $P \in \mathcal{P}$.*

A network invariant is an abstraction for all members of a family [29] and by this one has:

Theorem 5.2 *Let I be a network invariant for $\mathcal{P}$ and Φ a specification. If*

$I \models \Phi$, then for all $P \in \mathcal{P}$ we have $P \models \Phi$.

By this theorem it is sufficient to find a network invariant that satisfies a property to show that *all* members of the family have this property. Unfortunately, finding a network invariant is a difficult task. One might ask whether such a network invariant can always be found, or even whether one can compute one automatically. In fact, finding a network invariant is undecidable [29].

5.1 A Network Invariant for the Futurebus+ Single-Bus Case

In this section we describe how we have built a network invariant for single-bus configurations of the Futurebus+ system. A very similar, but erroneous, network invariant can be found in [12].

5.1.1 Building a parameterised model

The first step in the construction of the network invariant was defining a parameterised model of the protocol as a WS1S transition system. In this particular case it was a simple task. We used the model described in [11] and translated it into a BTS (see Def. 2.1). Using the method described in Sect. 3 we have build a WS1S-TS from this description. The memory line and the bus itself are not considered in the model. They were added by hand to the WS1S-TS.

Much effort has been invested into minimising the WS1S-TS. The transition relation of the automata constructed from the WS1S description of the system may grow exponentially in the number of variables occuring in the system's description. Therefore, we minimised the number of second-order variables and the syntactic representation of the transition relation. This is described in detail in [22].

5.1.2 Finding an abstraction relation

Our network invariant was automatically computed from the WS1S-TS using the manually supplied abstraction relation:

$$\begin{aligned}
I &\leftrightarrow (\forall i.\, p_i.state = \mathit{invalid}) \wedge \\
SU &\leftrightarrow (\forall i.\, p_i.state = \mathit{exclusive\text{-}modified} \wedge \\
&\qquad p_i.state \neq \mathit{exclusive\text{-}unmodified}) \wedge \\
EM &\leftrightarrow (\exists i.\, p_i.state = \mathit{exclusive\text{-}modified} \wedge \\
&\qquad \forall j.\, i \neq j \rightarrow p_j.state = \mathit{invalid}) \wedge \\
EU &\leftrightarrow (\exists i.\, p_i.state = \mathit{exclusive\text{-}unmodified} \wedge \\
&\qquad \forall j.\, i \neq j \rightarrow p_j.state = \mathit{invalid}) \wedge \\
BAD &\leftrightarrow \neg(I \vee SU \vee EM \vee EU) \vee \mathit{bus\text{-}error} \vee \mathit{error} \quad .
\end{aligned}$$

This is a "natural choice" for an abstraction relation. The idea is, that the abstract process should be able to "mimic" every behaviour of the system it is abstracting. The same idea is also used in [12]. We introduce an abstract

```
next(cmd) := case
  state=invalid: {none,read-shared,read-modified};
  state=shared-unmodified: {none,invalidate,read-shared,
                            read-modified};
  state=exclusive-unmodified: {none,copyback,read-shared,
                               read-modified};
  state=exclusive-modified: {none,copyback,read-shared,
                             read-modified};
esac;
```

Fig. 2. SMV Code of the Network-Invariant's Command Part

state *BAD*, which observes any violation of the safety properties defined in specifications (1) and (2).

The command part of the abstract system is given in Fig. 2. It differs from the original system only in the way the system issues the next command. By comparing Fig. 2 to Fig. 1, these changes become apparent:

- In *exclusive-modified* and *exclusive-unmodified* a *read-modified* and a *read-shared* transaction may be initiated.
- If a cache line is in *shared-unmodified* state, it may issue a *read-modified* and a *read-shared* command.

The next step is to prove whether the system constructed this way is indeed a network invariant. To do this we have to check whether the system abstract system and the abstraction relation satisfy Def. 5.1.

The first requirement ($P \sqsubseteq^{\alpha}_{\mathrm{L}} I$) holds, if the abstraction relation implies the identity relation, i.e., $\mathrm{id}_{\Sigma} \subseteq \alpha$, where Σ is the set of states of $\mathcal{S}(0,1)$. For the second requirement ($P \parallel I \sqsubseteq^{\alpha}_{\mathrm{L}} I$) we can use the fact that the parallel composition operation used here provides us with an intersection semantics (see [22] for a proof of this statement), i.e., $[\![P \parallel I]\!] = [\![P]\!] \cap [\![I]\!] = [\![P]\!]$, because $\mathrm{id}_{\Sigma} \subseteq \alpha$. Finally we have to check, whether the parallel composition operation is indeed monotonic with respect to our abstraction relation. Fortunately, this is the case, as stated by the following Lemma:

Lemma 5.3 *Let* $\mathcal{S}_1 = (\Theta_1, V, \mathcal{T}_1)$, $\mathcal{S}_2 = (\Theta_2, V, \mathcal{T}_2)$, *and* $\mathcal{S}_3 = (\Theta_3, V, \mathcal{T}_3)$ *be BTS,* Σ *the set of states of* $\mathcal{S}_i$ *for* $i \in \{1,2,3\}$, α *an abstraction such that* $\mathrm{id}_{\Sigma} \subseteq \alpha$, $S_1 \sqsubseteq^{\alpha}_{\mathrm{L}} S_2$, *and* $S_1 \sqsubseteq^{\alpha}_{\mathrm{L}} S_3$. *Then* $S_1 \parallel S_2 \sqsubseteq^{\alpha}_{\mathrm{L}} S_2 \parallel S_3$.

Proof. Because $\mathrm{id}_{\Sigma} \subseteq \alpha$ the claim is equivalent to $[\![S_1]\!] \cap [\![S_2]\!] \subseteq [\![S_2]\!] \cap [\![S_3]\!]$.

By $S_1 \sqsubseteq^{\alpha}_{\mathrm{L}} S_2$ we have $[\![S_1]\!] \cap [\![S_2]\!] = [\![S_1]\!]$. From $[\![S_1]\!] \subseteq [\![S_2]\!]$ and $[\![S_1]\!] \subseteq [\![S_3]\!]$ we have $[\![S_1]\!] \cap [\![S_1]\!] \subseteq [\![S_2]\!] \cap [\![S_3]\!]$, from which follows the claim.

Hence, for computing a network invariant in our framework, the abstraction relation only needs to contain the identity relation between the parameterised system and the abstraction.

Note, that the state *BAD* is not reached by the abstract system, i.e., the graph immediately shows the validity of specifications (1) and (2).

```
next(cmd) := case
  state=invalid: {none,copyback,read-shared,read-modified};
  state=shared-unmodified: {none,invalidate,copyback};
  state=exclusive-unmodified: {none,copyback,read-shared,
                               read-modified};
  state=exclusive-modified: {none,copyback,read-shared,
                             read-modified};
esac;
```

Fig. 3. SMV Code of the Command Part of [12]

The reasons for these are: Recall how the next command may be issued by a processor from Fig. 1. Assume the cache line is in *exclusive-modified* state. The processor must not, for example, issue a *read-shared* command. This can be done by a two processor configuration, which where the second processor is in *invalid* state.

5.1.3 Comparison to [12].

In [12] a system is presented from which the authors claim that it is a network invariant for the Futurebus+ Cache Coherence Protocol. Its command part is shown in Fig. 3. It is not a network invariant, because: A cache line in *invalid* state should not initiate a *copyback* transaction, because its data is, well, invalid. Such a transaction invalidates the content of the memory line. Moreover, a cache line in this model is not allowed to issue any *read* command if it is in *shared-unmodified* state. If one considers the system in *shared-unmodified* state and composes it with a cache line in *invalid* state, the system in [12] is no abstraction of this system, because it cannot mimic a *read* command issued by the *invalid* cache line.

5.1.4 Verification

The network invariant in Fig. 2 satisfies the specification of Sect. 4. The proof was established with PAX and NuSMV. The computations involved took less than 2 Minutes. The specifications (6) and (7) needed special treatment, because any read transaction may be *split*. This results in a computation invalidating this specification, where a transaction is always split. Using a proof rule from [4,5] we can prove that the system satisfies $\Box \Diamond \neg SR$. Hence we establish the specification for all single-bus configurations.

5.2 Generalising the Result

In this section we describe how to generalise the results from the linear case to arbitrary network topologies. The construction of the network invariant for the general case is based upon the following two observations:

(i) The memory board and the memory agent have a very similar behaviour. If one considers the communication of the memory agent on the bus,

one cannot distinguish it from a memory board (this is the intention of its design). We were able to show $MA \sqsubseteq_{\mathrm{L}} M$, where MA denotes the memory agent and M the memory board.

(ii) A processor and a cache agent have a very similar behaviour, too. As above, we have $CA \sqsubseteq_{\mathrm{L}} P$, where CA is the cache agent and P the processor with its cache line.

Using these abstractions and the fact that our parallel composition is monotonic with respect to $\sqsubseteq_{\mathrm{L}}$ (cf. [22]) the network invariant in question turns out to be the network invariant for the single-bus case (rf. Sect. 5.1). This will be established in the following paragraphs.

The first step is to show, that the bus-bridge BB does not interfere with normal operation. In our model of the protocol the bus bridge itself has an instant transmission time and only communicates the most necessary behaviour. The delays introduced by message propagation are simulated by having the agents non-deterministically split read-commands on their local bus. Then it easy to see, that $BB \sqsubseteq_{\mathrm{L}} B$, where B is a data bus. We have already shown, that $M \parallel B \parallel P \sqsubseteq_{\mathrm{L}} I$, where I is the network invariant. Using the monotonicity of $\parallel$ with respect to $\sqsubseteq_{\mathrm{L}}$ and the fact that $\sqsubseteq_{\mathrm{L}}$ is transitive, one has $MA \parallel BB \parallel CA \sqsubseteq_{\mathrm{L}} M \parallel B \parallel P \sqsubseteq_{\mathrm{L}} I$.

The second step is to put these results together. A single-bus system is composed of the data bus, a memory unit and an arbitrary number of processors. By the abstraction $MA \sqsubseteq_{\mathrm{L}} M$ one can replace the memory unit on a bus by an memory agent without changing its behaviour. Similarly any processor may be replaced by an cache agent. Hence the bus turned into a tree node. The main point is, that any of these substitutions have the same network invariant, because the memory-agent and the cache-agent are *implementations* of a memory or a cache, and each agent represents a complete multi-bus system, representable by the network invariant, on the bus.

Thus we have proven:

Proposition 5.4 *I is a network invariant for any configuration of the Futurebus+ Cache Coherence Protocol.*

This finishes the proof of the specification for all topologies of the Futurebus+ Cache Coherence Protocol.

6 Conclusion

By extending the methods presented in [3,4,5] we are able to compute a network invariant for the Futurebus+ Cache Coherence Protocol semi-automatically. This is, to the best of our knowledge, the first correct network invariant. A new progress property not shown before was established (see Equations (5)–(7)). This resulted in applying PAX to a large example. Using compositional reasoning we have shown that the network invariant for the general case of the Futurebus+ Cache Coherence Protocol is the same as the

network invariant for the protocol's single-bus case.

It turns out that the time needed to compute and verify the network invariant is far less the time needed to model-check a small sample configuration consisting of two processors. This makes the method presented here promising for the application to other large examples.

Related Work.

A precise model of the Futurebus+ cache coherence protocol was designed by E. Clarke et al. [11]. They used temporal logic model checking to show that the protocol satisfies a formal specification of cache coherence. They have verified a selection of examples and not a parameterised version. We have based our specification of cache coherence on theirs and extended it with a new progress property.

Parameterised versions of the protocol were verified in [20,6]. Both efforts only verify a subset of the specification given in [11].

In [20] the states of a network are encoded as a regular language over an alphabet of the states of its component processes and the transition relation is represented by a finite state transducer. This idea is improved in [1,7], where this method is called *regular model checking*. This method has been implemented in the Pen program [24]. We have based our method on similar ideas. For some systems a network invariant can be computed using an algorithm proposed by D. Lesens [23], where widening techniques [13] are used to compute a network invariant for linear parameterised networks. This idea is similar to ours. All those methods are semi-automatic or may fail to terminate.

We have used PAX to compute a network invariant [21,29] for single-bus configurations of the Futurebus+ Cache Coherence Protocol. Such network invariants are described in [12,22]. The network invariant given in [12] is not correct. Though the abstraction relations are similar, the proof in [12] was done by hand. Our semi-automatic method increases the confidence in the established proof.

In [16] it is shown that the verification problem for the linear Futurebus+ Cache Coherence Protocol is decidable. However, the verification with PAX is very fast. It only needed 2 Minutes to verify the system. Using a conventional model checker and 2 processors needed 36 hours to model check our specifications.

References

[1] P. A. Abdulla, A. Bouajjani, B. Johnson, and M. Nilsson. Handling global conditions in parameterized system verification. In Halbwachs and Peled [17].

[2] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent system. *Information Processing Letters*, 6(22), 1986.

[3] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS '00*, volume 1785 of *LNCS*. Springer, 2000.

[4] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying universal properties of parameterized networks. In M. Joseph, editor, *Proc. FTRTFT 2000*, volume 1926 of *LNCS*. Springer, 2000.

[5] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Verfication of parameterized protocols. *Journal of Universal Computer Science*, 7(2), 2001.

[6] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In A. J. Hu and M. Y. Vardi, editors, *Proc. CAV '98*, volume 1427 of *LNCS*. Springer, 1998.

[7] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touilli. Regular model checking. In E. A. Emerson and A. P. Sistla, editors, *Proc. CAV '00*, volume 1855 of *LNCS*. Springer, 2000.

[8] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6, 1960.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In Halbwachs and Peled [17].

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Workshop on Logics of Programs*, volume 131 of *LNCS*. Springer, 1981.

[11] 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. Claesen, editor, *Proc. 11th Int. Symp. Comp. Hardware Desc. Lang. App.* North Holland, 1993.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[13] R. Cousot and P. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on Princ. of Prog. Lang.*, 1977.

[14] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[15] C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amererican Mathematical Society*, 98, 1961.

[16] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. A. Henzinger, editors, *Proc. CAV '96*, volume 1102 of *LNCS*. Springer, 1996.

[17] N. Halbwachs and D. Peled, editors. *Proc. CAV '99*, volume 1633 of *LNCS*. Springer, 1999.

[18] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second order logic in practice. In *TACAS '95*, volume 1019 of *LNCS*. Springer, 1996.

[19] IEEE standard for Futurebus+—logical protocol specification, 1992.

[20] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. CAV '97*, volume 1254 of *LNCS*. Springer, 1997.

[21] R. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *ACM Symp. on Princ. of Dist. Comp.*, 1989.

[22] M. Kyas. Verifikation parameterisierter Netzwerke durch Abstraktion. Master's thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Kiel, 2000. In English, available at `http://www.informatik.uni-kiel.de/~mky/publications/`.

[23] D. Lesens. *Vérification et synthèse de systèmes réactifs.* PhD thesis, Institut National Polytechnique de Grenoble, September 1997.

[24] M. Nilsson. Analyzing parameterized distributed algorithms. Master's thesis, Dept. of Computer Systems, 1999.

[25] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[26] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and M. Montanari, editors, *Proc. 5th Int. Symp. on Programming*, volume 137 of *LNCS*. Springer, 1981.

[27] L. Stockmeyer. *The complexity of decision problems in automata theory and logic.* PhD thesis, Department of Electrical Engineering, M.I.T., Cambridge, M.A., 1974.

[28] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume III. Springer, 1997.

[29] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Proc. CAV '89*, volume 407 of *LNCS*. Springer, 1989.