Electronic Notes in Theoretical Computer Science 50 No. 4 (2001) – Proc. VEPAS 2001
URL: http://www.elsevier.nl/locate/entcs/volume50.html 15 pages

Networks of Processes with Parameterized State Space

K. Baukus K. Stahl

Institute of Computer Science and Applied Mathematics
CAU Kiel, Preusserstr. 1–9, D-24105 Kiel, Germany.
kba@informatik.uni-kiel.de
kst@informatik.uni-kiel.de

S. Bensalem Y. Lakhnech

VERIMAG, *Centre Equation, 2 Av. de Vignate,*
38610 Gières, France.
bensalem@imag.fr
lakhnech@imag.fr

Abstract

In general, the verification of parameterized networks is undecidable. In recent years there has been a lot of research to identify subclasses of parameterized systems for which certain properties are decidable. Some of the results are based on finite abstractions of the parameterized system in order to use model-checking techniques to establish those properties. In a previous paper we presented a method which allows to compute abstractions of a parameterized system modeled in the decidable logic WS1S. These WS1S systems provide an intuitive way to describe parameterized systems of finite state processes. In practice however, the processes in the network themselves are infinite because of unbounded data structures. One source of unboundedness can be the usage of a parameterized data structure. Another typical source may be the presence of structures ranging over subsets of participating processes. E.g., this is the case for group membership or distributed shared memory consistency protocols. In this paper we use deductive methods to deal with such networks where the data structure is parameterized by the number of processes and an extra parameter. We show how to derive an abstract WS1S system which can be subject to algorithmic verification. For illustration of the method we verify the correctness of a distributed shared memory consistency protocol using PVS for the deductive verification part and the tools PAX and SMV for the algorithmic part.

1 Introduction

Networks with an a priori unknown number of nodes occur everywhere in practice, e.g., processes running on a computer, machines in a token ring, a LAN or even the world wide web itself. Therefore, a lot of algorithms like mutual exclusion, leader election, group membership, or distributed memory algorithms are developed for such parameterized networks and they are expected to work properly for every concrete number of participants. Hence, although the problem is known to be undecidable (Apt and Kozen, [2]), there has been much interest in the verification of such protocols. Also, automated and semi-automated methods for the verification of restricted classes of parameterized networks have been developed.

Deductive methods presented in [18,25,5,23,14,19] are based on induction on the number of processes. These methods require finding a network invariant that abstracts any arbitrary number of processes with respect to a pre-order that preserves the property to be verified.

Algorithmic methods presented in [13,10,11] show that for classes of ring networks of arbitrary size and client-server systems, there exists k such that the verification of the parameterized network can be reduced to the verification of networks of size up to k.

A semi-automated method presented in [17] uses the idea of representing sets of states of parameterized networks by regular languages, where additionally finite-state transducers are used to compute predecessors. In [1,16] acceleration techniques are applied to consider the effect of taking infinitely often a transition.

In [3] we showed how to represent parameterized systems in the decidable logic WS1S, i.e., the current state of the system is modeled as a fixed number of finite subsets of the natural numbers and the transitions of the processes in the network are described in WS1S. Given a boolean abstraction relation in WS1S this allows to construct the abstract system automatically. The method was used to verify several parameterized protocols. With an additional marking algorithm and the lifting of fairness conditions ([4]) we were able to establish liveness properties for these protocols as well.

The method is implemented in a tool called PAX [1], that uses the decision procedures of Mona [15] to check the satisfiability of WS1S formulae.

In this paper we show how to combine this method with deductive verification in order to verify protocols which do not fulfill the above requirements, i.e., each process in the network has an unbounded state space. Here, we address different reasons for the unboundedness. First, we deal with the problem of having two parameters; one gives us the number of processes, the other parameterizes each process itself. Second, it is often the case that the processes render with data structures to keep information about the other processes,

[1] `http://www.informatik.uni-kiel.de/~kba/pax`

i.e., they have data structures ranging over subsets of the involved processes and are therefore unbounded.

We illustrate our method using a typical example: a distributed shared memory protocol. The protocol has two parameters, the number of processes and the number of memory pages. In order to guarantee exclusive write and multiple read the processes have to know the actual access privileges of the other processes. We give a generic abstraction for this type of protocols in the framework of PVS and show how to analyze the abstract WS1S system using pax and SMV to prove safety and liveness properties of the protocol.

2 Protocol Description

To illustrate our method we verify a distributed shared memory consistency protocol by Li and Hudak presented in [20]. The idea of distributed shared memory (DSM) is to allow processors in a distributed environment to utilize each other's local memories. DSM systems provide a virtual address space for a network of processors. They replicate or migrate data across the network to handle requests of threads running on the processors.

Li and Hudak's protocol is a multiple-reader, single-writer protocol; several processes are allowed to have read access to the same data while write access is exclusive. The data which is subject of the read/write requests is organized into *pages*. A *page table* on each processor maintains the current access status of the processor for each page. The status may be *read* or *write* and keeps the information whether the processor owns the page. The owner is the processor last having write privileges to a page. Moreover, the status may be *nil* if the processor has no local copy of the page or if the page has been modified by some other processor.

When a processor wants to upgrade its privileges (from nil to read or from read to write) it sends a corresponding request via broadcast. The owner handles the request; for a read request it adds the requesting processor to the *copy-set*, the list of processors with read access. Processors in the copy-set have to be informed when a processor wants write access. Each of these processors has to acknowledge that it changed its page table for that page to nil. Then, the ownership changes to the requesting processor. The pseudo-code in Figure 1 describes the various actions [12].

As communication mechanism we assume to have one central request queue where all the requests are sent to. Acknowledgments are sent directly to the requesting processors. The description of the protocol immediately gives us four verification goals:

Exclusive Ownership: For each page p there is always at most one owner.

Exclusive Write: Whenever there is a processor having write access for p then there is no processor with read access.

<table>
<tr><td>req_rd(p)
PTable[p].lock:=true;
broadcast read request for p;</td><td rowspan="2">send_rd_ack(p,i)
PTable[p].lock:=true;
IF PTable[p].owner THEN
PTable[p].copyset:=PTable[p].copyset ∪ {i};
PTable[p].access:=read;
send ack and p to i; FI
PTable[p].lock:=false;</td></tr>
<tr><td>get_rd_ack(p)
PTable[p].access:=read;
PTable[p].lock:=false;</td></tr>
<tr><td>req_wr(p)
PTable[p].lock:=true;
broadcast write request for p;</td><td rowspan="2">send_wr_ack(p,i)
PTable[p].lock:=true;
IF PTable[p].owner THEN
send p and copyset to i;
PTable[p].access:=nil;
PTable[p].owner:=false; FI
PTable[p].lock:=false;</td></tr>
<tr><td>get_wr_ack(p,copy-set)
PTable[p].access:=inv;
req_invalidate(p,copyset);</td></tr>
<tr><td>req_invalidate(p, copyset)
FOR k in copyset DO
send inv request for p to k;</td><td rowspan="2">send_inv_ack(p)
IF PTable[p].access=read THEN
PTable[p].access:=nil;
send acknowledgment; FI</td></tr>
<tr><td>read_nil_ack(i)
PTable[p].copyset:=
PTable[p].copyset \ {i};</td></tr>
<tr><td colspan="2">get_owner
IF PTable[p].copyset=∅ AND PTable[p].access=inv THEN
PTable[p].access:=write; PTable[p].copyset:=∅;
PTable[p].owner:=true; PTable[p].lock:=false; FI</td></tr>
</table>

Fig. 1. Pseudo-code of one processor according to the DSM protocol

Copy-Set Adequacy: Processors having read access to page p are always in the copy-set of the owner of p except for the owner itself.

Liveness: Whenever a processor requests access privileges it eventually obtains them.

3 Verification by Abstraction

First, let us recall some definitions and the idea of proving properties of systems by abstraction. Given a deadlock-free [2] transition system $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$ and a total abstraction relation $\alpha \subseteq \Sigma \times \Sigma_A$, we say that $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ is an *abstraction* of $\mathcal{S}$ w.r.t. α, denoted by $\mathcal{S} \sqsubseteq_\alpha \mathcal{S}_A$, if the following conditions are satisfied: (1) $\Theta \subseteq \alpha^{-1}(\Theta_A)$ and (2) $\tau \circ \alpha^{-1} \subseteq \alpha^{-1} \circ \tau_A$ for corresponding $\tau \in \mathcal{T}$, $\tau_A \in \mathcal{T}_A$.

In case Σ_A is finite, we call α finite abstraction relation. Let φ, φ_A be LTL formulae and let $[\![\varphi]\!]$ (resp. $[\![\varphi_A]\!]$) denote the set of models of φ (resp. φ_A). Then, from $\mathcal{S} \sqsubseteq_\alpha \mathcal{S}_A$, $\alpha^{-1}([\![\varphi_A]\!]) \subseteq [\![\varphi]\!]$, and $\mathcal{S}_A \models \varphi_A$ we can conclude $\mathcal{S} \models \varphi$. This statement, which is called preservation result, shows the interest of verification by abstraction: since if $\mathcal{S}_A$ is finite, it can automatically be checked whether $\mathcal{S}_A \models \varphi_A$. In fact, a similar preservation result holds for any temporal logic without existential quantification over paths, e.g., $\forall CTL^\star$, LTL, or $\mu_\Box$ [7,8,21].

If we have already proven some state property φ to be invariant in $\mathcal{S}$, i.e., $\mathcal{S} \models \Box\varphi$, we can strengthen condition (2) to (2′)

$$(\tau \cap \{(s_0, s_1) \mid s_0 \models \varphi, s_1 \models \varphi\}) \circ \alpha^{-1} \subseteq \alpha^{-1} \circ \tau_A \ .$$

This allows to search for some smaller abstract system in order to establish properties for the concrete system. We denote this type of abstraction by $\mathcal{S} \sqsubseteq^\varphi_\alpha \mathcal{S}_A$.

In case $\mathcal{S}$ is a fair transition system with $\mathcal{F}$ as fairness formula and if $\mathcal{F}_A$ is the fairness formula of $\mathcal{S}_A$, then by requiring $\alpha^{-1}([\![\neg\mathcal{F}_A]\!]) \subseteq [\![\neg\mathcal{F}]\!]$, we have the same preservation result as above. We indicate this type of abstraction by $\mathcal{S} \sqsubseteq^F_\alpha \mathcal{S}_A$.

3.1 System Reduction using PVS

Our aim is to verify the DSM protocol by Li and Hudak using abstraction techniques. In [3] we presented how to compute abstractions automatically for networks with finite processes modeled in the decidable logic WS1S. Unfortunately, Li and Hudak's protocol handles a parameterized number of pages (beside the parameterized number of processors) and each page table entry contains a set (*copy-set*) of processor indices.

To reduce the state space of the processors we introduce a *global-copy* (see Figure 2 for the PVS declaration). This variable is global to all processors. We added assignments to it whenever the processor having the ownership for one page updates its local *copy-set*. If we can prove that the local *copy-set* of some processor, whenever needed, coincides with the *global-copy* we can get rid of all the local *copy-sets*.

[2] Throughout this paper we only consider deadlock free transition systems which can be achieved by adding an idle transition.

```
Access: TYPE = {write, nil, read, invalidate}
Requests: TYPE = {read_req, write_req}
Direct: TYPE = {write_ack, read_ack, nil_req}

N, P: posnat

Processor: TYPE = below(N)
Page: TYPE = below(P)

Copy_set_type: TYPE = setof[Processor]

Req_channel: TYPE = list[[Processor, Requests, Page]]
Direct_channels: TYPE = [Processor → list[[Direct, Page]]]
Nil_Acks: TYPE = [Page → list[Processor]]

PageEntry: TYPE =
[# access: [Processor → Access],
   owner?, locked?: [Processor → bool],
   copy_set: [Processor → Copy_set_type],
   send_copy, global_copy: Copy_set_type #]

State: TYPE =
[# PageTable: [Page → PageEntry],
   Reqs: Req_channel,
   DirectCom: Direct_channels,
   Nils: Nil_Acks #]
```

Fig. 2. Parts of PVS theory

To deal with the second parameter (the page parameter) we identify a class of parameterized networks for which we can verify certain properties by instantiating the second parameter with 1.

These intermediate steps give us a reduced system presentable in WS1S but still allows to prove properties for the original protocol. We use PVS [22] to establish the reduction. Since PVS allows to use higher order logic for specifications it is straightforward to give a PVS model for the pseudo-code in Figure 1. Figure 3 shows an example transition expressed as a relation between pre- and post-state.

Using PVS it was straightforward to prove that indeed the *global-copy* matches with the local *copy-set* whenever needed. The property (see Figure 4) is inductive over all transitions and initially fulfilled and therefore invariant.

Generality of this step

As already mentioned network protocols often handle data structures ranging over sets of processors in the network. As examples we listed shared memory protocols such as the one described in Section 2 or group membership protocols. To illustrate how the idea to reduce the state space of each processor by introducing a global structure used by all processors can be applied to other protocols, let us consider a group membership protocol. Briefly, each

```
s, s1: VAR State

i: VAR Processor
p: VAR Page

req_rd(s, s1, i, p): bool =
    access(PageTable(s)(p))(i) = nil ∧
     ¬ locked?(PageTable(s)(p))(i) ∧
      s1 = s WITH [(PageTable)(p) := PageTable(s)(p)
                                  WITH [(locked?)(i) := TRUE],
                  Reqs := cons((i, read_req, p), Reqs(s))]
```

Fig. 3. Requesting read access

```
Global_Local(s, i, p): bool =
    (owner?(PageTable(s)(p))(i) ⊃
      copy_set(PageTable(s)(p))(i) = global_copy(PageTable(s)(p)))
```

Fig. 4. Equality of *global-copy* and *copy-set* for owner

processors keeps track which processors are still alive and vital part of the network. Due to continuous communication errors are detected and processors are removed from the membership-list of the well functioning processors. Two properties are of interest for those protocols; *agreement*, meaning that the well functioning processors agree on their membership-lists, and *validity*, meaning that an error will be detected eventually and then the membership-lists corresponds to the set of well functioning processors. We analyzed a synchronous group membership protocol by proving first agreement deductively. Then, we could reduce the system by using a global membership-list maintained by the processors working properly.

Now, back to our DSM protocol. Even if we remove all local copy-sets we have a system of processors where each of them maintains a page table for a parameterized number of pages. But, we observe (and can prove with PVS) that all transitions alter only the page entry for exactly one page and consume or produce only messages concerning this page. We fix these assumptions in the next definition.

Definition 3.1 Let $\mathcal{S}(N, P)$ be a parameterized system with N processors and a second parameter P. Let the processors communicate over some message queues $q_1, \ldots, q_k$ where each message is of the form (i, msg, p) with $i < N$, $p < P$, and msg of some finite type M. The state space of each processor j is an array $a[j][0..P-1]$ of size P and of finite type T.

We call $\mathcal{S}(N, P)$ *strictly parameterized* in P if each processor has initially the same configuration for each page:

$$\forall i < N : \forall p_1, p_2 < P : a[i][p_1] = a[i][p_2] \wedge \bigwedge_{l=1}^{k} q_l = \langle\rangle \ ,$$

and each transition in $\mathcal{S}(N,P)$ is either

- an internal step $a[i][p] = t_1 \wedge a' = a([i][p] \mapsto t_1') \wedge \bigwedge_{l=1}^{k} q_l' = q_l$
- or a communication step

$$\begin{aligned} &a[i][p] = t_1 \wedge a[j][p] = t_2 \wedge a' = a([i][p] \mapsto t_1', [j][p] \mapsto t_2') \wedge \\ &[q_l = \langle j, msg_1, p\rangle \cdot q_l' \wedge msg_1 = m \wedge][q_o' = q_o \cdot \langle j, msg_2, p\rangle \wedge] \\ &\bigwedge_{r \neq l,o} q_r' = q_r \ , \end{aligned}$$

where t_1, t_2, t_1', t_2' are constants in T and m, msg_1, msg_2 are constants in M. Concerning the communication step either the message consuming part $q_l = \langle j, msg_1, p\rangle \cdot q_l'$ or the message producing part $q_o' = q_o \cdot \langle j, msg_2, p\rangle$ may be empty. We call such transitions changing only array entries for p and handling only *p-messages*, *p-transitions*. □

As usual we denote with $[\![\mathcal{S}(N,P)]\!]$ the set of computations $\sigma = s_0, s_1, \ldots$ of $\mathcal{S}(N,P)$. Now, let $[\![\mathcal{S}(N,P)]\!] \downarrow_p$ denote those computations $\sigma^p = s_0^p, s_1^p, \ldots$ derived from the original ones by projecting the array $a[0..N-1][0..P-1]$ to the array entry $a[0..N-1][p]$ and projecting the contents of the communication channels to messages with tag p. Then, we can show:

Lemma 3.2 *For a system $\mathcal{S}(N,P)$ strictly parameterized in P we have for all P and $p < P$*

$$[\![\mathcal{S}(N,P)]\!] \downarrow_p \ = \ [\![\mathcal{S}(N,P)]\!] \downarrow_1 \ = \ [\![\mathcal{S}(N,1)]\!]^\tau \ ,$$

where $[\![\cdot]\!]^\tau$ denotes the set of computations with arbitrary interleaved idle transitions.

Proof. 'first equality': Concerning one processor i each computation starts with $a[i][p] = a[i][m] = t$ for $t \in T(N)$ and $p, m < P$. The message queues are empty. Hence, the same transitions for p and m are enabled which produce the same messages (msg, p) resp. (msg, m) and yield the same array entry t'. Hence, in each computation we can exchange p- and m-transitions. This gives us the first equality.
'second equality': If we consider in $\mathcal{S}(N,P)$ only 1-transitions to happen, trivially we have $[\![\mathcal{S}(N,1)]\!] \subseteq [\![\mathcal{S}(N,P)]\!]$. Since all p-transitions with $p \neq 1$ do not alter $a[i][1]$ for any processor i and do not consume or produce 1-messages they are idle transitions in $[\![\cdot]\!] \downarrow_1$. Hence, we have $[\![\mathcal{S}(N,P)]\!] \downarrow_1 \subseteq [\![\mathcal{S}(N,1)]\!]^\tau$.□

The introduction of idling transitions is needed because we have made no assumptions about fairness. Hence, it is not guaranteed that some p transitions occur or when they occur as long as other transitions are enabled. Nevertheless concerning invariance properties we get immediately:

Corollary 3.3 *Let $\mathcal{S}(N,P)$ a system strictly parameterized in P. Let $\varphi(p)$ be a state formula. Then, we have:*

$$\mathcal{S}(N,1) \models \Box\varphi(1) \Leftrightarrow \mathcal{S}(N,P) \models \forall p < P : \Box\varphi(p) \ .$$

□

To deal with liveness properties we need some assumptions about fairness. We call $\mathcal{S}(N,P)$ *weak fair* if all transitions continuously enabled from a certain point in a computation are eventually taken. If moreover $\mathcal{S}(N,1)$ never blocks a queue, i.e., messages in the queues are eventually consumed, we can deduce from Lemma 3.2:

Corollary 3.4 *Let $\mathcal{S}(N,P)$ be a system strictly parameterized in P and assume $\mathcal{S}(N,P)$ is weak fair. Let $\varphi(p)$ be a property expressed in LTL/X. Then, if $\mathcal{S}(N,1)$ never blocks a queue we have:*

$$\mathcal{S}(N,1) \models \varphi(1) \Leftrightarrow \mathcal{S}(N,P) \models \forall p < P : \varphi(p) \ .$$

□

Now, we observe that our reduced system is indeed strictly parameterized in P. Hence, we can get rid of the second parameter and are prepared to represent the resulting network in the framework of WS1S. Then, we use abstraction techniques explained in the next section to analyze it.

3.2 Verification using PAX

First we briefly recall the definition of weak second order theory of one successor (WS1S for short) [6,24].

Terms of WS1S are built up from the constant 0 and 1st-order variables by applying the successor function suc(t) ("$t+1$"). *Atomic formulae* are of the form b, $t = t'$, $t < t'$, $t \in X$, where b is a boolean variable, t and t' are terms, and X is a set variable (2nd-order variable). WS1S formulae are built up from atomic formulae by applying the boolean connectives as well as quantification over both 1st-order and 2nd-order variables.

WS1S formulae are interpreted in models that assign finite sub-sets of ω to 2nd-order variables and elements of ω to 1st-order variables. The interpretation is defined in the usual way.

Given a WS1S formula f, we denote by $[\![f]\!]$ the set of models of f. The set of free variables in f is denoted by $\mathit{free}(f)$.

Finally, we recall that by Büchi [6] and Elgot [9] the satisfiability problem for WS1S is decidable. Indeed, the set of all models of a WS1S formula is representable by a finite automaton (see, e.g., [24]).

Now, we introduce WS1S transition systems which are transition systems with variables ranging over finite sub-sets of ω and show how they can be used to represent the parameterized system from the previous section.

Definition 3.5 [WS1S Transition Systems] A *WS1S transition system* $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$ is given by the following components:

- $\mathcal{V} = \{X_1, \ldots, X_k\}$: A finite set of second order variables where each variable is interpreted as a finite set of natural numbers.

- Θ: A WS1S formula with $\mathit{free}(\Theta) \subseteq \mathcal{V}$ describing the initial condition of the system.
- $\mathcal{T}$: A finite set of transitions where each $\tau \in \mathcal{T}$ is represented as a WS1S formula $\rho_\tau(\mathcal{V}, \mathcal{V}')$, i.e., $\mathit{free}(\rho_\tau) \subseteq \mathcal{V} \cup \mathcal{V}'$. We use the primed version of the variables to denote the post-state. □

The computations of $\mathcal{S}$ are defined as usual. Moreover, let $[\![\mathcal{S}]\!]$ denote the set of computations of $\mathcal{S}$.

In the previous section we have proven that we can replace the local variables *copy-set* of each processor by one global variable *global-copy*. This and the argument that allows us to verify the system instantiated with only one page concerning the page parameter gives us the possibility to model the reduced system as a WS1S system.

As set of second order variables we choose

$$\begin{aligned}\mathcal{V} = \{\ & \mathit{Procs}, \mathit{Read}, \mathit{Write}, \mathit{Invalidate}, \mathit{Owner}, \mathit{Locked}, \mathit{Global\_Copy},\\ & \mathit{Read\_Req}, \mathit{Write\_Req}, \mathit{Nil\_Req}, \mathit{Read\_Ack}, \mathit{Write\_Ack}, \mathit{Nil\_Ack}\ \}\ .\end{aligned}$$

We have no set to represent the processors being in state *nil*. We assume those which are not in *read*, *write*, and *invalidate* to have nil access to the page. In WS1S it is not possible to represent queues, hence, we model them as sets. This, of course, gives us an abstraction. On the other hand we lose some fairness, e.g., when a request of processor i is in the queue, then the request is eventually granted when read by the owner, or the owner eventually stops to handle any request:

$$\Box(i \in \mathit{Read\_Req} \Rightarrow \Diamond(i \in \mathit{Read\_Ack}) \vee \Diamond\Box\neg\texttt{send\_rd\_ack})$$

Hence, we add those fairness conditions $\mathcal{F}$ to the WS1S system. The initial condition of our WS1S system reads as follows:

$$\begin{aligned}&(\exists n : \forall i : i < n \Rightarrow i \in \mathit{Procs})\\ \wedge\ &(\exists j : j \in \mathit{Procs} \wedge \mathit{Read} = \{j\} \wedge \mathit{Owner} = \{j\})\\ \wedge\ &\mathit{Locked} = \emptyset \wedge \mathit{Write} \cup \mathit{Invalidate} = \emptyset\\ \wedge\ &\mathit{Read\_Req} \cup \mathit{Write\_Req} \cup \mathit{Nil\_Req} = \emptyset\\ \wedge\ &\mathit{Read\_Ack} \cup \mathit{Write\_Ack} \cup \mathit{Nil\_Ack} = \emptyset\ .\end{aligned}$$

To illustrate how the transitions of our PVS specification (simplified by reduction to one page) can be expressed in WS1S we give here $\rho_{\texttt{req\_rd}}$:

$$\begin{aligned}\exists i :\ & i \notin \mathit{Read} \cup \mathit{Write} \cup \mathit{Invalidate} \cup \mathit{Locked}\\ & \wedge \mathit{Locked}' = \mathit{Locked} \cup \{i\} \wedge \mathit{Read\_Req}' = \mathit{Read\_Req} \cup \{i\}\\ & \wedge \bigwedge_{X \in \mathcal{V} \setminus \{\mathit{Read\_Req}, \mathit{Locked}\}} X' = X\ .\end{aligned}$$

Next, we want to analyze the WS1S system defined above by abstraction. Therefore, we apply the methods presented in [3,4]. Let $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$ be a given WS1S system and let α be a boolean abstraction function expressed as a WS1S formula $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$. Since the abstract variables are booleans, the

abstract system we construct is finite, and hence, can be subject to model-checking techniques. Moreover, we make use of the fact that both $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$ and the transitions in $\mathcal{T}$ are expressed in WS1S to give an effective construction of the abstract system.

Hence, the initial states of the abstract system we construct can be described by the formula

$$\exists \mathcal{V} : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \ .$$

As transitions the abstract system contains for each concrete transition τ an abstract transition τ_A, which is characterized by the formula

$$\exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A)$$

with free variables $\mathcal{V}_A$ and $\mathcal{V}'_A$. It is obvious that those definitions satisfy conditions (1) and (2) from Section 3. If we have already proven some invariance property ψ to hold for $\mathcal{S}$, we can choose

$$\exists \mathcal{V}, \mathcal{V}' : \psi(\mathcal{V}) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A) \wedge \psi(\mathcal{V}')$$

as abstract transitions which satisfy condition (2′).

To compute the abstract system, one has to find all states fulfilling these formulae, which is possible since they are WS1S formulae. This means, choosing properties $\varphi_i(\mathcal{V})$ of the concrete system as abstract variables $a_i \Leftrightarrow \varphi_i(\mathcal{V})$ allows us to compute automatically the abstract system according to the boolean abstraction function $\bigwedge_{i=1}^{n} a_i \Leftrightarrow \varphi_i(\mathcal{V})$.

Sometimes, we are interested in so-called *universal* progress or response properties, that is, properties that guarantee each single process i eventually makes some progress, or each request by i to j eventually gets a response by j. To prove those properties by abstraction the abstraction function has to focus on processors, i.e., the abstraction function contains i or i, j as free variables ($\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, i)$ or $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, i, j)$).

Then, the abstract system contains as abstract transitions

$$\exists \mathcal{V}, \mathcal{V}' : \exists i, j : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, i, j) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A, i, j)$$

(or those with invariance constraints) and starts in initial state:

$$\exists \mathcal{V} : \exists i, j : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, i, j) \ .$$

Both methods are implemented in PAX and we use them to analyze our system.

4 Verification Results

Using PVS it was easy to prove some intuitive properties to be inductive over all transitions and therefore being invariant. These properties state that the processor having write privileges for some page is also the owner, that ownership is exclusive, and that the owner knows the processors with read access (see Figure 5). Hence, we have already two of our four properties, namely exclusive ownership and copy-set adequacy. As stated in Section 3 we can use these invariants to strengthen our WS1S system.

```
Invariant1(s, i, p): bool =
    access(PageTable(s)(p))(i) = write ⊃
     ¬ locked?(PageTable(s)(p))(i) ∧ owner?(PageTable(s)(p))(i)

Invariant2(s, p): bool =
    (∃ (i: Processor):
      owner?(PageTable(s)(p))(i) ⊃
       (∀ (j: Processor): ¬ (j = i) ∧ ¬ owner?(PageTable(s)(p))(j)))

Invariant3(s, p): bool =
    ∀ (j: Processor): ∃ (i: Processor):
        (access(PageTable(s)(p))(j) = read ∧
          ¬ owner?(PageTable(s)(p))(j) ∧ owner?(PageTable(s)(p))(i))
         ⊃ (j ∈ copy_set(PageTable(s)(p))(i))
```

Fig. 5. Invariants proven with PVS

Using PAX we can successively check for more invariants. It is possible to strengthen the exclusive ownership property to

$$\begin{aligned} & Owner \cap Write\_Ack = \emptyset \\ \wedge\ & Owner \cap Invalidate = \emptyset \\ \wedge\ & Write\_Ack \cap Invalidate = \emptyset \\ \wedge\ & \exists j : \{j\} = Owner \cup Write\_Ack \cup Invalidate \end{aligned}$$

which states that to grant a write request the owner gives away the ownership of the page to acknowledge the request. The acknowledgment causes the requesting processor to go in the invalidate state to inform all processors having read privileges. We take the property given above as the definition for one abstract variable (or we can take one variable for each conjunct). The resulting state space of the abstract system has exactly one state, namely the state where the variable is (or all variables are) true.

The first of the remaining two properties, exclusive write access, can be expressed as:

(1) $\forall i, j : \Box(i \in Write \Rightarrow j \notin Read)$

This is because we have already proven exclusive ownership and that the processor with write privileges is the owner. The second one, that a request will be eventually granted, is described by the LTL formula:

(2) $\forall i : \Box(i \in Write\_Req \Rightarrow \Diamond(i \in Write \wedge i \in Owner))$

Both properties are universal as defined in the previous section, hence, we define an abstraction function concentrating on two processors i, j:

$$\begin{aligned} \alpha(\mathcal{V}, \mathcal{V}_A, i, j) \equiv\ & i \neq j \wedge \\ & \bigwedge_{X \in \mathcal{V}} a_i^X \Leftrightarrow i \in X \wedge \\ & \bigwedge_{X \in \mathcal{V}} a_j^X \Leftrightarrow j \in X \end{aligned}$$

The PAX tool generates a finite abstract system and translates it to the SMV input language. Moreover, PAX automatically adds new boolean variables to the SMV specification for each transition to monitor which transition was taken in the last step.

SMV verifies property 1 in about half a second. But SMV fails to prove property 2. It generates a counter example which loops with the transitions requesting for read or write access. At the concrete level this is not possible, in fact, each processor locks the page when performing a request. The next request by that processor can only be done after unlocking the page which corresponds to receiving the requested privileges. Hence, in the concrete system we can have only as much requests as processors. In [4] we presented a marking algorithm to transform such dependencies into liveness conditions stating that a transition decreasing some set can only be taken infinitely often when another one increasing the same set is also taken infinitely often. Applied to the set $\mathit{Procs} \setminus \mathit{Locked}$ that gives us the fairness condition:

$$\Box\Diamond(\texttt{req\_rd} \vee \texttt{req\_wr}) \Rightarrow \Box\Diamond(\texttt{get\_rd\_ack} \vee \texttt{get\_wr\_ack})$$

Note, the generated fairness conditions are guaranteed to hold in the concrete system and are therefore not part of its fairness requirements $\mathcal{F}$. In contrast, we also need some fairness condition $\mathcal{F}_A$ at the abstract level corresponding to the weak fairness assumptions $\mathcal{F}$ in the concrete system as defined in Section 3. Taking these fairness conditions together with the generated ones as assumptions to constrain our abstract system, SMV needs 15 seconds to establish property 2. By Corollary 3.4 we have established the correctness of the original distributed shared memory protocol.

5 Conclusions

We have shown how to represent parameterized networks where each processor has an unbounded state space in the framework of PVS. The use of higher order logic allows an intuitive modeling. For illustration of our method we have chosen a distributed shared memory protocol which is representative for a class of network protocols which maintain lists of other processors in the network. This makes the local state space of the processors unbounded. We used theorem proving to establish trace equality with another system maintaining only one global unbounded data structure. Besides the parameter specifying the number of participating processors another parameter specifies the data structures handled by the processors. We identified a subclass of doubly-parameterized systems for which we can reduce the verification task by analyzing the system instantiated with 1 as the data structure parameter. Hence, we were able to describe the reduced system as a WS1S system and apply the PAX tool to verify the systems properties by abstraction. Given the abstraction function – which is straight forward – the computation of the abstract system is fully automatically. The abstract system is finite state and can be subject to model-checking techniques. The properties proven also include

liveness properties which needed some extra fairness conditions generated by a marking algorithm and are guaranteed to hold in the concrete system.

While our method is not fully automatic and needs user interaction, it is far from pure theorem proving. Once we have come up with the WS1S system, generating abstractions and checking properties for the abstract system is an algorithmic task. Also, the verification of liveness properties by theorem proving would require a lot more user interaction than identifying fairness conditions which can be generated by the marking algorithm to exclude the counterexamples produced by the model-checker.

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, editors, *CAV '99*, volume 1633 of *LNCS*, pages 134–145. Springer, 1999.

[2] K. Apt and D. Kozen. Limits for Automatic Verification of Finit-State Concurrent Systems. *Information Processing Letters*, 22(6):307–309, 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, *TACAS'00*, volume 1785, pages 188 – 203. Springer, 2000.

[4] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. In M. Joseph, editor, *FTRTFT'00*, volume 1926, pages 291 – 304. Springer, 2000.

[5] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 1989.

[6] J.R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[7] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[8] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In E.-R. Olderog, editor, *Proceedings of PROCOMET '94*. North-Holland, 1994.

[9] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.

[10] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *22nd ACM Symposium on Principles of Programming Languages*, pages 85–94, 1995.

[11] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In *8th Conference on Computer Aided Verification*, LNCS 1102, pages 87–98, 1996.

[12] K. Fisler and C. Girault. Modelling and Model Checking a Distributed Shared Memory Consistency Protocol. In *ICATPN'98*. Springer, 1998.

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

[14] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 22(6/7), 1992.

[15] 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.

[16] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *TACAS'00*, volume 1785. Lecture Notes in Computer Science, 2000.

[17] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In O. Grumberg, editor, *Proceedings of CAV '97*, volume 1256 of *LNCS*, pages 424–435. Springer, 1997.

[18] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *ACM Symp. on Principles of Distributed Computing, Canada*, pages 239–247, Edmonton, Alberta, 1989.

[19] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL '97*, Paris, 1997.

[20] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[21] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1), 1995.

[22] S. Owre, J. Rushby, N. Shankar, and F. von Henke. "formal verification for fault-tolerant architectures: Prolegomena to the design of PVS". *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[23] Z. Stadler and O. Grumberg. Network grammars, communication behaviours and automatic verification. In *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, pages 151–165, Grenoble, France, 1989. Springer Verlag.

[24] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 134–191. Elsevier Science Publishers B. V., 1990.

[25] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants (extended abstract). In Sifakis, editor, *Workshop on Computer Aided Verification*, LNCS 407, pages 68–80, 1989.