ELSEVIER

Science of Computer Programming 43 (2002) 1–33

Science of Computer Programming

www.elsevier.com/locate/scico

State space generation for the HAVi leader election protocol

Yaroslav S. Usenko

CWI, P.O. Box 94079, 1090 GB Amsterdam, Netherlands

Received 18 June 1999; received in revised form 9 March 2001

Abstract

This paper describes two specifications of the leader election protocol from the home audio/video interoperability (HAVi) architecture. The specifications were written in two concurrent specification languages: μCRL and PROMELA. Two toolsets allowing generation of finite labeled transition systems, for μCRL and PROMELA, respectively, were applied in this case study. The results of the state space generation by both tools and some conclusions on the semantical differences between PROMELA and μCRL are presented in this paper. © 2002 Elsevier Science B.V. All rights reserved.

1. Introduction

Currently, in the field of software verification many of the existing state-of-the-art analysis methods are based on state space representations in the form of finite-state labeled transition systems (FLTS). It appears that even for completely different concurrent languages, FLTSs can be used to describe the behavior of specifications in these languages. There are number of tools to manipulate FLTSs, to check different kinds of equivalences and preorders, to find deadlocks, to check modal and temporal properties, to minimize FLTSs in different ways, etc. It is interesting to study the different verification tools supporting concurrent languages, by comparing how fast they can generate an FLTS, and how many states and transitions are contained in the resulting FLTS.

In this paper we consider two toolsets that allow state space generation—one for the algebraic concurrent language μCRL [9], and one for the imperative concurrent language PROMELA [14]—and compare the state spaces generated by them for one particular leader election protocol.

E-mail address: yaroslav.usenko@cwi.nl (Y.S. Usenko).

0167-6423/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-6423(01)00018-1

The μCRL toolset [7] has been developed at CWI to support formal reasoning about systems specified in μCRL. Its implementation is based on term rewriting [1] and linearization techniques [10]. It allows to generate state spaces, search for deadlocks, perform some optimizations on μCRL specifications, simulate them, and store the FLTSs into files readable by certain model checking and minimization tools, like CADP [5] developed at INRIA.

Spin [13] has been developed at Bell Labs and is one of the fastest and most widely used tools for protocol verification. It allows formal analysis of PROMELA specifications, model checking of LTL formulas [16], generation of state spaces, and searching for deadlocks.

In this case study we consider the leader election protocol from the home audio/video interoperability (HAVi) architecture [11]. Previously, this protocol was specified in PROMELA and LOTOS, and analyzed formally [17]. Here we take a more abstract definition of the protocol, to keep the specification relatively simple and free of many implementation details. In [17] several incorrect behaviors of the HAVi leader election protocol were found. We found an incorrect behavior of a similar kind in our model of the protocol using simulation of the μCRL specification.

As the first step, the leader election protocol was modeled in μCRL. After that we made a model in PROMELA which closely resembles the behavior of the μCRL model. To this end, we deliberately did not use some elements of PROMELA, as using these elements would give rise to semantical differences between the μCRL and the PROMELA models. For example, unlike μCRL, PROMELA has asynchronous communication built in. [1] As a result of not using such PROMELA features, the PROMELA model is quite different from what a straight formalization of the informal description of the protocol could be. However, it is semantically close to the model in μCRL, which enables a clear comparison between the state spaces of the two models. Finally, we generated the FLTSs for both models and checked them for the absence of deadlocks.

The structure of this paper is as follows. First, we describe the leader election protocol informally (Section 2). Then we present the specification in μCRL (Section 3) and some details about its specification in PROMELA (Section 4). We conclude with results of state space generation by the tools (Section 5). We assume a basic familiarity of the reader with μCRL and PROMELA; Section 3.1 contains an overview of the μCRL syntax that can also be found for instance in [8], and Section 4 contains an overview of PROMELA; the language definition of PROMELA can be found on the Spin WWW page [14]. For a more systematic introduction to μCRL see [6]. For a systematic treatment of ACP style process algebra, which is the basis of μCRL, see [2,3].

[1] Asynchronous communication in PROMELA does not allow one to clean the communication buffer, which is needed in the μCRL model of the HAVi leader election protocol.

2. Informal description

The informal description of the HAVi leader election protocol appears on pages 160–162 of [11]. We try to stay as close to the description in [11] as possible; however, our description differs in the places where abstractions from some implementation details were made in the μCRL and PROMELA models of the protocol. We put the emphasis on behavioral and communicational aspects and abstracted from the exact data definitions used in the protocol. This was done to reduce the sizes of the specifications and to make the case study applicable to the toolsets.

The system consists of a number of Device Control Module Managers (DCMM). Each DCMM has its own input buffer, from which it gets incoming messages that were sent to it by other DCMMs via the bus. The environment may influence the system by flipping (i.e. switching on or off) DCMMs, and it may observe that a DCMM has finished the election procedure. In both the PROMELA and the μCRL specification all of these components are modeled as processes which communicate synchronously.

The structure of the system is presented in Fig. 1. The communication between two DCMMs is done via the buffer of the receiving DCMM. We did not implement the communication via the bus, as this would make the specification of the bus more involved, and would not add much because communication via the buffer is asynchronous anyway. The environment flips the DCMMs synchronously and the bus observes that a DCMM was flipped in a synchronous manner as well.

Each DCMM has its unique ID number by which it can be addressed. If the environment flips a DCMM, the bus observes this change in the network by communicating with the DCMM in question. The bus informs all working DCMM processes about the changes in the network via their buffers, by first cleaning a buffer and then delivering a network reset message into this buffer.

The leader election is performed among the DCMMs that are "on" in the following way. After receiving a *NetworkReset(nst)* message, a DCMM starts to perform the

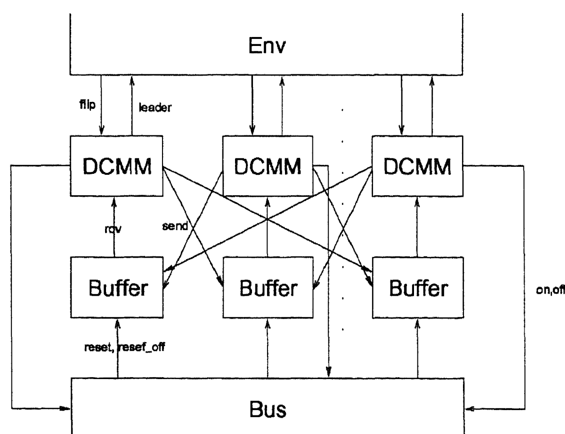

Fig. 1. Processes and communications in the system.

election procedure. It gets the status information about the network from the parameter *nst*. This status information says which DCMMs are currently on in the network. The function $il(N, nst)$ is then used to determine the ID of the initial leader. By comparing this ID with its own ID, the DCMM can decide whether it is the initial leader or an initial follower. The initial leader behaves as follows.

- From each initial follower *m* it awaits a *DMCapabilityDeclaration*(*m*, *URL*) message, from which it learns whether the DCMM *m* has the URL capability (has access to the Internet).
- Upon reception of the message from each initial follower that is on, the initial leader uses the function *fl*(*N*, *nst*, *URLs*) to determine the ID of the final leader.
- It sends a *DMLeaderDeclaration*(*m*,*fl*, *URLs*) message to each initial follower *m*, thus informing it about the final leader. The final leader is the last one to which this message is sent.
- Finally, it communicates with the environment by a leader action, indicating what it regards to be the final leader.

Each initial follower *m* behaves as follows:

- It keeps sending a *DMCapabilityDeclaration*(*m*, *URL*) message to the initial leader until it receives a *DMLeaderDeclaration*(*m*,*fl*, *URLs*) message from it.
- Finally, it communicates with the environment by a leader action, indicating what it regards to be the final leader.

It is important to realize that at any moment of the election any DCMM may be flipped, or may receive a *NetworkReset* message. In case a DCMM is switched on, it awaits for a *NetworkReset* message. In case of receiving a *NetworkReset* message, it (re)starts the election procedure. The DCMMs ignore any unexpected messages. The goal of the election procedure is to elect a final leader. This means that when no network resets occur any longer, each DCMM will eventually get information about the final leader, and this information will be the same for each DCMM.

3. Specification in μCRL

The complete μCRL specification can be found in Appendix A or obtained from the WWW. [2]

3.1. Overview of the μCRL syntax

Starting from a set Act of actions that can be parameterized with data, processes are defined by means of guarded recursive equations (these are explained at the end of this section) and the following μCRL operators.

[2] From `http://www.cwi.nl/~ysu/sources/HAVi` or by contacting the author.

First, there is a constant δ ($\delta \notin \mathsf{Act}$) that cannot perform any activity and is called deadlock or inaction.

Next, there are the sequential composition operator $\cdot$ and the alternative composition operator $+$. The process $x \cdot y$ first behaves as x and if x terminates successfully, continues to behave as y. The process $x + y$ can either behave as x or as y.

Interleaving parallelism is modeled by the operator $\|$. The process $x \| y$ is the result of interleaving actions of x and y, except that actions from x and y may also synchronize to a communication action, when this is explicitly allowed by a communication function. This is a partial, commutative and associative function $\gamma : \mathsf{Act} \times \mathsf{Act} \rightarrow \mathsf{Act}$ that describes how actions can communicate; parameterized actions $a(d)$ and $b(d')$ communicate to $\gamma(a, b)(d)$, provided $d = d'$.

To enforce that actions in processes x and y synchronize, we can prevent actions from happening on their own, using the encapsulation operator ∂_H. The process $\partial_H(x)$ can perform all actions of x except that actions in the set $H \subseteq \mathsf{Act}$ are blocked. So, assuming $\gamma(a, b) = c$, in $\partial_{\{a,b\}}(x \| y)$ the actions a and b are forced to synchronize to c.

We assume the existence of a special action τ ($\tau \notin \mathsf{Act}$) that is internal and cannot be observed directly. The hiding operator τ_I renames the actions in the set $I \subseteq \mathsf{Act}$ to τ. By hiding all internal communications of a process, only the external actions remain observable.

The following two operators combine data with processes. The sum operator $\sum_{d:D} p(d)$ describes the process that can execute the process $p(d)$ for any value d selected from the data domain D. The process $x \lhd b \rhd y$ (where b is a boolean) has the behavior of x if b is true and the behavior of y if b is false. Combining these two operations we get, for instance, that $\sum_{d:D}(a(d) \lhd d = 0 \rhd \delta)$ can only perform $a(0)$.

We apply the convention that $\cdot$ binds stronger than $\sum$, followed by $_ \lhd _ \rhd _$, the parallel operators, and $+$ binds weakest.

A set of recursion variables with data parameters is used to define processes recursively. A recursive equation is an equation defining a recursion variable as being equal to a process term that contains μCRL operators and recursion variables. For example, $\mathsf{X}(n : Nat) = a(n) \cdot \mathsf{X}(n + 1)$ is a recursive equation defining the recursive variable X which carries a data parameter that ranges over the natural numbers. For each natural number m, $\mathsf{X}(m)$ performs an infinite sequence of actions $\mathsf{a}(m) \cdot \mathsf{a}(m+1) \cdot \mathsf{a}(m+2) \cdot \ldots$. A recursive equation is completely guarded if all occurrences of recursion variables in it are always preceded by an action, and a recursive equation is guarded if there is an equivalent equation which is completely guarded. For example, the recursive equation in the example above is guarded, while $\mathsf{X} = \mathsf{X}$ is not.

3.2. Data types

The sorts *Bool* and *Nat* represent booleans and natural numbers, respectively. Sort **ABI** is a boolean array with natural indices. It is implemented by keeping the list of indices of elements that are true in ascending order. Sorts *Message* and *Status* are

described below. Finally, the sort *Queue* is a FIFO queue with elements of the sort *Message*. It is used in the Buffer process definition.

3.2.1. Constants

The initial parameters of the protocol are defined as constants. The value of nB determines the capacity of the buffers. We have to limit the capacity, because otherwise the state space would become infinite. The value of $initNDCMM$ is the number of DCMM processes in the system. The value of $initNst$ is the boolean array of size $initNDCMM$, representing the initial network status (which processes are "on" initially). The value of $initURLs$ contains the information on URL capabilities of the DCMM processes. The function il is defined as the minimal ID of a process that is "on". The function fl is the minimal ID of a URL capable process that is "on", or the minimal ID of a process that is "on" if all of the URL capable processes are "off".

map
$nB :\rightarrow Nat$
$initNDCMM :\rightarrow Nat$
$initNst :\rightarrow \mathbf{ABI}$
$initURLs :\rightarrow \mathbf{ABI}$
rew
$nB = 2$
$initNDCMM = 3$
$initNst = seton(0_0, 0)$
$initURLs = seton(0_0, 1)$
map
$il : Nat \times \mathbf{ABI} \rightarrow Nat$
$fl : Nat \times \mathbf{ABI} \times \mathbf{ABI} \rightarrow Nat$
var
$N : Nat$
$nst, URLs : \mathbf{ABI}$
rew
$il(N, nst) = if(eq(nst, 0_0), 0, min_on(nst))$
$fl(N, nst, URLs) = if(eq(nst, 0_0), 0,$
$\quad if(eq(URLs, 0_0), min_on(nst), min_on(URLs)))$

Here 0_0 is the constant of the sort **ABI** representing a boolean array in which all of the values are "false". The function $seton(abi, i)$ sets the ith element of the array abi to "true".

3.2.2. Messages

The sort *Message* is used to define all the messages that DCMM processes can receive. The use of abstract data types allows us to define messages having different parameters.

sort *Message*
func
$NetworkReset : \mathbf{ABI} \rightarrow Message$
$DMCapabilityDeclaration : Nat \times Bool \rightarrow Message$
$DMLeaderDeclaration : Nat \times \mathbf{ABI} \rightarrow Message$
map
$eq : Message \times Message \rightarrow Bool$

3.2.3. Status

The sort *Status* is a simple enumerated type used to represent the statuses in which a DCMM process can be. We could use different μCRL processes for each status, but in this case the two alternatives that are enabled in each status would be repeated in each such process. This could be avoided if we had a disrupt mechanism in μCRL. The drawback of our approach of having just one process is that we have a lot of parameters in each recursive call, most of which are not used in each state of the DCMM process.

The definition of the sort *Status* is a common way to represent enumerated types in μCRL. One could also use the sort *Nat* directly and the constructors *INIT*, etc., as maps to the corresponding naturals. However, such an approach leads to rewriting of the symbolic information to natural numbers, decreasing the readability of the output generated by the tools.

sort *Status*
func
$INIT, LE, LEIF, LEIL, LEILS, AOS, AO :\rightarrow Status$
map
$n : Status \rightarrow Nat$
$eq : Status \times Status \rightarrow Bool$
rew
$n(INIT) = 0 \; n(LE) = 1 \; n(LEIF) = 2 \; n(LEIL) = 3$
$n(LEILS) = 4 \; n(AOS) = 5 \; n(AO) = 6$
var $a, b : Status$
rew $eq(a, b) = eq(n(a), n(b))$

The meaning of each status abbreviation is explained below in the description of the processes.

3.3. Actions and communication function

The following actions are used in the specification. The names of the actions have the following intuition. The actions with underscores correspond to "send" actions, the actions without underscores to "read" actions, and the actions with double underscores to "communication" actions. The communication function is defined according to this intuition.

act

_flip, flip_on, flip_off, __flip	: *Nat*
_on, _off, on, off, __on, __off	: *Nat*
_send, send, _rcv, rcv, __send, __rcv	: *Nat* × *Message*
_reset, reset, __reset	: *Nat* × **ABI**
_reset_off, reset_off, __reset_off	: *Nat*
_leader	: *Nat* × *Nat*
j	

comm

_flip | flip_on	= __flip
_flip | flip_off	= __flip
_on | on	= __on
_off | off	= __off
_send | send	= __send
_rcv | rcv	= __rcv
_reset | reset	= __reset
_reset_off | reset_off	= __reset_off

3.4. Processes

3.4.1. DCMM process

The parameters of the process have the following meaning: *St* is the status of the process; *URL* is true if the DCMM has URL capabilities; *n* is the ID of the process; *N* is the total number of processes in the system, *nst* is the current network status; *wait* is the array of processes from which a message is awaited, or the array of processes to which a message still has to be sent; *URLs* is the array of URL capabilities of other processes, collected by the process; *il* and *fl* are the initial and final leader IDs, respectively; and *am_on* is true iff the process is on.

$$\mathsf{DCMM}(St{:}Status, URL{:}Bool, n{:}Nat, N{:}Nat, nst{:}\mathbf{ABI}, wait{:}\mathbf{ABI}, URLs{:}\mathbf{ABI}, il{:}Nat, fl{:}Nat, am_on{:}Bool) =$$

The following alternatives are enabled for any status of the DCMM process. It can be switched on, if it was off. In this case it communicates with the Bus process by an on action, and its status becomes *INIT* (Initial status). If the DCMM process is on, it can receive a *NetworkReset(nst1)* message and change its status to *LE* (Leader Election). Alternatively, it can be flipped off, communicate with the Bus process by off, and change its status to *INIT*.

$$\mathsf{flip_on}(n) \cdot \mathsf{_on}(n) \cdot \mathsf{DCMM}(INIT, URL, n, N, 0_0, 0_0, 0_0, 0, 0, \mathbf{t}) \lhd \neg am_on \rhd \delta$$
$$+$$
$$\sum_{nst1:\mathbf{ABI}} \mathsf{rcv}(n, NetworkReset(nst1)) \cdot \mathsf{DCMM}(LE, URL, n, N, nst1, 0_0, 0_0, 0, 0, \mathbf{t}) \lhd am_on \rhd \delta$$
$$+$$
$$\mathsf{flip_off}(n) \cdot \mathsf{_off}(n) \cdot \mathsf{DCMM}(INIT, URL, n, N, 0_0, 0_0, 0_0, 0, 0, \mathbf{f}) \lhd am_on \rhd \delta$$
$$+$$

If the status of the DCMM process is *LE*, the following alternatives may be enabled. In case the DCMM process is the only process in the network that is "on", it declares itself to be the final leader, informs the environment about it, and goes to autonomous operation. In case the DCMM process is not the initial leader, it sends its capabilities to the initial leader, and its status becomes *LEIF* (Leader Election Initial Follower). In case none of the two above applies, the DCMM process can receive a capability

declaration from a process m and then, depending on whether it still has to wait for messages from other processes, its status becomes either $LEIL$ (Leader Election Initial Leader) or $LEILS$ (Leader Election Initial Leader Sending). Finally, the DCMM process ignores any leader declaration messages in this status.

$$
\begin{array}{l}
(\;\; \mathsf{_leader}(n,n)\cdot \\
\quad \mathsf{DCMM}(AO, URL, n, N, nst, 0_0, upd(0_0, n, URL), 0, n, \mathbf{t}) \lhd n_on(nst) = 1 \rhd \delta \\
\;+ \\
\quad \mathsf{_send}(il(N,nst), DMCapabilityDeclaration(n, URL)) \\
\qquad \cdot\, \mathsf{DCMM}(LEIF, URL, n, N, nst, 0_0, 0_0, il(N,nst), 0, \mathbf{t}) \lhd il(N,nst) \neq n \rhd \delta \\
\;+ \\
\quad \Big(\sum_{m:Nat} \sum_{d:Bool} \\
\qquad\quad \mathsf{rcv}(n, DMCapabilityDeclaration(m,d)) \Big) \\
\qquad\qquad \cdot\, \mathsf{DCMM}(LEILS, URL, n, N, nst, setoff(nst,n), upd(upd(0_0), n, URL), m, d), \\
\qquad\qquad\quad 0, fl(N, nst, upd(upd(nst,n, URL), m, d)), \mathbf{t}) \\
\qquad\qquad \lhd n_on(nst) = 2 \rhd \\
\qquad\quad \mathsf{rcv}(n, DMCapabilityDeclaration(m,d)) \\
\qquad\qquad \cdot\, \mathsf{DCMM}(LEIL, URL, n, N, nst, setoff(setoff(nst,n), m), \\
\qquad\qquad\quad upd(upd(0_0, n, URL), m, d), 0, 0, \mathbf{t}) \\
\qquad) \lhd il(N,nst) = n \wedge n_on(nst) \neq 1 \rhd \delta \\
\;+ \\
\qquad \Big(\sum_{m:Nat} \sum_{URLs1:\mathbf{ABI}} \mathsf{rcv}(n, DMLeaderDeclaration(m, URLs1)) \Big) \\
\qquad\quad \cdot\, \mathsf{DCMM}(LE, URL, n, N, nst, 0_0, 0_0, 0, 0, \mathbf{t}) \\
\quad) \lhd St = LE \rhd \delta \\
+
\end{array}
$$

If the status of the DCMM process is $LEIF$, it behaves as an initial follower. This means that it can send its capabilities to the initial leader, receive a leader declaration, and ignore any capability declaration messages:

$$
\begin{array}{l}
(\;\; \mathsf{_send}(il, DMCapabilityDeclaration(n, URL)) \\
\qquad \cdot\, \mathsf{DCMM}(LEIF, URL, n, N, nst, 0_0, 0_0, il, 0, \mathbf{t}) \\
\quad + \\
\qquad \sum_{m:Nat} \sum_{URLs1:\mathbf{ABI}} \mathsf{rcv}(n, DMLeaderDeclaration(m, URLs1)) \\
\qquad\quad \cdot\, \mathsf{DCMM}(AOS, URL, n, N, nst, 0_0, URLs1, 0, m, \mathbf{t}) \\
\quad + \\
\qquad \Big(\sum_{m:Nat} \sum_{d1:Bool} \mathsf{rcv}(n, DMCapabilityDeclaration(m, d1)) \Big) \\
\qquad\quad \cdot\, \mathsf{DCMM}(LEIF, URL, n, N, nst, 0_0, 0_0, il, 0, \mathbf{t}) \\
) \lhd St = LEIF \rhd \delta \\
+
\end{array}
$$

If the status of the DCMM process is *LEIL*, it behaves as initial leader. This meant that it can receive a capability declaration from process m and then, depending on if this was the last message it was waiting for, its status becomes *LEIL* or *LEILS*. The DCMM process ignores any leader declarations in this state.

$$
\begin{array}{l}
\Bigg(\sum_{m:Nat}\sum_{d:Bool} \\
\qquad \mathsf{rcv}(n, DMCapabilityDeclaration(m,d)) \\
\qquad\quad \cdot\, \mathsf{DCMM}(LEILS, URL, n, N, nst, setoff(nst,n), upd(URLs,m,d), \\
\qquad\qquad 0, fl(N, nst, upd(URLs,m,d)), \mathbf{t}) \\
\qquad \lhd n_on(wait) = 1 \wedge wait[m] \rhd \\
\qquad \mathsf{rcv}(n, DMCapabilityDeclaration(m,d)) \\
\qquad\quad \cdot\, \mathsf{DCMM}(LEIL, URL, n, N, nst, setoff(wait,m), upd(URLs,m,d), 0, 0, \mathbf{t}) \\
\quad + \\
\quad \Bigg(\sum_{m:Nat}\sum_{URLs1:\mathbf{ABI}} \mathsf{rcv}(n, DMLeaderDeclaration(m, URLs1))\Bigg) \\
\qquad \cdot\, \mathsf{DCMM}(LEIL, URL, n, N, nst, wait, URLs, 0, 0, \mathbf{t}) \\
\quad) \lhd St = LEIL \rhd \delta \\
+
\end{array}
$$

If the status of the DCMM process is *LEILS*, it informs the initial followers about the final leader. If the final leader has to be informed, it is informed last. All messages are ignored by the process in this state. After informing the last initial follower, the status of the process becomes *AOS* (Autonomous Operation Sending).

$$
\begin{array}{l}
\Bigg(\sum_{m:Nat} \\
\qquad (\ \ \mathsf{_send}(m, DMLeaderDeclaration(fl, URLs)) \\
\qquad\qquad \cdot\, \mathsf{DCMM}(LEILS, URL, n, N, nst, setoff(wait,m), URLs, 0, fl, \mathbf{t}) \\
\qquad\quad \lhd m \neq fl \wedge n_on(wait) > 1 \rhd \delta \\
\qquad + \\
\qquad\quad \mathsf{_send}(m, DMLeaderDeclaration(fl, URLs)) \\
\qquad\qquad \cdot\, \mathsf{DCMM}(AOS, URL, n, N, nst, 0_0, URLs, 0, fl, \mathbf{t}) \\
\qquad\quad \lhd n_on(wait) = 1 \rhd \delta \\
\qquad) \lhd wait[m] \rhd \delta \\
\quad + \\
\quad \Bigg(\sum_{m:Nat}\sum_{URLs1:\mathbf{ABI}} \mathsf{rcv}(n, DMLeaderDeclaration(m, URLs1))\Bigg) \\
\qquad \cdot\, \mathsf{DCMM}(LEILS, URL, n, N, nst, wait, URLs, 0, fl, \mathbf{t}) \\
\quad + \\
\quad \Bigg(\sum_{m:Nat}\sum_{d1:Bool} \mathsf{rcv}(n, DMCapabilityDeclaration(m, d1))\Bigg) \\
\qquad \cdot\, \mathsf{DCMM}(LEILS, URL, n, N, nst, wait, URLs, 0, fl, \mathbf{t}) \\
\quad) \lhd St = LEILS \rhd \delta \\
+
\end{array}
$$

If the status of the DCMM process is AOS, it informs the environment about the result of the election, and its status becomes AO (Autonomous Operation). If the status of the DCMM process is AO, it performs a j loop. This is an abstraction of the autonomous behavior of the process.

$$
\begin{array}{l}
\quad \mathsf{_leader}(n,fl)\cdot \mathsf{DCMM}(AO,URL,n,N,nst,0_0,URLs,0,fl,\mathbf{t}) \lhd St = AOS \rhd \delta \\
+ \\
\quad \mathsf{j}\cdot \mathsf{DCMM}(AO,URL,n,N,nst,0_0,URLs,0,fl,\mathbf{t}) \lhd St = AO \rhd \delta
\end{array}
$$

3.4.2. Environment

In μCRL it is not necessary to specify the environment explicitly. The reactive system is described by its interaction with the environment. Everything else within the system may be abstracted from. However, for verification or testing purposes some assumptions about the environment have to be made. This can be done by specifying the assumed environment as a process, and putting it in parallel with the system.

In our particular case, the environment may flip DCMM processes in the system any number of times, and then stop. But it cannot stop when all of the DCMM processes are "off".

$$
\begin{array}{l}
\mathsf{Env}(N{:}Nat, nst{:}\mathbf{ABI}) = \sum_{m:Nat} \\
\quad (\quad \mathsf{_flip(m)}\cdot \mathsf{Env}(N, reverse(nst,m)) \\
\quad +\mathsf{_flip(m)}\cdot \delta \lhd n_on(reverse(nst,m)) > 0 \rhd \delta \\
\quad) \lhd N > m \rhd \delta
\end{array}
$$

3.4.3. Bus

The bus can observe changes in the network configuration and inform the active processes about these changes. It is specified with the help of two processes. The process Bus can communicate with a DCMM process by an action on or off to observe that this process was flipped. The process Bus1 is used to reset the buffers of all active processes in no particular order.

$$
\begin{array}{l}
\mathsf{Bus}(N{:}Nat, nstat{:}\mathbf{ABI}) = \\
\quad \sum_{m:Nat} \mathsf{on}(m)\cdot \mathsf{Bus1}(N, seton(nstat,m), seton(nstat,m)) \\
+ \\
\quad \sum_{m:Nat} \mathsf{off}(m)\cdot \mathsf{_reset_off}(m) \\
\qquad \cdot(\mathsf{Bus}(N, setoff(nstat,m)) \lhd n_on(nstat) = 1 \rhd \\
\qquad\quad \mathsf{Bus1}(N, setoff(nstat,m), setoff(nstat,m))) \\
\mathsf{Bus1}(N{:}Nat, nstat{:}\mathbf{ABI}, wait{:}\mathbf{ABI}) = \\
\quad \sum_{m:Nat} \mathsf{_reset}(m, nstat)\cdot(\mathsf{Bus}(N, nstat) \lhd n_on(wait) = 1 \rhd \\
\qquad\quad \mathsf{Bus1}(N, nstat, setoff(wait,m))) \lhd wait[m] \rhd \delta
\end{array}
$$

3.4.4. Buffer

The process Buffer is a FIFO queue of capacity nB. The FIFO queue can receive a message if it is not full, or send a message if it is not empty. By communicating with the Bus, the Buffer can be reset in two different ways: by an action reset or reset_off. In the first case it clears its message queue, and puts the network reset message into this queue. In the second case it just clears the queue.

$$
\begin{aligned}
&\mathsf{Buffer}(N{:}Nat, n{:}Nat, q{:}QMes) = \\
&\quad\left(\sum_{mes:Message} \mathsf{send}(n, mes)\cdot \mathsf{Buffer}(N, n, add(q, mes))\right) \lhd nB > size(q) \rhd \delta \\
&+ \\
&\quad \mathsf{_rcv}(n, first(q))\cdot \mathsf{Buffer}(N, n, remfirst(q)) \lhd \neg is_empty(q) \rhd \delta \\
&+ \\
&\quad \sum_{nst1:\mathbf{ABI}} \mathsf{reset}(n, nst1)\cdot \mathsf{Buffer}(N, n, add(\langle\rangle, NetworkReset(nst1))) \\
&+ \\
&\quad \mathsf{reset_off}(n)\cdot \mathsf{Buffer}(N, n, \langle\rangle)
\end{aligned}
$$

3.5. System

The whole system consists of several processes in parallel. First, three pairs of DCMM and Buffer processes are composed. Then they are merged together, and merged with the Bus process. Finally the Env is merged with the system.

$$
\begin{aligned}
&\mathsf{SYSTEMDCMM}(N{:}Nat, nstat{:}\mathbf{ABI}, URLs{:}\mathbf{ABI}) = \\
&\quad \partial_{\{\mathsf{_flip,flip_on,flip_off}\}}(\\
&\quad\quad \tau_{\{\mathsf{j,__on,__off,__reset,__reset_off}\}} \circ \partial_{\{\mathsf{_on,on,_off,off,_reset,reset,_reset_off,reset_off}\}}(\\
&\quad\quad\quad \tau_{\{\mathsf{__send}\}} \circ \partial_{\{\mathsf{_send,send}\}}(\\
&\quad\quad\quad\quad \tau_{\{\mathsf{__rcv}\}} \circ \partial_{\{\mathsf{_rcv,rcv}\}}(\\
&\quad\quad\quad\quad\quad \mathsf{DCMM}(INIT, URLs[0], 0, N, 0_0, 0_0, 0_0, 0, 0, nstat[0]) \parallel \mathsf{Buffer}(N, 0, \langle\rangle)) \\
&\quad\quad\quad\quad \parallel \tau_{\{\mathsf{__rcv}\}} \circ \partial_{\{\mathsf{_rcv,rcv}\}}(\\
&\quad\quad\quad\quad\quad \mathsf{DCMM}(INIT, URLs[1], 1, N, 0_0, 0_0, 0_0, 0, 0, nstat[1]) \parallel \mathsf{Buffer}(N, 1, \langle\rangle)) \\
&\quad\quad\quad\quad \parallel \tau_{\{\mathsf{__rcv}\}} \circ \partial_{\{\mathsf{_rcv,rcv}\}}(\\
&\quad\quad\quad\quad\quad \mathsf{DCMM}(INIT, URLs[2], 2, N, 0_0, 0_0, 0_0, 0, 0, nstat[2]) \parallel \mathsf{Buffer}(N, 2, \langle\rangle)) \\
&\quad\quad\quad) \parallel \mathsf{Bus}(N, nstat) \\
&\quad\quad) \parallel \mathsf{Env}(N, nstat) \\
&\quad)
\end{aligned}
$$

The system is initialized in the following way.

$$\mathbf{init}\ \mathsf{SYSTEMDCMM}(initNDCMM, initNst, initURLs)$$

4. From μCRL to PROMELA

PROMELA—the underlying language of SPIN—is a C-like imperative concurrent nondeterministic language. It has no explicit parallel operator, but has a process creation mechanism. Communication can happen via explicitly defined channels. It may either be synchronous or asynchronous. It is possible to pass data values during the communication. There are loops and goto statements. Nondeterminism is modeled by the following construction:

```
  if
  :: <alternative 1>
  :: <alternative 2>
...
  :: <alternative n>
  fi
```

If an alternative starts with a blocking statement, then it is disabled. The blocking statements are send and read statements in cases when synchronous communication is not possible, and any expression with value 0. Each process may have local variables. Shared variables are also allowed. To minimize the state space and interleavings, special constructions like `atomic{<block>}` and `d_step{<block>}` are allowed. Atomic sequences do not interleave with other processes executions. Sequences within `d_step` are considered to be one statement, meaning that no transfers of control to or from `d_step` are allowed, nor communications with `d_step`.

The PROMELA specification of the HAVi leader election protocol discussed in this paper was written based on the μCRL model, in order to preserve the semantics of this model as much as possible. The aim was to obtain the same behavior in the PROMELA model as in the μCRL model. This was achieved by a simulation of the behavior of μCRL constructions in PROMELA. Another approach would have been to use features of PROMELA for which there are no counterparts in μCRL. This would have lead to a more elegant PROMELA specification, which however would have differed from the μCRL model, thus obstructing a clear comparison between the state spaces of the two models.

Some crucial details of the implementation of μCRL constructions in PROMELA are described below. The source code of the PROMELA specification of the HAVi leader election protocol can be found in Appendix B or can be obtained from WWW. [3]

4.1. Abstract data types

There is no support for abstract data type specification in PROMELA. There is built-in support for arrays, structures and enumerated data types though. Operations on data types can be encoded as macro definitions or as in-line functions. Computations may

[3] From `http://www.cwi.nl/~ysu/sources/HAVi`, or by contacting the author.

be done within d_step blocks, allowing to consider long deterministic computations as one step.

4.2. Conditions and nondeterminism

The semantics of the conditional operator in PROMELA is slightly different from the semantics of conditions in μCRL. In PROMELA conditions are statements as in imperative languages, meaning that their execution causes a transition from one state to the other. In μCRL conditions are not transitions to other states, but restrictions under which such transitions are possible.

Therefore, we cannot simply translate a μCRL expression of the form

$$\begin{aligned} \mathsf{X}(d{:}N) = {}& _\mathsf{a}(d) \cdot \mathsf{X}(d) \lhd d < 5 \rhd \delta + \\ & _\mathsf{b}(d) \cdot \mathsf{X}(d) \lhd d < 7 \rhd \delta \end{aligned}$$

to

```
X:
  if
    :: d<5 -> a!d; goto X;
    :: d<7 -> b!d; goto X;
  fi;
```

because this would lead to different semantic behavior. For instance, if $d<5$ and there is another process Y willing to communicate with our process X via channel b, then one of the possible executions of the PROMELA specification above leads to a deadlock. Namely, since the condition $d<5$ is evaluated to true, X starts waiting for communication via a, while Y is waiting for communication via b. By contrast, the μCRL specification above does not contain a deadlock under these circumstances.

A semantically sound translation of the μCRL specification above is:

```
X:
  if
  :: d<5 -> if
            :: a!d; goto X;
            :: b!d; goto X;
            fi;
  :: (d<7)&&!(d<5) -> b!d; goto X;
  fi;
```

In the general case to correctly translate a μCRL expression with a choice of several conditions to PROMELA, we first need to make these conditions disjoint. Disjointness means that at most one condition can be true for any set of parameter values. It is always possible to make all conditions disjoint in this case; however, instead of n conditions in the original μCRL expression we may end up with $2^n - 1$ conditions in the resulting PROMELA expression.

4.3. Value-passing communication

In the value-passing communication style of μCRL, read and send actions are dual. This is due to commutativity of the communication function γ. The value-passing mechanism in μCRL is based on matching parameter values: $\mathsf{a}(e_1) \mid \mathsf{b}(e_2) = \gamma(\mathsf{a},\mathsf{b})(e_1)$ if $eq(e_1, e_2)$. That is why both read and send actions can be parameterized by arbitrary expressions. The standard value-passing communication is modeled in μCRL in the following way: one process performs a send action, e.g. $_\mathsf{a}(5)$; and the other process performs a receive action for an arbitrary value of the data domain, e.g. $\sum_{n:Nat} \mathsf{a}(n)$. By putting the two processes in parallel, and forcing them to communicate, we get

$$\partial_{\{_\mathsf{a},\mathsf{a}\}}\left(_\mathsf{a}(5) \middle\| \sum_{n:Nat} \mathsf{a}(n)\right) = __\mathsf{a}(5),$$

which is an action $__\mathsf{a}$ saying that the synchronous communication happened, and that the value 5 was passed during this communication. However, this value-passing mechanism allows to express more: for instance, the second process may decide to receive only naturals that are less than 10 and refuse to communicate (and thus receive) other values. This can be expressed as $\sum_{n:Nat}(\mathsf{a}(n) \lhd n < 10 \rhd \delta)$, which will communicate with $_\mathsf{a}(5)$, but not with $_\mathsf{a}(15)$, leading to a deadlock in the latter case.

In PROMELA value-passing communication is performed differently. Send statement $\mathsf{a}!e_1$ means that the value of e_1 is put into the channel a. Read statement $\mathsf{a}?m$ means that a value is read from the channel a and assigned to the variable m.

For the case of a read action a, the translation is performed in the following way:

$$\mathsf{X}(d{:}D) = \sum_{n:N} \mathsf{a}(n,d) \cdot \mathsf{X}(g(n,d)) \lhd b(d) \rhd \delta$$

becomes

```
X:
  b(d) -> a[d]?n; atomic{d_step{g(n,d)}; goto X}
```

Here we assume that a is an array of channels indexed by elements of D, and that the corresponding send statements take the form `a[d]!e`, not `a!d,e`. If D is an infinite set, then we cannot define such an array in PROMELA, and we need to consider the subset of elements of D for which b is true. If this subset is finite, we can make the array a to be indexed by those elements only.[4] A remedy to this lack of expressiveness in PROMELA can be found in [15]. We note, however, that the approach described there requires tripling of the communication channels, and the use of shared memory.

[4] Actually, this solution only works if the boolean b does not depend on the value that is being received.

4.4. Parameterized nondeterminism

Another μCRL construction is nonbounded and parameterized nondeterminism for actions that are not meant to be receive actions. Consider the following process equation:

$$\mathsf{X}(d{:}N) = \sum_{n:N} _\mathsf{a}(n,d) \cdot \mathsf{X}(g(n,d)) \lhd n \leqslant d \rhd \delta.$$

Here the set of possible alternatives depends on the value of d. Assuming that _a is not a read action, we can translate this process equation to PROMELA in the following way:

```
X:
  n=0;
TEMP:
  if
  :: n<=d -> a!n,d; atomic{d_step{g(n,d)}; goto X;}
  :: n<d -> atomic{n=n+1; goto TEMP}
  fi
```

In this case we again have an increase of the state space; this time it is linear in the number of alternatives in a state of the μCRL process.

5. State space generation

Spin and the μCRL toolset were used to generate the entire state spaces of specifications of the HAVi leader election protocol in PROMELA and μCRL, respectively, and to search for deadlocks in these specifications.

The process of state space generation in Spin is described in Chapter 13 of [12]. The basic idea is to generate an action/effect matrix for each statement of each process type, and to explore the state space step by step by allowing each process to perform a transition and adding the resulting state to the discovered state space.

State space generation for μCRL is described in [4]. First the specification is transformed into a linear form, which is a symbolic representation of a labelled transition system, and then the explicit labelled transition system is generated by the μCRL instantiator.

Unfortunately, it is not possible to get the state space as an output of Spin. Therefore, it is not possible to compare the generated state spaces. The following results were obtained by considering systems with two or three DCMMs and different buffer sizes. State spaces for four DCMMs could not be generated by either toolset. In the case with three DCMMs and buffer size two we could not get the state space analyzed by Spin. The results of state space generation are presented in Table 1.

Table 1
Results of state space generation

	Spin	μCRL
2 DCMMs Buffer size 2	States: 128,803 Transitions: 187,339 Elapsed time: 3.5 s Memory: 6.2 MB	States: 3842 Transitions: 13,460 Elapsed time: 7.6 s Memory: 6.8 MB
2 DCMMs Buffer size 5	States: 208,215 Transitions: 301,590 Elapsed time: 6.3 s Memory: 8.7 MB	States: 7292 Transitions: 26,048 Elapsed time: 12.9 s Memory: 7.1 MB
3 DCMMs Buffer size 1	States: 107,486,000 Transitions: 188,381,000 Elapsed time: 47 h : 01 min Memory: 8.35 GB	States: 576,120 Transitions: 3,290,223 Elapsed time: 25 min : 20 s Memory used: 25 MB
3 DCMMs Buffer size 2	States: $>265,798,000$ Transitions: $>449,935,000$ Elapsed time: >190 h Memory: >15.6 GB	States: 3,136,289 Transitions: 18,248,754 Elapsed time: 2 h : 10 min Memory used: 155 MB

Table 2
Command line arguments

Spin	μCRL

`> spin -av HAVi.spin`
`> cc -O3 -64 -w -o pan`
`-D_POSIX_SOURCE -DMEMCNT=35`
`-DSAFETY -DNOFAIR -DCOLLAPSE`
`-g pan.c`
`> ./pan -m <depth>`
	`> mcrl -regular -tbfile HAVi.mcrl`
	`> instantiator HAVi`

In order to enable the reader to reproduce our results, the precise command line arguments for both tools are given in Table 2. The invocation of Spin with the parameter `-av` generates a verifier in pan.c, which is used for the state space generation. The verifier is compiled using a C compiler with the maximal optimization (`-O3` option) to run on a 64 bit architecture (`-64` option). The option `-DMEMCNT=35` sets an upper bound on the amount of memory that can be allocated for a maximum of 2^{35} bytes. The option `-DSAFETY` optimizes the verifier for the case where no cycle detection is needed. The option `-DNOFAIR` disables the code for weak-fairness, and the option `-DCOLLAPSE` enables a state vector compression mode. The run-time option `-m<depth>` of `pan` sets the maximal search depth to `<depth>` steps. The value of `<depth>` we used was `30,000` for systems with two DCMMs, and `50,000,000` for three DCMMs. In the case of the μCRL toolset, first the linearizer was invoked with the option `-regular`, which does not allow the linearizer to introduce infinite data types, and `-tbfile`, to

generate the machine readable linear specification. Next, we use the `instantiatior` to generate the state space.

From Table 1 we can conclude that Spin generates more states per second, but the resulting state space is much larger than the one generated by the μCRL toolset.

The results shown above cannot be interpreted as a direct comparison of state space generation capabilities of Spin and the μCRL toolset, due to the differences in the underlying languages. Namely, the PROMELA code was derived from μCRL code instead of being written directly from the informal description. The PROMELA specification was optimized to use some of the PROMELA features that do not exist in μCRL. On the other hand, some of such features were not deployed for several reasons. The `unless` statement has unclear semantics when used in combination with synchronous communication. An attempt to use channels with nonzero capacity as storage instead of arrays lead to 250% increase of the state space.

We note that a native PROMELA specification of the same protocol was analyzed in [17]. That model is quite different from ours, as it employs most of the PROMELA communication primitives and contains more implementation details. The sizes of the state spaces of the PROMELA specification in [17] are comparable to the sizes of the state spaces of the PROMELA specification presented here. In [17] several incorrect behaviors of the HAVi leader election protocol were found. We found an incorrect behavior of a similar kind in our model of the protocol using simulation of the μCRL specification. This error is due to the fact that a node can be reset in the middle of the leader election as a result of a network change, and there can be a delay before another node may be reset. In this interval the second node can send a message to the first one, and the first one can declare the second one to be the leader, based on the information contained in this message. However, the second node is reset after this, in which case it will attempt to elect a new leader, while the first node will not participate in this election, as it is confident that the leader has already been elected.

It is interesting to note that although both Spin and the μCRL toolset use a similar approach to state space generation, namely exploitation of the reachable state space by analyzing the conditions under which the transitions from a given state are possible, the sizes of the resulting state spaces differ substantially on our specification of the HAVi leader election protocol. Two reasons for such differences are in the preprocessing that is done before the actual state space generation, and in the exact implementations of the algorithms in the toolsets. We believe that in order to uncover the exact differences in the state space generation algorithms of the two toolsets, one needs to have a close look at the source code of the implementations, and try out some small and specifically tailored examples.

Acknowledgements

Thanks go to Dragan Boshnachki and Judi Romijn for carefully reading and commenting on the PROMELA specification and the rest of the paper, as well as to Jan

Bergstra, Jan Friso Groote and Andre van Delft for helpful discussions. Many thanks to the anonymous referees, who helped to improve the structure of the paper, and to Wan Fokkink, who suggested improvements regarding the use of English and style.

Appendix A. μCRL source [5]

```
 1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 2 %%%    Constants, Parameters                         %%%
 3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 4 map
 5    nB:->NAT           % Limit for Buffer capasity
 6    initNDCMM:->NAT     % Initial Number of processes
 7    initNst:->ABI       % Initial Network status
 8    initURLS:->ABI      % Initial URL processes status
 9 rew
10    nB=2
11    initNDCMM=3
12    initNst=seton(0_0,0)
13    initURLs=seton(0_0,1)
14 map
15   il: NAT#ABI->NAT
16   fl: NAT#ABI#ABI->NAT
17 var
18   N: NAT
19   nst,URLs: ABI
20 rew
21   il(N,nst)=if(eq(nst,0_0),0,min_on(nst))      %Minimal on
22   fl(N,nst,URLs)=if(eq(nst,0_0),0,             %Minimal URL on or minimal
23                       if(eq(URLs,0_0),min_on(nst), %on if there is no URL.
24                        min_on(URLs)))
25
26 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27 %%%    Bool                                          %%%
28 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29 sort Bool
30 func
31   T,F:  -> Bool
32 map
33   and: Bool#Bool      -> Bool
34   or:  Bool#Bool      -> Bool
35   not: Bool           -> Bool
36   if:  Bool#Bool#Bool -> Bool
37   eq:  Bool#Bool      -> Bool
38 var
39   b,b1,b2: Bool
40 rew
41   and(T,b)=b     and(b,T)=b
42   and(b,F)=F     and(F,b)=F
```

[5] Note that the source code can also be obtained from `http://www.cwi.nl/~ysu/sources/HAVi` or by contacting the author.

```
43   or(T,b)=T       or(b,T)=T
44   or(b,F)=b       or(F,b)=b
45   not(F)=T        not(T)=F
46   if(T,b1,b2)=b1 if(F,b1,b2)=b2
47   eq(F,F)=T       eq(F,T)=F
48   eq(T,F)=F       eq(T,T)=T
49
50 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
51 %%%     NAT                                                        %%%
52 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
53 sort NAT
54 func
55   0:          -> NAT
56   x2p1:   NAT -> NAT
57   _x2p0:  NAT -> NAT
58 map
59   x2p0:   NAT      -> NAT
60   eq:     NAT#NAT -> Bool
61   1,2,3,4,5,6:          -> NAT
62   succ:    NAT          -> NAT
63   gt:      NAT#NAT      -> Bool
64   if:      Bool#NAT#NAT-> NAT
65 var
66   n,m: NAT
67 rew
68   x2p0(0)=0
69   x2p0(x2p1(n))=_x2p0(x2p1(n))
70   x2p0(_x2p0(n))=_x2p0(_x2p0(n))
71
72   eq(0,0)=T
73   eq(x2p1(n),0)=F
74   eq(0,x2p1(n))=F
75   eq(_x2p0(n),0)=F
76   eq(0,_x2p0(n))=F
77   eq(x2p1(n),_x2p0(m))=F
78   eq(_x2p0(n),x2p1(m))=F
79   eq(_x2p0(n),_x2p0(m)=eq(n,m)
80   eq(x2p1(n),x2p1(m))=eq(n,m)
81
82   1=x2p1(0)  2=_x2p0(1)
83   3=x2p1(1)  4=_x2p0(2)
84   5=x2p1(2)  6=_x2p0(3)
85
86   succ(0)=x2p1(0)
87   succ(x2p1(n))=_x2p0(succ(n))
88   succ(_x2p0(n))=x2p1(n)
89
90   gt(0,n)=F gt(x2p1(n),0)=T gt(_x2p0(n),0)=T
91
92   gt(x2p1(n),_x2p0(m))=not(gt(m,n))
93   gt(_x2p0(n),x2p1(m)=gt(n,m)
94
95   gt(x2p1(n),x2p1(m))=gt(n,m)
96   gt(_x2p0(n),_x2p0(m))=gt(n,m)
97
```

```
98    if(T,n,m)=n if(F,n,m)=m
99
100  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
101  %%%     ABI(Bool array with NAT indices)                               %%%
102  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
103  sort ABI
104  func
105       0_0     :                ->ABI
106       add     : ABI#NAT        ->ABI
107  map
108       rem     : ABI#NAT        ->ABI
109       upd     : ABI# NAT#Bool->ABI
110       n_on    : ABI            ->NAT
111       min_on  : ABI            ->NAT
112       setoff  : ABI#NAT        ->ABI
113       seton   : ABI#NAT        ->ABI
114       reverse: ABI#NAT         ->ABI
115       acc     : ABI#NAT        ->Bool
116       eq      : ABI#ABI        ->Bool
117       if      : Bool#ABI#ABI  ->ABI
118  var
119       n,m:NAT
120       abi,abi1:ABI
121       b1,b2:Bool
122  rew
123       rem(0_0,n)=0_0
124       rem(add(abi,m),n)=if(gt(m,n),add(abi,m),if(eq(n,m),abi,add(rem(abi,n),m)))
125
126       upd(0_0,n,F)=0_0
127       upd(0_0,n,T)=add(0_0,n)
128       upd(add(abi,m),n,F)=rem(add(abi,m),n)
129       upd(add(abi,m),n,T)=if(gt(m,n),add(add(abi,m),n),
130                                      if(eq(n,m),add(abi,m),add(upd(abi,n,T),m)))
131
132       n_on(0_0)=0 n_on(add(abi,n))=succ(n_on(abi))
133
134       min_on(0_0)=0 min_on(add(abi,n))=n
135
136       seton(abi,n)=upd(abi,n,T)  setoff(abi,n)=upd(abi,n,F)
137
138       reverse(abi,n)=upd(abi,n,not(acc(abi,n)))
139
140       acc(0_0,n)=F
141       acc(add(abi,m),n)=if(gt(m,n),F,if(eq(m,n),T,acc(abi,n)))
142
143       eq(0_0,0_0)=T eq(0_0,add(abi,n))=F eq(add(abi,n),0_0)=F
144       eq(add(abi,n),add(abi1,m))=and(eq(n,m),eq(abi,abi1))
145
146       if(T,abi,abi1)=abi if(F,abi,abi1)=abil
147
148  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
149  %%%     Messages                                                       %%%
150  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
151  sort Message
152  func
153        NetworkReset            : ABI       -> Message
```

```
154        DMCapabilityDeclaration : NAT#Bool -> Message
155        DMLeaderDeclaration     : NAT#ABI  -> Message
156  map
157        eq:Message#Message->Bool
158  var
159        n,m:NAT
160        abi,abi1:ABI
161        b1,b2:Bool
162  rew
163        eq(NetworkReset(abi),NetworkReset(abi1))=eq(abi,abi1)
164        eq(DMCapabilityDeclaration(n,b1),DMCapabilityDeclaration(m,b2))
165          =and(eq(n,m),eq(b1,b2))
166        eq(DMLeaderDeclaration(n,abi),DMLeaderDeclaration(m,abi1))
167          =and(eq(n,m),eq(abi,abi1))
168        eq(NetworkReset(abi),DMCapabilityDeclaration(n,b1))=F
169        eq(NetworkReset(abi),DMLeaderDeclaration(n,abi1))=F
170        eq(DMCapabilityDeclaration(n,b1),NetworkReset(abi))=F
171        eq(DMCapabilityDeclaration(n,b1),DMLeaderDeclaration(m,abi))=F
172        eq(DMLeaderDeclaration(n,abi),NetworkReset(abi1))=F
173        eq(DMLeaderDeclaration(n,abi),DMCapabilityDeclaration(m,b1))=F
174
175  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
176  %%%     Status                                                   %%%
177  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
178  sort Status
179  func
180    INIT,LE,LEIF,LEIL,LEILS,AOS,AO:->Status
181  map
182    n:Status->NAT
183    eq:Status#Status->Bool
184  rew
185    n(INIT)=0 n(LE)=1 n(LEIF)=2 n(LEIL)=3 n(LEILS)=4 n(AOS)=5 n(AO)=6
186  var a,b:Status
187  rew eq(a,b)=eq(n(a),n(b))
188
189  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
190  %%%     Actions                                                  %%%
191  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
192  act
193  _flip, flip_on, flip_off,__flip:NAT
194  _on, _off, on, off, __on, __off:NAT
195  _send, send, _rcv, rcv, __send, __rcv:NAT#Message
196  _reset, reset, __reset:NAT#ABI
197  _reset_off, reset_off, __reset_off:NAT
198  _leader:NAT#NAT
199  j
200
201  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
202  %%%   Communication Function                                     %%%
203  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
204  comm
205  _flip|flip_on=__flip
206  _flip|flip_off=__flip
207  _on|on=__on
208  _off|off=__off
209  _send|send=__send
```

```
210 _rcv|rcv=__rcv
211 _reset|reset=__reset
212 _reset_off|reset_off=__reset_off
213
214 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
215 %%%  DCMM Process                                                  %%%
216 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
217 proc
218 DCMM(St:Status, URL:Bool, n:NAT, N:NAT, nst:ABI, wait:ABI, URLs:ABI,
219      il:NAT, fu:NAT, am_on:Bool)=
220   flip_on(n)._on(n).DCMM(INIT,URL,n,N,0_0,0_0,0_0,0,0,T)
221      <|not(am_on)|>delta
222  +
223   sum(nst1:ABI,rcv(n,NetworkReset(nst1)).DCMM(LE,URL,n,N,nst1,0_0,0_0,0,0,T))
224      <|am_on|>delta
225  +
226   flip_off(n)._off(n).DCMM(INIT,URL,n,N,0_0,0_0,0_0,0,0,F)
227      <|am_on|>delta
228  +
229
230  ( _leader(n,n).DCMM(A0,URL,n,N,nst,0_0,upd(0_0,n,URL),0,n,T)
231       <|eq(n_on(nst),1)|> delta
232     +
233     _send(il(N,nst) ,DMCapabilityDeclaration(n,URL))
234         .DCMM(LEIF,URL,n,N,nst,0_0,0_0,il(N,nst),0,T)
235       <|not(eq(il(N,nst),n))|> delta
236     +
237     ( sum(m:NAT,sum(d:Bool, (
238           rcv(n,DMCapabilityDeclaration(m,d))
239            .DCMM(LEILS,URL,n,N,nst,setoff(nst,n),upd(upd(0_0,n,URL),m,d),0,
240              fl(N,nst,upd(upd(nst,n,URL),m,d)),T)
241      <|eq(n_on(nst),2)|>
242          rcv(n,DMCapabilityDeclaration(m,d))
243            .DCMM(LEIL,URL,n,N,nst,setoff(setoff(nst,n),m),
244              upd(upd(0_0,n,URL),m,d)0,0,T))))
245     )<|and(eq(il(N,nst),n),not(eq(n_on(nst),1)))|>delta
246     +
247     sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1))))
248         .DCMM(LE,URL,n,N,nst,0_0,0_0,0,0,T)
249  )<|eq(St,LE)|>delta
250  +
251
252  ( _send(il,DMCapabilityDecalaration(n,URL))
253         .DCMM(LEIF,URL,n,N,nst,0_0,0_0,il,0,T)
254    +
255     sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1))
256         .DCMM(AOS,URL,n,N,nst,0_0,URLs1,0,m,T)))
257    +
258     sum(m:NAT,sum(d1:Bool,rcv(n,DMCapabilityDeclaration(m,d1))))
259         .DCMM(LEIF,URL,n,N,nst,0_0,0_0,il,0,T)
260  )<|eq(St,LEIF)|>delta
261  +
262
263  ( sum(m:NAT,sum(d:Bool,(
264     rcv(n,DMCapabilityDeclaration(m,d))
```

```
265        ·DCMM(LEILS,URL,n,N,nst,setoff(nst,n),upd(URLS,m,d),0,
                fl(N,nst,upd(URLs,m,d)),T)
266       <|and(eq(n_on(wait),1),acc(wait,m))|>
267        rcv(n,DMCapabilityDeclaration(m,d))
268           ·DCMM(LEIL,URL,n,N,nst,setoff(wait,m),upd(URLS,m,d),0,0,T)))
269     +
270      sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1))))
271         ·DCMM(LEIL,URL,n,N,nst,wait,URLs,0,0,T)
272  )<|eq(St,LEIL)|>delta
273   +
274
275  (  sum(m:NAT,(
276        _send(m,DMLeaderDeclaration(fl,URLs))
277          ·DCMM(LEILS,URL,n,N,nst,setoff(wait,m),URLs,0,fl,T)
278       <|and(not(eq(m,fl)),gt(n_on(wait),1))|> delta
279        +
280        _send(m,DMLeaderDeclaration(fl,URLs))
281         ·DCMM(AOS,URL,n,N,nst,0_0,URLs,0,fl,T)
282       <|eq(n_on(wait),1)|> delta
283              )<|acc(wait,m)|>delta)
284   +
285    sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1))))
286         ·DCMM(LEILS,URL,n,N,nst,wait,URLs,0,fl,T)
287   +
288    sum(m:NAT,sum(d1:Bool,rcv(n,DMCapabilityDeclaration(m,d1))))
289         ·DCMM(LEILS,URL,n,N,nst,wait,URLs,0,fl,T)
290  )<|eq(St,LEILS)|>delta
291   +
292    _leader(n,fl).DCMM(AO,URL,n,N,nst,0_0,URLs,0,fl,T)
293      <|eq(St,AOS)|>delta
294   +
295    j.DCMM(AO,URL,n,N,nst,0_0,URLs,0,fl,T)
296      <|eq(St,AO)|>delta
297
298  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
299  %%%   Env Process                                                  %%%
300  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
301  Env(N:NAT,nst:ABI)=sum(m:NAT,(_flip(m).Env(N,reverse(nst,m))
302                               +_flip(m).delta<|gt(n_on(reverse(nst,m)),0)|>delta
303                               )<|gt(N,m)|>delta)
304
305  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
306  %%%   Bus Process                                                  %%%
307  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
308  Bus(N:NAT,nstat:ABI)=
309    sum(m:NAT,on(m).Bus1(N,seton(nstat,m),seton(nstat,m)))
310   +
311    sum(m:NAT,off(m)._reset_off(m)
312      ·(Bus(N,setoff(nstat,m))<|eq(n_on(nstat),1)|>
313       Bus1(N,setoff(nstat,m),setoff(nstat,m))))
314
315  Bus1(N:NAT,nstat:ABI,wait:ABI)=
316    sum(m:NAT,_reset(m,nstat).(Bus(N,nstat) <|eq(n_on(wait),1)|>
317                      Bus1(N,nstat,setoff(wait,m)))<|acc(wait,m)|>delta)
318
```

```
319 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
320 %%%   Message Queues                                               %%%
321 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
322 sort QMes
323 func empty    :                 -> QMes
324      and      : QMes#Message -> QMes
325
326 map  first    : QMes            -> Message
327      remfirst : QMes            -> QMes
328      is_empty : QMes            -> Bool
329      size     : QMes            -> NAT
330 var
331       mes1,mes2:Message
332       q: QMes
333  rew
334       first(add(empty,mes1))=mes1
335       first(add(add(q,mes2),mes1))=first(add(q,mes2))
336       remfirst(add(empty,mes1))=empty
337       remfirst(add(add(q,mes2),mes1))=add(remfirst(add(q,mes2)),mes1)
338       is_empty(empty)=T
339       is_empty(add(q,mes1))=F
340       size(empty)=0
341       size(add(q,mes1))=succ(size(q))
342
343  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
344  %%%  Buffer Process                                                %%%
345  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
346  proc
347  Buffer(N:NAT,n:NAT,q:QMes)=
348    sum(mes:Message,send(n,mes).Buffer(N,n,add(q,mes))) <|gt(nB,size(q))|> delta
349   +
350    _rcv(n,first(q)).Buffer(N,n,remfirst(q)) <|not(is_empty(q))|> delta
351   +
352    sum(nst1:ABI,reset(n,nst1).Buffer(N,n,add(empty,NetworkReset(nst1))))
353   +
354    reset_off(n).Buffer(N,n,empty)
355
356  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
357  %%%  The Whole System                                              %%%
358  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
359  SYSTEMDCMM(N:NAT,nstat:ABI,URLs:ABI)=
360  encap({_flip,flip_on,flip_off},
361    hide({j,__on,__off,__reset,__reset_off},
362    encap({_on,on,_off,off,_reset,reset,_reset_off,reset_off},
363        hide({__send},encap({_send,send},
364          (hide({__rcv},encap({_rcv,rcv},
365            DCMM(INIT,acc(URLs,0),0,N,0_0,0_0,0_0,0_0,0,0,acc(nstat,0))||
366            Buffer(N,0,empty))))
367           ||
368          (hide({__rcv},encap({_rcv,rcv},
369            DCMM(INIT,acc(URLs,1),1,N,0_0,0_0,0_0,0_0,0,0,acc(nstat,1))||
370            Buffer(N,1,empty))))
371           ||
372          (hide({__rcv},encap({_rcv,rcv},
373            DCMM(INIT,acc(URLs,2),2,N,0_0,0_0,0_0,0_0,0,0,acc(nstat,2))||
```

```
374           Buffer(N,2,empty))))
375       ))
376       ||
377       Bus(N,nstat)
378   ))
379   ||
380   Env(N,nstat)
381 )
382
383 init SYSTEMDCMM(initNDCMM,initNst,initURLs)
```

Appendix B. PROMELA source[6]

```
1  #define initNDCMM 3
2  #define nB 2
3
4  typedef ABI {bool a[initNDCMM]};
5  mtype = {NetworkReset, DMCapabilityDeclaration, DMLeaderDeclaration};
6  typedef Message {mtype MTYPE; byte NN; bool URL; ABI NST};
7
8  chan on = [0] of {byte};
9  chan off = [0] of {byte};
10 chan send[initNDCMM] = [0] of {Message};
11 chan rcv[initNDCMM] = [0] of {Message};
12 chan reset[initNDCMM] = [0] of {ABI};
13 chan reset_off[initNDCMM] = [0] of {bit};
14 chan flip[initNDCMM] = [0] of {bit};
15 chan leader = [0] of {byte, byte}
16
17 chan env = [0] of {ABI}
18 chan bus = [0] of {ABI} /* Due to the technical restrictions of spin we
19 cannot pass arrays as parameters for processes. So we use these channels to
20 pass nst to Env and Bus */
21
22 /* inlines use and sideeffect variable _i
23 (assumed that it is defined as byte) */
24
25 /* copies N first elements of array B
26 to the corresponding elements of A */
27 inline array_assign(A, B, N)
28 { _i=0; do
29         :: _i<N -> A.a[_i]=B.a[_i]; _i=_i+1
30         :: else -> break
31         od; _i=0;}
32
33 /* m := minimal m s.t. A[m].
34 0 if all elements of A are false */
35 inline array_min_true(A, N, m)
36 { _i=0; do
37         :: (_i<N) -> if
38                       :: !A.a[_i] -> _i=_i+1
```

[6] Note that the source code can also be obtained from `http://www.cwi.nl/~ysu/sources/HAVi` or by contacting the author.

```
39                      :: else -> break
40                      fi;
41          :: else -> break
42          od; m = (_i==N -> 0 : _i); _i=0;}
43
44  /* n_on := number of true elements of A */
45  inline array_n_true(A, N, n_on)
46  { n_on=0; _i=0; do
47                 ::(_i<N) -> n_on=(A.a[_i]->n_on+1:n_on);_i=_i+1
48                 ::else -> break
49                 od; _i=0;}
50
51  /* assign false to N first elements of A*/
52  inline array_false(A, N)
53  { _i=0; do
54          ::(_i<N) -> A.a[_i]=false; _i=_i+1
55          ::else -> break
56          od; _i=0;}
57
58  #define NETWORK_RESET_WAIT_URLS rcv[n]?NetworkReset,_,ib,nst;\
59  atomic{d_step{array_false(wait,N);array_false(URLs,N);\
60  il=0;fl=0;m=0;n_on=0};goto LE}
61
62  #define NETWORK_RESET_URLS rcv[n]?NetworkReset,_,ib,nst;\
63  atomic{d_step{array_false(URLs,N);\
64  il=0;fl=0;m=0;n_on=0};goto LE}
65
66  #define NETWORK_RESET rcv[n]?NetworkReset,_,ib,nst;\
67  atomic{d_step{il=0;fl=0;m=0;n_on=0};goto LE}
68
69  #define FLIP_OFF_NST_WAIT_URLS flip[n]?1;off!n;\
70  atomic{d_step{array_false(nst,N);array_false(wait,N);array_false(URLs,N);\
71  il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}
72
73  #define FLIP_OFF_NST_URLS flip[n]?1;off!n;\
74  atomic{d_step{array_false(nst,N);array_false(URLs,N);\
75  il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}
76
77  #define FLIP_OFF_NST flip[n]?1;off!n;\
78  atomic{d_step{array_false(nst,N);\
79  il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}
80
81  #define FLIP_OFF flip[n]?1;off!n;\
82  atomic{d_step{il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}
83
84  bool ib; hidden ABI iabi;
85
86  /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
87  %%%      DCMM Process                                          %%%
88  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
89  proctype DCMM(bool URL; byte n, N; bool _am_on)
90  {  bool am_on; ABI nst, wait, URLs;
91     byte il,fl,m,n_on; bool d; byte _i;
92
93     d_step{ am_on=_am_on; array_false(nst,N); array_false(wait,N);
94             array_false(URLs,N); il=0; fl=0; m=0; n_on=0; d=false; _i=0;}
```

```
 95  INIT:
 96    if
 97    :: !am_on -> flip[n]?1; on!n; atomic {am_on=true; goto INIT}
 98    :: am_on -> if
 99                :: NETWORK_RESET
100                :: FLIP_OFF
101                fi;
102    fi;
103
104  LE:
105    atomic{
106      d_step{array_min_true(nst,N,il);} /* il calculation */
107      if
108      :: il==n -> d_step{array_assign(wait,nst,N); wait.a[n]=false;
109                          URLs.a[n]=URL; il=0;} goto LEIL;
110      :: else
111      fi;}
112
113  LE1:
114    if
115    :: send[il]!DMCapabilityDeclaration(n,URL,iabi); goto LEIF
116    :: rcv[n]?DMLeaderDeclaration,_,ib,iabi; goto LE1;
117    :: rcv[n]?DMCapabilityDeclaration,_,ib,iabi; goto LE1;
118    :: NETWORK_RESET
119    :: FLIP_OFF_NST
120    fi;
121
122  LEIF:
123    if
124    :: send[il]!DMCapabilityDeclaration(n,URL,iabi); goto LEIF
125    :: rcv[n]?DMLeaderDeclaration,fl,ib,URLs; goto AOS
126    :: rcv[n]?DMCapabilityDeclaration,_,ib,iabi; goto LEIF
127    :: NETWORK_RESET
128    :: FLIP_OFF_NST
129    fi;
130
131  LEIL:
132    atomic{d_step{array_n_true(wait,N,n_on);}
133  LEIL1:
134      if
135      :: n_on==0 -> d_step{array_assign(wait,nst,N);
136                    wait.a[n]=false;
137
138                    array_min_true(nst,N,fl);
139                    array_min_true(URLs,N,m); /* final leader calculation */
140                    fl=(m==0->fl:m); m=0;}
141
142                    goto LEILS;
143      :: else
144      fi;}
145
146  LEIL2:
147    if
148    :: rcv[n]?DMCapabilityDeclaration,m,d,iabi;
149       atomic{d_step{n_on=(wait.a[m]->n_on-1:n_on);
```

```
150          wait.a[m]=false; URLs.a[m]=d; m=0; d=false}; goto LEIL1; }
151     :: rcv[n]?DMLeaderDeclaration,_,ib,iabi; goto LEIL2;
152     :: NETWORK_RESET_WAIT_URLS
153     :: FLIP_OFF_NST_WAIT_URLS
154     fi;
155
156  LEILS:
157     atomic{d_step{m=0; d=true;} /* final leader is informed the last */
158
159  LEILS1:
160       if
161       :: (d && (m==fl || (m<N && !wait.a[m])) ) -> m=m+1; goto LEILS1;
162       :: (m==N) -> d_step{d=false; m=fl} goto LEILS1;
163       :: (m==fl && !d && !wait.a[m]) -> m=0; goto AOS;
164       :: else
165       fi;}
166
167  LEILS2:
168     if
169     :: send[m]!DMLeaderDeclaration(fl,false,URLs);
170          d_step{wait.a[m]=false; m=(m==fl->m:m+1)} goto LEILS1;
171     :: rcv[n]?DMLeaderDeclaration,_,ib,iabi; goto LEILS2;
172     :: rcv[n]?DMCapabilityDeclaration,_,ib,iabi; goto LEILS2;
173     :: NETWORK_RESET_WAIT_URLS
174     :: FLIP_OFF_NST_WAIT_URLS
175     fi;
176
177  AOS:
178     if
179     :: leader!n,fl; goto AO;
180     :: NETWORK_RESET_URLS
181     :: FLIP_OFF_NST_URLS
182     fi;
183
184  AO:
185     if
186     :: NETWORK_RESET_URLS
187     :: FLIP_OFF_NST_URLS
188     :: goto AO;
189     fi;
190  }
191
192  /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
193  %%%      Bus Process                                           %%%
194  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
195  proctype Bus(byte N)
196  { ABI nst, wait; byte m, n_on, n_on_wait; byte _i;
197
198     d_step{array_false(nst,N); array_false(wait,N); m=0; n_on=0; _i=0;}
199     bus?nst;
200     d_step{array_n_true(nst,N,n_on);}
201
202  Bus_:
203     if
204     :: n_on==0 -> on?m; atomic{d_step{nst.a[m]=true; m=0; n_on=1;} goto Bus1}
```

```
205    :: else ->
206       if
207       :: on?m; atomic{d_step{nst.a[m]=true; m=0; n_on=n_on+1;} goto Bus1}
208       :: off?m; reset_off[m]!1;
209            atomic{d_step{nst.a[m]=false; m=0; n_on=n_on-1;} goto Bus1}
210         fi;
211    fi;
212
213  Bus1:
214    atomic{
215      if
216      ::  (m==N) -> m=0; goto Bus_;
217      ::  (m<N && !nst.a[m]) -> m=m+1; goto Bus1;
218      ::  else
219      fi;}
220
221    reset[m]!nst; atomic{m=m+1; goto Bus1};
222  }
223
224  #define BUFFER_RESET reset[n]?nst;atomic{d_step{queue_clean(nIn);\
225  queue[0].MTYPE=NetworkReset;array_assign(queue[0].NST,nst,N);\
226  array_false(nst,N);nIn=1}; goto Buffer_}
227
228  #define BUFFER_RESET_OFF reset_off[n]?1;\
229  atomic{d_step(queue_clean(nIn); nIn=0}; goto Buffer_}
230
231  /* inlines below use and sideeffect variable _j
232  (assumed that it is defined as byte) */
233
234  /* shifts queue[1..nIn-1] to queue[0..nIn-2]
235  (if nIn<=1 does nothing) */
236  inline queue_shift()
237  { _j=1;  do
238           ::_j<nIn-> queue[_j-1].MTYPE=queue[_j].MTYPE;
239                      queue[_j-1].NN=queue[_j].NN;
240                      queue[_j-1].URL=queue[_j].URL;
241                      array_assign(queue[_j-1].NST,queue[_j].NST,N);
242                      _j=_j+1
243           ::else-> break
244    od; _j=0;}
245
246  /* assignes default values to queue elements */
247  inline queue_clean(NNN)
248  { _j=0;  do
249           :: _j<NNN -> queue_clean_element(_j); _j=_j+1
250           :: else -> break
251           od; _j=0;}
252
253  /* assignes default value to an element */
254  inline queue_clean_element(el)
255  { queue[el].MTYPE=NetworkReset;
256    queue[el].NN=0;
257    queue[el].URL=false;
258    array_false(queue[el].NST,N);}
259
```

```
260 /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
261 %%%      Buffer Process                                       %%%
262 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
263 proctype Buffer(byte n, N)
264 { byte nIn,_i,_j; Message queue[nB]; ABI nst;
265
266   d_step(nIn=0; array_false(nst,N); queue_clean(nB); _i=0; _j=0;}
267
268 Buffer_:
269   if
270   :: (nIn<nB && nIn>0) ->
271        if
272        ::  send[n]?queue[nIn];
273            atomic{nIn=nIn+1; goto Buffer_};
274        ::  rcv[n]!queue[0];
275            atomic{d_step{queue_shift(); nIn=nIn-1;
276               queue_clean_element(nIn);} goto Buffer_}
277        ::  BUFFER_RESET
278        ::  BUFFER_RESET_OFF
279        fi;
280   :: (nIn==nB) -> if
281                   :: rcv[n]!queue[0];
282                      atomic{d_step{queue_shift(); nIn=nIn-1;
283                      queue_clean_element(nIn);} goto Buffer_}
284                   :: BUFFER_RESET
285                   :: BUFFER_RESET_OFF
286                   fi;
287   :: (nIn==0) -> if
288                  :: send[n]?queue[nIn]; atomic{nIn=1; goto Buffer_}
289                  :: BUFFER_RESET
290                  :: BUFFER_RESET_OFF
291                  fi;
292   fi;
293 }
294
295 /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
296 %%%      Env Process                                                 %%%
297 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
298 proctype Env(byte N)
299 { ABI nst; byte n_on,j; byte _i;
300
301   d_step{j=0; n_on=0; array_false(nst,N);_i=0}
302   env?nst;
303 Env_:
304   if
305   :: flip[j]!1;
306      atomic{ d_step{nst.a[j]=!nst.a[j]; j=0; array_n_true(nst,N,n_on);}
307              if
308              ::  (n_on) -> d_step{n_on=0; array_false(nst,N);} goto Env_End;
309              ::  (true) -> n_on=0; goto Env_;
310              fi;
311             }
312   :: leader?_,_; atomic{j=0; goto Env_;}
313   :: (j<(N-1)) -> atomic{j=j+1; goto Env_;}
314   fi;
315 Env_End:
```

```
316   leader?_,_; goto Env_End;
317 }
318
319 /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
320 %%%      Init                                                        %%%
321 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
322
323 init
324 { ABI nst, URLs; byte j; byte _i;
325   atomic{
326     d_step{ array_false(nst,initNDCMM); array_false(URLs,initNDCMM);
327             nst.a[0]=true; URLs.a[1]=true; j=0;}
328     do
329     :: j<initNDCMM -> run DCMM(URLs.a[j],j,initNDCMM,nst.a[j]);
330                      run Buffer(j,initNDCMM); j=j+1;
331     :: else -> break;
332     od; j=0;
333     run Bus(initNDCMM); run Env(initNDCMM);
334   }
335   bus!nst; env!nst;
336 }
```

References

[1] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, Cambridge, August 1999.

[2] J.C.M. Baeten, C. Verhoef, Concrete process algebra, in: S. Abramsky, D. Gabbay, T.S.E. Maibaum (Eds.), Handbook of Logic in Computer Science, Vol. 4, Chap. 2, Oxford University Press, Oxford, 1994.

[3] J.C.M. Baeten, W.P. Weijland, in: Process Algebra, Cambridge Tracts in Theoretical Computer Science, Vol. 18, Cambridge University Press, Cambridge, 1990.

[4] D. Dams, J.F. Groote, Specification and implementation of components of a μCRL toolbox, Logic Group Preprint Series 152, Department of Philosophy, Utrecht University, December 1995. Under revision for FAC.

[5] J.-C. Fernandez, H. Garavel, R. Mateescu, A. Kerbrat, L. Mounier, M. Sighireanu, CADP: a protocol validation and verification toolbox, Proc. 8th Conf. on Computer-Aided Verification, New Brunswick, NJ, USA, August 1996, pp. 437–440.

[6] J.F. Groote, The syntax and semantics of timed μCRL, Tech. Report SEN-R9709, CWI, Amsterdam, June 1997.

[7] J.F. Groote, B. Lisser, Tutorial and Reference Guide for the μCRL toolset version 1.0, CWI, Amsterdam, 1999. Available from URL http://www.cwi.nl/~mcrl/mutool.html.

[8] J.F. Groote, F. Monin, J. Springintveld, A computer checked algebraic verification of a distributed summation algorithm, Tech. Report 97-14, Department of Mathematics and Computing Science, Eindhoven University of Technology, October 1997.

[9] J.F. Groote, A. Ponse, The syntax and semantics of μCRL, in: A. Ponse, C. Verhoef, S.F.M. van Vlijmen (Eds.), Algebra of Communicating Processes 1994, Workshop in Computing Series, Springer, Berlin, 1995, pp. 26–62.

[10] J.F. Groote, A. Ponse, Y.S. Usenko, Linearization in Parallel pCRL, Tech. Report SEN-R0019, CWI, July 2000. Accepted by JLAP.

[11] Grundig, Hitachi, Matsushita, Philips, Sharp, Sony, Thomson, Toshiba, Specification of the Home Audio/Video Interoperability (HAVi) Architecture, November 19 1998. Version 1.0beta.

[12] G.J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[13] G.J. Holzmann, The model checker SPIN, IEEE Trans. Software Eng. 23 (5) (1997) 279–295.
[14] Bell Labs, Spin version 3.3: language reference. WWW page. http://cm.bell-labs.com/cm/cs/what/spin/Man/promela.html.
[15] Bell Labs, Spin newsletter. http://netlib.bell-labs.com/netlib/spin/news/news5.html#C, May 1995. No. 5.
[16] A. Pnueli, The temporal logic of programs, Proc. 18th IEEE Symp. on Foundation of Computer Science, 1977, pp. 46–57.
[17] J.M.T. Romijn, Model checking the HAVi leader election protocol, Tech. Report SEN-R9915, CWI, Amsterdam, 1999. Under revision for FAC.