Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

A Comparison of Spin and the muCRL Toolset on HAVi Leader Election Protocol

Y.S. Usenko

Software Engineering (SEN)

SEN-R9917 July 31, 1999

Report SEN-R9917
ISSN 1386-369X

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

A Comparison of Spin and the μCRL Toolset on HAVi Leader Election Protocol

Y.S. Usenko
Yaroslav.Usenko@cwi.nl
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
This paper describes an attempt to compare two toolsets allowing generation of finite labeled transition systems from underlying concurrent specification languages. The comparison is done on a specification of the leader election protocol from Home Audio/Video interoperability (HAVi) architecture. Some important semantical differences of PROMELA and μCRL are identified, that lead to big differences in size of the state spaces generated for equivalent specifications.

1991 Mathematics Subject Classification: 68Q22; 68Q45; 68Q60
1991 ACM Computing Classification System: D.1.3; D.2.4; D.2.8
Keywords and Phrases: μCRL, SPIN, Tool Comparison, State Space Generation, Specification, Deadlock Checking.
Note: Work carried out under project SEN 2.1 Process Specification and Analysis.

1. Introduction

With current state of the art in software verification a lot of the analysis methods are based on finite labeled transition systems (FLTS). There is a number of tools to manipulate with them, to check different kind of equivalences and preorder, to find deadlocks, to check modal and temporal properties, to minimize in different ways, etc. Therefore one of the major topics that remains for verification tools supporting concurrent languages is how fast can they generate FLTS, and how large these FLTS are.

It appears that for many completely different concurrent languages FLTS can be generated to describe behavior of specifications. Although not for any specification a FLTS can be generated, it is desirable to be able to do so in cases when it is possible. In this paper we consider two toolsets that allow state space generation–one for algebraic concurrent language μCRL [6], and one for imperative concurrent language PROMELA; and try to compare the state spaces generated by them.

The μCRL toolset [4] has been developed at CWI to support formal reasoning about systems specified in μCRL. It is based on term rewriting and linearization techniques. At the moment it allows to generate state spaces, search for deadlocks, do some optimization for μCRL specifications and simulate them. Spin [8] is one of the fastest and widely used tools for protocol verification. It allows formal analysis of PROMELA specifications, generation of state spaces, and searching for deadlocks.

To make a comparison we consider the leader election protocol from Home Audio/Video interoperability (HAVi) Architecture [7]. Previously this protocol has been specified in PROMELA and LOTOS, and analyzed formally [10]. Here we take a more abstract definition of the protocol to keep the specification relatively simple and free of many implementation details.

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

translation to Spin (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. Subsection 3.1 contains an overview of μCRL syntax that can also be found for instance in [5]. For more systematic and available introduction to μCRL see [3]. For systematic treatment of ACP style process algebra, which is the basis of μCRL see [1, 2].

2. Informal Description

The informal description of the protocol appears on pages 160-162 of [7]. The system consists of a number of Device Control Module Managers (DCMM). Each DCMM process has its own input buffer from which it gets incoming messages. All processes communicate with the Bus process and the Env process representing the environment. In [7] the bus is specified as IEEE P1394 Standard for a "High Performance Serial Bus" (FireWire). There is a formal specification of the Link Layer of IEEE P1394 Standard in μCRL[9]. We do not use this specification because of the complexity reasons. The processes and synchronous communications are presented on Figure 1.

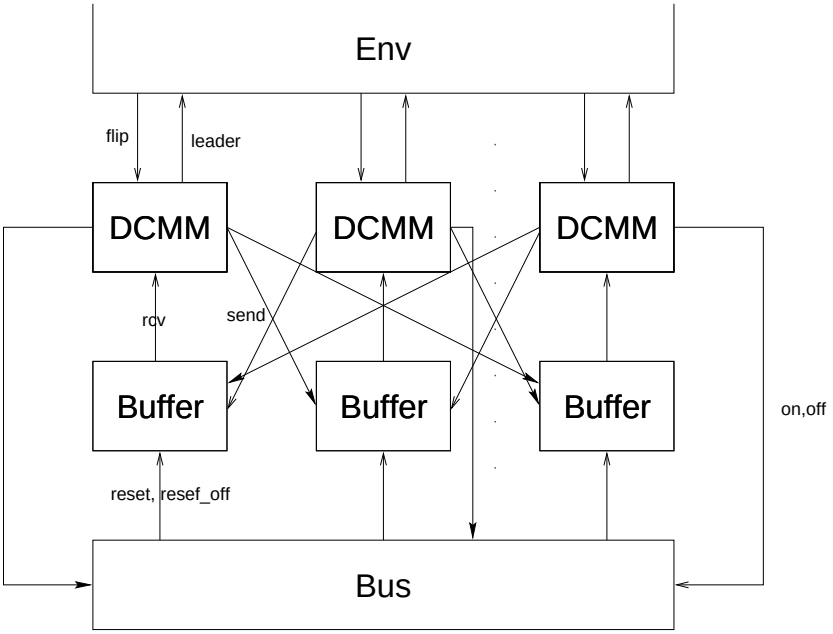

Figure 1: Processes and Communications in the System.

Each DCMM process has its unique ID number by which it can be addressed. The environment may flip (switch on or off) DCMM processes. The Bus observes such changes in the network communicating with the DCMM process that was flipped. It informs all working DCMM processes about changes in the network communicating with their buffers. As the result, the buffers become empty, and deliver the network reset to the corresponding DCMM processes.

The leader election is performed in the following way. After receiving a $NetworkReset(nst)$ message, a DCMM process has the status information about the network from parameter nst. This status information says which processes are "on" within the network. Each DCMM process uses function $il(N, nst)$ to determine the ID of the initial leader. By comparing this ID with its own ID, the DCMM knows if it is an initial leader or an initial follower. In the former case it behaves as follows.

- From each initial follower m it awaits $DMCapabilityDeclaration(m, URL)$ message, from which it knows if the process m has URL capability.
- It uses function $fl(N, nst, URLs)$ to determine the ID of the final leader.
- It sends $DMLeaderDeclaration(m, fl, URLs)$ message to each initial follower m. The final leader is the last one to which this message is sent.
- Finally, it communicates with process Env by leader action.

Initial follower m shall behave as follows.

- It keeps sending $DMCapabilityDeclaration(m, URL)$ message to the initial leader until it gets $DMLeaderDeclaration(m, fl, URLs)$ from it.
- Finally, it communicates with process Env by leader action.

After electing a leader DCMM processes inform the environment about the results of the election.

At any moment of the election any DCMM process may be flipped, or it may receive a *NetworkReset* message. In case a DCMM process was switched on, it awaits for a *NetworkReset* message. In case of receiving a *NetworkReset* message it restarts the election procedure. The DCMM processes 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 more, then each DCMM process will eventually get information about the final leader. And this information is the same for each DCMM process.

3. Specification in μCRL

The complete μCRL specification can be found in Appendix A. The language appears to be suitable for such kind of specification. The specification could be much shorter if there were libraries of standard data types available. The introduction of generic data types might be useful for this, but it introduces technical problems associated with rewriting. Emulation of generic data types using preprocessor might be considered as an alternative approach. The language allows to improve the specification in several ways, but the current tool support (the linearizer), either does not handle this features or produces results of much bigger complexity.

3.1 Overview of syntax

Starting from a set Act of actions that can be parameterized with data, processes are defined by means of guarded recursive equations and the following operators.

First, there is a constant δ ($\delta \notin \mathsf{Act}$) that cannot perform any action 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 successfully terminates continues to behave as y. The process $x + y$ can either do an action of x and continue to behave as x or do an action of y and continue to behave 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 $\gamma : \mathsf{Act} \times \mathsf{Act} \to \mathsf{Act}$ that describes how actions can communicate; parameterized actions $a(d)$ and $b(d')$ communicate to $\gamma(a, b)(d)$, provided $d = d'$. A specification of a process typically contains a specification of a communication function.

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 are blocked. So, assuming $\gamma(a,b) = c$, in $\partial_{\{a,b\}}(x \parallel y)$ 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 directly observed. The hiding operator τ_I renames the actions in the set I to τ. By hiding all internal communications of a process only the external actions remain.

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 some value d selected from the sort D. The conditional operator $\_ \lhd \_ \rhd \_$ describes the *then-if-else*. 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.

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

3.2 Data Types

The sorts $\mathbb{B}$ and $\mathbb{N}$ 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 there is an implementation of FIFO queues for *Message* as sort *Queue*. It is used in 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 may 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
$nB :\rightarrow \mathbb{N}$
$initNDCMM :\rightarrow \mathbb{N}$
$initNst :\rightarrow$ **ABI**
$initURLs :\rightarrow$ **ABI**
rew
$nB = 2$
$initNDCMM = 3$
$initNst = seton(0\_0, 0)$
$initURLs = seton(0\_0, 1)$
map
$il : \mathbb{N} \times$ **ABI** $\rightarrow \mathbb{N}$
$fl : \mathbb{N} \times$ **ABI** $\times$ **ABI** $\rightarrow \mathbb{N}$
var
$N : \mathbb{N}$
$nst, URLs :$ **ABI**
rew
$il(N, nst) = if(eq(nst, 0\_0), 0, min\_on(nst))$
$fl(N, nst, URLs) = if(eq(nst, 0\_0), 0,$
$\quad if(eq(URLs, 0\_0), min\_on(nst), min\_on(URLs)))$

3.2.2 Messages

The sort *Message* is used to define all messages DCMM processes can receive. The use of abstract data types allows us to define messages having different parameters. However, if we had more types of messages, the definition of the equality predicate *eq* would be much more complex.

sort *Message*
func
$NetworkReset : \mathbf{ABI} \rightarrow Message$
$DMCapabilityDeclaration : \mathbb{N} \times \mathbb{B} \rightarrow Message$
$DMLeaderDeclaration : \mathbb{N} \times \mathbf{ABI} \rightarrow Message$
map
$eq : Message \times Message \rightarrow \mathbb{B}$

3.2.3 Status

Sort *Status* is a simple enumerated type to represent 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 such an 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 that status of the DCMM process.

The definition of sort *Status* is a common way to represent enumerated types in μCRL. One could also use the sort $\mathbb{N}$ 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
$n : Status \rightarrow \mathbb{N}$
$eq : Status \times Status \rightarrow \mathbb{B}$
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 $eq(a, b) = eq(n(a), n(b))$

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 "communicate" actions. The communication function is defined according to this intuition.

act

_flip, flip_on, flip_off, __flip	$: \mathbb{N}$
_on, _off, on, off, __on, __off	$: \mathbb{N}$
_send, send, _rcv, rcv, __send, __rcv	$: \mathbb{N} \times Message$
_reset, reset, __reset	$: \mathbb{N} \times \mathbf{ABI}$
_reset_off, reset_off, __reset_off	$: \mathbb{N}$
_leader	$: \mathbb{N} \times \mathbb{N}$

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 it 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.

$\mathsf{DCMM}(St : Status, URL : \mathbb{B}, n : \mathbb{N}, N : \mathbb{N}, nst : \mathbf{ABI},$
$wait : \mathbf{ABI}, URLs : \mathbf{ABI}, il : \mathbb{N}, fl : \mathbb{N}, am\_on : \mathbb{B}) =$

The following alternatives are enabled in any status of the DCMM process. It can be switched on, if it was off. In this case it communicates with Bus process by on action, and it's status becomes $INIT$. If the DCMM process is on, it can receive a $NetworkReset(nst1)$ message and change it's status to LE. Or it can be flipped off, communicate with Bus process by off, and change it's 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 $LEIF$. In case when none of the two above applies, the DCMM process can receive capability declaration from a process m and then, depending on whether it still has to wait for messages from another processes, its status becomes either $LEIL$ or $LEILS$. Finally, the DCMM process ignores any leader declaration messages in this status.

$($ $\mathsf{\_leader}(n, n) \cdot$
$\mathsf{DCMM}(AO, URL, n, N, nst, 0\_0, upd(0\_0, n, URL), 0, n, \mathbf{t}) \lhd n\_on(nst) = 1 \rhd \delta$
$+$
$\mathsf{\_send}(il(N, nst), DMCapabilityDeclaration(n, URL)) \cdot$
$\mathsf{DCMM}(LEIF, URL, n, N, nst, 0\_0, 0\_0, il(N, nst), 0, \mathbf{t}) \lhd il(N, nst) \neq n \rhd \delta$
$+$
$(\sum_{m:\mathbb{N}} \sum_{d:\mathbb{B}}$
$\mathsf{rcv}(n, DMCapabilityDeclaration(m, d)) \cdot$

$$
\begin{array}{l}
\quad \mathsf{DCMM}(\mathit{LEILS}, \mathit{URL}, n, N, \mathit{nst}, \mathit{setoff}(\mathit{nst}, n), \mathit{upd}(\mathit{upd}(0\_0), n, \mathit{URL}), m, d), \\
\qquad 0, \mathit{fl}(N, \mathit{nst}, \mathit{upd}(\mathit{upd}(\mathit{nst}, n, \mathit{URL}), m, d)), \mathbf{t}) \\
\quad \lhd\, \mathit{n\_on}(\mathit{nst}) = 2 \,\rhd \\
\quad \mathsf{rcv}(n, \mathit{DMCapabilityDeclaration}(m, d)) \cdot \\
\quad \mathsf{DCMM}(\mathit{LEIL}, \mathit{URL}, n, N, \mathit{nst}, \mathit{setoff}(\mathit{setoff}(\mathit{nst}, n), m), \\
\qquad \mathit{upd}(\mathit{upd}(0\_0, n, \mathit{URL}), m, d), 0, 0, \mathbf{t}) \\
) \lhd \mathit{il}(N, \mathit{nst}) = n \wedge \mathit{n\_on}(\mathit{nst}) \neq 1 \rhd \delta \\
+ \\
(\textstyle\sum_{m:\mathbb{N}} \sum_{\mathit{URLs1}:\mathbf{ABI}} \mathsf{rcv}(n, \mathit{DMLeaderDeclaration}(m, \mathit{URLs1}))) \cdot \\
\quad \mathsf{DCMM}(\mathit{LE}, \mathit{URL}, n, N, \mathit{nst}, 0\_0, 0\_0, 0, 0, \mathbf{t}) \\
) \lhd \mathit{St} = \mathit{LE} \rhd \delta \\
+
\end{array}
$$

If the status of the DCMM process is *LEIF* it behaves as an initial follower. It can send its capabilities to the initial leader. It can receive a leader declaration. And it ignores any capability declaration messages in this status.

$$
\begin{array}{l}
(\quad \_\mathsf{send}(\mathit{il}, \mathit{DMCapabilityDeclaration}(n, \mathit{URL})) \cdot \\
\quad \mathsf{DCMM}(\mathit{LEIF}, \mathit{URL}, n, N, \mathit{nst}, 0\_0, 0\_0, \mathit{il}, 0, \mathbf{t}) \\
+ \\
\quad \textstyle\sum_{m:\mathbb{N}} \sum_{\mathit{URLs1}:\mathbf{ABI}} \mathsf{rcv}(n, \mathit{DMLeaderDeclaration}(m, \mathit{URLs1})) \cdot \\
\quad \mathsf{DCMM}(\mathit{AOS}, \mathit{URL}, n, N, \mathit{nst}, 0\_0, \mathit{URLs1}, 0, m, \mathbf{t}) \\
+ \\
\quad (\textstyle\sum_{m:\mathbb{N}} \sum_{d1:\mathbb{B}} \mathsf{rcv}(n, \mathit{DMCapabilityDeclaration}(m, d1))) \cdot \\
\quad \mathsf{DCMM}(\mathit{LEIF}, \mathit{URL}, n, N, \mathit{nst}, 0\_0, 0\_0, \mathit{il}, 0, \mathbf{t}) \\
) \lhd \mathit{St} = \mathit{LEIF} \rhd \delta \\
+
\end{array}
$$

If the status of the DCMM process is *LEIL*, it behaves as initial leader. 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.

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

If the status of the DCMM process is *LEILS*, it informs initial followers about the final leader. If the final leader has to be informed, it is informed the last. Any message is ignored by the process in this state. After informing the last initial follower, the status of the process becomes *AOS*.

$$
\begin{array}{l}
(\textstyle\sum_{m:\mathbb{N}} \\
\quad (\quad \_\mathsf{send}(m, \mathit{DMLeaderDeclaration}(\mathit{fl}, \mathit{URLs})) \cdot \\
\qquad \mathsf{DCMM}(\mathit{LEILS}, \mathit{URL}, n, N, \mathit{nst}, \mathit{setoff}(\mathit{wait}, m), \mathit{URLs}, 0, \mathit{fl}, \mathbf{t}) \\
\qquad \lhd\, m \neq \mathit{fl} \wedge \mathit{n\_on}(\mathit{wait}) > 1 \,\rhd \delta
\end{array}
$$

$$
\begin{array}{l}
\quad + \\
\quad\quad \_\mathsf{send}(m, DMLeaderDeclaration(fl, URLs)) \cdot \\
\quad\quad\quad \mathsf{DCMM}(AOS, URL, n, N, nst, 0\_0, URLs, 0, fl, \mathbf{t}) \\
\quad\quad \lhd\, n\_on(wait) = 1 \rhd \delta \\
\quad) \lhd wait[m] \rhd \delta \\
\quad + \\
\quad\quad (\sum_{m:\mathbb{N}} \sum_{URLs1:\mathbf{ABI}} \mathsf{rcv}(n, DMLeaderDeclaration(m, URLs1))) \cdot \\
\quad\quad\quad \mathsf{DCMM}(LEILS, URL, n, N, nst, wait, URLs, 0, fl, \mathbf{t}) \\
\quad + \\
\quad\quad (\sum_{m:\mathbb{N}} \sum_{d1:\mathbb{B}} \mathsf{rcv}(n, DMCapabilityDeclaration(m, d1))) \cdot \\
\quad\quad\quad \mathsf{DCMM}(LEILS, URL, n, N, nst, wait, URLs, 0, fl, \mathbf{t}) \\
\quad) \lhd St = LEILS \rhd \delta \\
+
\end{array}
$$

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

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

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 are "off".

$$
\begin{array}{l}
\mathsf{Env}(N : \mathbb{N}, nst : \mathbf{ABI}) = \sum_{m:\mathbb{N}} \\
\quad (\quad \_\mathsf{flip}(\mathsf{m}) \cdot \mathsf{Env}(N, reverse(nst, m)) \\
\quad + \_\mathsf{flip}(\mathsf{m}) \cdot \delta \lhd n\_on(reverse(nst, m)) > 0 \rhd \delta \\
\quad) \lhd N > m \rhd \delta
\end{array}
$$

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. Process Bus can communicate with a DCMM process by action on or off to observe that a process was flipped. Process Bus1 is used to reset buffers of all active processes in no particular order.

$$
\begin{array}{l}
\mathsf{Bus}(N : \mathbb{N}, nstat : \mathbf{ABI}) = \\
\quad \sum_{m:\mathbb{N}} \mathsf{on}(m) \cdot \mathsf{Bus1}(N, seton(nstat, m), seton(nstat, m)) \\
+ \\
\quad \sum_{m:\mathbb{N}} \mathsf{off}(m) \cdot \_\mathsf{reset\_off}(m) \cdot \\
\quad\quad (\mathsf{Bus}(N, setoff(nstat, m)) \lhd n\_on(nstat) = 1 \rhd \\
\quad\quad\quad \mathsf{Bus1}(N, setoff(nstat, m), setoff(nstat, m))) \\
\mathsf{Bus1}(N : \mathbb{N}, nstat : \mathbf{ABI}, wait : \mathbf{ABI}) = \\
\quad \sum_{m:\mathbb{N}} \_\mathsf{reset}(m, nstat) \cdot (\mathsf{Bus}(N, nstat) \lhd n\_on(wait) = 1 \rhd \\
\quad\quad \mathsf{Bus1}(N, nstat, setoff(wait, m))) \lhd wait[m] \rhd \delta
\end{array}
$$

3.4.4 Buffer

Process Buffer is a FIFO queue of capacity nB that can be reset in two different ways. It can receive a message if it is not full, or send a message if it is not empty. It can communicate with the Bus by action reset or reset_off. In the first case it clears its message queue, and puts the network reset message in it. In the second case it just clears the queue.

$$
\begin{aligned}
&\mathsf{Buffer}(N:\mathbb{N}, n:\mathbb{N}, q:QMes) = \\
&\quad (\textstyle\sum_{mes:Message} \mathsf{send}(n, mes) \cdot \mathsf{Buffer}(N, n, add(q, mes))) \lhd nB > size(q) \rhd \delta \\
&+ \\
&\quad \_\mathsf{rcv}(n, first(q)) \cdot \mathsf{Buffer}(N, n, remfirst(q)) \lhd \neg is\_empty(q) \rhd \delta \\
&+ \\
&\quad \textstyle\sum_{nst1:\mathbf{ABI}} \mathsf{reset}(n, nst1) \cdot \mathsf{Buffer}(N, n, add(\langle\rangle, NetworkReset(nst1))) \\
&+ \\
&\quad \mathsf{reset\_off}(n) \cdot \mathsf{Buffer}(N, n, \langle\rangle)
\end{aligned}
$$

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 to the system.

$$
\begin{aligned}
&\mathsf{SYSTEMDCMM}(N:\mathbb{N}, nstat:\mathbf{ABI}, URLs:\mathbf{ABI}) = \\
&\quad \partial_{\{\_flip,flip\_on,flip\_off\}}(\\
&\qquad \tau_{\{\_\_on,\_\_off,\_\_reset,\_\_reset\_off\}} \circ \partial_{\{\_on,on,\_off,off,\_reset,reset,\_reset\_off,reset\_off\}}(\\
&\qquad\quad \tau_{\{\_\_send\}} \circ \partial_{\{\_send,send\}}(\\
&\qquad\qquad \tau_{\{\_\_rcv\}} \circ \partial_{\{\_rcv,rcv\}}(\\
&\qquad\qquad\quad \mathsf{DCMM}(INIT, URLs[0], 0, N, 0\_0, 0\_0, 0\_0, 0, 0, nstat[0]) \parallel \mathsf{Buffer}(N, 0, \langle\rangle)) \\
&\qquad\qquad \parallel \tau_{\{\_\_rcv\}} \circ \partial_{\{\_rcv,rcv\}}(\\
&\qquad\qquad\quad \mathsf{DCMM}(INIT, URLs[1], 1, N, 0\_0, 0\_0, 0\_0, 0, 0, nstat[1]) \parallel \mathsf{Buffer}(N, 1, \langle\rangle)) \\
&\qquad\qquad \parallel \tau_{\{\_\_rcv\}} \circ \partial_{\{\_rcv,rcv\}}(\\
&\qquad\qquad\quad \mathsf{DCMM}(INIT, URLs[2], 2, N, 0\_0, 0\_0, 0\_0, 0, 0, nstat[2]) \parallel \mathsf{Buffer}(N, 2, \langle\rangle)) \\
&\qquad\quad) \parallel \mathsf{Bus}(N, nstat) \\
&\qquad) \parallel \mathsf{Env}(N, nstat) \\
&\quad)
\end{aligned}
$$

The system is initialized in the following way.

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

4. Translation 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 be either 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 expressions with value 0. Each process may have local variables. Shared variables are also allowed. To minimize 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. No transfers of control to or from, as well as communications are allowed within `d_step`. The source code of the PROMELA specification can be found in Appendix B. We tried to do the translation preserving the semantics as close as possible. Some of the crucial details of the translation are described below.

4.1 Abstract Data Types

There is no support for specifying abstract data types 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 inline functions. Computations may be done within atomic or d_step blocks, that allow to consider long deterministic computations as one step.

4.2 Conditions and Nondeterminism

The semantics of PROMELA conditional operator is slightly different from semantics of conditions in μCRL. In PROMELA conditions are statements and represent transitions to another states by their execution. In μCRL conditions are of a different kind than actions. They do not represent transitions to another state, but conditions under which such transitions are possible.

Therefore we cannot just translate μCRL expression of the form
$\mathsf{X}(d) = \_\mathsf{a}(d) \cdot \mathsf{X}(d) \lhd d < 5 \rhd \delta +$
$\quad \_\mathsf{b}(d) \cdot \mathsf{X}(d) \lhd d < 7 \rhd \delta$ to

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

because we get different semantic behavior. For instance, if $d < 5$ and there is another process Y willing to communicate with our process via channel b, then one of the possible executions in SPIN leads to a deadlock. Namely, when condition $d < 5$ is evaluated to true, and process X is waiting for communication via a, while Y is waiting for communication via b. In μCRL no execution leads to a deadlock in this case. So the semantically sound translation in this case could be:

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

In general case to make a correct translation we need first to make all conditions in each state of μCRL process disjoint. Disjointness means that at most one condition can be true for any parameter values. It is always possible to make all conditions disjoint in μCRL, however instead of n conditions in the original process we may get up to $2^n - 1$ conditions. In the worst case instead of one state with n conditions in μCRL we get an equivalent process in PROMELA with 2^n different states.

4.3 Parameterized Nondeterminism and Value-Passing Communication

In value-passing communication of μCRL read and send actions are dual. This is due to commutativity of communication function γ. The value-passing mechanism is based on matching parameter values: $\mathsf{a}(e_1)|\mathsf{b}(e_2) = \gamma(\mathsf{a},\mathsf{b})(e_1)$ if $eq(e_1, e_2)$. That is why both read and send actions can be parameterized by arbitrary expressions. In PROMELA value-passing communication is done differently. Send statement $\mathsf{a}!e_1$ means that the value of e_1 is put to the channel a. Read statement $\mathsf{a}?m$ means that a value is read from channel a and assigned to variable m.

Another problem of translation is non-bounded and parameterized nondeterminism possible in μCRL. Consider the following process equation.

$\mathsf{X}(d : N) = \sum_{n:N} \_\mathsf{a}(n, d) \cdot \mathsf{X}(g(n, d)) \lhd n \leq d \rhd \delta$ Here the set of possible alternatives depends on the value of d. Assuming that $\_\mathsf{a}$ 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 increase of state space which is linear in the number of alternatives in a state of the μCRL process.

For the case with read action a the translation is performed in the following way: $\mathsf{X}(d : D) = \sum_{n:N} \mathsf{a}(n, d) \cdot \mathsf{X}(g(n, d)) \lhd b(d) \rhd \delta$

```
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. If D is an infinite set, we cannot define such an array in PROMELA. The corresponding send action should also look as `a[d]!e`, not as `a!d,e`.

We can only transform equations where predicate b does not depend on n. This is due to the fact that it is not possible to analyze the value to be read from a channel before actually reading it. We can achieve similar to μCRL matching in communication by using arrays of channels. But this can only be done for bounded set of values. If we want to pass values from unbounded domains we need to use the original value passing mechanism of PROMELA.

5. State Space Generation

Spin and μCRL toolset were used to generate the entire state spaces and search for deadlocks for specifications in PROMELA and μCRL respectively. Unfortunately it is not possible to get the state transition system 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 DCMM processes and different buffer sizes. State spaces for four DCMM processes was not possible to generate by either toolset. In case with three DCMM processes we could not get the state space analyzed by Spin. The results of state space generation are presented in Table 1.

The command line arguments for both tools are given in Table 2. The parameter `<depth>` was `30000` for systems with 2 DCMMs, and `50000000` for 3 DCMMs.

From this we can conclude that Spin generates states faster, but the resulting transition system has more states. The results shown above cannot be interpreted as a direct comparison of state space

Table 1: Results of state space generation.

	Spin	μCRL
2 DCMMs Buffer size 2	states: 128,807 transitions: 187,343 elapsed time: 15.5s actual memory used for states: 8.6Mb total actual memory used: 4.2Mb	states: 3,842 transitions: 13,460 elapsed time: 14.6s total memory used: 6.8Mb
2 DCMMs Buffer size 5	states: 208,223 transitions: 301,598 elapsed time: 29,2s actual memory used for states: 12.2Mb total actual memory used: 6.9Mb	states: 7,292 transitions: 26,048 elapsed time: 24.9s total memory used: 7.1Mb
3 DCMMs Buffer size 2	states: >47,291,200 transitions: >78,622,800 elapsed time: >8h:04m:22.6s actual memory used for states: >1,665.1Mb total actual memory used: >5,621.3Mb	states: 3,136,289 transitions: 18,248,754 elapsed time: 5h:49m:24.2s total memory used: 155Mb

Table 2: Command line arguments.

Spin	μCRL

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

generation capabilities of Spin and μCRL toolset due to the differences in the underlying languages. However, the PROMELA specification was optimized to use many of its features that do not exist in μCRL. 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. The trick to use 3 channels instead of 1 to achieve conditional synchronous communication was not tried, and will be tried in the future.

The results may look misleading due to the fact that the PROMELA code was derived from μCRL code instead of being written directly from the informal description. However, a native PROMELA specification of the same protocol was analyzed in [10]. The model is quite different, as it employs most of the PROMELA communication primitives, but the sizes of state spaces were comparable to the ones presented here.

It is also quite difficult to analyze the correspondence between the two specifications presented here. Possibly a better comparison could be achieved using much smaller protocols available in the literature. However, the choice for HAVi leader election protocol was due to the fact that it is a

protocol some industrial groups are interested in now, and are going to use it in the future.

Acknowledgments

Thanks go to Dragan Boshnachki and Judi Romijn for carefully reading and commenting on PROMELA specification and the rest of the paper, as well as to Jan Bergstra, Jan Friso Groote and Andre van Delft for helpful discussions.

References

[1] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol 4*, chapter 2. Oxford University Press, 1994.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[3] Jan Friso Groote. The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, Amsterdam, June 1997.

[4] Jan Friso Groote and Bert Lisser. Tutorial and reference guide for the μCRL toolset version 1.0. Technical report, CWI, 1999. To appear, available from URL http://www.cwi.nl/~mcrl/mutool.html.

[5] J.F. Groote, F. Monin, and J. Springintveld. A computer checked algebraic verification of a distributed summation algorithm. Technical Report 97-14, Dept. of Mathematics and Computing Science, October 1997.

[6] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes 1994*, pages 26–62. Workshop in Computing Series, Springer-Verlag, 1995.

[7] Grundig, Hitachi, Matsushita, Philips, Sharp, Sony, Thomson, Toshiba. *Specification of the Home Audio/Video Interoperability (HAVi) Architecture*, November 19 1998. Version 1.0 beta. Available from URL http://www.havi.org/.

[8] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[9] Bas Luttik. Description and formal specification of the link layer of P1394. In Ignac Lovrek, editor, *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*. University of Zagreb, Croatia, June 18-19 1997.

[10] J.M.T. Romijn. Model checking the HAVi leader election protocol. Technical Report SEN-R9915, CWI, Amsterdam, 1999.

A. μCRL Source[1]

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Constants, Parameters                                            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map
   nB:->NAT                % Limit for Buffer capasity
   initNDCMM:->NAT         % Initial Number of processes
   initNst:->ABI           % Initial Network status
   initURLs:->ABI          % Initial URL processes status
rew
   nB=2
   initNDCMM=3
   initNst=seton(0_0,0)
   initURLs=seton(0_0,1)
map
  il: NAT#ABI->NAT
  fl: NAT#ABI#ABI->NAT
var
  N: NAT
  nst,URLs: ABI
rew
  il(N,nst)=if(eq(nst,0_0),0,min_on(nst))          %Minimal on
  fl(N,nst,URLs)=if(eq(nst,0_0),0,                 %Minimal URL on or minimal
                      if(eq(URLs,0_0),min_on(nst),  %on if there is no URL.
                        min_on(URLs)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Bool                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Bool
func
  T,F: -> Bool
map
  and: Bool#Bool      -> Bool
  or:  Bool#Bool      -> Bool
  not: Bool           -> Bool
  if:  Bool#Bool#Bool -> Bool
  eq:  Bool#Bool      -> Bool
var
  b,b1,b2: Bool
rew
  and(T,b)=b       and(b,T)=b
  and(b,F)=F       and(F,b)=F
  or(T,b)=T        or(b,T)=T
  or(b,F)=b        or(F,b)=b
  not(F)=T         not(T)=F
  if(T,b1,b2)=b1   if(F,b1,b2)=b2
  eq(F,F)=T        eq(F,T)=F
  eq(T,F)=F        eq(T,T)=T

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     NAT                                                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort NAT
func
  0:          -> NAT
  x2p1:   NAT -> NAT
  _x2p0:  NAT -> NAT
map
  x2p0:   NAT          -> NAT
  eq:     NAT#NAT      -> Bool
  1,2,3,4,5,6:         -> NAT
  succ:   NAT          -> NAT
  gt:     NAT#NAT      -> Bool
  if:     Bool#NAT#NAT -> NAT
var
  n,m: NAT
rew
  x2p0(0)=0
  x2p0(x2p1(n))=_x2p0(x2p1(n))
  x2p0(_x2p0(n))=_x2p0(_x2p0(n))
```

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

```
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)=abi1
```

```
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
200 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
201 %%%     Communication Function                                         %%%
202 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
203 comm
204 _flip|flip_on=__flip
205 _flip|flip_off=__flip
206 _on|on=__on
207 _off|off=__off
208 _send|send=__send
209 _rcv|rcv=__rcv
210 _reset|reset=__reset
211 _reset_off|reset_off=__reset_off
212
213 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
214 %%%     DCMM Process                                                   %%%
215 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
216 proc
217 DCMM(St:Status, URL:Bool, n:NAT, N:NAT, nst:ABI, wait:ABI, URLs:ABI,
218      il:NAT, fl:NAT, am_on:Bool)=
219   flip_on(n)._on(n).DCMM(INIT,URL,n,N,0_0,0_0,0_0,0,0,T)
220     <|not(am_on)|>delta
221  +
222   sum(nst1:ABI,rcv(n,NetworkReset(nst1)).DCMM(LE,URL,n,N,nst1,0_0,0_0,0,0,T))
```

```
223       <|am_on|>delta
224     +
225      flip_off(n)._off(n).DCMM(INIT,URL,n,N,0_0,0_0,0_0,0,0,F)
226       <|am_on|>delta
227     +
228
229     (  _leader(n,n).DCMM(AO,URL,n,N,nst,0_0,upd(0_0,n,URL),0,n,T)
230         <|eq(n_on(nst),1)|> delta
231      +
232       _send(il(N,nst),DMCapabilityDeclaration(n,URL)).
233           DCMM(LEIF,URL,n,N,nst,0_0,0_0,il(N,nst),0,T)
234         <|not(eq(il(N,nst),n))|> delta
235      +
236       ( sum(m:NAT,sum(d:Bool,(
237             rcv(n,DMCapabilityDeclaration(m,d)) .
238               DCMM(LEILS,URL,n,N,nst,setoff(nst,n),upd(upd(0_0,n,URL),m,d),0,
239                 fl(N,nst,upd(upd(nst,n,URL),m,d)),T)
240           <|eq(n_on(nst),2)|>
241             rcv(n,DMCapabilityDeclaration(m,d)) .
242               DCMM(LEIL,URL,n,N,nst,setoff(setoff(nst,n),m),
243                 upd(upd(0_0,n,URL),m,d),0,0,T))))
244       )<|and(eq(il(N,nst),n),not(eq(n_on(nst),1)))|>delta
245      +
246       sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1)))).
247           DCMM(LE,URL,n,N,nst,0_0,0_0,0,0,T)
248     )<|eq(St,LE)|>delta
249     +
250
251     ( _send(il,DMCapabilityDeclaration(n,URL)).
252           DCMM(LEIF,URL,n,N,nst,0_0,0_0,il,0,T)
253      +
254       sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1))).
255           DCMM(AOS,URL,n,N,nst,0_0,URLs1,0,m,T)))
256      +
257       sum(m:NAT,sum(d1:Bool,rcv(n,DMCapabilityDeclaration(m,d1)))).
258           DCMM(LEIF,URL,n,N,nst,0_0,0_0,il,0,T)
259     )<|eq(St,LEIF)|>delta
260     +
261
262     (  sum(m:NAT,sum(d:Bool,(
263          rcv(n,DMCapabilityDeclaration(m,d)).
264           DCMM(LEILS,URL,n,N,nst,setoff(nst,n),upd(URLs,m,d),0,fl(N,nst,upd(URLs,m,d)),T)
265         <|and(eq(n_on(wait),1),acc(wait,m))|>
266          rcv(n,DMCapabilityDeclaration(m,d)).
267           DCMM(LEIL,URL,n,N,nst,setoff(wait,m),upd(URLs,m,d),0,0,T))))
268      +
269       sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1)))).
270           DCMM(LEIL,URL,n,N,nst,wait,URLs,0,0,T)
271     )<|eq(St,LEIL)|>delta
272     +
273
274     (  sum(m:NAT,(
275           _send(m,DMLeaderDeclaration(fl,URLs)).
276             DCMM(LEILS,URL,n,N,nst,setoff(wait,m),URLs,0,fl,T)
277         <|and(not(eq(m,fl)),gt(n_on(wait),1))|> delta
278           +
279           _send(m,DMLeaderDeclaration(fl,URLs)).
280             DCMM(AOS,URL,n,N,nst,0_0,URLs,0,fl,T)
281         <|eq(n_on(wait),1)|> delta
282                 )<|acc(wait,m)|>delta)
283      +
284       sum(m:NAT,sum(URLs1:ABI,rcv(n,DMLeaderDeclaration(m,URLs1)))).
285           DCMM(LEILS,URL,n,N,nst,wait,URLs,0,fl,T)
286      +
287       sum(m:NAT,sum(d1:Bool,rcv(n,DMCapabilityDeclaration(m,d1)))).
288           DCMM(LEILS,URL,n,N,nst,wait,URLs,0,fl,T)
289     )<|eq(St,LEILS)|>delta
290     +
291      _leader(n,fl).DCMM(AO,URL,n,N,nst,0_0,URLs,0,fl,T)
292       <|eq(St,AOS)|>delta
293     +
294      tau.DCMM(AO,URL,n,N,nst,0_0,URLs,0,fl,T)
295       <|eq(St,AO)|>delta
296
297   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
298   %%%     Env Process                                                     %%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Env(N:NAT,nst:ABI)=sum(m:NAT,(_flip(m).Env(N,reverse(nst,m))
                              +_flip(m).delta<|gt(n_on(reverse(nst,m)),0)|>delta
                              )<|gt(N,m)|>delta)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Bus Process                                                        %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Bus(N:NAT,nstat:ABI)=
  sum(m:NAT,on(m).Bus1(N,seton(nstat,m),seton(nstat,m)))
 +
  sum(m:NAT,off(m)._reset_off(m).
    (Bus(N,setoff(nstat,m))<|eq(n_on(nstat),1)|>
     Bus1(N,setoff(nstat,m),setoff(nstat,m))))

Bus1(N:NAT,nstat:ABI,wait:ABI)=
  sum(m:NAT,_reset(m,nstat).(Bus(N,nstat)<|eq(n_on(wait),1)|>
                      Bus1(N,nstat,setoff(wait,m)))<|acc(wait,m)|>delta)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Message Queues                                                     %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort QMes
func empty    :              -> QMes
     add      : QMes#Message -> QMes

map  first    : QMes         -> Message
     remfirst : QMes         -> QMes
     is_empty : QMes         -> Bool
     size     : QMes         -> NAT
var
     mes1,mes2:Message
     q: QMes
rew
     first(add(empty,mes1))=mes1
     first(add(add(q,mes2),mes1))=first(add(q,mes2))
     remfirst(add(empty,mes1))=empty
     remfirst(add(add(q,mes2),mes1))=add(remfirst(add(q,mes2)),mes1)
     is_empty(empty)=T
     is_empty(add(q,mes1))=F
     size(empty)=0
     size(add(q,mes1))=succ(size(q))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Buffer Process                                                     %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc
Buffer(N:NAT,n:NAT,q:QMes)=
  sum(mes:Message,send(n,mes).Buffer(N,n,add(q,mes))) <|gt(nB,size(q))|> delta
 +
  _rcv(n,first(q)).Buffer(N,n,remfirst(q)) <|not(is_empty(q))|> delta
 +
  sum(nst1:ABI,reset(n,nst1).Buffer(N,n,add(empty,NetworkReset(nst1))))
 +
  reset_off(n).Buffer(N,n,empty)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     The Whole System                                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SYSTEMDCMM(N:NAT,nstat:ABI,URLs:ABI)=
encap({_flip,flip_on,flip_off},
  hide({__on,__off,__reset,__reset_off},
  encap({_on,on,_off,off,_reset,reset,_reset_off,reset_off},
      hide({__send},encap({_send,send},
        (hide({__rcv},encap({_rcv,rcv},
          DCMM(INIT,acc(URLs,0),0,N,0_0,0_0,0_0,0_0,0,0,acc(nstat,0))||
          Buffer(N,0,empty))))
         ||
        (hide({__rcv},encap({_rcv,rcv},
          DCMM(INIT,acc(URLs,1),1,N,0_0,0_0,0_0,0_0,0,0,acc(nstat,1))||
          Buffer(N,1,empty))))
         ||
        (hide({__rcv},encap({_rcv,rcv},
          DCMM(INIT,acc(URLs,2),2,N,0_0,0_0,0_0,0_0,0,0,acc(nstat,2))||
          Buffer(N,2,empty))))
      ))
```

```
        ||
        Bus(N,nstat)
    ))
    ||
    Env(N,nstat)
)

init SYSTEMDCMM(initNDCMM,initNst,initURLs)
```

B. PROMELA Source[2]

```
#define initNDCMM 3
#define nB 2

typedef ABI {bool a[initNDCMM]};
mtype = {NetworkReset, DMCapabilityDeclaration, DMLeaderDeclaration};
typedef Message {mtype MTYPE; byte NN; bool URL; ABI NST};

chan on = [0] of {byte};
chan off = [0] of {byte};
chan send[initNDCMM] = [0] of {Message};
chan rcv[initNDCMM] = [0] of {Message};
chan reset[initNDCMM] = [0] of {ABI};
chan reset_off[initNDCMM] = [0] of {bit};
chan flip[initNDCMM] = [0] of {bit};
chan leader = [0] of {byte, byte}

chan env = [0] of {ABI}
chan bus = [0] of {ABI} /* Due to the technical restrictions of spin we
cannot pass arrays as parameters for processes. So we use these channels to
pass nst to Env and Bus */

/* inlines use and sideeffect variable _i
(assumed that it is defined as byte) */

/* copies N first elements of array B
to the corresponding elements of A */
inline array_assign(A, B, N)
{ _i=0; do
        :: _i<N -> A.a[_i]=B.a[_i]; _i=_i+1
        :: else break
        od; _i=0;}

/* m := minimal m s.t. A[m].
0 if all elements of A are false */
inline array_min_true(A, N, m)
{ _i=0; do
        ::(_i<N) -> if
                    :: !A.a[_i] -> _i=_i+1
                    :: else -> break
                    fi;
        ::else ->   break
        od; m = (_i==N -> 0 : _i); _i=0;}

/* n_on := number of true elements of A */
inline array_n_true(A, N, n_on)
{ n_on=0; _i=0; do
                ::(_i<N) -> n_on=(A.a[_i]->n_on+1:n_on);_i=_i+1
                ::else -> break
                od; _i=0;}

/* assign false to N first elements of A*/
inline array_false(A, N)
{ _i=0; do
        ::(_i<N) -> A.a[_i]=false; _i=_i+1
        ::else -> break
        od; _i=0;}

#define NETWORK_RESET_WAIT_URLS rcv[n]?NetworkReset,_,ib,nst;\
atomic{d_step{array_false(wait,N);array_false(URLs,N);\
```

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

```
il=0;fl=0;m=0;n_on=0};goto LE}

#define NETWORK_RESET_URLS rcv[n]?NetworkReset,_,ib,nst;\
atomic{d_step{array_false(URLs,N);\
il=0;fl=0;m=0;n_on=0};goto LE}

#define NETWORK_RESET rcv[n]?NetworkReset,_,ib,nst;\
atomic{d_step{il=0;fl=0;m=0;n_on=0};goto LE}

#define FLIP_OFF_NST_WAIT_URLS flip[n]?1;off!n;\
atomic{d_step{array_false(nst,N);array_false(wait,N);array_false(URLs,N);\
il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}

#define FLIP_OFF_NST_URLS flip[n]?1;off!n;\
atomic{d_step{array_false(nst,N);array_false(URLs,N);\
il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}

#define FLIP_OFF_NST flip[n]?1;off!n;\
atomic{d_step{array_false(nst,N);\
il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}

#define FLIP_OFF flip[n]?1;off!n;\
atomic{d_step{il=0;fl=0;m=0;n_on=0;am_on=false};goto INIT}

bool ib; hidden ABI iabi;

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     DCMM Process                                                    %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
proctype DCMM(bool URL; byte n, N; bool _am_on)
{  bool am_on; ABI nst, wait, URLs;
   byte il,fl,m,n_on; bool d; byte _i;

   d_step{ am_on=_am_on; array_false(nst,N); array_false(wait,N);
           array_false(URLs,N); il=0; fl=0; m=0; n_on=0; d=false; _i=0;}
INIT:
  if
  :: !am_on -> flip[n]?1; on!n; atomic {am_on=true; goto INIT}
  :: am_on ->  if
               ::  NETWORK_RESET
               ::  FLIP_OFF
               fi;
  fi;

LE:
  atomic{
    d_step{array_min_true(nst,N,il);} /* il calculation */
    if
    :: il==n -> d_step{array_assign(wait,nst,N); wait.a[n]=false;
                       URLs.a[n]=URL; il=0;} goto LEIL;
    :: else
    fi;}

LE1:
  if
  ::  send[il]!DMCapabilityDeclaration(n,URL,iabi); goto LEIF
  ::  rcv[n]?DMLeaderDeclaration,_,ib,iabi; goto LE1;
  ::  rcv[n]?DMCapabilityDeclaration,_,ib,iabi; goto LE1;
  ::  NETWORK_RESET
  ::  FLIP_OFF_NST
  fi;

LEIF:
  if
  ::  send[il]!DMCapabilityDeclaration(n,URL,iabi); goto LEIF
  ::  rcv[n]?DMLeaderDeclaration,fl,ib,URLs; goto AOS
  ::  rcv[n]?DMCapabilityDeclaration,_,ib,iabi; goto LEIF
  ::  NETWORK_RESET
  ::  FLIP_OFF_NST
  fi;

LEIL:
  atomic{d_step{array_n_true(wait,N,n_on);}
LEIL1:
    if
    :: n_on==0 -> d_step{array_assign(wait,nst,N);
```

```
                    wait.a[n]=false;

                    array_min_true(nst,N,fl);
                    array_min_true(URLs,N,m);  /* final leader calculation */
                    fl=(m==0->fl:m); m=0;}

                    goto LEILS;
    :: else
    fi;}

LEIL2:
  if
  ::  rcv[n]?DMCapabilityDeclaration,m,d,iabi;
      atomic{d_step{n_on=(wait.a[m]->n_on-1:n_on);
        wait.a[m]=false; URLs.a[m]=d; m=0; d=false}; goto LEIL1; }
  ::  rcv[n]?DMLeaderDeclaration,_,ib,iabi; goto LEIL2;
  ::  NETWORK_RESET_WAIT_URLS
  ::  FLIP_OFF_NST_WAIT_URLS
  fi;

LEILS:
  atomic{m=0; d=wait.a[fl]; wait.a[fl]=false; /* final leader is informed the last */

LEILS1:
    if
    :: (m==N && d) -> d_step{m=fl; d=false;}
    :: (m==N && !d) -> m=0; goto AOS;
    :: (m<N && !wait.a[m]) -> m=m+1; goto LEILS1;
    :: else
    fi;}

LEILS2:
  if
  ::  send[m]!DMLeaderDeclaration(fl,false,URLs); wait.a[m]=false; goto LEILS1;
  ::  rcv[n]?DMLeaderDeclaration,_,ib,iabi; goto LEILS2;
  ::  rcv[n]?DMCapabilityDeclaration,_,ib,iabi; goto LEILS2;
  ::  NETWORK_RESET_WAIT_URLS
  ::  FLIP_OFF_NST_WAIT_URLS
  fi;

AOS:
  if
  ::   leader!n,fl; goto AO;
  ::   NETWORK_RESET_URLS
  ::   FLIP_OFF_NST_URLS
  fi;

AO:
  if
  ::   NETWORK_RESET_URLS
  ::   FLIP_OFF_NST_URLS
  ::   goto AO;
  fi;
}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Bus Process                                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
proctype Bus(byte N)
{ ABI nst, wait; byte m, n_on, n_on_wait; byte _i;

  d_step{array_false(nst,N); array_false(wait,N); m=0; n_on=0; _i=0;}
  bus?nst;
  d_step{array_n_true(nst,N,n_on);}

Bus_:
  if
  :: n_on==0 -> on?m; atomic{d_step{nst.a[m]=true; m=0; n_on=1;} goto Bus1}
  :: else -> if
             ::  on?m; atomic{d_step{nst.a[m]=true; m=0; n_on=n_on+1;} goto Bus1}
             ::  off?m; reset_off[m]!1; atomic{d_step{nst.a[m]=false; m=0; n_on=n_on-1;} goto Bus1}
             fi;
  fi;

Bus1:
  atomic{
```

```
212       if
213       :: (m==N) -> m=0; goto Bus_;
214       :: (m<N && !nst.a[m]) -> m=m+1; goto Bus1;
215       :: else
216       fi;}
217
218     reset[m]!nst; atomic{m=m+1; goto Bus1};
219   }
220
221   #define BUFFER_RESET reset[n]?nst;atomic{d_step{queue_clean(nIn);\
222   queue[0].MTYPE=NetworkReset;array_assign(queue[0].NST,nst,N);\
223   array_false(nst,N);nIn=1}; goto Buffer_}
224
225   #define BUFFER_RESET_OFF reset_off[n]?1;\
226   atomic{d_step{queue_clean(nIn); nIn=0}; goto Buffer_}
227
228   /* inlines below use and sideeffect variable _j
229   (assumed that it is defined as byte) */
230
231   /* shifts queue[1..nIn-1] to queue[0..nIn-2]
232   (if nIn<=1 does nothing) */
233   inline queue_shift()
234   {  _j=1; do
235           ::_j<nIn-> queue[_j-1].MTYPE=queue[_j].MTYPE;
236                      queue[_j-1].NN=queue[_j].NN;
237                      queue[_j-1].URL=queue[_j].URL;
238                      array_assign(queue[_j-1].NST,queue[_j].NST,N);
239                      _j=_j+1
240           ::else->break
241      od; _j=0;}
242
243   /* assignes default values to queue elements */
244   inline queue_clean(NNN)
245   {  _j=0; do
246           :: _j<NNN -> queue_clean_element(_j); _j=_j+1
247           :: else break
248           od; _j=0;}
249
250   /* assignes default value to an element */
251   inline queue_clean_element(el)
252   { queue[el].MTYPE=NetworkReset;
253     queue[el].NN=0;
254     queue[el].URL=false;
255     array_false(queue[el].NST,N);}
256
257   /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
258   %%%     Buffer Process                                                  %%%
259   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
260   proctype Buffer(byte n, N)
261   { byte nIn,_i,_j; Message queue[nB]; ABI nst;
262
263     d_step{nIn=0; array_false(nst,N); queue_clean(nB); _i=0; _j=0;}
264
265   Buffer_:
266     if
267     :: (nIn<nB && nIn>0) ->
268          if
269          ::  send[n]?queue[nIn];
270              atomic{nIn=nIn+1; goto Buffer_};
271          ::  rcv[n]!queue[0];
272              atomic{d_step{queue_shift(); nIn=nIn-1;
273                queue_clean_element(nIn);} goto Buffer_}
274          ::  BUFFER_RESET
275          ::  BUFFER_RESET_OFF
276          fi;
277     :: (nIn==nB) -> if
278                     ::  rcv[n]!queue[0];
279                         atomic{d_step{queue_shift(); nIn=nIn-1;
280                         queue_clean_element(nIn);} goto Buffer_}
281                     ::  BUFFER_RESET
282                     ::  BUFFER_RESET_OFF
283                     fi;
284     :: (nIn==0) ->  if
285                     ::  send[n]?queue[nIn]; atomic{nIn=1; goto Buffer_}
286                     ::  BUFFER_RESET
287                     ::  BUFFER_RESET_OFF
```

```
288                   fi;
289     fi;
290   }
291
292   /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
293   %%%     Env Process                                                      %%%
294   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
295   proctype Env(byte N)
296   { ABI nst; byte n_on,j; byte _i;
297
298     d_step{j=0; n_on=0; array_false(nst,N);_i=0}
299     env?nst;
300   Env_:
301     if
302     :: flip[j]!1;
303        atomic{ d_step{nst.a[j]=!nst.a[j]; j=0; array_n_true(nst,N,n_on);}
304                if
305                ::  n_on=0; goto Env_;
306                ::  (n_on!=0) -> d_step{n_on=0; array_false(nst,N);} goto Env_End;
307                fi;
308              }
309     :: leader?_,_; atomic{j=0; goto Env_;}
310     :: (j<(N-1)) -> atomic{j=j+1; goto Env_;}
311     fi;
312   Env_End:
313     leader?_,_; goto Env_End;
314   }
315
316   /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
317   %%%     Init                                                             %%%
318   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
319
320   init
321   { ABI nst, URLs; byte j; byte _i;
322     atomic{
323       d_step{ array_false(nst,initNDCMM); array_false(URLs,initNDCMM);
324               nst.a[0]=true; URLs.a[1]=true; j=0;}
325       do
326       :: j<initNDCMM -> run DCMM(URLs.a[j],j,initNDCMM,nst.a[j]);
327                         run Buffer(j,initNDCMM); j=j+1;
328       :: else break;
329       od; j=0;
330       run Bus(initNDCMM); run Env(initNDCMM);
331     }
332     bus!nst; env!nst;
333   }
```