Model Checking the HAVi Leader Election Protocol

Judi Romijn
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
judi@cwi.nl

ABSTRACT
The HAVi specification [9] proposes an architecture for audio/video interoperability in home networks. Part of the HAVi specification is a distributed leader election protocol. We have modelled this leader election protocol in Promela and Lotos and have checked several properties with the tool Spin and the tool Xtl (from the Cæsar/Aldébaran package).

It turns out that the protocol does not meet some safety properties and that there are situations in which the protocol may never converge to designate a leader. Our conclusion is that realistic timing requirements on sending and processing of messages should be added to the HAVi specification. Then a (timed) formal verification could give a definite answer with respect to correctness of the leader election protocol.

1991 Mathematics Subject Classification: 68M10, 68Q10, 68Q22, 68Q45, 68Q60, 68Q65
1991 ACM Computing Classification System: C.2.2, D.2.4, D.3.3, F.1.1, F.1.2, F.4.1, G.4
Keywords and Phrases: model checking, protocol verification, abstraction, process algebra, temporal logic, safety, liveness
Note: This research was carried out as part of the project "Specification, Testing and Verification of Software for Technical Applications" at the Stichting Mathematisch Centrum for Philips Research Laboratories under Contract RWC-061-PS-950006-ps.

Table of Contents

1 Introduction

The Home Audio/Video Interoperability (HAVi) project [9] is a joint effort by eight consumer electronics companies to solve interoperability problems for audio/video networks in the home environment.

The HAVi specification specifies a set of Application Programming Interfaces (APIs) and protocols that allow consumer electronics manufacturers and third parties to develop applications for the home network. Thus the home network is viewed as a distributed computing platform, and the primary goal of the HAVi architecture is to assure that products from different vendors can cooperate to perform application tasks. The HAVi architecture is supposed to work on top of an IEEE 1394 serial bus [16, 17].

There are two types of HAVi devices: controllers and controlled devices. The controller acts a host for controlled devices via a Device Control Module (DCM). Installation and allocation of such DCMs is done by a HAVi software element which is called the Device Control Module Manager (DCM Manager). Each controller is supposed to have a DCM Manager. All DCM Managers have to cooperate with each other to ensure that the installation and allocation of DCMs works properly. A complicating factor here is the dynamic plug-and-play character of the 1394 network. Each time when a change in the 1394 network occurs, the DCM Managers restart their activities by first choosing a leader among them, and then under the control of the designated leader, complete their DCM controlling tasks.

The purpose of the leader election is that the DCM Manager with the best capabilities will play a central role in the DCM controlling tasks. Since not all of these capabilities are persistent and globally available, the DCM Managers need to communicate to find out which one is the best candidate for leadership.

In this paper, we study the leader election protocol between the DCM Managers. Our goal is to analyse this protocol with several model checking tools, to determine whether the protocol is correct, and to compare the model checking tools. Our approach is to construct a model of the behaviour of the protocol in a suitable formal language, and to establish certain properties through model checking. Model checking is a verification approach where one checks whether a property holds by exploring the reachable state space of the model. The manual construction of such proofs is a tedious and error-prone process. Nowadays, there are several tools that fully automate the model checking process.

We present several models of the protocol leader election protocol in the formal languages Promela [11] and Lotos [7]. Several properties have been checked with the model checking tools Spin [11, 12] and Xtl [22] (part of the Cæsar/Aldébaran distribution [6]).

We have found errors in the formal models with both Spin and Xtl. It turns out that some safety properties are not met by the protocol and that there are situations in which the protocol may never converge to a designate a leader. The cause of these errors is that the HAVi specification is not detailed enough to ensure that HAVi compliant implementations are faultless. The errors occur when communication between different devices is faster than communication between components in one device. Besides our conclusions on the correctness of the HAVi protocol, we compare the two model checking tools.

As far as we know, the only other paper in which HAVi leader election protocol between DCM Managers is studied is [28]. Here, a comparison is made between the performance of state space exploration of Spin and the μCRL tool set [8]. The model of the protocol differs from ours and no model checking is performed.

This paper is organised as follows. Section 2 gives an informal description of the HAVi leader election protocol. Section 3 introduces the tools and languages used. Section 4 describes our model of the protocol. Section 5 gives the details of all the model checking experiments. Finally, Section 6 gives several conclusions that we drew from this experiment.

In the appendices, relevant excerpts from the HAVi and 1394 specifications and several code listings can be found.

Acknowledgements

I thank Eddy Zondag for his help in understanding and modelling the HAVi leader election protocol, Gerard Holzmann for his help with running Spin on large machines, Radu Mateescu for his help with Cæsar, Aldébaran and Xtl, and Yaroslav Usenko for questioning the validity of my models. Dennis Dams and Gerard Holzmann are thanked for their help in understanding the finer details of the CTL versus LTL problem. Finally, I thank Ron Koymans, Izak van Langevelde and Frits Vaandrager for their comments on previous versions of this paper.

2 The DCM Manager leader election protocol

The DCM Manager leader election protocol is described in the HAVi specification [9] at page 160. The protocol tries to find a suitable leader for the actual task of the DCM Managers, which is performed in the autonomous operation phase. We only study the leader election phase.

The parts of the HAVi specification and the IEEE 1394 standard that are relevant for this protocol are listed in Appendices A and B. Here, we give an informal explanation of the protocol, and the services that it requires from several HAVi components.

2.1 Protocol

Each DCM Manager enters the leader election phase upon initialisation and each time a bus reset event is received. First it obtains information on the current network topology, by sending a request to another HAVi element, the Communications Media Manager, which returns a list with all the devices that are active in the (1394) network. The list contains the Global Unique ID (GUID) of all devices in the network. The DCM Manager then questions the 1394 level of each active device to find out some more information. The information needed for this protocol is the HAVi type of the device is (FAV, IAV, BAV or LAV), and whether there is a DCM Manager present at the device (at FAV compulsory, at IAV optional). Based on this information, the DCM Manager selects an initial leader from the GUIDs of devices on which a DCM Manager is present. Since each DCM Manager uses the same procedure for the selection, all of them choose the same initial leader without communicating with each other. Each DCM Manager which is not the initial leader is called initial follower.

The initial leader waits for initialisation requests from all initial followers, in which they state their capability. Using this new information and the HAVi type of the devices, the initial leader decides which DCM Manager is the best candidate for the final leadership. One of the criterions is the HAVi controller type, which is found in the (static) information of the HAVi device and which can be accessed from outside the device. The other criterion is Internet access which is found in the request messages from the followers. Each initial follower is informed of the decision with an initialisation reply, and the DCM Manager which has been elected as final leader is informed last. After this, the leader election phase ends and the autonomous operation phase phase is entered. Here, each DCM Manager which is not the final leader is called final follower.

During or after the leader election phase, the network topology may change, which causes a bus reset phase to start. Whenever this happens, the DCM Managers should start anew with the leader election because the previously elected leader may have disappeared from the network or a more suitable candidate may have appeared. The DCM Managers are informed of a bus reset phase by the Communications Media Manager with an event. The HAVi specification does not lay down any implementation rules for the delivery of this event, such as timing requirements. So it is possible that the bus reset event is delivered after the bus reset phase has already ended. If multiple bus reset phases occur (almost) adjacently, the DCM Managers may get out of phase in their leader election. Then one DCM Manager might be sending its initialisation request to an initial leader which is not aware of any bus reset phase having taken place, or vice versa. To keep things in order, the DCM Manager which is to be the initial leader, must remember this role and answer initialisation requests with an initialisation reply, even after leader election has ended. During and after the protocol, all unexpected messages are ignored.

2.2 HAVi components

The DCM Managers use the services of the local elements Messaging System, Communication Media Manager, and Event Manager. These elements will be available at each HAVi device that contains a DCM Manager.

The **Messaging System** provides two services and two modes of sending messages to software elements, whether local or not. The service choices are to block while waiting for a response by the receiver or not to wait for a response. The modes are reliable or simple. The reliable mode implicates that the sender is informed by the Messaging Systems involved whether the message reached the receiver. The sender is blocked until such an acknowledgement arrives or a time-out occurs. The simple mode implicates no acknowledgement information from the Messaging Systems is given to the sender. The Messaging System on the device of the receiver delivers the message to the receiver via a call back function, which the receiver has dispensed to the Messaging System at start-up time. The Messaging System uses the 1394 network for the actual message passing. From the 1394 specification we learn that at the 1394 level, no messages can be sent between different devices while a bus reset is taking place.

The DCM Managers communicate with each other using the reliable method and the response service. The HAVi specification does not limit the nature of the call back function that the DCM Managers use. The DCM Managers use a timeout of 3 seconds on all messages.

The **Event Manager** accepts requests to post events and sends a message with the event through the Messaging System to every local software element that has subscribed to the event. A posting request must be sent through the Messaging System. The DCM Managers all subscribe to the BusReset event during initialisation.

The **Communication Media Manager** provides information on the network configuration which it gets from the 1394 layer. Upon the start of a bus reset phase, it posts the event BusReset. Since each FAV or IAV device has its own Communication Media Manager to signal the bus reset start, the BusReset event only needs to be sent to software elements on the same device. This means that the Messaging System can at all times deliver the messages containing this event to the interested parties, as long as the device is powered up.

The Communication Media Manager also allows software elements to request network information in the form of a GUIDList. This service is only available outside bus reset phases, after the Communication Media Manager has received the information from 1394. This information is to be asked with a message through the Messaging System.

An example scenario In Figure 1 we show an example scenario in the following happens. A bus reset period starts. The Communication Media Manager posts the BusReset to the Event Manager. The Event Manager delivers the BusReset to the DCM Manager. The DCM Manager reacts by requesting the GUIDList from the Communication Media Manager. This list is available only when the bus reset period has ended.

3 Languages and tools

This section gives a short introduction to the languages and tools used for formalisation and verification of the leader election protocol. For details we refer to the documentation cited below.

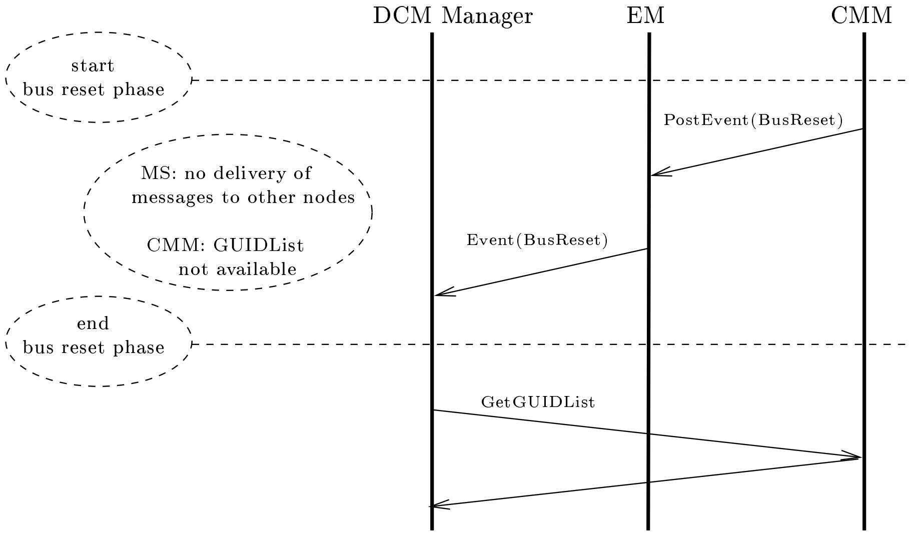

Figure 1: A bus reset scenario

3.1 Spin and Promela

Spin [11, 12] is a tool that supports simulation and verification of Promela [11] models of distributed systems. Models in Promela (a Process Meta Language) consist of definitions of process behaviour, with variable assignments, sequential and alternative composition, repetition and dynamic process creation. Communication between processes happens on synchronous or asynchronous channels. Synchronous communication always involves two processes. The support of data types is very limited: basic types are booleans and naturals, from which arrays and record structures can be built.

Verification is supported through detection of deadlocks, invalid end-states or non-progress loops, through violation of assertions and through LTL [24, 20] properties. The verification is done on the fly: the global state space is not constructed, but explored directly from an interpreted version of the Promela code.

3.2 Lotos, Cæsar/Aldébaran and Xtl

Lotos [7] is a standardised language for abstract modelling of distributed systems. Lotos models consist of a data part and a behaviour part: the data part is expressed in ACT-ONE, an algebraic formalism for abstract data types, and the behaviour part is expressed in process algebra with sequential, alternative and parallel composition, and recursion. Communication happens on synchronous gates and can involve more than two processes.

The Cæsar/Aldébaran tool set [6] facilitates simulation and verification of Lotos models. Simulation and detection of deadlocks, livelocks et cetera can be done on the fly.

The Xtl tool [22] (which is part of the Cæsar/Aldébaran tool set) facilitates the verification of temporal properties over Lotos models. First the global state space must be generated (with Cæsar), then Xtl can verify a property given in one of the following logics: HML [10], CTL [1], LTAC [26], ACTL [4, 3] and the modal μ-calculus [18]. It is even possible to define one's own modal logic in terms of the libraries provided by Xtl (including greatest and least fixpoint operators).

4 Modelling decisions

In this section, our model of the protocol is explained. What is presented here is the result of a process of experimenting with different models, imposing and lifting restrictions until a satisfactory model with a manageable size was obtained.

First we explain the general modelling decisions, and give a description of the processes involved. Then the details of the Promela and Lotos models are explained (See Appendices C and D for code listings). Unless stated otherwise, the explanations refer to the model of the protocol with 3 DCM Managers and asynchronous communication between the DCM Managers. In the remainder of this section we abbreviate DCM Manager (DM), Communication Media Manager (CMM), and Messaging System (MS).

4.1 General description

Restrictions on the network Each of the following choices is a restriction on what is allowed by the HAVi model. These restrictions are imposed in order to obtain a model of manageable size.

We study only situations with one network in which maximally three devices are active, and demand that in the start state no device is powered on.

The HAVi device types are FAV, IAV, BAV and LAV. We assume that there only are FAV devices in the network, and that on each of these devices there is a DM present.

A bus reset in the 1394 network may be caused by a change in the network topology (a device being added to or removed from the network), by a device in the network being powered up or down, by race conditions in the 1394 protocol or by other error situations. We model the cause of a bus reset as the power change of zero or more devices in the network. Here, zero power changes represents some other cause of bus reset, and the power change of a device also represents the connecting or disconnecting of that device (when a device is disconnected but still powered up, it operates in a new network consisting of just itself; we only study one network). The network behaviour is modelled with the process Bus_Reset.

From IEEE 1394 we learn that the worst-case time delay between the start of the bus reset phase and the moment that the last device in the network notices the bus reset is less than 167 microseconds. The duration of the bus reset phase until normal operation resumes is at least 414 and maximally 581 microseconds. We restrict the bus reset phase delay to zero, which means that the bus reset phase starts at the same time at all devices in the network. For our verification purposes we only want to consider properties that concern situation in which a bus reset is not taking place. Therefore it is convenient to have the start of the bus reset phase actually precede the change of network which causes the bus reset phase.

In the HAVi design, each DCM Manager use a capability and a preference in the leader election protocol. We restrict ourselves to the capability UrlCapable, which indicates whether a device has Internet access (true) or not (false). We assume that the value of UrlCapable does not change.

In a 1394 network a device may be unplugged (powered off), and then plugged back in (powered on). This may cause the device to get a different 1394 physical ID and HAVi SEID (Software Element ID) once it is back in the network, than the 1394 and HAVi IDs it had before. Since each device has a globally unique ID (GUID) which does not change, and other devices can find out about this through the GUIDList which is managed by their CMM, we only identify devices with their GUID and do not model the physical ID.

Which HAVi components? We model the DM, the MS and the CMM with separate processes, which are described below. We do not include a process for the Event Manager. The only event posted to this component will be the BusReset, and all different scenarios of delivery of this event can be modelled by one synchronous communication between the CMM and the DM. If the delivery is unsuccessful, the communication does not occur. An extra process Bus_Reset is needed to model

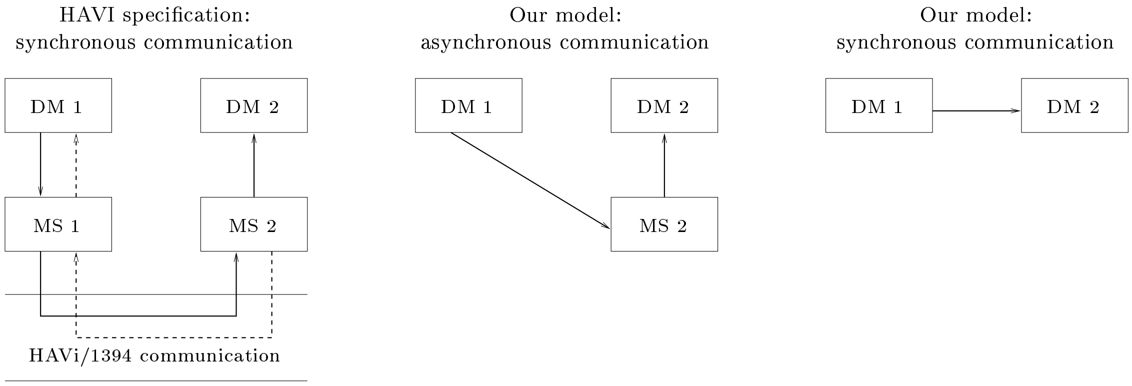

Figure 2: DCM Manager to DCM Manager communication

the behaviour of the 1394 network.

Process Bus_Reset This process determines whether a new bus reset period will start, and which devices (hence which DMs and CMMs) will be powered up or down. Both of these choices are non-deterministic, hence in a verification all possibilities will be considered. Whenever a device is powered up or down, the DM, CMM and MS on that device are informed by Bus_Reset in a synchronous manner. The power changes are determined in increasing order of device ID.

Process CMM This process controls the GUIDList, in which all devices present in the network are listed. It also signals any start or end of a bus reset period on the 1394 network, and passes this information on to the DM and the MS on the same device.

When several bus reset periods follow each other with little time in between, it is possible that a CMM has not posted the occurrence of a previous bus reset, when the next is already taking place. The HAVi specification does not define whether both bus reset events should be posted or just one. We choose to have the new bus reset overrule the previous one, and have only the last bus reset notification being posted and delivered.

Process MS This process takes care of the communication between the DMs and acts as a buffer. All message transfers that use the MS, are performed in reliable mode, therefore we model such a message transfer as one communication involving just the sending and the receiving component. The message transfer is shown in Figure 2. The HAVi design is that DM 1 sends a message, intended for DM 2, to the MS 1 (which is on the same device as DM 1). MS 1 sends the message on the network to MS 2, which delivers it to DM 2. After sending the message, DM 1 will wait for an error message or an acknowledgement of successful delivery to DM 2. DM 1 only continues its operation after such a notification. In Figure 2, continuous arrows show how a message is transported through the HAVi architecture from DM 1 to DM 2, and the dashed arrows show how the notifications are generated and returned. In case of erroneous transfer, the message may not reach MS 2 or DM 2. Successful delivery to DM 2 means that either DM 2 is interrupted to receive the message (synchronous communication) or the message is put into a buffer designated by DM 2 (asynchronous communication). We have modelled the synchronous version of this communication with direct synchronous communication between DM 1 and DM 2 (and then there is no need for any MS process), and the asynchronous version by synchronous communication between DM 1 and MS 2. In the latter case, DM 2 can get the message from MS 2 by synchronous communication. Note that MS 1 is not used in this communication scheme. This modelling choice is made to limit the possibilities for the communication, which is reasonable since we are only interested in the communication succeeding (modelled by the message put into the buffer) or failing (modelled by the communication not occurring at all). Of course, the size of the buffer maintained by the MS limits the number of messages that can be sent to a DM before it actually receives them.

So, in short, in the case of synchronous communication between DMs, there will be no MS process in our model. In the case of asynchronous communication between DMs, there will be an MS process which acts as a buffer for incoming messages directed to the DM at the same device. The buffer size is a parameter for the model; in all our models the buffer size is 1. In case of asynchronous communication, the DM will empty the buffer in the event of a bus reset period or whenever the power is switched off.

Process DCM_Manager The general task of the DM is explained in Section 2. Our model follows this procedure as closely as possible, except for a few modelling choices.

1. In our model we skip the subscription that the DM uses to inform the EM that it wants to receive all bus reset events. We also skip the registration of the call back function that the DM must dispense to the MS.
2. From the two parameters that the DM uses in the protocol, we only consider UrlCapable (Internet access).
3. The HAVi method of electing the initial leader, is to choose the DM on the device with the highest *bit order reversed* ID. Since our assignment of IDs to DMs is arbitrary, we just choose the DM with the lowest ID for initial leader.
4. The selection of the final leader in the HAVi design should be an arbitrary choice of the devices with the best capabilities. We study networks with only FAV devices on which a DM is present, hence we let the device with the lowest ID and UrlCapable set to true be the final leader (which is not arbitrary, but does limit the size of the state space). If no device has special capabilities, the HAVi design allows the initial leader to elect an arbitrary device for final leader. In this case, we have the initial leader elect itself for final leader (which also limits the state space size).
5. In the HAVi protocol, each initial follower will send its initialisation request to the initial leader, and will resend the request if a reply was not received before a timeout occurs (which is after 3 seconds). All our models are without timing information. Hence we let the initial follower choose arbitrarily between resending the request and receiving the reply. In this manner we cover all possibilities. Note that this choice does not introduce new behaviour, that is, behaviour that is not permitted by the HAVi specification.

4.2 Promela

The Promela code is listed in Appendix C.1.

At the beginning of the code, some type definitions are given for the global variables in which all information about the several DMs is stored. These variables must be global (as opposed to local for the DCM_Manager process that actually 'owns' the information) in order to facilitate access to this information in the model checking process. Then some channels are defined which are used for communication between DMs (asynchronous in this model), between DM and CMM (synchronous), and between DM and the Bus_Reset process. We have chosen to model the asynchronous communication between DMs with the channel `chanDM`, and not a separate MS process. This helps in keeping the model simple and the state space small.

The statement labels indicate the state a process is in. By default, execution of a process starts at the first statement. When a change of state is desired, this is done with the `goto <label>` statement, where `<label>` is the target state.

Many statements in the processes are not meant to be executed in an interleaved manner with other activities in the network. With the `atomic` attribute, we can express that the statements in its scope are to be executed without creating new states in between. This also helps in keeping the state space small.

Most of the processes should be interrupted when the power on the corresponding device is switched off. This interruption is modelled with the operator `unless`.

Process Bus_Reset First a bus reset period is started. The decision to actually execute this statement is made non-deterministically, meaning that in the whole state space it is always possible to postpone this branch of execution as long as there is another statement that can be executed.

The start of the bus reset period is forwarded to all the CMMs that are up, by setting the corresponding `delivery` boolean in the global `BusResetDelivery` array. If this boolean was true already because a previous bus reset was not handled yet by a CMM, the boolean will remain true, so only the last bus reset notification is delivered.

Whenever a device is powered up or down, a `power_change` message is sent to the corresponding DM, which will in its turn inform the CMM. If Promela supported multi-way communication, both DM and CMM could be informed at the same time. As it is, this modelling choice keeps the size of the state space manageable.

The model in Appendix C.3 allows maximally two bus reset periods. This makes the behaviour of the model finite and allows us to search for invalid end states. The reason for this is given in Section 5.4.

Process CMM Whenever the device of the CMM is being powered up or down, it will get a message on the (synchronous) channel `chanCMM` and goto the corresponding state. Whenever the device of the CMM is up and the start of a bus reset period is marked in the `delivery` variable in the global array `BusResetDelivery`, then the CMM forwards this information to the DM on the same device, by means of a message on `chanCMM`, which will interrupt the DM.

The control of the GUIDList is not done explicitly by CMM, since it is not possible to send the list on a channel (because Promela forbids the sending of array structures). Instead, we have the DMs read a global variable, which is permitted only outside bus reset periods.

Process DCM_Manager This process actually performs the tasks of the leader election protocol. If it becomes initial leader, it needs a local array for keeping track of information received from initial followers. This information is stored the array `InfoHost`. When the power of the device is not on, the process just waits for a message that power has been turned on. When this happens, it forwards the information to the local CMM (which is one of many ways to solve the multi-way synchronisation which is not provided by Promela), and starts the leader election protocol.

First the DM needs the contents of the GUIDList. This list can be obtained as soon as the latest bus reset period has ended. The information from global array `Global` is copied into the appropriate entry of global array `Local` (this entry is owned by this process; the array is global only for model checking purposes). From the GUIDList, the initial leader is determined. The DM continues being either the initial leader, or an initial follower.

Being the initial leader is not kept track of in a variable in this model. Only final leadership is recorded in a `fleader` variable for each DM. The model in Appendix C.2 records both kinds of leadership in a `leader` variable for each DM. Which model is used during verification depends on the property that is to be checked,

The initial leader and follower tasks are described in Section 2.

The final leader and followers should perform the tasks of the autonomous operation phase, but we have not modelled this behaviour. Therefore a final leader or follower DM does nothing, except wait for a new bus reset period, a power change or (in case it was the initial leader) answer initialisation requests with the initialisation reply.

Process MS This process is not present in the Promela model. We choose to have the asynchronous channel `chanDM` perform the desired functionality. This decision forces us to put the awareness of a bus reset taking place (hence no communication on the network possible) in a different process. We choose to have the sending DM inspect the global variable `BusResetPeriod`.

Process Init Here, all the processes are actually started, and the non-deterministic choice for UrlCapable to be true or false is made. Note that the process Assertion is also started, which is not listed in the Promela model itself. This process monitors the property which is to be checked. The properties are explained in Section 5.

4.3 Lotos

The Lotos code is listed in Appendices D.1, D.2 and D.3.

We will not explain the data parts any further, since they mostly speak for themselves.

As for the behaviour part, this is modelled with a process definition for each small part of the protocol's behaviour. Because of the cyclic character of the whole protocol and certain parts of it, recursion is often used in these process definitions. Cæsar does not allow some forms of recursion, which are part of Lotos, such as recursion in combination with a communication operator. This means that we cannot recursively instantiate the DM processes.

Most of the processes should be interrupted when the power on the corresponding device is switched off. This is done with the reception of a `power_change` message at the right-hand side of the disrupt operator `[>`. This operator works as follows: (1) the process `A[>B` can perform an action from process `A` and then behave as `A'[>B` (with `A'` the remainder of `A`), and (2) `A[>B` can perform an action from process `B` and then behave as `B'` (with `B'` the remainder of `B`).

Because of the enforced multi-way synchronisation, a process must sometimes participate in a communication even if the power of its device is off. The subprocess FlushBusReset takes care of this.

Process LE This is the top process expression which initialises all the subprocesses that are to communicate with each other. In the initialisation of this process, the network consists of 3 DMs that are not up. This parameter can then be passed to subprocesses. There are also gates, which are used for synchronous communication, which can be multi-way. For instance, process BusReset communicates over gate `gBusReset` with all instances of process CMM and of process MS. A communication from BusReset on this gate can only take place if all of the processes mentioned participate in it (enforced synchronisation). The instantiation of the DM processes is a non-deterministic choice between instantiation with UrlCapable set to either true or false.

Process BusReset As in the Promela model, the only option of this process is to start a bus reset period, but this choice may be delayed if there is any other activity in the network. Starting with ID 1, the subprocess BusReset2 decides non-deterministically for each device whether its power status changes or not. The operator `+` works modulo `4`, so `3+1` yields `0`, and at this ID the subprocess BusReset2 ends the bus reset period, and calls the top process again.

Process CMM The states of this process are reflected in the subprocesses CMMDown, CMMUp, CMMReady, CMMDeliver and CMMDeliver2.

Whenever CMMUp is executed, the CMM has to get the GUIDList first, which is available only at the end of the bus reset period. This information is sent by BusReset. After this, the CMM is ready for normal operation.

CMMReady is the normal situation when the CMM is up. It can send the GUIDList on gate `gInfo` to the DM with the same ID, or signal a bus reset start. After a bus reset start, the CMM executes CMMDeliver.

In CMMDeliver two things must happen: an update of the GUIDList must be received from BusReset, and a `bus_reset_event` must be sent to the DM on this device. If the message to DM is sent first, then all the CMM can do is wait for the reception of the new GUIDList, after which it is ready for normal operation. If the bus reset period ends before the message to DM is sent, the process CMMDeliver2 is executed.

In CMMDeliver2 the GUIDList is available again for the DM on this device, the `bus_reset_event` must be sent to the DM, and a new bus reset period may start.

At any point in this behaviour, the power of the device may be switched off, which is handled with the disrupt operator `[>`.

Process MS The states of this process are reflected in the subprocesses MSDown, MSUp, MSSuspend, and MSReady. Of these we only explain the latter two.

MSSuspend is executed whenever the MS is up, but a bus reset is taking place. No communication is possible on the network, but the DM at the same device may still receive messages from the buffer. This state is left as soon as the bus reset period ends.

MSReady is the normal situation when the MS is up.

At any point in this behaviour, the power of the device may be switched off, which is handled with the disrupt operator `[>`.

Process DCM_Manager The states of this process are reflected in the subprocesses DMDown, DMUp, DMif, DMSendRequest, DMil, DMElect, DMSendReply, DMff, DMffi, DMfl and DMfli.

In DMUp, the leader election process starts. The DM gets the GUIDList from the CMM on the same device, and uses the function `i_leader` to compute the initial leader.

In DMif and DMSendRequest, the initial follower's actions are executed (See Section 2).

In DMil, DMElect and DMSendReply, the initial leader's actions are executed (See Section 2).

In DMff, DMffi, DMfl and DMfli, the final leader and followers should determine resource allocation of the DCM units in the network, but we have not modelled this behaviour. Therefore a final leader or follower DM does nothing, except wait for a new bus reset period, a power change or (in case it was the initial leader, which is in DMffi and DMfli) answer initialisation requests with the initialisation reply.

At any point in this behaviour, the power of the device may be switched off, which is handled with the disrupt operator `[>`.

5 Model checking experiments

In order to check that the protocol works as intended, we have checked four properties on several models of the protocol. Each of the following sections is dedicated to one property. The properties are listed in this section in an informal manner and in a notation slightly different from the actual input for the tools. For the exact definitions of the properties, we refer to the Appendices C.5, D.4 and D.6.

The properties presented here were devised after the models of the protocol had been constructed. This has both advantages and disadvantages. A disadvantage is that it turns out to be rather difficult to express properties for our specific models. In fact we have had to change them slightly to make some information visible. An advantage is that the models have not been tailored towards the properties that should be checked except the changes mentioned. A potential danger is that the the model does not resemble the protocol close enough anymore, and the properties to be checked trivially hold.

Since the behaviour of the protocol is unpredictable during bus resets or the period that the CMMs need to deliver the bus reset event, we only demand that the properties be true for stable situations, that is, in states where it is not the case that a bus reset is taking place or a bus reset event should still be delivered. Since a new bus reset period may start at any moment after the previous bus reset has ended and since we have included this possibility in our models with non-deterministic choice, we get the behaviour depicted in Figure 3 from our models. Suppose that $s_1, s_2, s_3, \ldots, s_n$ in Figure 3 are stable states, which means that no bus reset is taking place, and all events concerning the last bus reset have been delivered. We see that from s_1 it is possible that a new bus reset period starts, but it is also possible that some other behaviour takes place on the transition to s_2. If we establish a property in terms of behaviour, we can only capture the desired behaviour from s_1 by using an *exists* quantifier: from s_1 there exists a behaviour which satisfies a certain requirement. Moreover, in our models the amount of activity that concerns the protocol is bounded. After a certain point, the protocol is stuck or completed, and the only possible behaviour is that a new bus reset period starts. So it is not possible to express a property as follows: "for all behaviours: if no bus reset starts in this behaviour then fulfill a requirement".

Expressing properties for Promela models Safety properties can be checked in Spin through the use of `assertion` statements. We use a process with only such an assertion statement in the verification for checking whether there is a state in which the assertion is false. If this happens, Spin reports this as an error and stops the verification. An error trace is saved which can be used for diagnostic purposes.

Liveness properties can be checked in Spin through the use of LTL [24, 20] formulas, which are translated into `never` claims. A never claim is a process which will only terminate if the corresponding LTL formula was violated. Actually, never claims represent ω-regular properties. Spin checks whether never claims hold in the initial state. This means that if the never claims is already satisfied by the initial state, no further exploration of the state space is needed.

Both assertions and LTL formulas are expressed in terms of predicates, which range over values of variables. It is also possible to check a pattern of communications, but not in combination with checks of state variable values. Since in our case, it is by far easiest to find error situations by referencing the state variable values, we stick to the assertions and never claims.

Expressing properties for Lotos models We express safety and liveness properties in ACTL [4, 3] for the verification of the Lotos models is done with A property is checked by Xtl on the reachable state space, by checking for each reachable state whether the property holds in that state.

Since the model checker Xtl is only used on state spaces which have been generated from the Lotos model, the information of state variables is lost. Actually, the states are identified by natural numbers in the state graph accepted by Xtl. This means that we cannot express properties in terms of values of state variables, and we can just observe the occurrences of actions. A consequence of this approach is that safety properties can only be expressed as liveness properties. With the ACTL logic we are able to observe certain patterns of occurrences of actions. In order to still reference state variable values, one could build self-loops into the Lotos model. which give the values of the state variables in

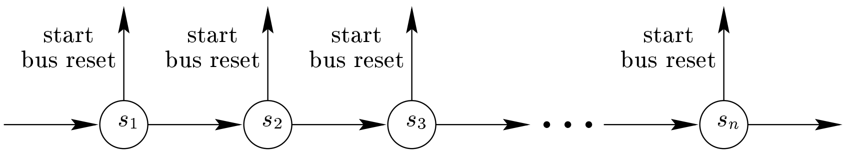

Figure 3: Protocol behaviour

that state. However, this was not a feasible approach in our case.

An action can be observed by comparing an action label from a transition to a label set in the property that is being checked. Comparing an action label to the label set **T** (**F**) always succeeds (fails). Label sets can be constructed from syntactic expressions that capture one or more action labels, and boolean operators. For instance, it is straightforward to construct a label set that succeeds when compared to the label `BUS_RESET_START` or the label `POWER_CHANGE` and fails otherwise.

In order to enable the checking of not just communications between the DCM Managers, but also other important actions, the model contains a few extra observable events. These the occurrences of communication on the special gate `GEvent`. In this way we observe a DCM Manager electing itself for initial or final leader.

We now give an overview of the ACTL operators used, and their informal meaning[1].

$\mathbf{T}, \neg, \wedge, \vee, \rightarrow$ Boolean true, negation, and, or, implication

$[a]\ \phi$ For every transition $s \xrightarrow{a} t$ from the current state: formula ϕ must hold in the target state t

$\forall \mathbf{G}_a \phi$ For each (possibly finite) path from the current state where all actions are either a or τ, formula ϕ must hold in every state

$\exists(\phi_a \mathbf{U}_b \psi)$ There exists a path from the current state along which for a finite fragment formula ϕ holds in each state and all actions are either a or τ, and this fragment is immediately followed by a transition $s \xrightarrow{b} t$, and in state t formula ψ holds.

For a complete list of ACTL operators and a formal definition, we refer to [10, 4, 3].

The standard library in the Cæsar/Aldébaran distribution for using these operators is the `actl.xtl` library (implemented by Mateescu [22]) which establishes the validity of a formula by checking whether the formula holds in all reachable states of the Lotos model. This library is not implemented in such a way that it gives diagnostics in case a property is not true. Diagnostics can be obtained by using the `walk_actl.xtl` library (implemented by Pecheur [23]), which also implements the ACTL operators mentioned, and which tries to find an error trace. This implementation establishes the validity of a formula by checking whether the formula holds in the initial state of the Lotos model. Of course, in general the use of this library is more costly since there is more administration involved in finding the trace, and a lot of backtracking occurs.

5.1 Safety: At most one leader

It is never the case that more than one DCM Manager is a (initial or final) leader.

Spin We use an `assertion` statement, and check the following formula:

$$\forall d, d'.(\neg \mathit{bus\_reset} \wedge \mathit{leader}(d) \wedge \mathit{leader}(d') \ \rightarrow \ (d = d'))$$

This property does not hold for any of the models. It turns out that the error trace found by Spin for the property checked in Section 5.3 is also an error trace for this property. See Figure 4 for the trace and Section 5.3 for an explanation. In Section 5.5 we discuss whether the HAVi protocol is wrong.

Xtl What we want to establish, is that there are not multiple InitialLeader or FinalLeader events in between bus reset periods. Since we can check for patterns of actions, we formulate the property as follows: if a bad pattern of Initial or FinalLeader events occurs, then we are not in a stable situation (where no bus reset is taking place and the last bus reset events have all been delivered). This boils

[1]Note that the [] operator does not have the ACTL interpretation, but the interpretation of the Hennessy-Milner modal logic [10]. Since the Xtl library for ACTL is defined using the Xtl libraries for the Hennessy-Milner modal logic and the modal μ-calculus [18], we can use operators from these logics in any ACTL expression

down to expressing that when a bad pattern does occur outside bus reset periods, apparently a bus reset event must still be delivered.

We check the following formula:

$$([b_1]\,\forall\mathbf{G}_{i_1}([i_3]\,\forall\mathbf{G}_{i_1}([i_3]\,\exists(\mathbf{T}_{i_2}\mathbf{U}_{b_2}\mathbf{T})))\ \wedge\ ([b_1]\,\forall\mathbf{G}_{i_1}([f]\,\forall\mathbf{G}_{i_1}([i_4\cup f]\,\exists(\mathbf{T}_{i_2}\mathbf{U}_{b_2}\mathbf{T})))$$

$$\begin{aligned}\text{where}\quad b_1 &= \mathit{BusResetEnd}\\ b_2 &= \mathit{BusResetEvent}\\ i_1 &= \mathit{Ignore}_1 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{Initleader} \vee \mathit{FinalLeader})\\ i_2 &= \mathit{Ignore}_2 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart})\\ i_3 &= \mathit{InitLeader}\\ i_4 &= \mathit{InitLeader} \vee \mathit{FinalLeader}\\ f &= \mathit{FinalLeader}\end{aligned}$$

This formula expresses two patterns that should be followed by a bus reset event being delivered. Both patterns start with the end of a bus reset period, and do not allow the start of a new bus reset period by the use of the action label sets Ignore_1 and Ignore_2. The first pattern checks the double occurrence of the InitialLeader event. The second pattern checks the occurrence of a FinalLeader event, followed by either an InitialLeader or FinalLeader event. The action label sets in the subscript of the **G** and **T** symbols enable the actions in the subscripts to occur in any sequence in between.

This property holds for all models. Since we found errors in the Promela models for this property using Spin two questions remain, namely whether the error behaviour found with Spin also occurs here and if so, why it is not found with the ACTL formula used. Simulating the behaviour from the Spin error trace is possible for the Lotos model with two DCM Managers and synchronous behaviour. As to the second question. The answer is that the label set Ignore_1 is too restrictive. The idea of checking a pattern when a bus reset event has completed turns out counterproductive. We might have checked *all* occurrences of the FinalLeader event followed by bad patterns, and qualified the occurrence of a BusResetStart, BusResetEnd or BusResetEvent as a good pattern. In any case, it appears that the formulation of the property in this setting is very complicated. In Section 5.5 we discuss whether the HAVi protocol is wrong.

5.2 Safety: Best candidate becomes final leader

It is never the case that a final leader is selected which is not UrlCapable, while there is a DCM Manager active in the network which is UrlCapable.

Spin We use an `assertion` statement, and check the following formula:

$$\neg\mathit{bus\_reset} \wedge \forall d.((\mathit{f\_leader}(d) \wedge \neg\mathit{url\_capable}(d))\ \rightarrow\ \forall d'.(\mathit{up}(d') \rightarrow \neg\mathit{url\_capable}(d')))$$

This property holds for all models except for the setting with three DCM Managers and asynchronous communication. However, the error found here reveals problems with the interpretation and execution of the Promela code rather than an error in the protocol. In fact, we can reason why in our model the property should be true for any number of DCM Managers with either synchronous or asynchronous communication. The idea is that upon receipt of a bus reset event, each DCM Manager will clear the information of being final leader and ask for the new network topology (the GUIDList). Since the start of a bus reset period causes the delivery of a bus reset event at some time, in a stable situation all bus reset events have been delivered, and each DCM Manager must have the correct network topology information. So after the last bus reset event delivery to a DCM Manager, it cannot choose a non UrlCapable final Leader if there is a UrlCapable DCM Manager present. So the only way in which a non UrlCapable DCM Manager can still be the final leader in a stable situation, while a UrlCapable DCM Manager is present, is to receive an InitReply with its identity from the initial leader, when the initial leader has not received the latest bus reset event. But we have modelled the final leader election by having the initial leader choose itself, if no UrlCapable Manager is present. So it cannot ever send an InitReply with the identity of another, non UrlCapable DCM Manager. It is clear that although the property must hold in our models, it does not hold when we lift the restriction that the initial leader chooses itself for final leader when no UrlCapable DCM Manager is present. In Section 5.5 we discuss whether the HAVi protocol is wrong.

Xtl The situation that a DCM Manager is up and UrlCapable is signalled by the request from such a DCM Manager to the initial leader, in which the UrlCapable parameter is true. Whenever such a request is followed by the election of a final leader which is not UrlCapable, there must be a bus reset event pending that needs to be delivered.

We check the following formula:

$$[u]\,\forall \mathbf{G}_{i_1}([f]\,\exists(\mathbf{T}_{i_2}\mathbf{U}_b\mathbf{T}))$$

$$\begin{aligned} \text{where}\quad & b = \mathit{BusResetEvent} \\ & i_1 = \mathit{Ignore}_1 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{BusResetEnd} \vee \mathit{FinalLeader}) \\ & i_2 = \mathit{Ignore}_2 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart}) \\ & f = \mathit{FinalLeaderNotUrlCapable} \\ & u = \mathit{RequestUrlCapable} \end{aligned}$$

This property holds for all models. See the paragraph above on Spin experiments for this property, for a discussion whether this property holds in general or not. In Section 5.5 we discuss whether the HAVi protocol is wrong.

5.3 Safety: All agree on the final leader

Whenever a final leader is selected, all DCM Managers agree on the identity of this leader. Of course this can only be checked as soon as all DCM Managers have been informed of the decision of the initial leader. Since the final leader is informed last of the decision (and whenever this happens to be also the initial leader, it will 'inform itself last'), this can be checked as soon as one of the DCM Managers has been elected for final leader.

Spin We use an `assertion` statement, and check the following formula:

$$\forall d.(\neg \mathit{bus\_reset} \wedge \mathit{f\_leader}(d) \;\rightarrow\; \forall d'.(\mathit{up}(d') \rightarrow \mathit{leader\_id}(d') = d))$$

This property does not hold for any of the models. In Figure 4 an error trace constructed by Spin for the model with two DCM Managers and synchronous communication is listed. This trace describes the following behaviour. In the first bus reset period both DCM Managers are powered up. They start leader election, in which DCM Manager A is the initial leader and DCM Manager B is the initial follower. B is UrlCapable and A is not. B sends A an InitRequest, A computes the final leader which is B, and sends the InitReply to B. A new bus reset period starts and ends without change in the network topology. The CMM on the device of B delivers the bus reset event to B, and B starts afresh with the leader election. B is again initial follower and sends A an InitRequest. A does not know about the second bus reset period so it is in its final follower phase where it answers any InitRequest with the same InitReply as before. A sends B the InitReply and B concludes it is the final leader. Now the CMM on the device of A delivers the bus reset event to A, and A starts afresh with the leader election. A is again initial leader and does not know the identity of the final leader to be elected, while B still thinks it is final leader. In this state the property checked is violated. In Section 5.5 we discuss whether the HAVi protocol is wrong.

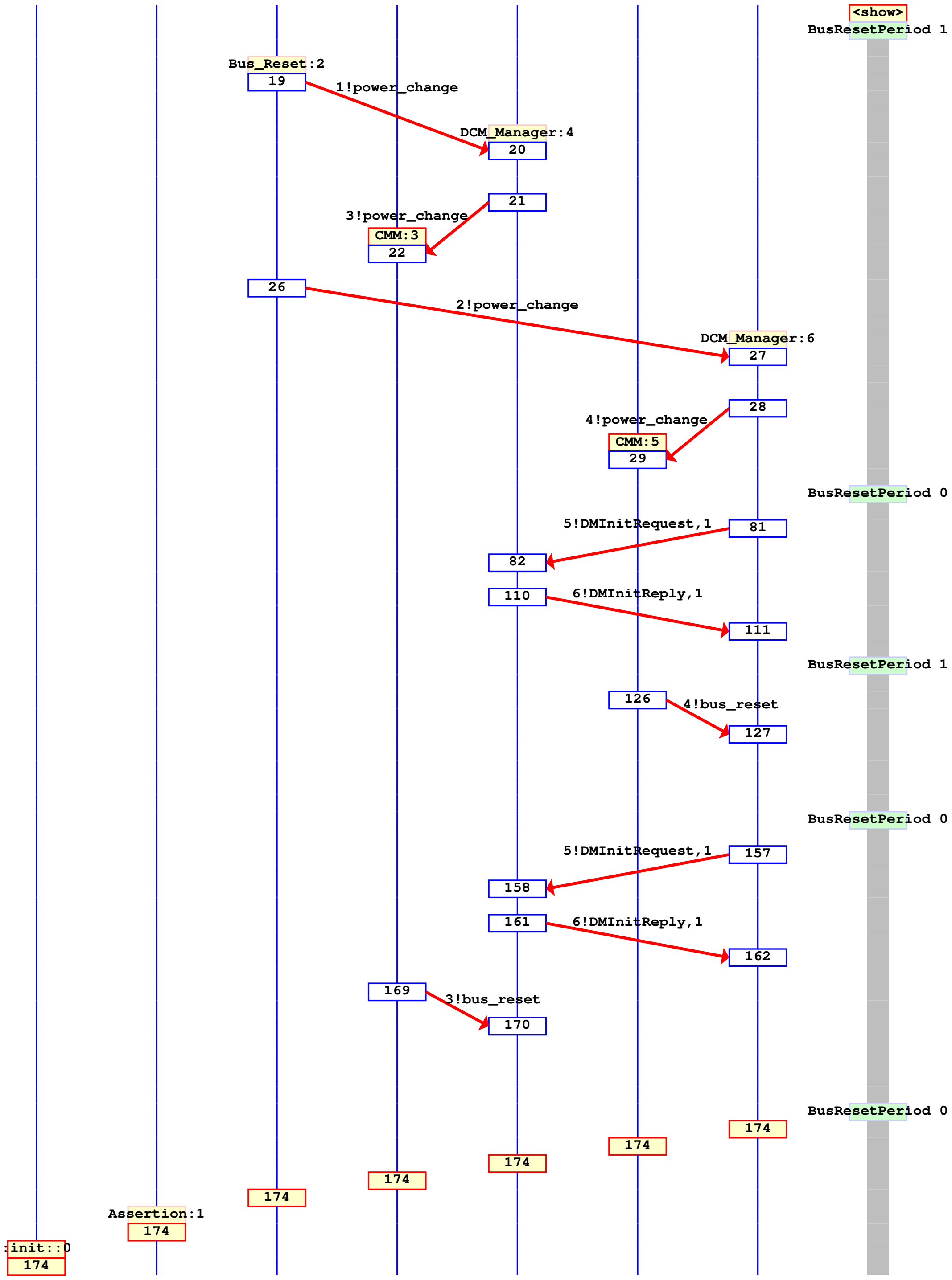

Figure 4: The Spin error trace for ‘one leader’ and ‘same final leader’

```
AG_A(A, F) is FALSE
  0  : (0, "GBUSRESET !BUS_RESET_START", 5036)
  1  : (5036, "GUPDOWN !1 !POWER_CHANGE", 3437)
  2  : (3437, i, 4798)
  3  : (4798, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(FALSE))", 4797)
  4  : (4797, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(FALSE))", 4790)
  5  : (4790, "GBUSRESET !BUS_RESET_START", 4789)
  6  : (4789, "GEVENT !INIT_LEADER !1", 4769)
  7  : (4769, i, 133)
  8  : (133, "GUPDOWN !2 !POWER_CHANGE", 142)
  9  : (142, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(TRUE))", 658)
  10 : (658, "GEVENT !FINAL_LEADER !1 !FALSE", 150)
  11 : (150, "GINFO !2 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 2542)
  12 : (2542, "GDMOUT !1 !CONSM(DMINITREQUEST,2,TRUE)", 552)
  13 : (552, "GDMIN !1 !CONSM(DMINITREQUEST,2,TRUE)", 2524)
  14 : (2524, "GDMOUT !2 !CONSM(DMINITREPLY,1,FALSE)", 316)
  15 : (316, "GINFO !1 !BUS_RESET_EVENT", 303)
  16 : (303, "GDMOUT !1 !EMPTY", 1924)
  17 : (1924, "GDMOUT !1 !CONSM(DMINITREQUEST,2,TRUE)", 1921)
Box(A, F) is FALSE
  18 : (1921, "GDMIN !2 !CONSM(DMINITREPLY,1,FALSE)", 1909)
AG_A(A, F) is FALSE
  19 : (1909, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 1486)
  20 : (1486, "GEVENT !INIT_LEADER !1", 1662)
  21 : (1662, "GDMIN !1 !CONSM(DMINITREQUEST,2,TRUE)", 1507)
Box(A, F) is FALSE
  22 :  (1507, "GDMOUT !2 !CONSM(DMINITREPLY,2,FALSE)", 1548)
EU_A_B(F, A, B, G) is FALSE
*Failure.*
```

Figure 5: The Xtl error trace for ‘same final leader’

Xtl We can only check that everyone has the same leader identity by checking the parameters of messages/events concerning the final leader. We require the leader identity parameter to be equal for all such actions in stable situations. So the property must express that whenever two actions carry a different leader identity outside a bus reset period, apparently a bus reset event must still be delivered.

We check the following formula:

$$
\begin{aligned}
&\forall d.\ [l_d]\ \forall \mathbf{G}_{i_1}([l_{\neg d}]\ \exists(\mathbf{T}_{i_2}\mathbf{U}_b\mathbf{T}))\\
&\quad \text{where} \quad b = \mathit{BusResetEvent}\\
&\qquad\qquad i_1 = \mathit{Ignore}_1\\
&\qquad\qquad\quad = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{BusResetEnd} \vee \mathit{InitReply} \vee \mathit{FinalLeader})\\
&\qquad\qquad i_2 = \mathit{Ignore}_2 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{BusResetEnd})\\
&\qquad\qquad l_d = (\mathit{InitReply} \vee \mathit{FinalLeader}) \text{ with leader identity } d\\
&\qquad\qquad l_{\neg d} = (\mathit{InitReply} \vee \mathit{FinalLeader}) \text{ with leader identity not equal to } d
\end{aligned}
$$

This property holds only when communication between DCM Managers is synchronous. In the asynchronous case an erroneous initialisation reply may be lingering in someones input queue, after the corresponding bus reset event has been handled by the sender of the erroneous message. In Figure 5 an error trace constructed with the `walk_actl` library is listed. The behaviour described by this trace is as follows. In the first bus reset period DCM Manager A is powered up. A is not UrlCapable. A starts the leader election and elects itself for initial leader. In the second bus reset period DCM

Manager B is powered up. B is UrlCapable. After the second bus reset, A has not received the bus reset event yet. A elects itself for final leader which completes the leader election. B elects A for initial leader and sends an InitRequest. A receives the InitRequest from the MS and sends an InitReply with its own identity for final leader. Now A receives the bus reset event and starts the leader election anew. B has not received the InitReply from the MS yet and sends a second InitRequest to A. Now B receives the InitReply from the MS and concludes that A is the final leader. A elects itself for initial leader, and receives the second InitRequest that B sent from the MS. A elects B for final leader and sends an InitReply with the identity of B for final leader. The property is violated.

Since we found errors for the Promela models with synchronous communication using Spin, two questions remain, namely whether the error behaviour found with Spin also occurs here and if so, why it is not found with the ACTL formula used. In Section 5.1 we have already simulated the error behaviour found by Spin and depicted in Figure 4 on the Lotos model with two DCM Managers and synchronous communication. As to the second question. The ACTL formula used only checks communication involving leader identities. Here we are really hampered by the fact that for the current Lotos models it is not possible to include state information in the formula. It turns out that in the synchronous Lotos models a bus reset event will appear in between the two events carrying a different leader identity. Since such a pattern is in general not erroneous, it is not possible with this approach to find the erroneous behaviour constructed with Spin. In Section 5.5 we discuss whether the HAVi protocol is wrong.

5.4 Liveness: Eventually there will always be a final leader

Whenever there is at least one DCM Manager active in the network, there should eventually be a final leader. The property we check is whether from each stable state in which at least one DCM Manager is up there exists a path on which no bus reset period starts and a final leader is chosen. It may be argued that this property is too strong since it assumes that there exists a path on which bus reset periods can be delayed until after the election of the final leader. If the environment would violate this assumption, the property would be false even when the protocol was correct. There are two reasons for our approach. First, we know that in our models the choice between a bus reset period starting and any other activity is non-deterministic. So bus reset periods can be delayed as long as other activity is possible. Second, the alternative property to be checked would be: ‘After the handing out of the GUIDList, each path leads to a new bus reset period or a final leader being elected’. This formula requires that during and after the leader election activity, the DCM Managers can perform idle/internal actions indefinitely, in order to distinguish between situations where leader election is interrupted by a bus reset period and situations where leader election does not terminate for some other reason, i.e. livelock rather than deadlock, since in case of a deadlock a bus reset period is forced to start. Moreover, the models already contain a livelock when there are more two initial followers of which one keeps sending InitRequests and the other never gets a turn. The problem with livelocks is that the property should then be checked under certain fairness aspects. This makes the situation increasingly complex, and we have chosen to stick with the first formulation.

Spin The only way to model a liveness property like this and have Spin check its validity, is with an LTL formula. We have been able to express this without too much trouble in ACTL, as can be seen below. However, the expressivity of LTL and branching time logics like ACTL is not comparable [27]. When we try to express the property to be checked in LTL and formulate it as follows, we get an expression which is not in LTL syntax:

$$\Box((\neg \mathit{bus\_reset} \wedge (\exists d.\ \mathit{up}(d))) \ \rightarrow\ \exists(\neg \mathit{bus\_reset}\ \mathbf{U} \neg \mathit{bus\_reset} \wedge \exists d.\ \mathit{f\_leader}(d)))$$

Because of the $\exists$ operator, this is not an LTL formula. However, we do need an $\exists$ operator to express the behaviour that the Promela models should have (See also Figure 3). The reason is that an LTL formula is interpreted to be true if and only if it holds for each behaviour of the model. So if it is only

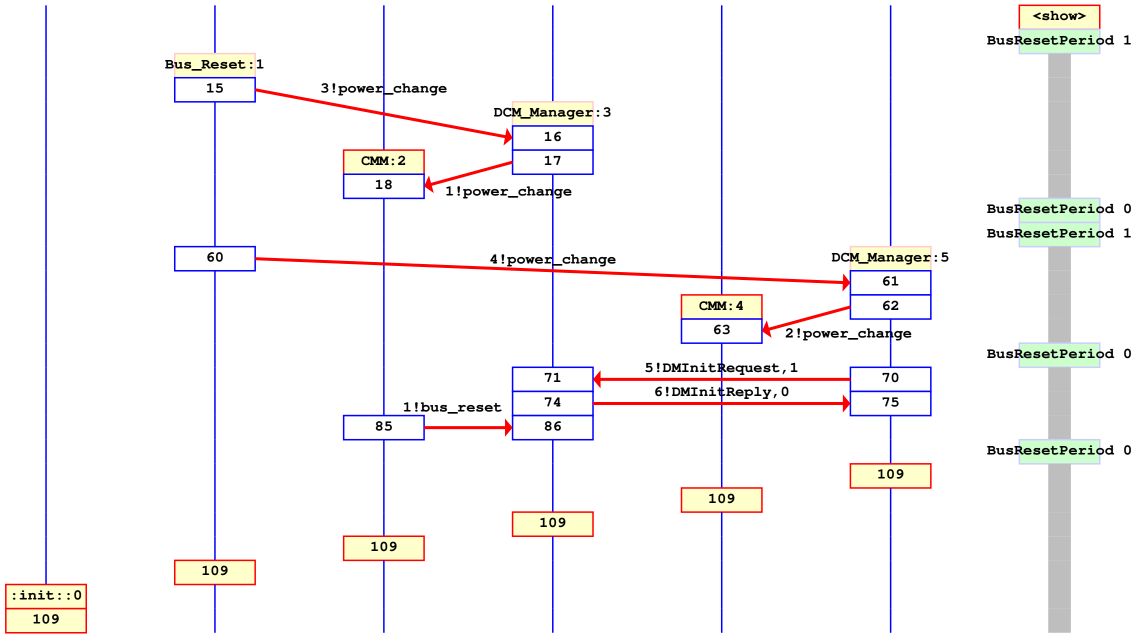

Figure 6: The Spin error trace for 'always final leader'

possible to express desired or undesired properties for one behaviour. But the property that we desire to have is that there always exists a good path. The property that we desire not to have is that there is no state from which there are only bad paths. This cannot be expressed in LTL. This problem has been discussed via e-mail [2, 13], but no solution was found, other than to change the model such that there is a fixed number of bus reset periods, after which the network remains stable. Then Spin's capability to find invalid end states can be used to check that the protocol ends up with a leader, or identify a finite path as undesirable with LTL. A drawback of this approach is that it is not a priori clear how many bus reset periods should be allowed to obtain correctness for the more general model. However, we already found errors in the Spin models for other properties, and in the Lotos models for this property. In the Spin models, errors occur already after two bus reset periods. We have changed all models such that at most two bus reset periods can take place, and added labels to indicate what states in the model are valid end states. Then it turns out that all new models have an invalid end state, which indicates that the protocol ends without electing a final leader even though at least one DCM Manager is up.

In Figure 6 the error trace constructed by Spin for the model with two DCM Managers and synchronous communication is listed. This trace describes the following behaviour. In the first bus reset period DCM Manager A is powered up. The first bus reset period is immediately followed by a second, in which DCM Manager B is powered up. A and B are both not UrlCapable. After the end of the second bus reset period, A does not receive the bus reset event yet. Now both A and B start the leader election, in which DCM Manager A is the initial leader and DCM Manager B is the initial follower. B sends A an InitRequest, A computes the final leader which is A, and sends the InitReply to B. B concludes that A is the final leader which completes the leader election. Now the CMM on the device of A delivers the bus reset event to A, and A starts afresh with the leader election. A is again initial leader and waits for the InitRequest from B, while B has already completed leader election. Since there is no action possible we are in an end state, and since for A the leader election has not been

```
AG_A(A, F) is FALSE
  0  : (0, "GBUSRESET !BUS_RESET_START", 962)
  1  : (962, "GUPDOWN !1 !POWER_CHANGE", 72)
  2  : (72, i, 1024)
  3  : (1024, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(FALSE))", 1023)
  4  : (1023, "GBUSRESET !BUS_RESET_START", 820)
  5  : (820, i, 612)
  6  : (612, "GUPDOWN !2 !POWER_CHANGE", 542)
  7  : (542, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(TRUE))", 288)
  8  : (288, "GINFO !2 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 97)
  9  : (97, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 335)
  10 : (335, "GEVENT !INIT_LEADER !1", 231)
  11 : (231, "GDM !1 !2 !DMINITREQUEST !FALSE", 199)
  12 : (199, "GDM !2 !1 !DMINITREPLY !1", 995)
  13 : (995, "GINFO !1 !BUS_RESET_EVENT", 95)
Box(A, F) is FALSE
  14  :     (95, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 1003)
EU_A_B(F, A, B, G) is FALSE
*Failure.*
```

Figure 7: The Xtl error trace for 'always final leader'

completed, it is an invalid end state. In Section 5.5 we discuss whether the HAVi protocol is wrong.

Xtl We check whether a DCM Manager is up in a stable state by observing the transaction in which the CMM hands out the GUIDList. We check whether a final leader is elected by observing the FinalLeader event. We demand that there exists a path from each GUIDList transaction on which no bus reset period starts and on which a FinalLeader event occurs.

We check the following formula:

$$[g]\, \exists(\mathbf{T}_i \mathbf{U}_f \mathbf{T})$$
$$\text{where} \quad i = \mathit{Ignore}$$
$$g = \mathit{GetGUIDList}$$
$$f = \mathit{FinalLeader}$$

This formula does not hold for any of the models.

In Figure 7 an error trace constructed with the `walk_actl` library is listed. By coincidence, the behaviour described by this trace is the same as the behaviour described by the error trace found by Spin for this property. See earlier in this section for an explanantion of the behaviour. In Section 5.5 we discuss whether the HAVi protocol is wrong.

5.5 Is the HAVi protocol wrong?

The error traces given in Figures 4, 5, 6 and 7 show that either our model of the protocol or the HAVi specification itself must be wrong.

The error traces indicate that problems occur when the delivery of a bus reset event message is delayed beyond the duration of the sending and delivery of both a message and a response between different devices. In the case of synchronous communication, another cause of problems is the availability of the GUIDList before the delivery of the corresponding bus reset event.

If all assumptions and restrictions that we made in our model are correct, then these scenarios may occur in an implementation that is totally compliant with this version of the HAVi specification, because of two reasons. First, the HAVi specification does not lay down how long messages may be on their way in the system. Second, the delivery of any event has to go through the Event Manager. The Event Manager may cause a delay of the event for several reasons. It is not known how many events the Event Manager may get due to a bus reset period, which need to be delivered, and in what manner these events are processed. Furthermore, there may be many components that listen to the bus reset event and in a sequential approach to delivery of the events, the DCM Manager may very well be the last of them to receive this message.

If our assumptions are not correct, then obviously it is hard to say whether the protocol would be correct or not. However, all of the assumptions we made are restrictions on configurations or scenarios permitted by the HAVi document which means that we only exclude some HAVi behaviour. So the error behaviour we found would almost certainly be present in a model with fewer restrictions. In fact, the chances are high that with fewer restrictions more erroneous behaviour could be found in the protocol. We already argued in Section 5.2 that lifting the restriction that the initial leader chooses itself for final leader when no UrlCapable DCM Managers are present, will lead to violations of the property 'the best candidate becomes final leader'. Other generalisations we could make are: several types of devices in the network, physical IDs that change, bus reset periods that start and end at different moments in different devices, no difference between processing of events and messages, et cetera. Also, it may still be the case that one or more of the software elements used for this protocol have a potential deadlock in their behaviour, and thus prevent the DCM Managers from completing their leader election.

Our conclusion is that for the HAVi leader election protocol to be correct (meaning that any implementation that complies with HAVi works correctly), the HAVi specification should have requirements added on the duration of delivery of events related to the duration of communication between devices. Since the disruption by bus reset periods makes it difficult to establish such requirements, we think the easiest solution is to establish real-time constraints on the duration of sending and processing messages and events, which are realistic for HAVi-compliant implementations. This information should then be checked in a timed formal verification. Since timed model checking is beyond the scope of this experiment, we cannot give an estimate of time bounds that would work, or say whether such time bounds exist.

5.6 Statistics

The statistics for model checking the different models with the Spin tool set (version 3.2.4, version 3.3.0 beta-13 May 1999) and the Cæsar/Aldébaran tool set (Cæsar version 5.3, Aldébaran version 6.4, Xtl version 1.1) are given in Tables 1, 2 and 3. All experiments with Spin were done on an SGI IRIX64 6.5 machine with 42 Gbyte of memory. All Cæsar/Aldébaran experiments were done on a SUN Ultra 5_10 SunOS 5.6 machine with 1 Gbyte of memory.

A few remarks are in order.

- Spin, Cæsar, Aldébaran and Xtl all generate C code which after compilation performs the state space generation, minimisation and/or exploration.
- The memory numbers mentioned in Table 3 indicate the amount of memory used by the verifier generated by Xtl in C code, compiled to executable form. However, C compilation takes at least 6 Mb. For the `walk_actl` library, C compilation takes at least 12 Mb for the models with 2 DCM Managers.
- For the Spin experiments, the memory usage is provided in the output of Spin. Note that this is always a little higher than the memory usage observed with the UNIX command 'top'. For the Cæsar, Aldébaran and Xtl experiments, the memory usage is obtained by observing the outcome of the UNIX command 'top'.
- For all experiments, the timing information is obtained by the UNIX command 'time'.

model	states	trans	holds?	mem (Mb)	time (h:m:s)	Spin
one leader						
2 sy	18K	93K	F	136	0:00:04	3.2.4
2 as	23K	108K	F	135	0:00:06	3.3.0 beta
3 sy	781K	4.7M	F	161	0:03:59	3.3.0 beta
3 as	2.8M	18M	F	230	0:15:30	3.3.0 beta
best final leader						
2 sy	167K	806K	T	140	0:00:43	3.3.0 beta
2 as	418K	2.1M	T	149	0:01:57	3.3.0 beta
3 sy	44M	279M	T	1767	7:32:22	3.3.0 beta
3 as	194M	3.8G	F	7778	49:43:03	3.3.0 beta
same final leader						
2 sy	16K	77K	F	135	0:00:04	3.3.0 beta
2 as	19K	91K	F	135	0:00:05	3.3.0 beta
3 sy	407K	2.5M	F	148	0:02:05	3.3.0 beta
3 as	1.7M	10M	F	190	0:08:54	3.3.0 beta
always final leader						
2 sy	17K	58K	F	135	0:00:04	3.3.0 beta
2 as	27K	98K	F	135	0:00:06	3.3.0 beta
3 sy	674K	3.2M	F	134	0:03:32	3.3.0 beta
3 as	1.5M	7.5M	F	182	0:07:52	3.3.0 beta

Table 1: Spin statistics: state space generation + model checking

- Normally, Lotos state space generation is done with Cæsar in the `.bcg` format, which is very compact. However, Cæsar does not always create the smallest state space possible, and for the models in this case this means that state space generation gets stuck at an unknown portion of the desired total, and fails due to lack of memory. So we turned to an alternative route, and generated the state spaces separately for each instance of each process in the main parallel composition expression. This again is done with Cæsar. The state spaces generated are first minimised with respect to strong bisimulation equivalence (with Aldébaran and the bmin criterion), which is also done in the `.bcg` format. Then these minimised state spaces must be combined into one state space. This is done with Aldébaran and works only if the separate state spaces are in the `.aut` format. The target state space is then also in the `.aut` format. The `.bcg` version is computed and then minimised.

  When generating the state space for one of the communicating processes, often the receipt of a messages is not restricted other than by all possible instantiations of the parameters of the communication. These parameter values had to be restricted in the separate process definitions to make state space generation manageable. Without these restrictions, it was not possible to generate a state space for the DCM Manager process with the lowest identity, in the case of asynchronous communication and 3 DCM Managers.

- All state space generation sizes in Table 2 are for a state space in the `.bcg` format, except the *comb network* entries which represent a state space in the `.aut` format. Minimised state spaces are always in the `.bcg` format. In some cases, the `.bcg` version has fewer states for the same state space than the original `.aut` version.

- In Table 3, the full state space size is listed for each model. When using the `actl` library, the full state space is explored, even when errors are found. When using the `walk_actl` library, the

2 DCM Managers, synchronous								
	generating				minimising			
per process	*states*	*trans*	*mem (Mb)*	*time (h:m:s)*	*states*	*trans*	*mem (Mb)*	*time (h:m:s)*
DM 1	16K	79K	3	0:00:11	32	144	9	0:00:04
DM 2	1.0K	5.4K	3	0:00:03	21	96	4	0:00:02
Bus_Reset	46	59	3	0:00:02	16	24	4	0:00:01
CMM 1,2	12K	55K	3	0:00:09	12	49	5	0:00:02
Other 1,2	2	8	3	0:00:02	1	4	4	0:00:01
comb network	1.5K	5.0K	5	0:00:01	1.2K	4.0K	4	0:00:01
2 DCM Managers, asynchronous								
DM 1	404K	3.0M	28	0:05:12	37	233	183	0:01:18
DM 2	2.1K	18K	3	0:00:05	27	170	4	0:00:01
Bus_Reset	46	59	3	0:00:02	16	24	4	0:00:01
CMM 1,2	12K	55K	3	0:00:09	12	49	5	0:00:02
MS 1,2	2.3K	19K	3	0:00:04	27	159	4	0:00:02
comb network	6.3K	23K	7	0:00:07	5.1K	19K	5	0:00:02
3 DCM Managers, synchronous								
DM 1	2.1M	16M	140	0:43:46	63	474	897	0:08:52
DM 2	177K	1.4M	12	0:02:31	37	255	92	0:00:39
DM 3	4.6K	38K	3	0:00:07	25	174	4	0:00:02
Bus_Reset	186	243	2	0:00:02	40	64	3	0:00:02
CMM 1,2,3	297K	2.1M	79	0:14:00	20	93	139	0:00:49
Other 1,2,3	2	40	2	0:00:02	1	20	3	0:00:01
comb network	58K	247K	47	0:00:52	44K	193K	24	0:00:21
3 DCM Managers, asynchronous								
DM 1	2.0M	11M	109	0:24:29	55	360	620	0:05:33
DM 2	509K	3.8M	31	0:06:52	35	199	233	0:01:36
DM 3	9.8K	105K	3	0:00:13	31	254	10	0:00:03
Bus_Reset	186	243	3	0:00:02	40	64	4	0:00:02
CMM 1,2,3	297K	2.1M	111	0:13:57	20	93	139	0:00:47
MS 1,2,3	3.9K	47K	3	0:00:08	35	279	6	0:00:02
comb network	1.0M	5.2M	358	0:31:12	748K	3.9M	423	0:10:21

Table 2: Cæsar/Aldébaran statistics: state space generation

verification stops after the construction of the first diagnostic trace. We do not know how many states and transitions were explored by `walk_actl` to construct the diagnostic traces.

- The Promela models for 2 DCM Managers are more efficient than the ones with 3 DCM Managers in the sense that they use the datatype `bit` instead of `byte` for the Id parameter in the general process `DCM_Manager`.
- With Spin we first tried to explore the whole state space. Whenever an error was found, we reran the verification with a smaller search depth (option '-m' at run time) to see if a smaller error trail could be found. In this way we found the trails reported in Table 1, which are the shortest trails we could find. Sometimes the search for a shorter trail involves the exploration or more states and transitions, due to the order in which the depth-first search is performed.

  Only after completion of the verification experiments, we found that the option '-DREACH' (to be used at compile time) guarantees a complete search of the truncated state space. This explains

one leader					
model	*states*	*trans*	*holds?*	*mem (Mb)*	*time (h:m:s)*
2 sy	1.2K	4.0K	T	3	0:00:08
2 as	5.1K	19K	T	3	0:00:13
3 sy	44K	193K	T	4	0:05:11
3 as	748K	3.9M	T	68	25:12:18
best final leader					
2 sy	1.2K	4.0K	T	3	0:00:05
2 as	5.1K	19K	T	3	0:00:08
3 sy	44K	193K	T	4	0:02:17
3 as	748K	3.9M	T	68	10:31:08
same final leader					
2 sy	1.2K	4.0K	T	3	0:00:05
2 as	5.1K	19K	F	3	0:00:11
3 sy	44K	193K	T	4	0:05:51
3 as	748K	3.9M	F	69	29:05:41
error trace same f_leader					
2 as	5.1K	19K	F	5	0:00:26
3 as	748K	3.9M	F	199	15:22:48
always final leader					
2 sy	1.2K	4.0K	F	3	0:00:05
2 as	5.1K	19K	F	3	0:00:08
3 sy	44K	193K	F	4	0:03:28
3 as	748K	3.9M	F	69	18:57:39
error trace always final leader					
2 sy	1.2K	4.0K	F	3	0:00:05
2 as	5.1K	19K	F	3	0:00:08
3 sy	44K	193K	F	6	0:05:32
3 as	748K	3.9M	F	199	14:16:50

Table 3: Xtl statistics: model checking

why we found a shorter error trail with Spin version 3.2.4 in one case than with Spin version 3.3.0 beta. The '-DREACH' option may increase memory usage and duration of verification experiments. It is very well possible that with this option we would have been able to find the error in the model with three DCM Managers and asynchronous communication for the property 'best final leader' with a much smaller search depth. Without the '-DREACH' option we did not find an error with search depth '-m1000' but ran out of memory.

- Checking the property 'best final leader' for the Promela model with 3 DCM Managers and asynchronous communication was done with the new Spin 3.3.0 beta option '-DSC' to keep the major part of the depth first search stack on disk, and not in memory. Otherwise this experiment would have taken much more memory. The stack file size was 281 Mbyte.

- All experiments with Spin were first done on Promela models in which the global variable `m` was 'hidden', which means that it is not part of the state vector. In this situation Spin did not explore the entire state space. Major parts of the code were unreachable because of using the hidden variable inside two branches of an 'if' statement inside an atomic statement. The predicate 'hidden' should not be used this way but this was not listed in the manuals (it is in the

Spin on-line manual now). The difference in semantics between the simulator and the verifier made the situation increasingly unclear, since the parts of the state space that were unreachable to the verifier, were reachable in simulation. Some improvements have been made in Spin 3.3.0 beta to the semantics of the simulator.

- All experiments in Spin were done without partial order reduction by using the compile time option '-DNOREDUCE'. The reason for this is that the use of synchronous communication in the escape guard of an unless command is not compatible with the partial order reduction, hence when using partial order reduction it is possible that error behaviour is missed.

- The error traces produced by Spin can be simulated interactively. The figures in this paper are the message sequence charts that were created during such simulation. The figures have been adjusted a little to improve the presentation in black and white. Each thin vertical line in the figure refers to a process in the Promela model, arrows between process lines refers to communication. The thick vertical line refers to the global variable BusResetPeriod in the Promela model. The numbers in the figures refer to steps in the error trail.

- The error traces produced by Xtl were found with the use of the `walk_actl` library. Traces are only produced in case of a universal property that does not hold, or an existential property that does hold. Since we used universal properties, we got traces only in case of an error. The error traces were constructed from end to beginning, and have been reversed in the figures to improve the presentation. The layout of the steps is:
`<step nr> : (<source state>, <transition label>, <target state>)`
The transition labels consist of the gate and the offers exchanged at the gate (each offer is preceded by `!`). In between of the steps, messages occur that indicate that a temporal operator from the formula checked does not hold at that point.

6 Conclusions

We have modelled the leader election protocol among DCM Manager components in the HAVi architecture, and found that this protocol does not meet some safety requirements and that it does not always converge to a situation with a leader actually elected. The errors are due to the absence of requirements on how long it takes for messages and events to reach their destination. It is expected that if these requirements are added, a formal verification will be able to show whether the restricted protocol works correctly.

6.1 Concerning Spin

Using Promela and Spin Promela is an easy language at first, and more difficult at second sight. The basic language constructs have an intuitive meaning, but combining many aspects such as rendezvous communication, the atomic and the unless construct makes behaviour more fuzzy. The treatment of data is manageable as long as the data is not too involved. In our case, we are clearly overstepping the bounds of the type of model for which Promela was designed.

The graphical interface of Spin is attractive, and it is easy to use. The semantics of the simulator and the verifier have been made more alike recently, which is very important since simulation is often used as a justification for having modelled things right. We are in favour of the semantics being exactly the same for simulator and verifier. After a while, we turned to the command-line use of the tools rather than the graphical interface. This was partly due to the experimental use of Spin on a 64 bit machine.

Expressing safety properties in assertions is very straightforward. Expressing liveness properties in LTL is rather cumbersome and proved impossible in our case, mostly because of the nature of LTL and the nature of the protocol. However, the possibility to track invalid end states was a simple way around this, although it implied changing the models.

Performance of Spin As can be seen in Table 1, the performance of Spin is quite good, as long as the number of DCM Managers remains small, and there are no asynchronous channels. We achieved the best performance by using all the advice given in Spin's Help Section on reducing the state space size. Of course, when the communication channels in a Promela model are asynchronous rather than synchronous, the state space grows tremendously because of all possibilities of interleaving the sending and receiving of messages with other activities. Spin uses a partial order reduction technique [15] to reduce the model checking effort. This technique identifies transitions as independent and takes only one of the many orders in which these transitions might be explored. The independence criterion holds for transitions that (1) access only local variables, (2) access only communication channels to which the executing process has exclusive read or write access. In our case, we could not use the partial order reduction because we had synchronous communication in the escape guard of an unless command. If we had been able to use partial order reduction, then we would not have had a great benefit for the following reasons. In our models, most variables have to be used in the verification and are global, and all communication channels for which exclusive read or write access can be guaranteed, are declared as arrays of channels which prohibits the use of the exclusive access declaration construct. The latter is a syntactical restriction for which some escape routes are available, such as the creation of a process where a channel from an array is bound to an ordinary channel, on which the exclusive read or write access can be declared. The other restriction is at the core of the reduction method, and cannot be lifted.

Another important memory usage-increasing factor for our models is probably that, whereas using `atomic` sequences does reduce the number of states, still the number steps performed in one such `atomic` sequence is reflected in the 'search depth' of the tool. This search depth is limited by the user, and determines the portion of the state space to be explored, the size of the heap that is to be allocated for the search, and hence the amount of memory used for the verification.

What one would like to have (and what might help to improve the performance of Spin tremendously) is to be able to define functions that perform computations without adding to the state space size, and `atomic` sequences to be truly atomic. One would then lose the possibility of exactly tracing down a statement where error situations occur or simulating per statement, but we feel that when using `atomic` sequences, it is fair to not have those possibilities anymore. Since the focus of Spin is on synchronisation and not on computation, there is no plan to improve Spin in this respect [14].

Multi-way synchronisation It is difficult to model multi-way synchronisation in Promela and keep the state space small. Channels are by definition one-to-one, and several processes glancing a global variable or a channel cannot be forced to do this in one atomic action. There is no plan to improve Spin in this respect [14].

Data structures Spin forbids the initialisation of processes with a parameter which is a non-basic data structure, such as an array or record. This hampers the construction of generic models. Recently, the sending of messages with an array as parameter became possible.

Never claims and traces A mixture of 'never claim' and 'trace' processes will probably affect the performance of Spin very badly. Nevertheless, the possibility to use assertions (that reference global state variables) in the 'trace' process seems like a desirable and useful feature for Spin. This is also a planned improvement for Spin [14].

6.2 Concerning Cæsar/Aldébaran and Lotos

Using Lotos, Cæsar, Aldébaran and Xtl Lotos is a hard language at first, and a precise language at second sight. It can be hard to grasp the meaning of the language constructs at first, but they have a clear semantics and do not become more complicated when combined. Modelling data is not very hard as long as the data is of a constructive and simple nature. Constructions like sets are not easy to model, but lists are.

The graphical interface of the tool set is easy to use. The simulator has the same semantics as the verifiers, which makes simulation a good means for validation of models. After a while, we turned to the command-line use of the tools rather than the graphical interface.

Expressing properties in an action based logic like ACTL turned out to be quite hard. This is partly due to the nature of the protocol, with bus reset periods disrupting normal behaviour. However, the greatest difficulty is caused by the fact that we cannot use state information in the formulas since Xtl does not yet work on the fly. Using ACTL, we were not able to find some violations of safety properties which we found with assertions in the Promela models.

Performance of Cæsar For this protocol, the performance of the Cæsar generator is poor. It does not produce the minimal graph under strong bisimulation equivalence, but generates far more states. Judging from the Lotos code and Table 2, we think this is caused by the use of the abstract data types. Apparently, terms which are equal on the basis of the data models are not recognised as such during state space generation.

If it were not for the Aldébaran possibility to compose several communicating components, we would not have been able to construct a complete a state space even for 2 DCM Managers. Actually, for the Lotos model with synchronous communication between DCM Managers, and 2 DCM Managers in the network, Cæsar generated about 1.3M states and 2.5M transitions in one hour, and then got stuck due to lack of memory. It is hard to say whether the error traces present in this model, would have been found with a far more restricted model of the protocol.

In order to use the Aldébaran facility of combining state spaces, we had to enumerate some datatypes, which affected the genericity of the Lotos model. We also had to restrict the possibilities for communication, which proved essential when generating the state space for the asynchronous case with 3 DCM Managers.

State variables It is awkward not to be able to check the values of variables in the Lotos model. This could be done by adding extra self-loops per process with this information, but due to the poor performance of Cæsar this approach was not feasible. Currently, work is going on to make Xtl work on-the-fly [21].

6.3 Comparison of the tools

Models, state spaces The models in Promela or Lotos are hard to compare. Some tasks can be performed in one `atomic` sequence in Promela (but do increase the size of the verification itself) which take several atomic actions in Lotos. In Lotos, the data types and process parameters allow for computations being made without state space enlargement. In Promela, most computations must be translated into (parts of) `atomic` sequences. In Promela one would like a little more support for data types and functions. The Lotos models with asynchronous communication and 3 DCM Managers are about as general as they can be. With the current tool support, state space generation becomes impossible with any generalisation of the behaviour.

LTL versus CTL We have found an error in the protocol with an ACTL property which we cannot express in LTL, and which we could only find with Spin by changing the models. The LTL versus CTL issue is the inspiration of many papers and discussions of which we only cite [25, 5, 27, 19]. Some attempts have been made at unification of the two approaches (See for instance [19]). However, the property that we expressed in ACTL turns out to be a classic example of the difference in expressivity between the two paradigms.

State space sizes The state spaces are smaller for the Lotos models than for the Promela models, when the models are fully explored. On the one hand, this definitely is a flattered view, since generating the state space for a complete Lotos model as such gives tremendously high numbers. On the other hand, the Spin sizes hide the actual number of statements that must be executed to reach a certain state. Because of the `atomic` predicate, the number of statements may be much higher. This does not affect the state space size, but it does affect the amount of memory used for the verification.

When errors are found, Spin stops immediately, hence explores only part of the state space. In Xtl, the library `actl` always explores the full model. The library `walk_actl` stops immediately when a diagnostic trace is constructed.

Memory usage when model checking It turns out that we needed much less memory for the Xtl verifications than with the Spin tool, which is probably due to the state space sizes being larger for Promela, and `atomic` sequences consisting of more steps causing more memory to be used than one statement. When verifying a property with the `walk_actl` library, much more memory is used then with the `actl` library, which we think is due to backtracking and overhead for the diagnostic trace.

Size of generated code The size of C code generated by Spin is manageable considering the state space size. For state space generation from Lotos models, the C files become larger. Finally, large state spaces cause Xtl to generate very large C files in which very many variables are allocated (a stack size greater than 2 Gbyte).

Expressing the properties to be checked The properties verified with Spin and with Xtl are not comparable. In Spin we used assertions (and tried in vain to use LTL) in terms of state variable values. In Xtl we used ACTL properties in terms of observable actions.

We would like to use state information from the Lotos process parameters in the properties to be verified with Xtl.

In Spin one would like to reference the values of state variables in a `trace` process, where the occurrences of communications can be checked. The combination of these features, which is as yet forbidden, would be very useful. This is a planned improvement.

Comparing model checking times When errors can be found, Spin is overall faster than Xtl except when the models become very large. For full state space exploration, Xtl is faster, probably due to the state space sizes. It should be noted that Spin builds the state space anew during exploration whereas Xtl checks properties on state spaces that have already been built, so in this case one should add the state space generation times to the model checking time. Both approaches have advantages and drawbacks in terms of efficiency.

Tailoring models for model checking We had slightly different Promela models depending on whether we were checking properties concerning final leadership, properties concerning any type of leadership, or the property which we could not express in LTL. In the former two cases, differences were only in the variables used for observation. In the latter, a fundamental change was made to the environment behaviour by having maximally two bus reset periods instead of arbitrarily many. If we had used one general model for all properties, we would have had a much larger state space. This was not necessary with the Lotos model because the experiments there were based on observing actions rather than state variable values and it was possible to express all properties in ACTL. The addition of the events that signal the election of a leader in the Lotos models do not seem to enlarge the state space size as much as the state variables in the Promela models do.

Efficiency of model checking When verifying an ACTL property with the `actl.xtl` library, Xtl visits all reachable states, thus verification does not stop as soon as the property is found to be false, and it cannot become true anymore, or vice versa. When using the `walk_actl.xtl` library, Xtl will stop as soon as a diagnostic trace has been constructed. This will be a trace showing truth in the case of an 'exists' property, and it will be a trace showing falsity in the case of a 'for all' property.

Spin uses partial order reduction [15] to improve efficiency. We already mentioned that a small change in the Promela syntax accepted by Spin can increase the benefit of this reduction technique. In our case the partial order reduction cannot be exploited because of the combination of rendez-vous communication and unless constructs. This may be a consideration when constructing models.

Spin stops the verification as soon as an error is found. A diagnostic trace leading to the error situation is presented to the user. The trace may reveal the falsity of the property to be checked, but also a dynamic error because an array index is out of range, et cetera.

6.4 Concerning this experiment

It appears that the combined approach of having different models of the same protocol and different verification techniques, gives better results, for several reasons:

1. The restrictions of the different modelling languages force one to think carefully about how to model all the aspects of the protocol.
2. The different verification techniques enable establishing different kinds of properties for the protocol.
3. One approach acts as a debugger for the other, in the sense that
    - Mistakes at the syntactic or semantic level are generally not made in the exact same manner during the different modelling efforts.
    - Results can be checked in two different situations.
    - Negative results obtained on one side and not on the other can still be 'checked' by simulating with the counterexample, and validating whether the error behaviour is also present in the model for which this could not be verified.

Thus, the results are more convincing than when only one modelling/verification approach is applied.

References

[1] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[2] D. Dams. Personal communication, October 1998.

[3] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25:761–778, 1993.

[4] R. de Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Proceedings of Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.

[5] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, June 1987.

[6] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, August 1996. Information and tool set available from URL `http://www.inrialpes.fr/vasy/pub/cadp.html`.

[7] International Organisation for Standardization. Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. ISO/IEC 8807, 1989.

[8] 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 '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

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

[10] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[11] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

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

[13] G.J. Holzmann. Personal communication, November 1998.

[14] G.J. Holzmann. Personal communication, June 1999.

[15] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of Formal Description Techniques (FORTE)*, pages 197–211, Bern, Switzerland, October 1994. Chapman & Hall.

[16] IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, August 1996.

[17] IEEE Computer Society. Draft Standard for a High Performance Serial Bus (Supplement). P1394a Draft 2.0, March 1998.

[18] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[19] O. Kupferman and M.Y. Vardi. Relating linear and branching model checking. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, New York, June 1998. Chapman & Hall.

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[21] R. Mateescu. Personal communication, December 1998.

[22] R. Mateescu and H. Garavel. XTL: A meta-language and tool for temporal logic model-checking. In T. Margaria and B. Steffen, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98*, number NS-98-4 in BRICS Notes Series, July 1998.

[23] C. Pecheur. Advanced modelling and verification techniques applied to a cluster file system. Technical Report 3416, INRIA Rhône-Alpes, May 1998.

[24] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 46–57. IEEE, 1977.

[25] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proceedings of 12th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, pages 15–32. Springer-Verlag, 1985.

[26] J.-P. Queille and J. Sifakis. Fairness and related properties in transition systems – a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[27] C. Stirling. Modal and temporal logics. In S. Abramsky, Dov M. Gabbay, and T. S.Ẽ. Maibaum, editors, *Handbook of Logic in Computer Science. Volume 2. Background: Computational Structures*, pages 477–563. Oxford University Press, 1992.

[28] Y.S. Usenko. A comparison of spin and the μcrl toolset on havi leader election protocol. Technical Report SEN-R99??, CWI, Amsterdam, 1999. Submitted.

A Excerpts from the HAVi Specification

The following parts of text are quoted from [9]. Most of the quotes are followed by our interpretation.

General

Page 15, Section 2.4.4 (Software Elements)

> The following table summarises which architectural elements are present for the various device categories, which are absent and which are optional.

Device Type/Element	FAV	IAV	BAV	LAV
DCM Manager	$\surd$	$[\surd]$		
Event Manager	$\surd$	$\surd$		
Messaging System	$\surd$	$\surd$		
1394 Communication Media Manager	$\surd$	$\surd$		

Interpretation So on each FAV device a DCM Manager must be present, while on an IAV device it is optional. The Event Manager, Messaging System and 1394 Communication Media Manager must be present on each FAV and IAV device. Neither BAV nor LAV devices have any of these software elements.

Page 82, Section 5.1.1 (HAVi API Descriptions)

> Communication Type:
>
> - messaging (M) - communication is via the Messaging System. This form of communication is initiated by the client.

DCM Manager

Page 155, Section 5.8.3 (DCM Manager API)

> **DCMManager::DMInitialization**
> The initial leader shall accept this message at any time, replying with the selected output parameters. An initial follower shall ignore any message until it has received the reply. (Note that a network reset event shall reinitiate the leader election phase on all DCM Managers)

Interpretation So the initial leader remembers its role even after leader election has ended. The word 'ignore' probably means 'accepts but throws away'.

Page 160, Section 5.8.5 (DCM Management Protocol)

> The DCM management system is constructed from a distributed group of DCM Managers on FAV and IAV devices. DCM Managers interact on a peer-to-peer basis to implement the DCM management service, while in turn using services of local system elements. These are the CMM, Messaging System, Event Manager, and DCM code units. Each DCM Manager can read SDD data directly from devices connected to the HAVi 1394 network. The DCM management protocol supports the use of device storage and Internet access facilities.
>
> In a nutshell, each DCM Manager starts with a leader election after a network reset event is received. One DCM Manager will be selected as the leader. All DCM Managers

are followers, and subordinate to the leader (i.e., the leader DCM Manager also plays the role of a follower in this protocol description). Leader and followers subsequently collaborate to install DCM code units autonomously for each guest found on the network, if none is installed for it already. In addition to automatic DCM management, each DCM Manager may also accept method invocations. The leader will control most of the protocol activities.

DCM Managers can support URL access facilities, and may announce this during the leader election. FAV devices capable of doing this shall do so. IAV devices may, but need not announce such a capability. Any device announcing such a capability shall be selected as the leader, where a FAV device is favoured over an IAV device.

The following abstractions are made in the description of the protocol:

- At any moment a network reset event can occur. Each DCM Manager will (re)start the leader election when this event is received as soon as possible. A pending message send or receive action shall be aborted due to network reset event.
- Protocol messages shall be implemented by MsgSendRequestSync and MsgSendResponse. The specified timeout values are not critical. Except for DMGetDCM, they shall be set to 3 seconds. The sending DM shall retry sending the request message each time a timeout condition occurs (or until a network reset event is received). Only for DMGetDCM, the sender may decide to stop resending the message after a limited number of timeouts.
- Potential deadlock conditions shall be prevented, e.g., by applying multi-threading in DCM Manager implementations. Unexpected messages shall be ignored by DCM Managers.

Interpretation The DCM Manager uses reliable message sending with the synchronous waiting-for-a-response method for communication to other DCM Managers.

We do not know what callback function will be used by the DCM Manager for receiving messages, but we assume that it is either a synchronous function (with an interrupt-like method) or an asynchronous function (that puts the message in some buffer).

Multi-threading is probably advised for the situations in which the DCM Manager is waiting for a response message, so that it can still receive other messages (like the bus reset event).

Page 161, Section 5.8.5.1 (Leader Election)

After a host device is powered up, or after a network reset event is received, each DCM Manager will enter this activity. (A reset or power up/down of a device shall cause network reset events on all other host devices.)

All DCM Managers shall behave as follows:

- The GUID list is retrieved through CMM1394::GetGUIDList. The relevant HAVi SDD data of all devices are retrieved: HAVi_Device_Type, HAVi_DCM_Manager, Model_ID, Model_Vendor_Id, Node_Unique_ID (GUID). Devices without such SDD data are classified as LAV devices.
- Each FAV or IAV device without a DCM Manager (derived from HAVi_DCM_Manager) shall be ignored, and shall not be a host for any guest on the HAVi 1394 network.
- From all host GUIDs, the highest bit order-reversed GUID is calculated, and the DCM Manager on the device with this GUID is declared the initial leader. The reversal prevents devices from certain vendors acting as the initial leader in many network configurations (since the GUID starts with a vendor identifier.) Note that all devices read the same GUID list, and will declare the same DCM Manager as initial leader.

At this time, each DCM Manager knows if it is the initial leader or an initial follower. Each device knows which other DCM Managers there are, and which SEIDs they have. (The SEID is the concatenation of the device GUID and the fixed DCM Manager software element handle.) Message passing between DCM Managers is enabled. Each DCM Manager shall be registered.

The initial leader shall behave as follows to select and announce the final leader:

- From all identified initial followers, it awaits a DMInitialization request with zero or more declared URL capabilities and URL access capable guest GUIDs from the initial followers.
- The selection of the final leader is as follows:
  - If there are FAV devices with a declared URL access capability, one of them is selected.
  - Otherwise, if there are IAV devices with a declared URL access capability, one of them is selected.
  - Otherwise, if there are FAV devices, one of them is selected.
  - Otherwise, if there are IAV devices, one of them is selected.
- The DMInitialization reply is sent to all initial followers. The final leader is the last one to which this message shall be sent. The reply carries the GUID of the final leader and the GUID of an arbitrarily selected URL capable guest, if any.

An initial follower shall send a DMInitialization request to the initial leader and awaits the reply. Upon a timeout, the request is resent.

Each DCM Manager now knows which one is the final leader, and which others are final followers. The autonomous operation starts.

Interpretation The only URL information that is of use to the leader election protocol is the URL access capability of device itself on which the DCM Manager resides.

The timeout, which causes the initialisation request to be resent, could occur in a number of situations. We assume that besides the obvious error situations, the following scenarios are reasonable:

- The message was sent but is delayed on its way to the destination and the timeout occurs before the acknowledgement of delivery is received by the sending DCM Manager.
- The message was sent, and acknowledgement has been received, but the response (whether sent or not) has not been received yet.

So a timeout occurring does not mean that the destination DCM Manager did not receive the message.

Messaging System

Page 26, Section 3.2.1.2.2 (Service Description)

Message Transfer Modes:

Simple mode is very basic: no control is performed by the Messaging System. The message is sent on the network and that is all.

Reliable mode is more complicated and expects the destination device to acknowledge the message.

Note that at the originating side, the calling software element is blocked until it gets the acknowledgement.

To avoid blocking a software element indefinitely an acknowledgement timeout is used. Its value shall be 30 seconds.

Interpretation We do not know what callback function the DCM Manager will give to the Messaging System, but we assume that it is either a synchronous function (with an interrupt-like method) or an asynchronous function (that puts the message in some buffer).

We assume that a multi-threading implementation is meant to have the DCM Manager receive other messages, not while it is busy sending a message itself, but while it is waiting for a response.

Page 35, 3.2.3.5 (Messaging System Description)

> The message passing API will provide a synchronous service allowing a caller to block until a response is received. As shown in the following figure, the caller asks to send a function call through the message passing API. The local Messaging System sends a request message according to the request mode (reliable mode in this example) and waits for the response or a timeout condition. The remote Messaging System receives the request and passes it to the destination software element. The destination element sends its function response message using a normal send in simple form. The requester's Messaging System receives the response and transmits it to the requester.

Interpretation Here the proposed multi-threading implementation can help the DCM Manager to receive messages while it is still waiting for a response.

Note that it is not said anywhere how long a message can be 'en route' from the source to the destination, let alone that the destination DCM Manager will process the message, send a response and have this delivered in the 3 seconds proposed. So we assume that Initialisation requests, once sent, could still be on their way when the sending DCM Manager experiences a timeout and will resend the request.

Page 94, Section 5.3.4 (Messaging System API)

> **MsgCallback**
>
> - sourceId: the 80-bit software element identifier of the software element that issued the message
> - state: the status of the message.
>   - SUCCESS if everything worked fine
>   - MSG::EALLOC if the message passing cannot deliver the entire received message due to a lack of resources
>   - MSG::EDISAPPEAR if the supervision of the software element (described in sourceId) has been detected as disappeared. The software element sourceId is no longer reachable (device unplugged, or software element performed a MsgClose). Fields other than sourceId are undefined.
> - payload: consists of the MessageLength and MessageBody
>
> This function is the callback supplied by a software element. This call back is invoked by the Messaging System each time an incoming message (incoming reliable request or simple messages) is received for that software element. It may also be invoked to notify the software element about a disappearance of a target software element (this service is provided only after a MsgWatchOn request).
>
> After the callback returns, and depending on the return code (SUCCESS or MSG::EFAIL), the Messaging System acknowledges the message: if the callback returns with SUCCESS, the Messaging System generates an ACK message. If the callback returns with EFAIL, the Messaging System generates an NOACK message

Warning: a callback function is not allowed to call blocking functions. A callback always executes in the context of the Messaging System, and is not allowed to block the Messaging System. Applications should treat the callback as an interrupt.

Page 95, 5.3.4 (Messaging System API)

MsgOpen

- callback: the call back function that the messaging system calls when it receives a message for that software element
- seid: the 80 bits software element identifier that has been assigned to the software element.

This function is called by a software element that requires the services of the Messaging System. This function provides a unique software element identifier to the software element which is to be used by the software element to register and to communicate with other software elements. This function also allows the calling software element to provide a call back function that will be used by the Messaging System when an incoming message (either a reliable request, or a simple message) has to be passed to the software element.

Interpretation It seems that for events and messages the same callback function is used. Nevertheless, we assume that a DCM Manager can make a difference between messages and (urgent) bus reset events, either in the callback function itself or by some combination of the callback function and the opCode which the DCM Manager gives to the Event Manager when it registers its interest in the bus reset events.

Page 101, Section 5.3.3 (Messaging System API)

MsgSendResponse
This function is used to send a "function response" message (see 3.2.3.2) to one software element which has previously performed a MsgSendRequest call. According to the chosen transfer mode, the function returns immediately or once the sending is completed (message acknowledge received).
MsgSendRequestSync
This function is used by a software element when it wants to send a "function call" message (see 3.2.3.2) to one destination software element and block until the response is received (synchronous mode, see section 3.2.3.5). All requests are sent in "reliable mode". A timeout value of zero defaults to the system timeout value. The timeout condition overrides the messaging system timeout (ackTimeout) condition.

Interpretation As we already saw, for the DCM Managers communicating with each other, the timeout is 3 seconds. It is not clear how this value overrides the ackTimeout. It could be that the whole process of delivering the "function call", the destination processing this call and finally returning the response, should not take longer than 3 seconds. Or the timer that signals this timeout could be restarted after the acknowledgement of delivery of the "function call" was received.

In our approach, this timing information is not modelled.

Communication Media Manager for 1394

Page 23, Section 3.1 (CMM description)

> 1394 bus is a dynamically configurable network. After each bus reset, a device may have a completely different physical ID than it had before. If a HAVi component or an application has been communicating with a device in the network, it may want to continue the communication after a bus reset, though the device may have a different physical ID. To identify a device uniquely regardless of frequent bus resets, The Global Unique ID (GUID) is used by CMM and other HAVi entities. GUID is a 64 bit number that is composed of 24 bits of node-vendor ID and 40 bits of chip ID. While a device's physical ID may change constantly, its GUID is permanent. CMM makes device GUID information available for its clients.

Interpretation In order to know what devices there are in the network, the CMM must use the topology map of 1394. This gives only physical IDs subject to change. These physical IDs must then be used to get the GUID for every device in the network, by asynchronous 1394 read operation (transaction layer)?? Then CMM can use new GUID list to compare with old one, detect devices who have left or connected and post events accordingly.

Page 23, Section 3.1 (CMM description)

> One of the advanced features the 1394 bus provides to the HAVi system is its support for dynamic device actions such as hot plugging and unplugging. To fully support this up to the user level, HAVi system components or applications need to be aware of these network changes. CMM works with the Event Manager to detect and announce such dynamic changes in network configuration. Since any topology change within the 1394 bus will cause a bus reset to occur, the CMM can detect topology changes and post an event to the Event Manager about these changes along with associated information. The Event Manager will then distribute a related event (called a network reset) to all interested HAVi entities or applications.

Interpretation DCM Managers are interested in this event.

Page 86, Section 5.2.1 (CMM1394 Services)

> **CMM1394::GetGUIDList**: Communication Type = M

Interpretation So in order to use this service, the DCM Manager must send a message through the Messaging System to the CMM.

Page 86, Section 5.2.2 (CMM1394 API)

> **CMM1394::GetGUIDList**
>
> Get GUID lists of both active and non-active devices on the network. The first item returned in activeGuidList shall be the GUID of the local device. A device is defined as active if it can process HAVi messages (IAV or FAV) or respond to commands from HAVi software elements (BAV or LAV). For FAV, IAV, or BAV devices, an SDD entry (HAVi_Device_Status) in the HAVi_unit_directory can be read to determine the status of the device (see section 9.7.7). A value of one indicates that the device is active. A value of zero indicates that the device is not active. An LAV device is considered active whenever its GUID is visible on the network.
>
> Since each device on the network can be identified by its GUID, the GUID list gives all devices available in the system.
>
> Error codes
>
> - ENOTREADY: GUID list is not available yet - system may be updating it

Interpretation By the explanation of Communication Types at Page 82 (Section 5.1.1), we conclude that whenever the DCM Manager wants to have the GUIDList, it sends a message to the CMM through the Messaging System.

Page 93, Section 5.2.4 (CMM1394 Events)

> **NetworkReset**
> NetworkReset is a local event. This event is generated whenever there is a change in the home network topology (e.g. a connection of a new device). The CMM is generally the component posting this event. As opposed to the NewDevices and GoneDevices events, the CMM does not gather GUID list of the changed devices. This event is intended for target software elements that are only interested in knowing when network topology has changed but are not interested in specifics of the change.

Interpretation Since DCM Managers need to know about bus resets, and then find out new or disappeared devices later with GetGUIDList, they only need to subscribe to the event NetworkReset. We assume that they only listen to the event NetworkReset.

Page 353, 10.3 (Scenarios)

> **A new BAV or LAV is plugged into the network**
> In this scenario, a new BAV or LAV device is plugged into the 1394 bus.
>
> - The CMM of each FAV or IAV generates locally a NetworkReset event (and also the NewDevices event)
> - DCM Managers have previously registered interest in NetworkReset event and so receive the event. Using the CMM1394::GetGUIDList method, they get the GUIDs of the new and gone devices on the network.

Interpretation Indeed this scenario indicates that DCM Managers only listen to event NetworkReset.

Event Manager

Page 36, Section 3.3 (Event Manager Description)

> If a Software Element wishes to be notified when a particular event is posted, it must register such intention with its local Event Manager.
>
> When a software element posts an event, it does so via a service provided by Event Manager. The Event Manager checks its internal table and notifies those software elements who have registered this event.
>
> An Event Manager notifies software elements by using the HAVi Messaging System; in particular, it sends a notification message to the software element that is to be notified.

Interpretation The messaging system is used for events. Probably this is done with the simple mode without waiting for a response, since event messages are only sent to Software Elements on the same device.

Page 105, Section 5.4.3 (Event Manager API)

> **EventManager::Register**
> EventManager::Register adds the software element (that has sent the message) to the Event Manager's internal table. A new entry is created to register the software element and the list of events it wishes to "listen to". There is no limit on the internal table size so long as Event Manager can find enough system resources to maintain the table. When any of the events in the software element's "interested" event list occurs, the Event Manager sends a notification message to the software element. The message contains opCode, the EventID representing the event, and possibly additional information about the event. When the software element receives the event notification message, it uses the opCode to determine how to process the message. It is therefore the responsibility of the software element to define the operation code for its event-notification message processing procedure call, and to pass it to the Event Manager at event registration time.

Interpretation We do not know what opCode the DCM Managers will use, but assume that it enables them to distinguish between ordinary messages and urgent events, even before the callback function should put messages in a buffer.

Page 108, Section 5.4.3 (Event Manager API)

> **EventManager::PostEvent**
> Post the specified event to the home network. Posting an event means notifying all "target" software elements (i.e., those including the event in their "listen-to" even list) regardless of their location on the network. Event notification messages are sent by the Event Manager to all target software elements. The event poster simply sends a message to an Event Manager indicating its intention to post the specified event. It is the responsibility of the Event Managers to ensure that all target software elements receive notification.

Page 110, Section 5.4.5 (Event Manager Protocol)

> When an event is posted globally to all Event Managers in the home network, the following mechanism is used:
>
> The event poster sends a EventManager::PostEvent message to its local Event Manager requesting the event to be posted on its behalf. The message contains the SEID of the event poster, the EventID of the event to be posted, whether the event is to be delivered locally or globally, and possibly additional information about the event.
>
> The local Event Manager checks if any software element residing locally have the posted event in its "listen-to" event list. All target software elements that meet this condition will get an event notification message from this Event Manager. The message contains the operation code selected by the target software, the SEID of event poster, the EventID of the posted event and possibly additional information about the event. The target software element, upon receiving the notification message, will presumably respond to the event.

Interpretation All of the messages are sent through the Messaging System.

B Excerpts from the IEEE 1394 Standard

The following parts of text are quoted from [16] and [17]. Most of the quotes are followed by our interpretation.

1394-1995, Page 200, Section 8.2.1

Serial Bus control request (SB_CONTROL.request)
This service shall provide the following actions:

- Present Status. The Serial Bus management layer shall return status to the application via the Serial Bus control confirmation service.

1394-1995, Page 201, Section 8.2.2

Serial Bus control confirmation (SB_CONTROL.confirmation)
This service shall communicate the following parameters after a control request of Present Status:

- Bus Manager ID. The 6-bit physical ID of the bus manager. If no bus manager is active, this parameter shall have a value of $3F_16$. This parameter is available only if the node is bus-manager or isochronous-resource-manager capable.

Interpretation If an application (the HAVi 1394 Communication Media Manager for instance) wants to know the network configuration, it is likely that the SB_CONTROL.request service is used to ask for 'Present Status', and a SB_CONTROL.confirmation with the 'Bus Manager ID' is received. The Bus Manager is supposed to have a TOPOLOGY_MAP register which can be read through an asynchronous (transaction layer) READ request.

1394-1995, Page 201, Section 8.2.3

Serial Bus event indication
BUS EVENT with values

- BUS RESET START. A bus reset has started
- BUS RESET COMPLETE. A bus reset process has completed. In the cable environment, this is indicated by the first subaction gap after the bus reset has started.

Interpretation It is not stated explicitly, but we expect from this text that these events are issued by the serial bus management layer each time a bus reset process starts or is completed, and that these events are caught by the HAVi 1394 Communication Media Manager.

1394-1995, Page 204, Section 8.3.1.6

A node that is capable of becoming the bus manager shall

b) Implement the SPEED_MAP and TOPOLOGY_MAP registers

1394-1995, Page 229, Section 8.4.2

Bus configuration procedures
When a bus reset occurs, all asynchronous and isochronous traffic on the Serial Bus ceases. Asynchronous may resume as soon as the self-identify process that follows a bus reset has completed. Previously established isochronous data streams, either talker or listener, are to resume as soon as possible after the self-identify process completes. In general, the roles of cycle master, isochronous resource manager, and bus manager shall be redetermined before new allocation of isochronous resources can be performed and before the new topology and speed maps can be made available.

Interpretation So messages through the HAVi Messaging System cannot be sent from one device to another while a bus reset is taking place. Sending may proceed as soon as the self-identify process is completed. The new GUIDList which must come from the topology map can actually be obtained only after the new cycle master, isochronous resource manager and bus manager have been elected.

1394-1995, Page 319, Annex H.1

> **Bus configuration timeline**
> (see Figure H-1) At point B, the following are true:
>
> - bus manager has made the TOPOLOGY_MAP registers available

Interpretation So it takes at most 625 μs after completion of the self-identify process to enable the reading of GUIDList.

1394a March 15, 1998, Page 18, Section 3.2.1

> **Arbitrated (short) bus reset**
> The last phase, self-identify, requires approximately one microsecond per node or about 70 μs worst case when there are 63 nodes. Tree identify is also quite rapid and takes less than 10 μs. The longest phase is bus reset and it lasts about 167 μs while the BUS_RESET signal is propagated.
>
> The reason for the long duration of BUS_RESET is that a transmitting node is unable to detect this arbitration line state. It is only after packet transmission is complete that the node will observe the reset. Hence BUS_RESET must be asserted longer than the longest possible packet transmission. This guarantees the success of bus reset regardless of the bus activity in progress.

Interpretation So it takes less than 167 μs after the start of the bus reset period for all devices to notice that a bus reset has started. After sending the bus reset signal for 167 μs, all nodes propagate the idle signal for another 167 μs. Based on these numbers, we get a best-case bus reset phase duration of 414 μs and a worst-case bus reset phase duration of < 581 μs.

C The input files for Spin

C.1 Promela model for 3 DCM Managers with asynchronous communication (final leader)

The Promela model of the leader election protocol with 3 DCM Managers, asynchronous communication between the DCM Managers, and tailored to verify final leader properties.

For an explanation of the following code, we refer to Section 4.2.

```
/* +-----------------------------------------------------------------------+
   |                          PROMELA SPECIFICATION                        |
   |                                                                       |
   | file          : leader_election.spin                                  |
   | author        : Judi Romijn (judi@cwi.nl)                             |
   | creation date : May 6, 1998                                           |
   | last modified : May 31, 1999                                          |
   +-----------------------------------------------------------------------+ */

/* ----CONSTANTS, SHORTHANDS------------------------------------------------ */
```

```
#define maxHost 3       /* maximum nr of host nodes (each has unique ID) */
#define maxMessages 1

/* ----PROPERTY DEFS--------------------------------------------------------- */


/* ----TYPE DEFS------------------------------------------------------------- */

mtype = {power_change, bus_reset, DMInitRequest, DMInitReply }

typedef FieldHost {     /* information about a host node */
        bool rec_req;   /* did I receive a request already? */
};
typedef Field {         /* general information about a host node */
        bool up;        /* is it powered up? */
};
typedef Field2 {        /* for all CMMs */
        bool delivery;  /* whether a bus reset event should be delivered */
};
typedef Array {
        Field network[maxHost];
        bool fleader;   /* final leader? */
        byte Leader;    /* Leader id */
        bool UrlCapable;        /* Am I in UrlCapable mode? (constant!!!!) */
}

/* ----CHANNELS-------------------------------------------------------------- */

chan chanUpDown[maxHost] = [0] of { mtype }             /* synchronous! */
chan chanCMM[maxHost] = [0] of { mtype }                /* synchronous! */
chan chanDM[maxHost] = [maxMessages] of { mtype, byte }

/* ----GLOBAL VARIABLES------------------------------------------------------ */

/* semaphor variables */
show bool BusResetPeriod=0;     /* are we in a bus_reset period? */
show Field2 BusResetDelivery[maxHost];  /* bus_reset events to be delivered? */

show Field Global[maxHost];     /* 'up' info per host node */
show Array Local[maxHost];      /* as Global, but one whole array per host */

/* scratch variables that are referenced within atomic sequences only */
  hidden byte k;
  mtype m;  /* ERRORS if hidden! */

/* ----PROCESS DEFINITIONS--------------------------------------------------- */


/* ----PROCESS Bus_Reset----------------------------------------------------- */
proctype Bus_Reset()
{
  byte j=0;                     /* for running through array */

  do
  :: d_step
```

```
      { BusResetPeriod = 1;      /* start of a bus_reset period */
        j=0;
        do
        :: (j<maxHost) ->
               if
               :: (Global[j].up) -> BusResetDelivery[j].delivery=1;
                                 /* start one more bus_reset delivery cycle */
               :: (!Global[j].up) -> skip;
               fi;
               j++;
        :: (j>=maxHost) -> break;
        od;
        j=0;                     /* change network topology */
      };
      do         /* decide new power status per host node */
      :: atomic{(j<maxHost) ->
            if
            :: Global[j].up = !Global[j].up;             /* come up/go down */
               chanUpDown[j]!power_change;
            :: skip;                                     /* stay up/down */
            fi;
            j++;
         }

      :: atomic{(j>=maxHost) ->
            BusResetPeriod = 0;  /* now GUIDlist is stable again */
            j=0;
            break;
         }
      od;
    od
  }


  /* ----PROCESS CMM---------------------------------------------------------- */
  proctype CMM(byte Id)
  {
    downCMM:
      { (0);
      } unless {atomic{chanCMM[Id]?power_change;
                   goto upCMM;};
               };
    upCMM:
      { do
        :: atomic
           { (BusResetDelivery[Id].delivery) ->
                 BusResetDelivery[Id].delivery = 0;
                 chanCMM[Id]!bus_reset;
           }
        od;
      } unless {atomic{chanCMM[Id]?power_change;
                   BusResetDelivery[Id].delivery = 0;
                   goto downCMM;};
               };

```

```
}


/* ----PROCESS DCM_Manager--------------------------------------------------- */
proctype DCM_Manager(byte Id)
{ byte j = 0;

  FieldHost InfoHost [maxHost];         /* all info per host node */

  /* the following is invalid because of array reference */
  /* xr chanDM[Id];                      to support partial order reduction */

  down_DM:
  { (0);                                /* unexecutable, forces to wait */
  } unless { atomic{ chanUpDown[Id]?power_change;
                     Local[Id].network[Id].up = Global[Id].up;
                     chanCMM[Id]!power_change;
                     goto leader_election_DM };
  };

  leader_election_DM: /* first part of leader election */
  { d_step              /* GetGUIDList + AsyncRead */
    { if
      :: (!BusResetPeriod) ->           /* as soon as allowed */
          j=0;          /* copy GUIDlist to local var */
          do
          ::(j<maxHost) -> Local[Id].network[j].up = Global[j].up;
                           j++;
          ::(j>=maxHost) -> break;
          od;
          j=0;
      fi;
    };
    d_step              /* compute initial leader */
    { j=0;
      do
      ::(j<maxHost && Local[Id].network[j].up==1) ->            /*found one! */
           break;
      ::(j<maxHost && Local[Id].network[j].up==0) -> j++;       /* not yet */
      ::(j>=maxHost) -> break;                          /* no candidate left */
      od;
      Local[Id].Leader = j;     /* maxHost if no init Leader candidate */
      j=0;
    };
    atomic              /* am I leader or not? */
    { if
      :: (Local[Id].Leader == Id) ->
                           goto init_leader_DM;
      :: (Local[Id].Leader != Id) -> goto init_follower_DM;
      fi;
    };

  init_follower_DM:
    { atomic
       { (!BusResetPeriod && Global[Local[Id].Leader].up) ->
```

```
          chanDM[Local[Id].Leader]!DMInitRequest(Id); /* send until */
     };
    goto init_follower_DM;
  } unless {atomic
     { chanDM[Id]?m(k) ->
         if
         :: (m==DMInitReply) ->             /* declaration recvd */
             Local[Id].Leader = k;
         :: (m!=DMInitReply) ->     /* unexpected message: ignore */
            goto init_follower_DM;
         fi;
     };
    };
  atomic{ if
          :: (Local[Id].Leader==Id ) -> Local[Id].fleader = 1;
                                     goto final_leader_DM;
          :: (Local[Id].Leader!=Id) -> goto final_follower_DM;
          fi;
       };

init_leader_DM:       /* copy info from local DM and compute #(host & up) */
  atomic
  { j=0;
    k=0;
    do
    :: (j<maxHost) ->
           InfoHost[j].rec_req = 0;
           k = (Local[Id].network[j].up==1 -> k+1 : k);
           j++;
    :: (j>=maxHost) -> break;
    od;
    j = k;
  };
  do          /* then wait for mes DMInitRequest from all up DMs */
  :: atomic
     { (j>1) ->
         chanDM[Id]?m(k) ->
           if
           :: (m==DMInitRequest) ->           /* expected message */
               if
               :: (Local[Id].network[k].up && !InfoHost[k].rec_req) ->
                     InfoHost[k].rec_req = 1;
                     j--;
               :: (!Local[Id].network[k].up || InfoHost[k].rec_req) -> skip;
               fi;
           :: (m != DMInitRequest) -> skip /* ignore unexp mes */
           fi;
     };
  :: atomic{ (j==1) ->                /* got mes from all DMs except myself */
                break };
  od;
  atomic               /* search for final leader: up and UrlCapable */
  { j = 0;
    do
    :: (j<maxHost && Local[Id].network[j].up && Local[j].UrlCapable) ->
```

```
          break;                                              /* found! */
      :: (j==maxHost) -> break;                       /* no candidate left */
      :: (j<maxHost && (!Local[Id].network[j].up || !Local[j].UrlCapable)) ->
          j++;                                        /* not yet */
      od;
      Local[Id].Leader = j;     /* maxHost if no candidate (no DM UrlCapable) */
      j=0;
    };
    d_step{ if
           :: (Local[Id].Leader>=maxHost) ->      /* all DM's not UrlCapable */
                 Local[Id].Leader = Id;          /* anyone can be leader: me */
           :: (Local[Id].Leader<maxHost) -> skip;
           fi;
           j=0 };
    do
    :: atomic
       { (j<maxHost && j!=Local[Id].Leader
          && j!=Id && Local[Id].network[j].up
          && !BusResetPeriod && Global[j].up) ->
             chanDM[j]!DMInitReply(Local[Id].Leader);
       };
          j++;
    :: d_step{ (j==Local[Id].Leader || j==Id) -> j++; }
    :: atomic{ (j==maxHost) -> j=0; break; }
    :: d_step{ (j<maxHost && !Local[Id].network[j].up) -> j++; }
    od;
    if                  /* done! */
    :: atomic{ (Local[Id].Leader != Id  /* Leader informed last */
                 && !BusResetPeriod
                 && Global[Local[Id].Leader].up) ->
              chanDM[Local[Id].Leader]!DMInitReply(Local[Id].Leader);
         };
           goto final_follower_i_DM;
    :: atomic{ (Local[Id].Leader == Id) ->
              Local[Id].fleader = 1;
              goto final_leader_i_DM;
    };
    fi;

  final_follower_DM:
    do
    :: chanDM[Id]?m(k);
    od;

  final_follower_i_DM:
    do
    :: atomic{ chanDM[Id]?m(j) ->
                 if
                 :: (m==DMInitRequest) -> /* never unexpected mes */
                       atomic{
                          (!BusResetPeriod && Global[j].up) ->
                          chanDM[j]!DMInitReply(Local[Id].Leader);};
                       j=0;
                 :: (m != DMInitRequest) -> skip /* ignore unexp mes */
                 fi;
```

```
       }
    od;

  final_leader_DM:
    do
    :: chanDM[Id]?m(k);
    od;

  final_leader_i_DM:
    do
    :: atomic{ chanDM[Id]?m(j) ->
                 if
                 :: (m==DMInitRequest) -> /* never unexpected mes */
                       atomic{
                          (!BusResetPeriod && Global[j].up) ->
                          chanDM[j]!DMInitReply(Id);};
                       j=0;
                 :: (m != DMInitRequest) -> skip /* ignore unexp mes */
                 fi;
       }
    od;

  } unless
    { if
      :: atomic{ chanCMM[Id]?bus_reset;
                 Local[Id].Leader=0;    /* clear scratch+non-hidden vars */
                 Local[Id].fleader = 0;         /* new leader to be elected */
                 j=0;
                 do
                 :: (j<maxHost) -> InfoHost[j].rec_req=0; /* clear InfoHost */
                                   Local[Id].network[j].up=0; /* clear Local */
                                   j++;
                 :: (j>=maxHost) -> j=0;break;
                 od;
                 do                             /* empty the queue */
                 :: (chanDM[Id]?[m(k)]) -> chanDM[Id]?m(k);
                 :: else -> break;
                 od;
                 goto leader_election_DM };
      :: atomic{ chanUpDown[Id]?power_change;
                 Local[Id].network[Id].up = Global[Id].up;
                 Local[Id].fleader = 0;         /* new leader to be elected */
                 Local[Id].Leader=0;    /* clear scratch+non-hidden vars */
                 j=0;
                 do
                 :: (j<maxHost) -> InfoHost[j].rec_req=0; /* clear InfoHost */
                                   Local[Id].network[j].up=0; /* clear Local */
                                   j++;
                 :: (j>=maxHost) -> j=0;break;
                 od;
                 do                             /* empty the queue */
                 :: (chanDM[Id]?[m(k)]) -> chanDM[Id]?m(k);
                 :: else -> break;
                 od;
                 chanCMM[Id]!power_change;
```

```
                 goto down_DM };
      fi;
    } /* end unless */
}


/* ----INIT PROCESS---------------------------------------------------------- */
init{
  /* byte j; */


  atomic{ run Assertion();
          run Bus_Reset();
                        /* run copy of DCM_Manager for each ID */
          if
          :: Local[0].UrlCapable = 0;   /* not in UrlCapable mode */
          :: Local[0].UrlCapable = 1;   /* in UrlCapable mode */
          fi;
          if
          :: Local[1].UrlCapable = 0;   /* not in UrlCapable mode */
          :: Local[1].UrlCapable = 1;   /* in UrlCapable mode */
          fi;
          if
          :: Local[2].UrlCapable = 0;   /* not in UrlCapable mode */
          :: Local[2].UrlCapable = 1;   /* in UrlCapable mode */
          fi;
          run CMM(0);
          run DCM_Manager(0);
          run CMM(1);
          run DCM_Manager(1);
          run CMM(2);
          run DCM_Manager(2);
        };

}

```

C.2 Promela model for 3 DCM Managers with asynchronous communication (leader)

The Promela model of the leader election protocol with 3 DCM Managers, asynchronous communication between the DCM Managers, and tailored to verify general leader properties.

The differences with the model tailored to verify final leader properties is the moment at which the `leader` of a DCM Manager variable is set to true. In the former model this was only after becoming the final leader, and hence the name of the variable was `fleader`. In this model, any initial or final leadership is reason to set the variable `leader` to true.

We now list the difference between this model (>) and the model in Appendix C.1 (<).

```
32c32
<  bool fleader; /* final leader? */
---
>  bool leader; /* leader? */
167c167
<        :: (Local[Id].Leader == Id) ->
---
```

```
>       :: (Local[Id].Leader == Id) -> Local[Id].leader = 1;
190c190
<             :: (Local[Id].Leader==Id ) -> Local[Id].fleader = 1;
---
>             :: (Local[Id].Leader==Id ) -> Local[Id].leader = 1;
242c242,244
<             :: (Local[Id].Leader<maxHost) -> skip;
---
>             :: (Local[Id].Leader<maxHost && Id==Local[Id].Leader) -> skip;
>             :: (Local[Id].Leader<maxHost && Id!=Local[Id].Leader) ->
>                   Local[Id].leader = 0;
265d266
<               Local[Id].fleader = 1;
312c313
<                  Local[Id].fleader = 0; /* new leader to be elected */
---
>                  Local[Id].leader = 0; /* new leader to be elected */
327c328
<                  Local[Id].fleader = 0; /* new leader to be elected */
---
>                  Local[Id].leader = 0; /* new leader to be elected */
```

C.3 Promela model for 3 DCM Managers with asynchronous communication (end states)

The Promela model of the leader election protocol with 3 DCM Managers, asynchronous communication between the DCM Managers, tailored to verify final leader properties and with a restricted number of bus reset periods.

The differences with the model tailored to verify final leader properties is the that there are maximally two bus reset periods. This makes the behaviour of the model finite and allows us to search for invalid end states.

We now list the difference between this model (>) and the model in Appendix C.1 (<).

```
62a63
>   byte HowManyBusResets=2; /* for verification purposes */
66c67,69
<      { BusResetPeriod = 1; /* start of a bus_reset period */
---
>      { (HowManyBusResets>0) ->/* only a limited number of bus reset periods */
>        HowManyBusResets--;
>        BusResetPeriod = 1; /* start of a bus_reset period */
96c99,101
<    od
---
>    :: (HowManyBusResets==0) -> break;
>    od;
>
104,105c109,112
<     { (0);
<     } unless {atomic{chanCMM[Id]?power_change;
---
>     { {skip;
> end_cmm1:
>       (0)}
>      unless {atomic{chanCMM[Id]?power_change;
```

```
107c114
<               };
---
>               }};
109c116,118
<     { do
---
>     { {skip;
> end_cmm2:
>       do
115,116c124,125
<       od;
<     } unless {atomic{chanCMM[Id]?power_change;
---
>       od}
>       unless {atomic{chanCMM[Id]?power_change;
119c128
<               };
---
>               }};
134,135c143,146
<   { (0); /* unexecutable, forces to wait */
<   } unless { atomic{ chanUpDown[Id]?power_change;
---
>   { skip;
> end_dcm1:
>     {(0)} /* unexecutable, forces to wait */
>     unless { atomic{ chanUpDown[Id]?power_change;
139a151
>   };
191,192c203,204
<                                           goto final_leader_DM;
<              :: (Local[Id].Leader!=Id) -> goto final_follower_DM;
---
>                                           goto end_final_leader_DM;
>              :: (Local[Id].Leader!=Id) -> goto end_final_follower_DM;
263c275
<             goto final_follower_i_DM;
---
>         goto end_final_follower_i_DM;
266c278
<                goto final_leader_i_DM;
---
>                goto end_final_leader_i_DM;
270c282
<   final_follower_DM:
---
>   end_final_follower_DM:
275c287
<   final_follower_i_DM:
---
>   end_final_follower_i_DM:
289c301
<   final_leader_DM:
---
```

```
>   end_final_leader_DM:
294c306
<   final_leader_i_DM:
---
>   end_final_leader_i_DM:
352,353c364
<   atomic{ run Assertion();
<           run Bus_Reset();
---
>   atomic{ run Bus_Reset();
```

C.4 Promela model for 3 DCM Managers with synchronous communication

The Promela model of the leader election protocol with 3 DCM Managers, synchronous communication between the DCM Managers and tailored to verify final leader properties can be obtained from the model in Appendix C.1 with the following UNIX `diff` code. The other models with synchronous communication are obtained similarly from the models in Appendices C.2 and C.3).

```
12d11
< #define maxMessages 1
41c40
< chan chanDM[maxHost] = [maxMessages] of { mtype, byte }
---
> chan chanDM[maxHost] = [0] of { mtype, byte } /* synchronous! */
58d56
<
320,323d317
<                 do                              /* empty the queue */
<                 :: (chanDM[Id]?[m(k)]) -> chanDM[Id]?m(k);
<                 :: else -> break;
<                 od;
336,339d329
<                 do                              /* empty the queue */
<                 :: (chanDM[Id]?[m(k)]) -> chanDM[Id]?m(k);
<                 :: else -> break;
<                 od;
351d340
<
```

C.5 Promela assertions for 3 DCM Managers

The properties described in these assertions are explained informally in Section 5.1, 5.2, and 5.3. All assertions are the same for synchronous or asynchronous communication between the DCM Managers.

Promela assertion: At most one leader

```
/* ----TO BE VERIFIED------------------------------------------------------- */
/*     'there is at most one leader'
*/

proctype Assertion()
{
  assert(
       ! (   (!BusResetPeriod)
```

```
          && (!BusResetDelivery[0].delivery)
          && (!BusResetDelivery[1].delivery)
          && (!BusResetDelivery[2].delivery)
          && (  (Local[0].leader && Local[1].leader)
              ||(Local[0].leader && Local[2].leader)
              ||(Local[1].leader && Local[2].leader)
             )
         )
      );
}
```

Promela assertion: Best final leader

```
/* ----TO BE VERIFIED-------------------------------------------------------- */
/*     'if a UrlCapable dcm is up, the final leader is
        always in UrlCapable mode'
*/

proctype Assertion()
{
    assert(
       ! (   (!BusResetPeriod)
          && (!BusResetDelivery[0].delivery)
          && (!BusResetDelivery[1].delivery)
          && (!BusResetDelivery[2].delivery)
          && (  (  (Local[0].fleader && !Local[0].UrlCapable)
                 &&(  (Global[1].up && Local[1].UrlCapable)
                    ||(Global[2].up && Local[2].UrlCapable)))
              ||(  (  (Local[1].fleader && !Local[1].UrlCapable)
                    &&(  (Global[0].up && Local[0].UrlCapable)
                       ||(Global[2].up && Local[2].UrlCapable)))
                 ||(  (Local[2].fleader && !Local[2].UrlCapable)
                    &&(  (Global[0].up && Local[0].UrlCapable)
                       ||(Global[1].up && Local[1].UrlCapable))))
             )
         )
      );
}
```

Promela assertion: Same final leader

```
/* ----TO BE VERIFIED-------------------------------------------------------- */
/*     'if there is a final leader, then everyone agrees on who this is'
*/

proctype Assertion()
{
    assert(
        ! (   (!BusResetPeriod)
           && (!BusResetDelivery[0].delivery)
           && (!BusResetDelivery[1].delivery)
           && (!BusResetDelivery[2].delivery)
           && (  (  Local[0].fleader
                  ||(Local[1].fleader || Local[2].fleader)
```

```
                )
             &&(  (   Global[0].up
                  && Global[1].up
                  &&(Local[0].Leader!=Local[1].Leader))
                ||(   Global[0].up
                  && Global[2].up
                  &&(Local[0].Leader!=Local[2].Leader))
                ||(   Global[1].up
                  && Global[2].up
                  &&(Local[1].Leader!=Local[2].Leader))
                )
             )
         )
      );
}
```

D The input files for Cæsar/Aldébaran and Xtl

D.1 ACT-ONE naturals library for 3 DCM Managers

The library `MY_NATURALS.lib` contains the naturals modulo 3, with the `+` and `-` operators, and some boolean operators.

```
type My_Natural is Boolean
   sorts Nat
   opns 0 (*! constructor *),
        1 (*! constructor *),
        2 (*! constructor *),
        3 (*! constructor *) : -> Nat
        _+_ ,
        _-_ : Nat, Nat -> Nat
        _==_ ,
        _<>_ ,
        _<_ ,
        _<=_ ,
        _>_ ,
        _>=_ : Nat, Nat -> Bool
   eqns
      forall m, n : Nat
      ofsort Nat        (* 3 + 1 or 1 + 3 reaches 0 again *)
         m + 0 = m;
         0 + m = m;
         1 + 1 = 2;
         1 + 2 = 3;
         1 + 3 = 0;
         2 + 1 = 3;
         (* 2 + 2 = 4; *)
         (* 2 + 3 = 5; *)
         3 + 1 = 0;
         (* 3 + 2 = 5; *)
         (* 3 + 3 = 6; *)
      ofsort Nat        (* I do not want to give equations for m-n with m<n! *)
         m - 0 = m;
         1 - 1 = 0;
```

```
         2 - 1 = 1;
         2 - 2 = 0;
         3 - 1 = 2;
         3 - 2 = 1;
         3 - 3 = 0;
      ofsort Bool
         0 == 0             = true;
         1 == 1             = true;
         2 == 2             = true;
         3 == 3             = true;
         0 == 1             = false;
         0 == 2             = false;
         0 == 3             = false;
         1 == 0             = false;
         1 == 2             = false;
         1 == 3             = false;
         2 == 0             = false;
         2 == 1             = false;
         2 == 3             = false;
         3 == 0             = false;
         3 == 1             = false;
         3 == 2             = false;
      ofsort Bool
         m <> n             = not (m == n);
      ofsort Bool
         m < 0              = false;
         0 < 1              = true;
         0 < 2              = true;
         0 < 3              = true;
         1 < 1              = false;
         1 < 2              = true;
         1 < 3              = true;
         2 < 1              = false;
         2 < 2              = false;
         2 < 3              = true;
         3 < 1              = false;
         3 < 2              = false;
         3 < 3              = false;
      ofsort Bool
         m <= n             = (m < n) or (m == n);
      ofsort Bool
         m >= n             = not (m < n);
      ofsort Bool
         m > n              = not (m <= n);
endtype
```

D.2 ACT-ONE data part for 3 DCM Managers with asynchronous communication

The following data part is tailored towards the situation of three DCM Managers in a setting with asynchronous communication between the DCM Managers (through a messaging system).

However, the data parts for two DCM Managers, or for a setting with synchronous communication are very similar to this particular listing.

```
library MY_NATURAL endlib
library X_BOOLEAN endlib
```

```

type Message is BOOLEAN, MY_NATURAL
   sorts Message1, Message2, MesFrame
   opns
     init_leader     (*! constructor *),
     final_leader    (*! constructor *),
     bus_reset_start (*! constructor *),
     bus_reset_end   (*! constructor *),
     bus_reset_event (*! constructor *),
     power_change    (*! constructor *),
     GUID_list       (*! constructor *),
     empty           (*! constructor *) : -> Message1

     DMInitRequest   (*! constructor *),
     DMInitReply     (*! constructor *) : -> Message2

     _==_ : Message2, Message2 -> Bool
     _<>_ : Message2, Message2 -> Bool

     consm (*! constructor *) : Message2, Nat, Bool -> MesFrame
     mes   : MesFrame -> Message2
     id    : MesFrame -> Nat
     UrlCapable : MesFrame -> Bool

   eqns forall   m,m1,m2: Message2, n:Nat, b:Bool

        ofsort Message2
           mes(consm(m,n,b)) = m
        ofsort Nat
           id(consm(m,n,b)) = n
        ofsort Bool
           UrlCapable(consm(m,n,b)) = b;
           DMInitRequest == DMInitRequest = true;
           DMInitReply == DMInitReply = true;
           DMInitReply == DMInitRequest = false;
           DMInitRequest == DMInitReply = false;
           m1 <> m2 = not(m1 == m2)
endtype

type MessageList is Message
  sorts Buffer
  opns
    emptyb  (*! constructor *) : -> Buffer
    addb    (*! constructor *) : MesFrame, Buffer -> Buffer
    headb   : Buffer -> MesFrame
    tailb   : Buffer -> Buffer

  eqns forall   l : Buffer,
                e : MesFrame

       ofsort MesFrame
          headb(addb(e,l))  = e

       ofsort Buffer
          tailb(emptyb)     = emptyb;
```

```
          tailb(addb(e,l))  = l

endtype

type EnrichedMessageList is MessageList
   opns
     append: MesFrame, Buffer -> Buffer
     length: Buffer -> Nat
     MaxBuf: -> Nat
   eqns forall m,m1,m2: MesFrame, buf:Buffer
        ofsort Buffer
           append(m,emptyb) = addb(m,emptyb);
           append(m1,addb(m2,buf)) = addb(m2,append(m1,buf))
        ofsort Nat
           length(emptyb) = 0;
           length(addb(m,buf))= 1 + length(buf);
           MaxBuf = 1
endtype

type Node is BOOLEAN, MY_NATURAL
   sorts Node
   opns
     consn (*! constructor *) : Bool -> Node
     up   : Node -> Bool
     count_up : Node -> Nat

   eqns forall n:Node, b1,b2:Bool
        ofsort Bool
           up(consn(b2)) = b2;
        ofsort Nat
           up(n) => count_up(n) = 1;
           not(up(n)) => count_up(n) = 0;

endtype


type NodeTuple is Node
  sorts Network
  opns
    consnet (*! constructor *) : Node, Node, Node -> Network
    firstn  : Network -> Node
    secondn : Network -> Node
    thirdn  : Network -> Node

  eqns forall   l : Network,
                n1,n2,n3 : Node

       ofsort Node
          firstn(consnet(n1,n2,n3))  = n1;
          secondn(consnet(n1,n2,n3)) = n2;
          thirdn(consnet(n1,n2,n3))  = n3;


endtype
```

```
type EnrichedNodeTuple is NodeTuple
   opns
     nr_uphosts : Network -> Nat
     flip : Nat, Network -> Network
     i_leader: Network -> Nat
   eqns forall n1,n2,n3,n4,n5,n6: Node
        ofsort Network
           flip(1,consnet(n1,n2,n3))
               = consnet(consn(not(up(n1))),n2,n3);
           flip(2,consnet(n1,n2,n3))
               = consnet(n1,consn(not(up(n2))),n3);
           flip(3,consnet(n1,n2,n3))
               = consnet(n1,n2,consn(not(up(n3))));
        ofsort Nat
           nr_uphosts(consnet(n1,n2,n3))
               = count_up(n1) + (count_up(n2) + count_up(n3));
           up(n1) => i_leader(consnet(n1,n2,n3)) = 1;
           not(up(n1)) and up(n2) => i_leader(consnet(n1,n2,n3)) = 2;
           not(up(n1)) and (not(up(n2)) and up(n3)) =>
               i_leader(consnet(n1,n2,n3)) = 3;
           not(up(n1)) and (not(up(n2)) and not(up(n3))) =>
               i_leader(consnet(n1,n2,n3)) = 0;
endtype

type Host is BOOLEAN, MY_NATURAL
   sorts Host
   opns
     consh (*! constructor *) : Nat, Bool, Bool -> Host
     id      : Host -> Nat
     UrlCapable   : Host -> Bool
     rec_req : Host -> Bool

   eqns forall n:Nat, b1:Bool, b2:Bool
        ofsort Nat
           id(consh(n,b1,b2)) = n
        ofsort Bool
           UrlCapable(consh(n,b1,b2)) = b1;
           rec_req(consh(n,b1,b2)) = b2
endtype

type HostList is Host
  sorts Hosts
  opns
    emptyh  (*! constructor *) : -> Hosts
    addh    (*! constructor *) : Host, Hosts -> Hosts
    headh   : Hosts -> Host
    tailh   : Hosts -> Hosts

  eqns forall   l : Hosts,
                e : Host

       ofsort Host
          headh(addh(e,l))  = e

       ofsort Hosts
```

```
          tailh(emptyh)     = emptyh;
          tailh(addh(e,l))  = l

endtype

type EnrichedHostList is HostList, EnrichedNodeTuple
   opns
     _==_: Hosts, Hosts -> Bool
     _<>_: Hosts, Hosts -> Bool
     init_hosts: Network -> Hosts
     chge_rec: Nat, Bool, Hosts -> Hosts
     rec: Nat, Hosts -> Bool
     f_leader: Nat, Hosts -> Nat
   eqns forall host,host1,host2: Host, hosts,hosts1,hosts2: Hosts,
               n1,n2,n3:Node, n:Nat, b:Bool
        ofsort Bool
           emptyh == emptyh = true;
           addh(host,hosts) == emptyh = false;
           emptyh == addh(host,hosts) = false;
           addh(host1,hosts1) == addh(host2,hosts2)
           = ((id(host1)==id(host2)) and ((UrlCapable(host1) iff UrlCapable(host2))
             and ((rec_req(host1) iff rec_req(host2)) and (hosts1==hosts2))));
           hosts1 <> hosts2 = not(hosts1 == hosts2);
        ofsort Hosts
           not(up(n1)) and (not(up(n2)) and not(up(n3)) ) =>
             init_hosts(consnet(n1,n2,n3))
               = emptyh;
           not(up(n1)) and (not(up(n2)) and up(n3) ) =>
             init_hosts(consnet(n1,n2,n3))
               = addh(consh(3,false,false),emptyh);
           not(up(n1)) and (up(n2) and not(up(n3)) ) =>
             init_hosts(consnet(n1,n2,n3))
               = addh(consh(2,false,false),emptyh);
           up(n1) and (not(up(n2)) and not(up(n3)) ) =>
             init_hosts(consnet(n1,n2,n3))
               = addh(consh(1,false,false),emptyh);
           not(up(n1)) and (up(n2) and up(n3) ) =>
             init_hosts(consnet(n1,n2,n3))
               = addh(consh(2,false,false),
                 addh(consh(3,false,false),emptyh));
           up(n1) and (not(up(n2)) and up(n3) ) =>
             init_hosts(consnet(n1,n2,n3))
               = addh(consh(1,false,false),
                 addh(consh(3,false,false),emptyh));
           up(n1) and (up(n2) and not(up(n3)) ) =>
             init_hosts(consnet(n1,n2,n3))
               = addh(consh(1,false,false),
                 addh(consh(2,false,false),emptyh));
           up(n1) and (up(n2) and up(n3) ) =>
             init_hosts(consnet(n1,n2,n3))
               = addh(consh(1,false,false),
                 addh(consh(2,false,false),
                 addh(consh(3,false,false),emptyh)));
           chge_rec(n,b,emptyh) = emptyh;
           (n <> id(host)) =>
```

```
              chge_rec(n,b,addh(host,hosts)) = addh(host,chge_rec(n,b,hosts));
          (n == id(host)) =>
              chge_rec(n,b,addh(host,hosts)) = addh(consh(n,b,true),hosts)
       ofsort Bool
              rec(n,emptyh) = true;
          (n<>id(host)) =>
              rec(n,addh(host,hosts)) = rec(n,hosts);
          (n==id(host)) =>
              rec(n,addh(host,hosts)) = rec_req(host)
       ofsort Nat
          f_leader(n,emptyh) = n;
          not(UrlCapable(host)) =>
              f_leader(n,addh(host,hosts)) = f_leader(n,hosts);
          UrlCapable(host) =>
              f_leader(n,addh(host,hosts)) = id(host)
endtype

```

D.3 Lotos behaviour part for 3 DCM Managers with asynchronous communication

```
specification leader_election
              [ginfo, gUpDown, gBusReset, gDMin, gDMout, gEvent]
              : noexit

behaviour

LE [ ginfo, gUpDown, gBusReset, gDMin, gDMout, gEvent ]
   ( consnet(consn(false), consn(false), consn(false)) ) (* all dcms down *)

where

process LE [ gInfo, gUpDown, gBusReset, gDMin, gDMout, gEvent ]
           ( net:Network )
  : noexit :=

  ( BusReset[gUpDown,gBusReset,gEvent] (net)
  )

  |[gUpDown, gBusReset]|

  ( ( ( DCM_Manager[gInfo,gUpDown,gDMin,gDMout,gEvent]
        (1,true)
        []
        DCM_Manager[gInfo,gUpDown,gDMin,gDMout,gEvent]
        (1,false) )
      |||
      ( DCM_Manager[gInfo,gUpDown,gDMin,gDMout,gEvent]
        (2,true)
        []
        DCM_Manager[gInfo,gUpDown,gDMin,gDMout,gEvent]
        (2,false) )
      |||
      ( DCM_Manager[gInfo,gUpDown,gDMin,gDMout,gEvent]
        (3,true)
```

```
   []
   DCM_Manager[gInfo,gUpDown,gDMin,gDMout,gEvent]
   (3,false) )
 )
 |[gInfo,gUpDown,gDMin,gDMout]|
 ( ( CMM[gInfo,gUpDown,gBusReset](1)
     |[gBusReset]|
     CMM[gInfo,gUpDown,gBusReset](2)
     |[gBusReset]|
     CMM[gInfo,gUpDown,gBusReset](3)
   )
   |[gUpDown,gBusReset]|
   ( MS[gUpDown,gBusReset,gDMout,gDMin](1)   (* DM's out is MS's in and vv *)
     |[gBusReset]|
     MS[gUpDown,gBusReset,gDMout,gDMin](2)   (* DM's out is MS's in and vv *)
     |[gBusReset]|
     MS[gUpDown,gBusReset,gDMout,gDMin](3)   (* DM's out is MS's in and vv *)
   )
 )
)

where

 process BusReset [ gUpDown, gBusReset , gEvent ]
     ( net: Network )

   : noexit :=

   gBusReset ! bus_reset_start
   ; BusReset2[gUpDown,gBusReset,gEvent]
     (net,1)

   where

     process BusReset2 [ gUpDown, gBusReset, gEvent ]
         (net: Network, j:Nat)

       : noexit :=

     ( [j==0]
         -> ( gBusReset !  bus_reset_end ! net
              ; BusReset[gUpDown,gBusReset,gEvent](net)
            )
     )
     []
     ( [j<>0]
         -> ( ( gUpDown ! j ! power_change
                ; BusReset2[gUpDown,gBusReset,gEvent]
                  (flip(j,net),j+1)
              )
              []
              ( i
                ; BusReset2[gUpDown,gBusReset,gEvent]
                  (net,j+1)
              )
```

```
               )
        )
        endproc (* BusReset2 *)

    endproc (* BusReset *)

    process FlushBusReset[gBusReset]
      : noexit :=

      ( gBusReset ! bus_reset_start
        ; FlushBusReset[gBusReset] )
      []
      ( choice b1,b2,b3:Bool
        [] ( gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
             ; FlushBusReset[gBusReset] ) )

    endproc (* FlushBusReset *)

    process CMM[ gInfo, gUpDown, gBusReset ]
            ( Id: Nat )
      : noexit :=

      CMMDown[gInfo,gUpDown,gBusReset](Id)

    where

      process CMMDown[ gInfo, gUpDown, gBusReset ]
              ( Id: Nat )
        : noexit :=

        FlushBusReset[gBusReset]
        [> ( gUpDown ! Id ! power_change
             ; ( choice b1,b2,b3:Bool
                 [] gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
                 ; CMMUp[gInfo,gUpDown,gBusReset]
                   (Id,consnet(consn(b1),consn(b2),consn(b3))) ) )
      endproc (* CMMDown *)

      process CMMUp[ gInfo, gUpDown, gBusReset ]
              ( Id: Nat , net: Network )
        : noexit :=

        CMMReady[gInfo,gUpDown,gBusReset](Id,net)
        [> ( gUpDown ! Id ! power_change
             ; CMMDown[gInfo,gUpDown,gBusReset](Id) )

      where
        process CMMReady[ gInfo, gUpDown, gBusReset ]
              ( Id: Nat , net: Network )
        : noexit :=

        ( gInfo ! Id ! GUID_list ! net
          ; CMMReady[gInfo,gUpDown,gBusReset](Id,net) )
        []
        ( gBusReset ! bus_reset_start
```

```
        ; CMMDeliver[gInfo,gUpDown,gBusReset](Id) )

      endproc (* CMMReady *)

      process CMMDeliver[ gInfo, gUpDown, gBusReset ]
            ( Id: Nat )
      : noexit :=

      ( gInfo ! Id ! bus_reset_event
        ; ( choice b1,b2,b3:Bool
            [] (gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
                ; CMMReady[gInfo,gUpDown,gBusReset]
                  (Id,consnet(consn(b1),consn(b2),consn(b3))))))
      []
      ( choice b1,b2,b3:Bool
               [] (gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
                   ; CMMDeliver2 [gInfo,gUpDown,gBusReset]
                     (Id,consnet(consn(b1),consn(b2),consn(b3)))
                  )
      )
      endproc (* CMMDeliver *)

      process CMMDeliver2[ gInfo, gUpDown, gBusReset ]
            ( Id: Nat , net: Network )
      : noexit :=

      ( gInfo ! Id ! GUID_list ! net
        ; CMMDeliver2[gInfo,gUpDown,gBusReset] (Id,net) )
      []
      ( gInfo ! Id ! bus_reset_event
        ; CMMReady[gInfo,gUpDown,gBusReset] (Id,net) )
      []
      ( gBusReset ! bus_reset_start
        ; CMMDeliver[gInfo,gUpDown,gBusReset](Id) )
      endproc (* CMMDeliver2 *)

    endproc (* CMMUp *)

  endproc (* CMM *)

  process MS[ gUpDown, gBusReset, gin, gout ]
          ( Id: Nat )
    : noexit :=

    MSDown[gUpDown,gBusReset,gin,gout](Id)

  where

    process MSDown[ gUpDown, gBusReset, gin, gout ]
            ( Id: Nat )
      : noexit :=

      FlushBusReset[gBusReset]
      [> ( gUpDown ! Id ! power_change
           ; MsUp[gUpDown,gBusReset,gin,gout](Id,emptyb) )
```

```
 endproc (* MSDown *)

 process MSUp[ gUpDown, gBusReset, gin, gout ]
     ( Id: Nat, buf: Buffer )
  : noexit :=

  MSSuspend[gUpDown,gBusReset,gin,gout](Id,buf)
  [> gUpDown ! Id ! power_change
     ; MSDown[gUpDown,gBusReset,gin,gout](Id)

  where

  process MSSuspend[ gUpDown, gBusReset, gin, gout ]
      ( Id: Nat, buf: Buffer )
   : noexit :=

   ( gin ! Id ! empty
    ; MSSuspend[gUpDown,gBusReset,gin,gout] (Id,emptyb) )
   []
   ( [length(buf)>0]
     -> ( gout ! Id ! headb(buf)
          ; MSSuspend[gUpDown,gBusReset,gin,gout] (Id,tailb(buf)) ) )
   []
   ( choice b1,b2,b3:Bool
    [] (gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
        ; MSReady[gUpDown,gBusReset,gin,gout](Id,buf)) )

  endproc (* MSSuspend *)

  process MSReady[ gUpDown, gBusReset, gin, gout ]
      ( Id: Nat, buf: Buffer )
   : noexit :=

   ( [length(buf)<maxBuf]
    -> ( choice m:Message2,j:Nat,b:Bool
         [] gin ! Id ! consm(m,j,b)
            ; MSReady[gUpDown,gBusReset,gin,gout]
              (Id,append(consm(m,j,b),buf)) ) )
   []
   ( gin ! Id ! empty
    ; MSReady[gUpDown,gBusReset,gin,gout] (Id,emptyb) )
   []
   ( [length(buf)>0]
    -> ( gout ! Id ! headb(buf)
         ; MSReady[gUpDown,gBusReset,gin,gout] (Id,tailb(buf)) ) )
   []
   ( gBusReset ! bus_reset_start
    ; MSSuspend[gUpDown,gBusReset,gin,gout](Id,buf) )
  endproc (* MSReady *)

 endproc (* MSUp *)

endproc (* MS *)
```

```
  process DCM_Manager[ ginfo, gUpDown, gDMin, gDMout, gEvent ]
          ( Id: Nat , UrlCapable: Bool )

    : noexit :=

    downDM[ginfo,gUpDown,gDMin,gDMout,gEvent](Id,UrlCapable)

    where

      process downDM[ ginfo, gUpDown, gDMin, gDMout, gEvent ]
              ( Id: Nat , UrlCapable: Bool )
        : noexit :=

        stop
        [> (gUpDown! Id ! power_change                (* Disrupt !! *)
            ; leDM[ginfo,gUpDown,gDMin,gDMout,gEvent]
              (Id,UrlCapable) )

      endproc (* downDM *)

      process leDM[ ginfo, gUpDown, gDMin, gDMout, gEvent ]
              ( Id: Nat , UrlCapable: Bool )
        : noexit :=

        ( choice b1,b2,b3:Bool
          [] gInfo ! Id ! GUID_list ! consnet(consn(b1),consn(b2),consn(b3))
             ; ( [i_leader(consnet(consn(b1),consn(b2),consn(b3)))==Id]
                   -> gEvent ! init_leader ! Id
                      ; ilDM[gDMin,gDMout,gEvent]
                        (Id,UrlCapable,consnet(consn(b1),consn(b2),consn(b3)))
                 []
                 [i_leader(consnet(consn(b1),consn(b2),consn(b3)))<>Id]
                   -> ifDM[gDMin,gDMout,gEvent]
                      (Id,UrlCapable,consnet(consn(b1),consn(b2),consn(b3)),
                       i_leader(consnet(consn(b1),consn(b2),consn(b3))))
             )
        )
        [> ( ( gInfo ! Id ! bus_reset_event           (* Disrupt !! *)
               ; ( gDMout ! Id ! empty
                   ; leDM[ginfo,gUpDown,gDMin,gDMout,gEvent](Id,UrlCapable))
                 []
                 ( gUpDown ! Id ! power_change
                   ; downDM[ginfo,gUpDown,gDMin,gDMout,gEvent](Id,UrlCapable))
             )
             []
             ( gUpDown ! Id ! power_change
               ; downDM[ginfo,gUpDown,gDMin,gDMout,gEvent](Id,UrlCapable))
           )
      endproc (* leDM *)

      process ifDM[ gDMin, gDMout, gEvent ]
              ( Id: Nat , UrlCapable: Bool , net: Network , leader: Nat )
        : noexit :=

        DeclareCapability[ gDMin, gDMout, gEvent](Id,UrlCapable,leader)
```

```

        [> ( choice j:Nat, b:Bool
             [] gDMin ! Id ! consm(DMInitReply,j,b)
                ; ( [j==Id]
                      -> gEvent ! final_leader ! Id ! UrlCapable
                         ; flDM[gDMin,gDMout,gEvent]
                            (Id,UrlCapable,net)
                    []
                    [j<>Id]
                      -> ffDM[gDMin,gDMout,gEvent]
                             (Id,UrlCapable,net,j)
               )
            )

        where
          process DeclareCapability[ gDMin, gDMout, gEvent ]
                  ( Id:Nat , UrlCapable: Bool , leader: Nat )
          : noexit :=
            (gDMout ! leader ! consm(DMInitRequest,Id,UrlCapable)
             ; DeclareCapability[gDMin,gDMout,gEvent](Id,UrlCapable,leader))
            []
            (choice j:Nat, b:Bool
             [] gDMin ! Id ! consm(DMInitRequest,j,b)
                ; DeclareCapability[gDMin,gDMout,gEvent](Id,UrlCapable,leader))

          endproc (* DeclareCapability *)
      endproc (* ifDM *)

      process ilDM[ gDMin, gDMout, gEvent ]
              ( Id: Nat , UrlCapable: Bool , net: Network )
        : noexit :=

        Elect[gDMin,gDMout,gEvent]
             (Id,UrlCapable,net,init_hosts(net),nr_uphosts(net))

        where
          process Elect[ gDMin, gDMout, gEvent ]
                  ( Id: Nat , UrlCapable: Bool , net: Network ,
                    hosts: Hosts, nr: Nat)
            : noexit :=
              ([nr==1]
                -> DeclareLeader[gDMin,gDMout,gEvent]
                   (Id,UrlCapable,net,hosts,f_leader(Id,chge_rec(Id,UrlCapable,hosts))))
              []
              ([nr>1]
                -> ( choice m:Message2,j:Nat,b:Bool
                     [] gDMin ! Id ! consm(m,j,b)
                     ; ( ([m==DMInitRequest and not(rec(j,hosts))]
                         -> Elect[gDMin,gDMout,gEvent]
                            (Id,UrlCapable,net,chge_rec(j,b,hosts),nr-1))
                         []
                         ([m<>DMInitRequest or rec(j,hosts)]
                         -> Elect[gDMin,gDMout,gEvent]
                            (Id,UrlCapable,net,hosts,nr))
                       )
```

```
                ))
        endproc (* Elect *)

        process DeclareLeader[ gDMin, gDMout, gEvent ]
                ( Id: Nat , UrlCapable: Bool , net: Network ,
                  hosts: hosts , leader: Nat )
          : noexit :=
            ([hosts==emptyh]
              -> ( [leader==Id]
                     -> (gEvent ! final_leader ! Id ! UrlCapable
                         ; fliDM[gDMin,gDMout,gEvent]
                           (Id,UrlCapable,net))
                   []
                   [leader<>Id]
                     -> (gDMout ! leader ! consm(DMInitReply,leader,false)
                         ; ffiDM[gDMin,gDMout,gEvent]
                           (Id,UrlCapable,net,leader))
                ))
            []
            ([hosts<>emptyh and ((id(headh(hosts))==Id)or(id(headh(hosts))==leader))]
              -> DeclareLeader[gDMin,gDMout,gEvent]
                 (Id,UrlCapable,net,tailh(hosts),leader))
            []
            ([hosts<>emptyh and ((id(headh(hosts))<>Id)and(id(headh(hosts))<>leader))]
              -> (gDMout ! id(headh(hosts)) ! consm(DMInitReply,leader,false)
                 ; DeclareLeader[gDMin,gDMout,gEvent]
                   (Id,UrlCapable,net,tailh(hosts),leader)))
        endproc (* DeclareLeader *)

    endproc (* ilDM *)

    process ffDM[ gDMin, gDMout, gEvent ]
            ( Id: Nat , UrlCapable: Bool , net: Network , leader: Nat )
      : noexit :=

        choice m:Message2,j:Nat,b:Bool
        [] gDMin ! Id ! consm(m,j,b)
           ; ffDM[gDMin,gDMout,gEvent] (Id,UrlCapable,net,leader)

    endproc (* flDM *)

    process ffiDM[ gDMin, gDMout, gEvent ]
            ( Id: Nat , UrlCapable: Bool , net: Network , leader: Nat )
      : noexit :=

      ( choice m:Message2,j:Nat,b:Bool
        [] gDMin ! Id ! consm(m,j,b)
           ; ( ([m==DMInitRequest]
                 -> (gDMout ! j ! consm(DMInitReply,leader,false)
                     ; ffiDM[gDMin,gDMout,gEvent]
                       (Id,UrlCapable,net,leader)))
              []
               ([m<>DMInitRequest]
                 -> ffiDM[gDMin,gDMout,gEvent]
                    (Id,UrlCapable,net,leader))
```

```
                 )
          )

        endproc (* ffiDM *)

        process flDM[ gDMin, gDMout, gEvent ]
                ( Id: Nat , UrlCapable: Bool , net: Network )
          : noexit :=

            choice m:Message2,j:Nat,b:Bool
            [] gDMin ! Id ! consm(m,j,b)
               ; flDM[gDMin,gDMout,gEvent] (Id,UrlCapable,net)

        endproc (* flDM *)

        process fliDM[ gDMin, gDMout, gEvent ]
                ( Id: Nat , UrlCapable: Bool , net: Network )
          : noexit :=

          ( choice m:Message2,j:Nat,b:Bool
            [] gDMin ! Id ! consm(m,j,b)
               ; ( ([m==DMInitRequest]
                    -> (gDMout ! j ! consm(DMInitReply,Id,false)
                        ; fliDM[gDMin,gDMout,gEvent]
                          (Id,UrlCapable,net)))
                 []
                   ([m<>DMInitRequest]
                    -> fliDM[gDMin,gDMout,gEvent]
                       (Id,UrlCapable,net))
                 )
          )

        endproc (* fliDM *)

    endproc (* DCM_Manager *)

endproc (* LE *)

endspec (* leader_election *)
```

D.4 ACTL properties for 3 DCM Managers with asynchronous communication

The properties described in these formulas are explained informally in Section 5.1, 5.2, 5.3, and 5.4.

ACTL property: At most one leader

```
(*========================================================================*)
(* Libraries used *)

library actl.xtl end_library

(*========================================================================*)
(* Basic predicates over actions *)
```

```
let

  BusResetStart   : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_START" ),
  BusResetEnd     : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_END" _ ),
  BusResetEvent   : labelset  = EVAL_A( GINFO _ !"BUS_RESET_EVENT" ),
  InitLeader      : labelset  = EVAL_A( GEVENT !"INIT_LEADER" _ ),
  FinalLeader     : labelset  = EVAL_A( GEVENT !"FINAL_LEADER" _ _ )

in let

  InitOrFinalLeader : labelset  = InitLeader or FinalLeader,
  Ignore1 : labelset  = (not(BusResetEvent or BusResetStart
                             or Initleader or FinalLeader)),
  Ignore2 : labelset  = (not(BusResetStart or BusResetEvent))

in

(*=========================================================================*)
(* Safety properties *)

print ("    Safety properties:") fby

(* If more than one DCM Manager becomes initial or final leader,
   then a busreset event must be pending *)

PRINT_FORM ("\tProperty   \n
    If more than one DCM Manager becomes initial or final leader,\n
    then a bus reset event must be pending : \n",

     Box( BusResetEnd,
          AG_A( Ignore1,
                Box( InitLeader,
                     AG_A( Ignore1,
                           Box( Initleader,
                                 EU_A_B( true,
                                         Ignore2,
                                         BusResetEvent,
                                         true
                                       )
                              )
                         )
                   )
              )
        )
     and
     Box( BusResetEnd,
          AG_A( Ignore1,
                Box( Finalleader,
                     AG_A( Ignore1,
                           Box( InitOrFinalLeader,
                                EU_A_B( true,
                                        Ignore2,
                                        BusResetEvent,
                                        true
                                      )
```

```
                              )
                         )
                   )
              )
        )

)

nop

end_let
end_let
```

ACTL property: Best final leader

```
(*==========================================================================*)
(* Libraries used *)

library actl.xtl end_library

(*==========================================================================*)
(* Basic predicates over actions *)

let

  ReqUrlCapable  : labelset  =
                 EVAL_A ( GDMIN _ ?m:string
                         where
                           (m="CONSM (DMINITREQUEST, 1, TRUE)")
                           or
                           (m="CONSM (DMINITREQUEST, 2, TRUE)")
                           or
                           (m="CONSM (DMINITREQUEST, 3, TRUE)")
                        )
                 or
                 EVAL_A ( GDMOUT _ ?m:string
                         where
                           (m="CONSM (DMINITREQUEST, 1, TRUE)")
                           or
                           (m="CONSM (DMINITREQUEST, 2, TRUE)")
                           or
                           (m="CONSM (DMINITREQUEST, 3, TRUE)")
                        ),
  BusResetStart   : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_START" ),
  BusResetEnd     : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_END" _ ),
  BusResetEvent   : labelset  = EVAL_A( GINFO _ !"BUS_RESET_EVENT" ),
  FinalLeader     : labelset  = EVAL_A( GEVENT !"FINAL_LEADER" _ _ ),
  FLNotUrlCapable : labelset  = EVAL_A( GEVENT !"FINAL_LEADER" _ ?b:boolean
                          where not(b) )

in let

  Ignore1 : labelset  = (not(BusResetEvent or BusResetStart or BusResetEnd
                             or FinalLeader)),
  Ignore2 : labelset  = (not(BusResetStart or BusResetEvent))
```

```
in

(*==========================================================================*)
(* Safety properties *)

print ("    Safety properties:") fby

(* If a DCM Manager becomes final leader in not UrlCapable mode,
   and there were InitRequests with UrlCapable=true
   then a busreset event must be pending *)

PRINT_FORM ("\tProperty   \n
    If a DCM Manager becomes final leader in not UrlCapable mode, \n
    and there were InitRequests with UrlCapable=true \n
    then a bus reset event must be pending : ",

                Box( ReqUrlCapable,
                     AG_A( Ignore1,
                           Box( FLNotUrlCapable,
                                EU_A_B( true,
                                        Ignore2,
                                        BusResetEvent,
                                        true
                                      )
                              )
                         )
                   )

)

nop

end_let
end_let
```

ACTL property: Same final leader

```
(*==========================================================================*)
(* Libraries used *)

library actl.xtl end_library

(*==========================================================================*)
(* Maximum number of DCM Managers *)

def N () : integer = 3 end_def

(*==========================================================================*)
(* Basic predicates over actions *)

macro  InitReplj (j) =
              if (j=1)
              then ( EVAL_A ( GDMOUT _ ? m:string
                              where
                                (m="CONSM (DMINITREPLY, 1, FALSE)")
                            )
```

```
                ) or
                ( EVAL_A ( GDMIN _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 1, FALSE)")
                         )
                )
           else_if (j=2)
           then ( EVAL_A ( GDMOUT _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 2, FALSE)")
                         )
                ) or
                ( EVAL_A ( GDMIN _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 2, FALSE)")
                         )
                )
           else  (* (j=3) *)
                ( EVAL_A ( GDMOUT _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 3, FALSE)")
                         )
                ) or
                ( EVAL_A ( GDMIN _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 3, FALSE)")
                         )
                )
           end_if
end_macro
macro  InitReplnotj (j) =
           if (j=1)
           then ( EVAL_A ( GDMOUT _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 2, FALSE)")
                             or
                             (m="CONSM (DMINITREPLY, 3, FALSE)")
                         )
                ) or
                ( EVAL_A ( GDMIN _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 2, FALSE)")
                             or
                             (m="CONSM (DMINITREPLY, 3, FALSE)")
                         )
                )
           else_if (j=2)
           then ( EVAL_A ( GDMOUT _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 1, FALSE)")
                             or
                             (m="CONSM (DMINITREPLY, 3, FALSE)")
                         )
                ) or
                ( EVAL_A ( GDMIN _ ? m:string
```

```
                           where
                             (m="CONSM (DMINITREPLY, 1, FALSE)")
                             or
                             (m="CONSM (DMINITREPLY, 3, FALSE)")
                         )
                  )
              else  (* (j=3) *)
                  ( EVAL_A ( GDMOUT _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 1, FALSE)")
                             or
                             (m="CONSM (DMINITREPLY, 2, FALSE)")
                         )
                  ) or
                  ( EVAL_A ( GDMIN _ ? m:string
                           where
                             (m="CONSM (DMINITREPLY, 1, FALSE)")
                             or
                             (m="CONSM (DMINITREPLY, 2, FALSE)")
                         )
                  )
              end_if
end_macro
macro  FinalLeaderj (j) = EVAL_A( GEVENT !"FINAL_LEADER" ?n:integer _
                                 where (n=j))
end_macro
macro  FinalLeadernotj (j) = EVAL_A( GEVENT !"FINAL_LEADER" ?n:integer _
                                 where (n<>j))
end_macro

macro  IRorFLj(j) = InitReplj(j) or FinalLeaderj(j)  end_macro
macro  IRorFLnotj(j) = InitReplnotj(j) or FinalLeadernotj(j)  end_macro

let
    InitReply : labelset
              = EVAL_A ( GDMOUT _ ? m:string
                        where
                          (m="CONSM (DMINITREPLY, 1, FALSE)")
                          or
                          (m="CONSM (DMINITREPLY, 2, FALSE)")
                          or
                          (m="CONSM (DMINITREPLY, 3, FALSE)")
                          or
                          (m="CONSM (DMINITREPLY, 1, TRUE)")
                          or
                          (m="CONSM (DMINITREPLY, 2, TRUE)")
                          or
                          (m="CONSM (DMINITREPLY, 3, TRUE)")
                      )
                or
                EVAL_A ( GDMIN _ ? m:string
                        where
                          (m="CONSM (DMINITREPLY, 1, FALSE)")
                          or
                          (m="CONSM (DMINITREPLY, 2, FALSE)")
```

```
                           or
                           (m="CONSM (DMINITREPLY, 3, FALSE)")
                           or
                           (m="CONSM (DMINITREPLY, 1, TRUE)")
                           or
                           (m="CONSM (DMINITREPLY, 2, TRUE)")
                           or
                           (m="CONSM (DMINITREPLY, 3, TRUE)")
                       ),
    BusResetStart      : labelset = EVAL_A (GBUSRESET !"BUS_RESET_START"),
    BusResetEnd        : labelset = EVAL_A (GBUSRESET !"BUS_RESET_END" _),
    BusResetEvent      : labelset = EVAL_A (GINFO _ !"BUS_RESET_EVENT" ),
    FinalLeader        : labelset = EVAL_A (GEVENT !"FINAL_LEADER" _ _)
in let
    Ignore1   : labelset = not( BusResetEvent or BusResetStart or BusResetEnd
                                or InitReply or FinalLeader ),
    Ignore2   : labelset = not( BusResetEvent or BusResetStart or BusResetEnd )
in

(*==============================================================================*)
(* Safety properties *)

print ("\n    Safety properties:\n\n") fby

PRINT_FORM ("\tProperty   \n
    If init replies/leader events carry a different Leader Id\n
    then a bus reset event must be pending :\n ",

  forall j: integer among {1 ... N} in
       Box( IRorFLj(j),
            AG_A( Ignore1,
                  Box( IRorFLnotj(j),
                       EU_A_B( true,
                               Ignore2,
                               BusResetEvent,
                               true
                             )
                     )
                )
           )
  end_forall

)

nop

end_let
end_let
```

ACTL property: Eventually final leader

```
(*==============================================================================*)
(* Libraries used *)

library actl.xtl end_library
```

```
(*==========================================================================*)
(* Basic predicates over actions *)

let
    FinalLeader      : labelset = EVAL_A (GEVENT !"FINAL_LEADER" _ _),
    BusResetStart    : labelset = EVAL_A (GBUSRESET !"BUS_RESET_START"),
    InfoGUIDlist     : labelset = EVAL_A (GINFO _ !"GUID_LIST" _)

in let
    Ignore : labelset = not(BusResetStart or FinalLeader)
in

(*==========================================================================*)
(* Liveness properties *)

print ("    Liveness properties:\n\n") fby

(* AlwaysFinalLeaderIfOneDMUpAndNotBusResetStart *)

print ("\tProperty   'Always Final Leader If One DM Up And Not BusResetStart' : ") fby

PRINT_FORM(
 Box( InfoGUIDlist,
      EU_A_B( true,
              Ignore,
              FinalLeader,
              true
            )
    )

)

nop

end_let
end_let
```

D.5 Lotos behaviour for 3 DCM Managers with synchronous communication

This Lotos behaviour model uses the library listed in Appendix D.1, and a data part very similar to the listing in Appendix D.2 (it can be obtained from the latter by deleting the `MesFrame` and `Buffer` definitions).

```
specification leader_election
              [gInfo, gUpDown, gBusReset, gDM, gEvent]
              : noexit

behaviour

LE [ gInfo, gUpDown, gBusReset, gDM, gEvent ]
   ( consnet(consn(false), consn(false), consn(false)) ) (* all dcms down *)

where

process LE [ gInfo, gUpDown, gBusReset, gDM, gEvent ]
```

```
         ( net:Network )
: noexit :=

( BusReset[gUpDown,gBusReset,gEvent] (net)
)

|[gUpDown, gBusReset]|

( ( ( ( DCM_Manager[gInfo,gUpDown,gDM,gEvent]
        (1,true)
        []
        DCM_Manager[gInfo,gUpDown,gDM,gEvent]
        (1,false) )
      |||
      OtherCommunications[gDM](1) )
    |[gDM]|
    ( ( DCM_Manager[gInfo,gUpDown,gDM,gEvent]
        (2,true)
        []
        DCM_Manager[gInfo,gUpDown,gDM,gEvent]
        (2,false) )
      |||
      OtherCommunications[gDM](2) )
    |[gDM]|
    ( ( DCM_Manager[gInfo,gUpDown,gDM,gEvent]
        (3,true)
        []
        DCM_Manager[gInfo,gUpDown,gDM,gEvent]
        (3,false) )
      |||
      OtherCommunications[gDM](3) )
  )
  |[gInfo,gUpDown]|
  ( ( CMM[gInfo,gUpDown,gBusReset](1)
      |[gBusReset]|
      CMM[gInfo,gUpDown,gBusReset](2)
      |[gBusReset]|
      CMM[gInfo,gUpDown,gBusReset](3)
    )
  )
)

where

  process BusReset [ gUpDown, gBusReset , gEvent ]
          ( net: Network )

    : noexit :=

    gBusReset ! bus_reset_start
    ; BusReset2[gUpDown,gBusReset,gEvent]
      (net,1)

    where

```

```
        process BusReset2 [ gUpDown, gBusReset, gEvent ]
                (net: Network, j:Nat)

          : noexit :=

        ( [j==0]
            -> ( gBusReset !  bus_reset_end ! net
                 ; BusReset[gUpDown,gBusReset,gEvent](net)
               )
        )
        []
        ( [j<>0]
            -> ( ( gUpDown ! j ! power_change
                   ; BusReset2[gUpDown,gBusReset,gEvent]
                     (flip(j,net),j+1)
                 )
                 []
                 ( i
                   ; BusReset2[gUpDown,gBusReset,gEvent]
                     (net,j+1)
                 )
               )
        )
        endproc (* BusReset2 *)

    endproc (* BusReset *)

    process FlushBusReset[gBusReset]
      : noexit :=

      ( gBusReset ! bus_reset_start
        ; FlushBusReset[gBusReset] )
      []
      ( choice b1,b2,b3:Bool
        [] ( gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
             ; FlushBusReset[gBusReset] ) )

    endproc (* FlushBusReset *)

    process CMM[ gInfo, gUpDown, gBusReset ]
            ( Id: Nat )
      : noexit :=

      CMMDown[gInfo,gUpDown,gBusReset](Id)

    where

      process CMMDown[ gInfo, gUpDown, gBusReset ]
              ( Id: Nat )
        : noexit :=

        FlushBusReset[gBusReset]
        [> ( gUpDown ! Id ! power_change
             ; ( choice b1,b2,b3:Bool
                 [] gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
```

```
           ; CMMUp[gInfo,gUpDown,gBusReset]
             (Id,consnet(consn(b1),consn(b2),consn(b3))) ) )
endproc (* CMMDown *)

process CMMUp[ gInfo, gUpDown, gBusReset ]
        ( Id: Nat , net: Network )
  : noexit :=

  CMMReady[gInfo,gUpDown,gBusReset](Id,net)
  [> ( gUpDown ! Id ! power_change
       ; CMMDown[gInfo,gUpDown,gBusReset](Id) )

where
  process CMMReady[ gInfo, gUpDown, gBusReset ]
        ( Id: Nat , net: Network )
  : noexit :=

  ( gInfo ! Id ! GUID_list ! net
    ; CMMReady[gInfo,gUpDown,gBusReset](Id,net) )
  []
  ( gBusReset ! bus_reset_start
    ; CMMDeliver[gInfo,gUpDown,gBusReset](Id) )

  endproc (* CMMReady *)

  process CMMDeliver[ gInfo, gUpDown, gBusReset ]
        ( Id: Nat )
  : noexit :=

  ( gInfo ! Id ! bus_reset_event
    ; ( choice b1,b2,b3:Bool
        [] (gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
            ; CMMReady[gInfo,gUpDown,gBusReset]
              (Id,consnet(consn(b1),consn(b2),consn(b3))))))
  []
  ( choice b1,b2,b3:Bool
           [] (gBusReset ! bus_reset_end ! consnet(consn(b1),consn(b2),consn(b3))
               ; CMMDeliver2 [gInfo,gUpDown,gBusReset]
                 (Id,consnet(consn(b1),consn(b2),consn(b3)))
              )
  )
  endproc (* CMMDeliver *)

  process CMMDeliver2[ gInfo, gUpDown, gBusReset ]
        ( Id: Nat , net: Network )
  : noexit :=

  ( gInfo ! Id ! GUID_list ! net
    ; CMMDeliver2[gInfo,gUpDown,gBusReset] (Id,net) )
  []
  ( gInfo ! Id ! bus_reset_event
    ; CMMReady[gInfo,gUpDown,gBusReset] (Id,net) )
  []
  ( gBusReset ! bus_reset_start
    ; CMMDeliver[gInfo,gUpDown,gBusReset](Id) )
```

```
        endproc (* CMMDeliver2 *)

      endproc (* CMMUp *)

    endproc (* CMM *)

    process OtherCommunications[ gDM ]
            ( Id: Nat )
      : noexit :=

        ( gDM ? j:Nat ? k:Nat ! DMInitRequest ? b:Bool
          [(j<>Id) and (j<>0) and (k<>Id) and (k<>0)]
          ; OtherCommunications [gDM](Id))
        []
        ( gDM ? j:Nat ? k:Nat ! DMInitReply ? l:Nat
          [(j<>Id) and (j<>0) and (k<>Id) and (k<>0) and (l<>0)]
          ; OtherCommunications [gDM](Id))

    endproc (* OtherCommunications *)

    process DCM_Manager[ gInfo, gUpDown, gDM, gEvent ]
            ( Id: Nat , UrlCapable: Bool )
      : noexit :=

      downDM[gInfo,gUpDown,gDM,gEvent](Id,UrlCapable)

      where

        process downDM[ gInfo, gUpDown, gDM, gEvent ]
                ( Id: Nat , UrlCapable: Bool )
          : noexit :=

          stop
          [> (gUpDown! Id ! power_change                (* Disrupt !! *)
              ; leDM[gInfo,gUpDown,gDM,gEvent]
                (Id,UrlCapable) )

        endproc (* downDM *)

        process leDM[ gInfo, gUpDown, gDM, gEvent ]
                ( Id: Nat , UrlCapable: Bool )
          : noexit :=

          ( choice b1,b2,b3:Bool
            [] let net:Network=consnet(consn(b1),consn(b2),consn(b3))
               in ( gInfo ! Id ! GUID_list ! net
                    ; ( [i_leader(net)==Id]
                           -> gEvent ! init_leader ! Id
                              ; ilDM[gDM,gEvent](Id,UrlCapable,net)
                        []
                         [i_leader(net)<>Id]
                           -> ifDM[gDM,gEvent]
                              (Id,UrlCapable,net,i_leader(net))
                      )
                  )
```

```
    )
    [> ( ( gInfo ! Id ! bus_reset_event           (* Disrupt !! *)
           ; leDM[gInfo,gUpDown,gDM,gEvent](Id,UrlCapable))
         []
         ( gUpDown ! Id ! power_change
           ; downDM[gInfo,gUpDown,gDM,gEvent](Id,UrlCapable))
       )
  endproc (* leDM *)

  process ifDM[ gDM, gEvent ]
          ( Id: Nat , UrlCapable: Bool , net: Network , leader: Nat )
    : noexit :=

    DeclareCapability[ gDM, gEvent](Id,UrlCapable,leader)

    [> ( gDM ! Id ! leader ! DMInitReply ? j:Nat[j<>0]
         ; ( [j==Id]
               -> ( gEvent ! final_leader ! Id ! UrlCapable
                    ; flDM[gDM,gEvent](Id,UrlCapable,net) )
             []
             [j<>Id]
               -> ffDM[gDM,gEvent](Id,UrlCapable,net,j)
           )
       )

    where
      process DeclareCapability[ gDM, gEvent ]
              ( Id:Nat , UrlCapable: Bool , leader: Nat )
      : noexit :=

        ( gDM ! leader ! Id ! DMInitRequest ! UrlCapable
          ; DeclareCapability[gDM,gEvent](Id,UrlCapable,leader))
        []
        ( gDM ! Id ? k:Nat ! DMInitRequest ? b:Bool
          [(k<>Id) and (k<>0)]
          ; DeclareCapability[gDM,gEvent](Id,UrlCapable,leader))

      endproc (* DeclareCapability *)
  endproc (* ifDM *)

  process ilDM[ gDM, gEvent ]
          ( Id: Nat , UrlCapable: Bool , net: Network )
    : noexit :=

    Elect[gDM,gEvent]
         (Id,UrlCapable,net,init_hosts(net),nr_uphosts(net))

    where
      process Elect[ gDM, gEvent ]
              ( Id: Nat , UrlCapable: Bool , net: Network ,
                hosts: Hosts, nr: Nat)
        : noexit :=
          ([nr==1]
            -> DeclareLeader[gDM,gEvent]
               (Id,UrlCapable,net,hosts,f_leader(Id,chge_rec(Id,UrlCapable,hosts))))
```

```
        []
        ([nr>1]
          -> ( gDM ! Id ? j:Nat ! DMInitRequest ? b:Bool
               [(j<>Id) and (j<>0)]
               ; ( [not(rec(j,hosts))]
                     -> Elect[gDM,gEvent]
                        (Id,UrlCapable,net,chge_rec(j,b,hosts),nr-1)
                 []
                   [rec(j,hosts)]
                     -> Elect[gDM,gEvent]
                        (Id,UrlCapable,net,hosts,nr)
                 )
             )
        )
        []
        ( gDM ! Id ? k:Nat ! DMInitReply ? l:Nat
          [(k<>Id) and (k<>0) and (l<>0)]
          ; Elect[gDM,gEvent] (Id,UrlCapable,net,hosts,nr))
    endproc (* Elect *)

    process DeclareLeader[ gDM, gEvent ]
            ( Id: Nat , UrlCapable: Bool , net: Network ,
              hosts: hosts , leader: Nat )
      : noexit :=
        ([hosts==emptyh]
          -> ( [leader==Id]
                 -> (gEvent ! final_leader ! Id ! UrlCapable
                     ; fliDM[gDM,gEvent]
                       (Id,UrlCapable,net))
               []
               [leader<>Id]
                 -> (gDM ! leader ! Id ! DMInitReply ! leader
                     ; ffiDM[gDM,gEvent]
                       (Id,UrlCapable,net,leader))
             ))
        []
        ([hosts<>emptyh and ((id(headh(hosts))==Id)or(id(headh(hosts))==leader))]
          -> DeclareLeader[gDM,gEvent]
             (Id,UrlCapable,net,tailh(hosts),leader))
        []
        ([hosts<>emptyh and ((id(headh(hosts))<>Id)and(id(headh(hosts))<>leader))]
          -> (gDM ! id(headh(hosts)) ! Id ! DMInitReply ! leader
             ; DeclareLeader[gDM,gEvent]
               (Id,UrlCapable,net,tailh(hosts),leader)))
    endproc (* DeclareLeader *)

endproc (* ilDM *)

process ffDM[ gDM, gEvent ]
        ( Id: Nat , UrlCapable: Bool , net: Network , leader: Nat )
  : noexit :=

    ( gDM ! Id ? k:Nat ! DMInitRequest ? b:Bool
      [(k<>Id) and (k<>0)]
      ; ffDM[gDM,gEvent] (Id,UrlCapable,net,leader))
```

```
            []
            ( gDM ! Id ? k:Nat ! DMInitReply ? l:Nat
              [(k<>Id) and (k<>0) and (l<>0)]
              ; ffDM[gDM,gEvent] (Id,UrlCapable,net,leader))

        endproc (* flDM *)

        process ffiDM[ gDM, gEvent ]
                ( Id: Nat , UrlCapable: Bool , net: Network , leader: Nat )
          : noexit :=

            ( gDM ! Id ? j:Nat ! DMInitRequest ? b:Bool
              [j<>Id and (j<>0)]
              ; ( gDM ! j ! Id ! DMInitReply ! leader
                  ; ffiDM[gDM,gEvent] (Id,UrlCapable,net,leader)))
            []
            ( gDM ! Id ? k:Nat ! DMInitReply ? l:Nat
              [(k<>Id) and (k<>0) and (l<>0)]
              ; ffiDM[gDM,gEvent] (Id,UrlCapable,net,leader))

        endproc (* ffiDM *)

        process flDM[ gDM, gEvent ]
                ( Id: Nat , UrlCapable: Bool , net: Network )
          : noexit :=

            ( gDM ! Id ? k:Nat ! DMInitRequest ? b:Bool
              [(k<>Id) and (k<>0)]
              ; flDM[gDM,gEvent] (Id,UrlCapable,net))
            []
            ( gDM ! Id ? k:Nat ! DMInitReply ? l:Nat
              [(k<>Id) and (k<>0) and (l<>0)]
              ; flDM[gDM,gEvent] (Id,UrlCapable,net))

        endproc (* flDM *)

        process fliDM[ gDM, gEvent ]
                ( Id: Nat , UrlCapable: Bool , net: Network )
          : noexit :=

            ( gDM ! Id ? j:Nat ! DMInitRequest ? b:Bool
              [j<>Id and(j<>0)]
               ; ( gDM ! j !Id ! DMInitReply ! Id
                   ; fliDM[gDM,gEvent] (Id,UrlCapable,net)))
            []
            ( gDM ! Id ? k:Nat ! DMInitReply ? l:Nat
              [(k<>Id) and (k<>0) and (l<>0)]
              ; fliDM[gDM,gEvent] (Id,UrlCapable,net))

        endproc (* fliDM *)

    endproc (* DCM_Manager *)

endproc (* LE *)
```

```
endspec (* leader_election *)
```

D.6 ACTL properties for 3 DCM Managers with synchronous communication

The properties described in these formulas are explained informally in Section 5.1, 5.2, 5.3, and 5.4.

ACTL property: At most one leader

```
(*========================================================================*)
(* Libraries used *)

library actl.xtl end_library

(*========================================================================*)
(* Basic predicates over actions *)

let

  BusResetStart   : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_START" ),
  BusResetEnd     : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_END" _ ),
  BusResetEvent   : labelset  = EVAL_A( GINFO _ !"BUS_RESET_EVENT" ),
  InitLeader      : labelset  = EVAL_A( GEVENT !"INIT_LEADER" _ ),
  FinalLeader     : labelset  = EVAL_A( GEVENT !"FINAL_LEADER" _ _ )

in let

  InitOrFinalLeader : labelset  = InitLeader or FinalLeader,
  Ignore1 : labelset  = (not(BusResetEvent or BusResetStart
                             or Initleader or FinalLeader)),
  Ignore2 : labelset  = (not(BusResetStart or BusResetEvent))

in

(*========================================================================*)
(* Safety properties *)

print ("    Safety properties:") fby

(* If more than one DCM Manager becomes initial or final leader,
   then a busreset event must be pending *)

PRINT_FORM ("\tProperty   \n
    If more than one DCM Manager becomes initial or final leader,\n
    then a bus reset event must be pending : \n",

     Box( BusResetEnd,
          AG_A( Ignore1,
                Box( InitLeader,
                     AG_A( Ignore1,
                           Box( Initleader,
                                EU_A_B( true,
                                        Ignore2,
                                        BusResetEvent,
                                        true
```

```
                                    )
                              )
                          )
                    )
               )
          )
       and
       Box( BusResetEnd,
            AG_A( Ignore1,
                  Box( Finalleader,
                       AG_A( Ignore1,
                             Box( InitOrFinalLeader,
                                  EU_A_B( true,
                                          Ignore2,
                                          BusResetEvent,
                                          true
                                        )
                                  )
                             )
                       )
                  )
          )

)

nop

end_let
end_let
```

ACTL property: Best final leader

```
(*===========================================================================*)
(* Libraries used *)

library actl.xtl end_library

(*===========================================================================*)
(* Basic predicates over actions *)

let

  ReqUrlCapable  : labelset  =
                 EVAL_A ( GDM _ _ !"DMINITREQUEST" ?b:boolean
                          where (b) ),
  BusResetStart   : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_START" ),
  BusResetEnd     : labelset  = EVAL_A( GBUSRESET !"BUS_RESET_END" _ ),
  BusResetEvent   : labelset  = EVAL_A( GINFO _ !"BUS_RESET_EVENT" ),
  FinalLeader     : labelset  = EVAL_A( GEVENT !"FINAL_LEADER" _ _ ),
  FLNotUrlCapable : labelset  = EVAL_A( GEVENT !"FINAL_LEADER" _ ?b:boolean
                          where not(b) )

in let

  Ignore1 : labelset  = (not(BusResetEvent or BusResetStart or BusResetEnd
                             or FinalLeader)),
```

```
  Ignore2 : labelset  = (not(BusResetStart or BusResetEvent))

in

(*=============================================================================*)
(* Safety properties *)

print ("    Safety properties:") fby

(* If a DCM Manager becomes final leader in not UrlCapable mode,
   and there were InitRequests with UrlCapable=true
   then a busreset event must be pending *)

PRINT_FORM ("\tProperty   \n
    If a DCM Manager becomes final leader in not UrlCapable mode, \n
    and there were InitRequests with UrlCapable=true \n
    then a bus reset event must be pending : ",

                Box( ReqUrlCapable,
                     AG_A( Ignore1,
                           Box( FLNotUrlCapable,
                                 EU_A_B( true,
                                         Ignore2,
                                         BusResetEvent,
                                         true
                                       )
                               )
                         )
                   )

)

nop

end_let
end_let
```

ACTL property: Same final leader

```
(*=============================================================================*)
(* Libraries used *)

library actl.xtl end_library

(*=============================================================================*)
(* Maximum number of DCM Managers *)

def N () : integer = 3 end_def

(*=============================================================================*)
(* Basic predicates over actions *)

macro  InitReplyj (j) = EVAL_A ( GDM _ _ !"DMINITREPLY" ?n:integer
                                  where (n=j))
end_macro
macro  InitReplynotj (j) = EVAL_A ( GDM _ _ !"DMINITREPLY" ?n:integer
```

```
                                  where (n<>j))
end_macro
macro  FinalLeaderj (j) = EVAL_A( GEVENT !"FINAL_LEADER" ?n:integer _
                                  where (n=j))
end_macro
macro  FinalLeadernotj (j) = EVAL_A( GEVENT !"FINAL_LEADER" ?n:integer _
                                  where (n<>j))
end_macro

macro  IRorFLj(j) = InitReplyj(j) or FinalLeaderj(j)  end_macro
macro  IRorFLnotj(j) = InitReplynotj(j) or FinalLeadernotj(j)  end_macro

let
    InitReply         : labelset = EVAL_A (GDM _ _ !"DMINITREPLY" _),
    BusResetStart     : labelset = EVAL_A (GBUSRESET !"BUS_RESET_START"),
    BusResetEnd       : labelset = EVAL_A (GBUSRESET !"BUS_RESET_END" _),
    BusResetEvent     : labelset = EVAL_A (GINFO _ !"BUS_RESET_EVENT" ),
    FinalLeader       : labelset = EVAL_A (GEVENT !"FINAL_LEADER" _ _)
in let
    Ignore1   : labelset = not( BusResetEvent or BusResetStart or BusResetEnd
                                or InitReply or FinalLeader ),
    Ignore2   : labelset = not( BusResetEvent or BusResetStart or BusResetEnd )
in

(*=========================================================================*)
(* Safety properties *)

print ("\n    Safety properties:\n\n") fby

(* Between bus resets, at most 1 leader *)

PRINT_FORM ("\tProperty   \n
    Between BusReset Periods all init replies/leader events carry\n
    the same Leader Id :\n ",

  forall j: integer among {1 ... N} in
       Box( IRorFLj(j),
            AG_A( Ignore1,
                  Box(IRorFLnotj(j),
                          EU_A_B( true,
                                  Ignore2,
                                  BusResetEvent,
                                  true
                                )
                     )
                 )
            )
  end_forall

)

nop

end_let
end_let
```

ACTL property: Eventually final leader

```
(*==========================================================================*)
(* Libraries used *)

library actl.xtl end_library

(*==========================================================================*)
(* Basic predicates over actions *)

let
    FinalLeader      : labelset = EVAL_A (GEVENT !"FINAL_LEADER" _ _),
    BusResetStart    : labelset = EVAL_A (GBUSRESET !"BUS_RESET_START"),
    InfoGUIDlist     : labelset = EVAL_A (GINFO _ !"GUID_LIST" _)

in let
    Ignore : labelset = not(BusResetStart or FinalLeader)
in

(*==========================================================================*)
(* Liveness properties *)

print ("    Liveness properties:\n\n") fby

(* AlwaysFinalLeaderIfOneDMUpAndNotBusResetStart *)

print ("\tProperty   'Always Final Leader If One DM Up And Not BusResetStart' : ") fby

PRINT_FORM(
 Box( InfoGUIDlist,
      EU_A_B( true,
              Ignore,
              FinalLeader,
              true
            )
    )

)

nop

end_let
end_let
```