Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

The COMPASS Examples Compendium

Deliverable Number: D31.3d

Version: 0.5

Date: October 2013

Public Document

http://www.compass-research.eu

Contributors:

Peter Gorm Larsen, Aarhus University
Klaus Kristensen, Bang & Olufsen
Jim Woodcock, University of York
Simon Foster, University of York
Richard Payne, University of Newcastle
Uwe Schulze, Bremen University
Stefan Hallerstede, Aarhus University
Adalberto Cajueiro, UFPE

Editors:

Peter Gorm Larsen, AU
Stefan Hallerstede, AU

Reviewers:

Document History

Ver	Date	Author	Description
0.1	03-09-2013	Peter Gorm Larsen	Initial Template for examples
0.2	22-09-2013	Klaus Kristensen	Added the AV Device Discovery example
0.3	26-09-2013	Peter Gorm Larsen	Added more examples
0.4	08-10-2013	Jim Woodcock	Added more simple examples and overall table
0.5	29-11-2013	Stefan Hallerstede	Added Leader Election example (by Klaus Kristensen et al.); formatting

Contents

1 Introduction 6

2 The Stack Example 9
2.1 Case Description 9
2.2 Analysing using the Debugger 9
2.3 Potential Adjustments to Experiment with 9

3 The Incubator Monitor Example 10
3.1 Case Description 10
3.2 Analysing using the Debugger 10

4 The Bit Register Example 11
4.1 Case Description 11
4.2 Analysing using the Debugger 11
4.3 Potential Adjustments to Experiment with 11
4.4 Analysing using the Model Checker 11

5 The Simple Airport Control Example 13
5.1 Case Description 13

6 The Dwarf Example 14
6.1 Case Description 14
6.2 Analysing using the Debugger 15
6.3 Analysing using Proof Obligations and Theorem Proving 15
6.4 Potential Adjustments to Experiment with 15

7 The Library Example 16
7.1 Case Description 16

8 The Mini Mondex Example 17
8.1 Case Description 17
8.2 Analysing using Model Checking 17

9 The Turn Indicator Example 19
9.1 Case Description 19
9.2 Analysing using the Debugger 20

10 The AV Device Discovery Example 22
10.1 Case Description 22
10.2 Current Model limitations and Bugs 24

10.3 Analysing using the Debugger 24
10.4 Analysing using Model Checking 25

11 The Leadership Election Example 30
11.1 Case Description 30
11.2 Current Model limitations and errors 34
11.3 The CML model 36

1 Introduction

The purpose of this document is to provide an overview of a number of the public CML examples. The underlying idea is that stakeholders who are interested in experimenting with the COMPASS technology can use this as a starting point[1].

This deliverable is structured in different sections, each of which provides a brief (1 to 4 pages) introduction to one example model. Depending upon your own background and your interest in exploring the COMPASS technology and the particular features that interest you, you may wish to start with a different section than that presented first. The different sections and the examples they present illustrate different aspects as explained here:

- Section 2 illustrates a simple stack example.
- Section 3 illustrates a simple incubator monitor example.
- Section 4 illustrates a simple bit register with potential underflow and overflow of operations.
- Section 5 illustrates a simple airport traffic control system where different airplanes need to ask permission for landing.
- Section 6 illustrates a small railway dwarf control example.
- Section 7 illustrates a small library example.
- Section 8 illustrates a small smart card electronic cash system example (see `http://en.wikipedia.org/wiki/Mondex` for the inspiration for this work).
- Section 9 illustrates the functionality of a turn indicator for a car.
- Section 10 illustrates an example for discovery of Audio/Video devices from Bang & Olufsen.
- Section 11 illustrates an example for leadership election in a distributed Audio/Video network from Bang & Olufsen.

Depending upon what features from CML you would in particular like to examine Table 1 may also give you a hint about the best examples for you to explore. Note that there is a tendency that the simpler examplers are provided first so if you are

[1]The corresponding sources of all the examples can be accessed by importing them inside the tool.

entirely new to CML (and to its descendants VDM and CSP/Circus) it is probably advisable to start from the beginning.

CML model(s)	Basic data types	Set types	Sequence types	Mapping types	Invariants	Basic processes	Actions	Recursive processes	Parameterised processes
Stack	x		x			x			
Incubation Monitor	x					x			
Bit Register	x				x	x			
Simple Airport Control	x	x			x	x			
Incubator Monitoring	x				x	x			
Incubator Control	x				x	x			
Library	x	x	x	x	x	x	x		
Mini Mondex	x	x			x	x	x		x
Turn Indicator									
Device Discovery									
Leadership Election									

Table 1: Overview of characteristics for CML models

2 The Stack Example

2.1 Case Description

The basic stack example is shown in virtually every notation. Thus, this is initialising the stack to be empty and then has `Push` and `Pop` operations that can be called and different channels that connects the `Stack` process with the outside world.

2.2 Analysing using the Debugger

The `Stack` process can be debugged and it is possible to add **`token`** values as for example "`mk_token(42)`". Here the natural behaviour of the stack can be exercised. Note that you also can make breakpoints and inspect the different variables when stopped at such breakpoints.

2.3 Potential Adjustments to Experiment with

The current version of the `Stack` does not have an upper limit on its size. You can try to update it to have that. This can be done by introducing an invatiant on `stack` and an extra guard in the `Cycle` action when pushing a new element. Then you can see what effects that has on the debugger and on the proof obligations being generated.

3 The Incubator Monitor Example

3.1 Case Description

The basic incubation monitor example is simply something that continuous monitor a temperature that either can be increased with an `Increment` operation or decreased with a `Decrement` operation. This example illustrate the use of guards since there is a minimum and a maximum temperature.

3.2 Analysing using the Debugger

The `IncubatorMonitor` process can be debugged but no inoput is expected from the user at all (except for chosing to go up or down in temperature. Note that you also can make breakpoints and inspect the `temp` variable when stopped at such breakpoints.

4 The Bit Register Example

4.1 Case Description

This simple bit register example has been provided in VDM by a student called Andreas Müller from Austria (see `http://moodle.risc.jku.at/pluginfile.php/414/mod_resource/content/0/vdm-vortrag.pdf`). It has been translated to CML by Jim Woodcock and used for testing different features of the COMPASS tools.

This example represent the main operators you can conduct on a very simple calculator where you can have registers you can store and load values to/from and then addition and subtraction. Note that it on purpose only handles bytes up to 255 so it is possible to see the effect of overflows and underflows.

4.2 Analysing using the Debugger

Just like for the `Stack` example the `RegisterProc` process can be debugged and it is possible to load values into the register and try to add and subtract them from each other. Here the natural behaviour of the bit register (including underflow and overflow scenarios) can be exercised. Note that you also can make breakpoints and inspect the different variables when stopped at such breakpoints.

4.3 Potential Adjustments to Experiment with

In the current version there are a number of proof obligations generated that you will not be able to prove correct. Please update the argument types for some of the operations so this can be improved.

4.4 Analysing using the Model Checker

To analyse the bit register one needs to set the maximum value for bytes to be 16 and the initial value to be 15 (due to the performance of FORMULA). The code analysed in the model checker is given as follows:

```
-- The original value for MAX is 255.
-- Due to performance questions we use 8
values
```

```
 MAX : nat = 8

functions
   oflow : int*int -> bool
   oflow(i,j) == i+j > MAX

   uflow : int*int -> bool
   uflow(i,j) == i-j < 0

channels
   init, overflow, underflow
   inc, dec : int

process RegisterProc =
 begin

  state
  reg : int := 0

  operations
   INIT : () ==> ()
   INIT() == reg := MAX - 1

   ADD: int ==> ()
   ADD(i) == reg := reg + i

   SUB: int ==> ()
   SUB(i) == reg := reg - i

  actions

   REG =
  (inc.1 -> [not oflow(reg,1)] & ADD(1);REG)
  []
  (dec.1 -> [not uflow(reg,1)] & SUB(1);REG)
   @init -> INIT(); REG
 end
```

5 The Simple Airport Control Example

5.1 Case Description

This example originally came from [CK04] in a VDM-SL setting. It has been translated to CML by Jim Woodcock. This small example illustrate the logic behind granting permissions for landings and takeoffs for an airport. This is made at a very abstract level but it illustrates how success and failure of operations can be signalled over dedicated channels for this.

6 The Dwarf Example

6.1 Case Description

This CML model is developed jointly by Peter Gorm Larsen and Jim Woodcock is inspired by the Dwarf Signal control system described by Marcus Montigel, Alcatel Austria AG. This model was combined by a VDM model made by Peter Gorm Larsen and a CSP model made by Jim Woodcock and that both have been presented at a FM Railway workshop. Dwarf signals are mounted along railway tracks and give indications to the train drivers about when they are allowed to proceed or need to stop. The lights giving the signal for the drivers looks as sketched in Figure 1. If lights L1 and L2 are turned on it means stop, if L2 and L3 are turned on it means drive while a warning signal is shown by L1 and L3 being turned on (in this case the driver needs to proceed with great care).

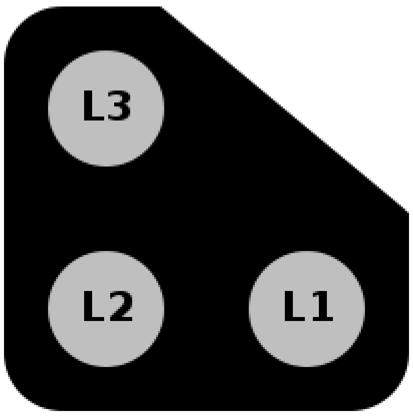

Figure 1: A small dwarf light with three different lights

Different safety requirements can be formulated as:

- Only one lamp may be changed at once
- All three lamps must never be on concurrently
- The signal must never be dark except if the dark aspect has to be shown or there is lamp failure
- The change to and from dark is allowed only from stop and to stop.

Below you can see how such requirements can be analysed using the CML tool support.

6.2 Analysing using the Debugger

The Dwarf model includes four different tests that one can try to experiment with in the debugger. If you use `TEST1` you can step through the `TEST1` action step by step and potentially incorporate additional `tock` steps moving the time at any point of time after `init`. In addition it is possible to `shine` the `dw.currentstate`. Trying to violate the safety requirements mentioned above can be experimented with using some of the other `TEST`'s. In these cases the preconditions will be violated and a run-time error will be issued.

6.3 Analysing using Proof Obligations and Theorem Proving

The theorem prover plugin allows the generation of an Isabelle theory for the Dwarf model. Using the Isabelle perspective, model-based theorems may be stated and discharged with the `cml_auto_tac` proof tactic. Such theorems include checking that an initialised state of the signal satisfies the safety invariants, for example:

```
lemma NeverShowAll_Init:
"|NeverShowAll(mk_DwarfType(&stop,&stop,{},{},&stop,&stop))|

 = |true|"
  by (cml_auto_tac)
```

The proof obligation generator produces four `Operation Postcondition` proof obligations. These relate to the correctness of the operation body with respect to the operation bodies. At present these can not yet be automatically discharged.

6.4 Potential Adjustments to Experiment with

7 The Library Example

7.1 Case Description

This CML model has been developed by Peter Gorm Larsen and Jim Woodcock. This is a classic standard example that has been treated by a large collection of formal methods [Win88]. The informal requirements tackled by the different notations was formulated as:

> Consider a small library database with the following transactions:
>
> 1. Checkout a copy of a book / Return a copy of a book;
> 2. Add a copy of a book to the library / Remove a copy of a book from the library;
> 3. Get the list of books by a particular author or in a particular subject area;
> 4. Find out the list of books currently checked out by a particular borrower;
> 5. Find out what borrower last checked out a particular copy of a book.
>
> There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:
>
> 1. All copies in the library must be available for checkout or be checked out.
> 2. No copy of the book may be both available and checked out at the same time.
> 3. A borrower may not have more than a predefined number of books checked out at one time

As such the model presented here does not present any new findings but it is used to illutrate how it can be formulated using CML.

8 The Mini Mondex Example

8.1 Case Description

This CML model is developed by Jim Woodcock inspired by the original work done with the Modex secure card system [SCW00, WSC+08].

8.2 Analysing using Model Checking

Due to performance issues and limitations on communication parameters, it is possible to check only one `Card` process. The analysed code is given as follows.

```
values
 N: nat = 2
 V: nat = 1
 M: nat = N*V

types
  Index = nat
 inv i == i in set {1,...,N}

  Money = nat
 inv m == m in set {0,...,M}


channels
  pay, transfer: Index * Index * Money
  accept , reject: Index

process Card = val i: Index @
begin
  state value: nat := 0

  operations
 Init: () ==> ()
 Init() == value := V

 Credit: nat ==> ()
 Credit(n) == value := value + n

 Debit: nat ==> ()
 Debit(n) == value := value - n

  actions
 Transfer = pay.i?j?n ->
 ( [n > value] & reject!i -> Skip
   []
   [n <= value] & transfer.i.j!n -> accept!i -> Debit(n) )

 Receive = transfer?j.i?n -> Credit(n)

 Cycle = ( Transfer [] Receive ); Cycle

@ Init(); Cycle

end

process OneCard = Card(1)

--THE FOLLOWING PROCESSES MAKE THE ANALYSIS VERY SLOW
process Cards =
  || i: Index @
 [ {| pay.i,transfer.i,accept.i, reject.i |}
   union
   {| transfer.j.i.n | j:Index,n:Money|} ] Card(i)

process Network = Cards \\ {|transfer|}
```

9 The Turn Indicator Example

9.1 Case Description

The CML model is developed by Jan Peleska and Uwe Schulze and it is inspired by collaborative work that has been done together with Daimler also using SysML and automating testing using RT Tester [Pel02, BPS12].

The model is a simplified version of a controller in a car responsible for turn indication flashing and emergency flashing and the interactions between these two actions. The controller reacts on three external inputs:

- `TurnIndLvr`: The turn indication lever with allowed values `<neutral>`, `<left>` and `<right>`
- `EmerFlash`: A button to command Emergency Flashing with the states **`true`** = pressed = emergency flashing active and **`false`** = released = emergency flashing inactive.
- `Voltage`: The voltage available from the power source of the car. Values between 103 and 140 represent a value between 10.3 and 14 volt and are acceptable for flashing. In case of lower or higher values flashing is disabled.

The controller sends indications to the rest of the car through the channels

- `FlashLeft`: The value **`true`** switches the left turn indication light on and **`false`** switches off the turn indication light.
- `FlashRight`: The value **`true`** switches the right turn indication light on and **`false`** switches off the turn indication light.

The main process is composed of several actions:

- `FLASH_CTRL`: This action on the input channels `TurnIndLvr` and `EmerFlash` and calculates in which flashing state the system is. The system can either be not flashing at all, perform turn indication flashing or can perform emergency flashing. This action also handles the possible overrides, e.g. emergency flashing overriding turn indication flashing (if the emergency switch is pressed while turn indication flashing is active) or turn indication flashing overriding emergency flashing (when turn indication flashing is commanded while emergency flashing is active). The action uses the internal events `newFlashState` and `newLvrState` to communicate with the other the turn indication actions.

- OUTPUT_CTRL : This action controls the actual switching of the left and right turn indication lights. It reacts on state changes received from FLASH_CTRL through newFlashState and newLvrState to calculate which lights to switch on or off. The current voltage is received from the action VOLTAGE through the channel newVoltage and is used to check if the voltage is acceptable for flashing or not. If any flashing is to be performed, this action controls the length of the flashing periods and also provides tip flashing functionality.
- SWITCH_OFF and NEW_DIRECTION: These actions a used to interrupt OUTPUT_CTRL whenever during active flashing the voltage is not acceptable anymore or the turn indication lever state changes. The internal events switchOff and newDirection are used to communicate these situations.
- VOLTAGE: This simple action receives new voltage values from the environment and communicates them to the other turn indication actions using the channel newVoltage.
- TIMER: This action implements two timers (Timer220 and Timer340) with set and elapsed events that are used by OUTPUT_CTRL to model on and off the flashing periods.

9.2 Analysing using the Debugger

The process TurnIndCtrlProcess can be debugged and the system can be stimulated to turn indication flashing or emergency flashing. Every execution of the model should start with an initialsation trace

```
[FlashLeft.false,
 FlashRight.false,
 newLvrState.<neutral>,
 newFlashState.mk_FlashState(false, false, 0)]
```

initialising the flashing outputs. The initial values for the flash states are already defined in the model. After the model is initialised, new values for the voltage, the turn indication lever state or the emergency switch state can be stimulated through the channels TurnIndLvr, EmerFlash and Voltage.

Note that the timer process TIMER in the current version of the model only waits for 2 tock events (Timer220) or 3 tock events (Timer340) instead of 220 respectively 340 tock events. With the original values, a lot of tock events would have to be performed manually when debugging the flashing cycles.

10 The AV Device Discovery Example

10.1 Case Description

This Device Discovery CML model represents some SoS aspects of the control layer (CL) part of the B&O AV Architecture. The CL is the top layer which the user interacts with using a local control interface (hence the term *control layer*). The CL part of the AV Architecture has the following responsibilities:

- Responsible for device discovery and device management
- Responsible for service discovery and service management
- Responsible for connectivity between remote and local services
- Responsible for event propagation between local and remote services

The Device Discovery model represents the communication protocols between the announcement- and discovery processes in an IP network. Device Discovery (DD) refers to the process of identifying devices present on a network. The DD abstraction contains two logical mechanisms and processes:

- Device Announcement Mechanism (DAM)
- Device Discovery Mechanism (DDM (uses DAM))

The Device Discovery model has 3 main CML processes.

- The `SourceProduct_DD_SD_InterfaceProtocolView` process models the device announcement protocol for a B&O AV source product.
- The `TargetProduct_DD_SD_InterfaceProtocolView` process models the device discovery protocol for a B&O AV Target product.
- The `Beo_DD_SD_InterfaceProtocolViews` process models the AV SoS containing the product processes in an IP network.

The CML processes in the model are based on the Interface Protocol View (PDV) from the COMPASS Interface pattern D22.3 [PHP+13]. In the model each CML process represents one PDV and one PDV represent an AV Control Layer product role (source- or target role D21.4 [HPH+13]). In the model both product roles can be powered on, or off by events on their control layer channel. The "`SourceProductPowerchannel`" and "`TargetProductPowerchannel`" in the PDVs represent these channels. The Source products SysML PDV state machine diagram, which the CML PDV are translated from is showed in Figure 2. In the CML PDVs SysML states are translated to CML actions post-fixed with state in their action name.

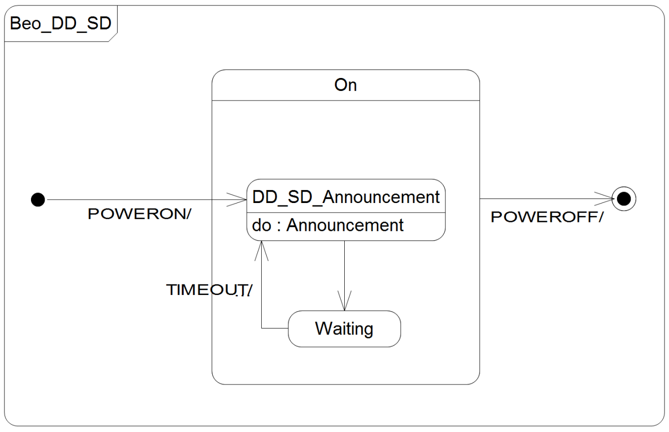

Figure 2: Source product SysML state machine

In the model products communicate using the `IPMulticastChannel` channel. The `IPMulticastChannel` represents the IP transport layer logic. In the PDVs we simulate the asynchronous behavior of the IP multicast logic by using CML timeouts for breaking channel synchronization between the processes. Note that at any time during announcement and query operations states, the products can be interrupted by power-off events on their control channels. The interruptible parts of the CML model are translated from SysML activity diagrams. The activity diagram in Figure 3 shows a refinement of the announcement operation in the `DD_SD_Announcement_State` for the AV source product protocol.

The state machine for the target product is equal to the source products state machine. The only difference is that the target product performs a discovery operation in the power on state, there the source product performs a announce operation in the power on state.

A detailed description of the AV Control layer, and the pattern level translation meta-model used for translating SysML based views to CML models and CML semantics can be found in D21.4. The Device Discovery model presented here is part of set of Control layer models. Descriptions of the other CL models can also be found in D21.4.

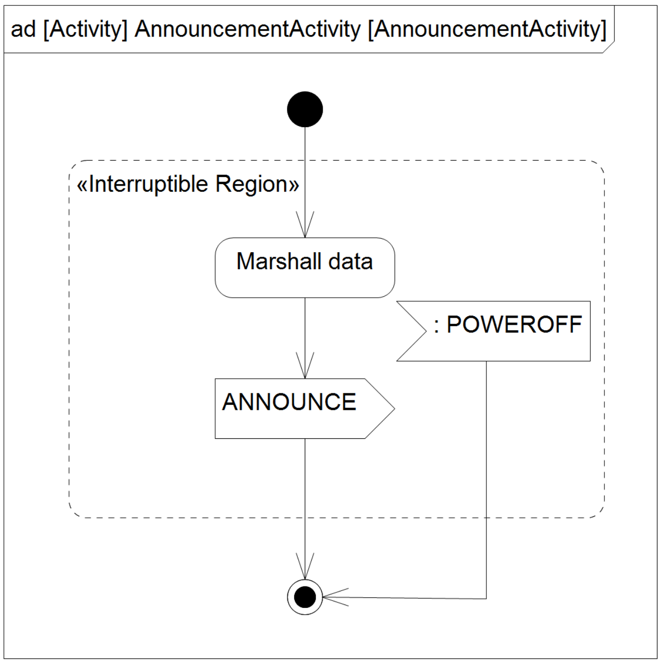

Figure 3: Refinement of the announcement operation

10.2 Current Model limitations and Bugs

The target product model does not simulate the device management consistency part of the DD protocol. The device management will be added later.

The processes control statements do not used the VDM `pre_funtion` option. This is currently not working correct in the tool. For now the model use Boolean helper functions for conditional checks.

The IP transport layer should be modelled as a separate process, enabling product scalability of the SoS.

The model is very CSP heavy. Next version should be more CML pure.

Currently inspection of variables values is not working on windows

10.3 Analysing using the Debugger

The Device Discovery model contains some test processes you can perform. Launch the `Beo_DD_SD_InterfaceProtocolViews` process.

The first thing you need to do, is to power on the products in the network Click on the `SourceProductPowerChannel.<POWER_ON>` and `TargetProduct-PowerChannel.<POWER_ON>` events in the debuggers event option windows.

Now both products start their announcement- and discovery logic. You now have following options:

- To simulate a multicast time-out, click `tock` twice for the process you wish to time-out.
- To let the target product discover the source product, click the `IPMulticastChannel.2` options in the CML event option window. You can verify this by looking at the values in the `product_DRs` set in the `TargetProduct_DD_SD_InterfaceProtocolView` process. The set should now contain the value 2.

Once the source product have been discover, if should not be discover again. Test this by adding a break point in the `updateProductSet` operation in the `TargetProduct_DD_SD_InterfaceProtocolView` process. This break point should never be hit after the product has been found once.

Try during execution of the models to interrupt the processes protocol actions by fire the `SourceProductPowerChannel.<POWER_OFF>` or `TargetProduct-PowerChannel.<POWER_OFF>` event.

Shut down the SoS debug by killing the Debugger process pressing the red box.

For getting a better understanding of the products protocol semantics, run the `SourceProduct_DD_SD_InterfaceProtocolView` or the `TargetProduct_-DD_SD_InterfaceProtocolView` process by themself. Click on the events offed by the CML event option window, and observe the simulated Device Discovery semantics.

You can also run the `Test_1` process and just click on the event offered to you in the CML event option window.

10.4 Analysing using Model Checking

Except for the processes using set manipulation, all other processes can be analysed in the model checker. Simplification were performed such as replacement of `mu` construct with explicit recursive processes, time information was removed and types had to be limited to a finite set. The analysed code is given as follows.

```
types
```

```
 Device_Record = nat -- unique device ID
  inv i == i in set {1,...,3}
 ServiceCallData = nat
  inv i == i in set {1,...,3}
 DeviceEvent = (<INTERUPT>|<POWER_ON>|<POWER_OFF>)
 TCP_PortEvent = (<CONNECT>|<DISCONNECT>)

  channels
--- channels for the DD and SD part of the SoS
 IPMulticastChannel:Device_Record
 SourceProductPowerChannel:DeviceEvent
 TargetProductPowerChannel:DeviceEvent

--- channels for the service part of the SoS
   SourceProductEventChannel:DeviceEvent
   TargetProductEventChannel:DeviceEvent
   BrowsingInterfaceCallChannel:ServiceCallData
   BrowsingInterfaceReplyChannel:ServiceCallData
   IPTCPChannel:ServiceCallData
   IPTCPPORTChannel:TCP_PortEvent
   LocalInterfaceChannel:ServiceCallData

  values
 timeout : nat = 1 -- generic IP timeout
 sourceBroadcastTime : nat = 2
 targetBroadcastTime : nat = 2
 ServiceCallDataERROR:ServiceCallData = 0

 functions
   IsDeviceInSet : Device_Record * (set of Device_Record) -> bool
   -- helper function
   IsDeviceInSet(d, s) == d in set s

  channels
--- channels for the DD and SD part of the SoS
 IPMulticastChannel:Device_Record
 SourceProductPowerChannel:DeviceEvent
 TargetProductPowerChannel:DeviceEvent

--- channels for the service part of the SoS
   SourceProductEventChannel:DeviceEvent
   TargetProductEventChannel:DeviceEvent
   BrowsingInterfaceCallChannel:ServiceCallData
   BrowsingInterfaceReplyChannel:ServiceCallData
   IPTCPChannel:ServiceCallData
   IPTCPPORTChannel:TCP_PortEvent
   LocalInterfaceChannel:ServiceCallData

  --chansets --CHANSET DO NOT WORK
```

```
-- CML process modelling the B&O DD protocol for
-- the source product in the network
  process SourceProduct_DD_SD_InterfaceProtocolView =
  begin
   state
   product_DR:Device_Record := 2 -- local device id
   operations
   createDDSDObject :()==>()
   createDDSDObject()== product_DR := 2
   -- just simulate updating the device record
   actions
 Off_State = SourceProductPowerChannel.<POWER_ON>->On_State
 On_State =
 DD_SD_Announcement_State /_\(SourceProductPowerChannel.<POWER_OFF>->Off_State)
 DD_SD_Announcement_State = (MarshallData_State;
DD_SD_Announcement_State)
 MarshallData_State = ( createDDSDObject();IPMulticastChannel!product_DR->Skip)
 [_> Skip
   @ Off_State
  end

-- CML process modelling the B&O DD protocol for
-- the target product in the network
  process TargetProduct_DD_SD_InterfaceProtocolView =
  begin
   state
   product_DRs:set of Device_Record := {} -- set of Ddiscoveed products

   operations
   resetProductSet:()==>()-- reset the device set then the product is power on
   resetProductSet() == product_DRs := {}

   updateProductSet:Device_Record ==> ()
   -- add device to the set, if the pre condition holds
   updateProductSet(dr)== product_DRs := product_DRs union {dr}


   actions
 Off_State = resetProductSet(); TargetProductPowerChannel.<POWER_ON>-> On_State
 On_State = Start_Discovery_State  /_\
 (TargetProductPowerChannel.<POWER_OFF> -> Off_State)
 Start_Discovery_State = DD_SD_Discovery_State ; Start_Discovery_State
 --run the Discovery logic
 DD_SD_Discovery_State =  IPMulticastChannel?pr ->
    (if IsDeviceInSet(pr,product_DRs) then updateProductSet(pr) else Skip; Skip)
[_> Skip
 @ Off_State      -- simulate IP multicast logic using timeouts
  end
```

```
-- SoS leveel process, contaning one source- and one target product
  process Beo_DD_SD_InterfaceProtocolViews =
 SourceProduct_DD_SD_InterfaceProtocolView
 [|{IPMulticastChannel}|]
 TargetProduct_DD_SD_InterfaceProtocolView

  --- test process for playing the SoS level process
  process Test_TurnOnProduct =
  begin
   actions
 SoS_On = SourceProductPowerChannel.<POWER_ON>->
 TargetProductPowerChannel.<POWER_ON> -> SoS_Off
 SoS_Off = SourceProductPowerChannel.<POWER_OFF>->
 TargetProductPowerChannel.<POWER_OFF> -> Skip

  @ SoS_On
  end
  -- test process combining the test process and the SoS process
  process Test_1 = Test_TurnOnProduct
   [|{SourceProductPowerChannel,TargetProductPowerChannel}|]
    Beo_DD_SD_InterfaceProtocolViews


-------------------------------------------------------------------------
-- section contain the processes for calling AV services over the network,
-- they are currently not described in the case doc

  process TargetProduct_SR_InterfaceProtocolView =
  begin
   actions
 Off_State = (TargetProductEventChannel.<POWER_ON>->On_State)
 On_State = WaitingOnFunctionCall_State /_\
 TargetProductEventChannel.<POWER_OFF> -> Off_State
 WaitingOnFunctionCall_State = BrowsingInterfaceCallChannel?x ->
 ConnectToDevice_State(x)
 ConnectToDevice_State= x:ServiceCallData @
(IPTCPPORTChannel.<CONNECT> ->
   SendingBrowsingRequest_State(x))
  [_> BrowsingInterfaceReplyChannel!ServiceCallDataERROR ->
   WaitingOnFunctionCall_State
 SendingBrowsingRequest_State = x:ServiceCallData @ SendCallData_State(x)
 /_\ ( IPTCPPORTChannel.<DISCONNECT> ->
   BrowsingInterfaceReplyChannel!ServiceCallDataERROR ->
   WaitingOnFunctionCall_State
    []
    TargetProductEventChannel.<INTERUPT> ->
    WaitingOnFunctionCall_State)
 SendCallData_State = x:ServiceCallData @ ( IPTCPChannel!x -> IPTCPChannel?y ->
  ReplayTofunctionCall_State(y))
 [_> BrowsingInterfaceReplyChannel!ServiceCallDataERROR ->
 WaitingOnFunctionCall_State
 ReplayTofunctionCall_State = x:ServiceCallData @
 BrowsingInterfaceReplyChannel!x ->
    WaitingOnFunctionCall_State
   @ Off_State
  end

  process SourceProduct_SR_InterfaceProtocolView =
  begin
   actions
 Off_State = SourceProductEventChannel.<POWER_ON>->On_State
 On_State = WaitingOnServiceCall_State /_\
 SourceProductEventChannel.<POWER_OFF>->
 Off_State
 WaitingOnServiceCall_State = IPTCPPORTChannel.<CONNECT>->ReadCallData_State
 ReadCallData_State = ReadCallData /_\ (SourceProductEventChannel.<INTERUPT> ->
   WaitingOnServiceCall_State [] IPTCPPORTChannel.<DISCONNECT> ->
   WaitingOnServiceCall_State)
 ReadCallData = (IPTCPChannel?x->LocalInterfaceChannel!x->
 LocalInterfaceChannel?y->
 IPTCPChannel!y->WaitingOnServiceCall_State) [_> WaitingOnServiceCall_State
   @ Off_State
  end

  process Beo_RS_InterfaceProtocolViews =
TargetProduct_SR_InterfaceProtocolView
  [|{IPTCPChannel,IPTCPPORTChannel}|]  SourceProduct_SR_InterfaceProtocolView
```

11 The Leadership Election Example

11.1 Case Description

A Bang & Olufsen (B&O) home Audio/Video (AV) network consists of several devices (such as audio, video, gateway and legacy audio products) which may be produced by competing manufacturers and distributed across a user's home. Such a network is an SoS: it exhibits the dimensions typical of an SoS as described in (Fitzgerald et al. 2013). The individual CSs exhibit a (potentially) wide variation in autonomy. They all operate at the behest of the user, but the fact that they may be legacy or non-B&O systems means that they may only offer a limited degree of controllability from the point of view of the SoS. The CSs exhibit operational independence; they provide stand-alone streaming or content browsing experiences, e.g. watching TV or selecting music to play. The CSs are typically distributed in different zones/rooms, the AV content can be local or remote, and the location of content source is often transparent to the user. Geographical distribution leads to emergent behaviors such as making sound follow the user around, driven by contracts between streaming and clock systems. The CSs undergo evolutionary development. The stakeholders will have an evolution vision that is not necessarily compliant with that of B&O. There is dynamic reconfiguration behavior in that products join or leave the SoS during streaming or browsing operations; products can be turned off by users or enter power-saving mode. While products have no interdependence, CSs rely on each other in order to deliver the emergent behavior that fulfills the SoS goal.

Constituent systems may join or leave the network at any time, but a consistent user experience (such as a playlist, current song, etc.) must be provided, and this requires availability and consistency of the system configuration data. In order to do this, a publish/subscribe architecture is employed. This in turn, requires that the underlying network is able to elect a leader from among the CSs. As there is no centralized control, the ability to elect a leader is a required emergent property of the SoS.

The leadership problem is a distributed consensus problem in a network with unreliable processes and asynchronous communication. When the CSs of the network are in an election state, no publisher is present and the multi-room experience space is inconsistent and unavailable. During the election, the devices must react to a set of local transition rules that will guarantee the desired emergent property of a leader in the network, and allow the network to enter the publisher-subscriber state.

The Emergent Leadership model has 3 main CML processes.

- The *Node* process models the LE devices behaviour in the network.
- The *Transport layer* process models the network layer and network protocols behaviour.
- The *SoS_Election* process models the SoS level behaviour by combining a set of node processes and the transport layer process.

Figure 4 present the interface connectivity view (ICV) for a simple SoS consisting of only two LE Devices (Node(s)) and a single Transport Layer; a larger SoS would have multiple LE Devices connected to the single Transport Layer. The

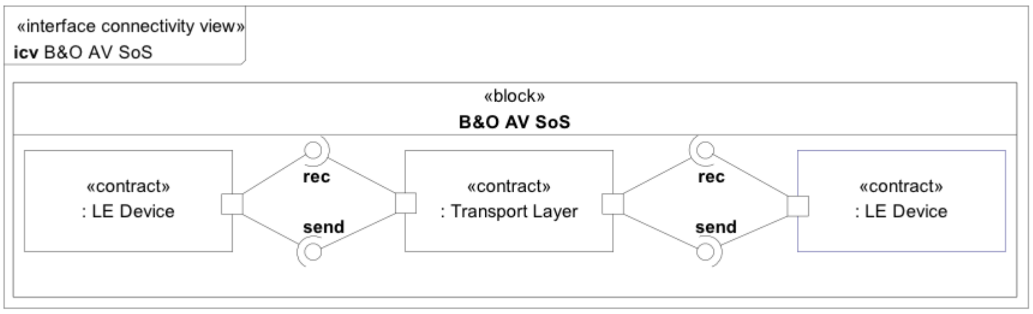

Figure 4: Interface connectivity view for B&O AV SoS

view shows that there are only two interfaces between these contractual specifications: rec and send. The rec interface is provided by the LE Device and required by the Transport Layer, whilst the send interface is provided by the Transport Layer and required by the LE Device. The interface naming has been given from the perspective of the LE Device.

The Interface protocol in Figure 5 shows the state machine for the ports of the transport layer process.

Figure 6 presents a contract definition view, which provides more details of the LE Device contract. The diagram includes the (private) operations and values of the contractual specification, along with the public operation provided by the rec interface. These operations are used to perform two main functions: to receive and process information from other LE Devices, and to use this processed information to make claims about the devices leadership status. The device transmits information about its claim and the strength of that claim to the Transport Layer through the Transport Layers send operation.

Figure 7 present the contract protocol view for the LE Device contract. This

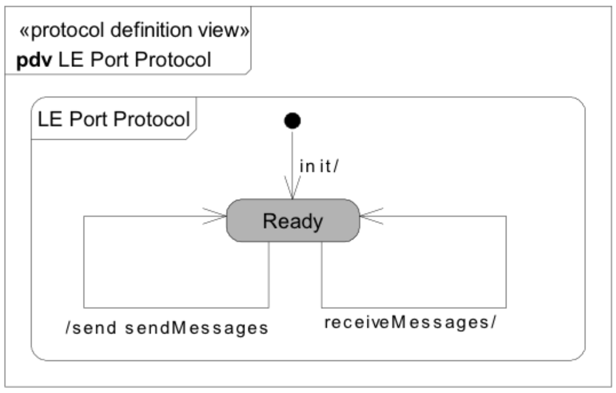

Figure 5: Protocol view for LE Port in B&O AV SoS

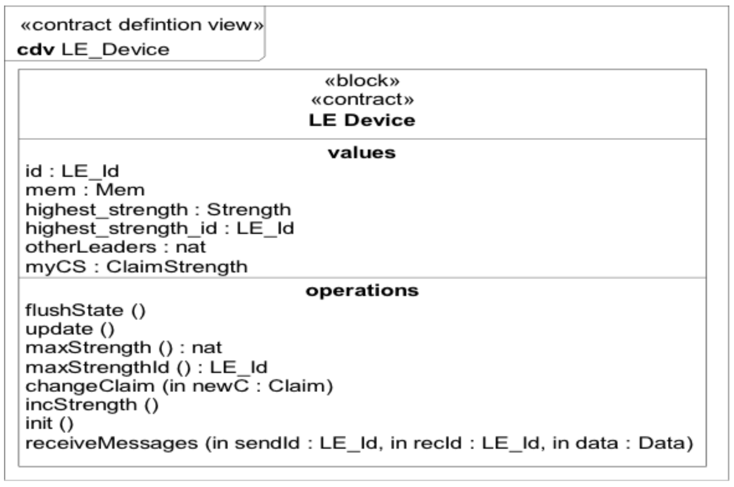

Figure 6: Contract definition view depicting LE Device contract specification

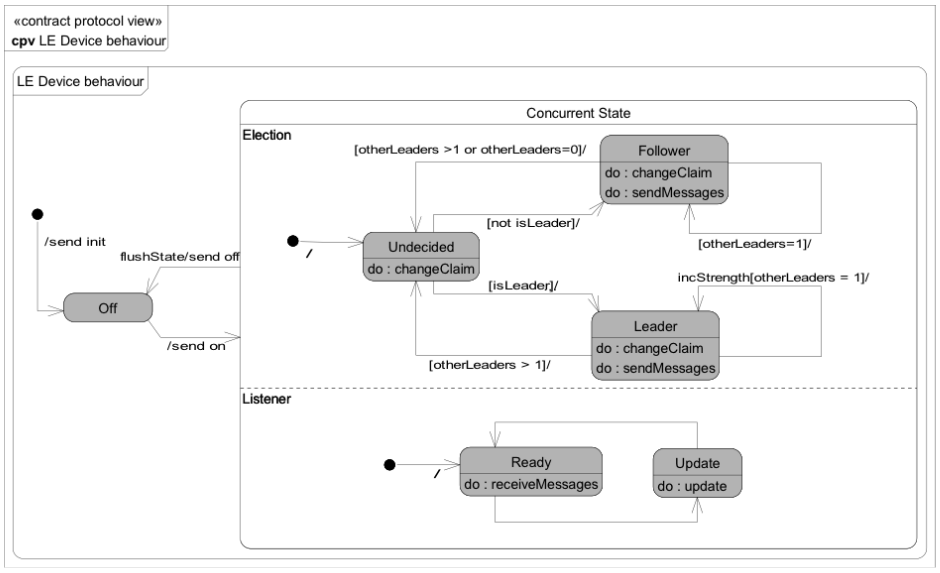

Figure 7: Contract definition view depicting LE Device contract specification

diagram allows us to describe the ordering of operation calls performed by an LE Device. Using this diagram, we describe the states the device may enter (*Leader*, *Follower* or *Undecided*).

From the diagram, we see that after initialization, a LE Device enters the Off state. From here, it may be turned on and it enters a parallel state with two sub-states: the *Listener* and *Election* states. In the *Listener* state, the LE Device repeatedly receives messages from its environment and performs the update operation. In the *Election* state, the LE Device initially enters the *Undecided* state, where it updates its claim to be undecided. After some time, the LE Device queries its updated attributes (these are updated in the *Listener* state), and decides if it can be a leader on the network.

If the LE Device can be a leader, it enters the Leader state, updates its claim and sends a message to all other devices it knows about. From this state, the LE Device continuously checks if it may be a leader. If, after becoming a leader, it receives a message from an existing leader, the LE Device enters the *Undecided* state. If the LE Device may remain a leader, then it increases its strength of the claim to be a leader and again sends a message to all other devices. Allowing successful leaders to increase the strength of their claim is a heuristic to increase the likelihood of successful leaders remaining as leaders, and thereby reduce the number of new elections.

If the LE Device is not a leader, it enters the *Follower* state. Once in this state, the

LE Device updates its claim to be a follower sends a message to all other devices. Once a LE Device is a follower, it remains in the *Follower* state unless either there are no leaders, or there is another LE Device claiming to be leader.

11.2 Current Model limitations and errors

The predicate logic regarding strength calculations need to be improved.

The model has two version of the *amLeader* operation; this is because of a bug in the interpreter, which currently not allows an action calling the same operations twice.

Remove the debug events.

11.2.1 Analysing using the Debugger

The following debug steps cover are analyzing these leadership specifications

- Node joins empty network
  - Excepted result
    - The node becomes the leader
- Node joins network which has a leader
  - Excepted result
    - The newly joined node becomes a follower.
- Leader leaves network
  - Excepted result
    - The follower becomes leader.

The process you want to run in the debugger mode is the *SoS_Election* process. Lunch the process from the process option window ,this will take you to the debug view First you need to *init* the network layer process and the node processes. Click init in the event option window. The next steps will test the *Node joins empty network* specification against the model.

The event option window will now give you the option to turn the nodes in the network on. Click on node 0, this will result in node 0 enter the network in the

Undecided state. In this state the node will to see if where a leader in the network is.

Node 0 is communication with the transport layer, but since where are no other node in the network the communication will time out, simulate that be clicking on the tock event in the event option window 4 times.

In the event option window you will now see some "debug" info, saying the `highest_is.0.1`, meaning node 0 with strength 1 is the leader. Click on this and move on. You will now get another debug event saying `leaderClaim.0.true`, this meant the node 0 has the leadership claim. Just click on the event and move on. Now you can see that node 0 is broadcasting to the network it's leader ship claim. The event `n_send.0.1.mk_CS(<leader>,0)` means node 0 is sending to node 1, that node 0 is the leader with strength 1. We have now finished the steps for the Node joins empty network specification. Now let do the steps for the Node joins network which has a leader specification.

Turn on node 1 from the event option window. Node 1 is now in the *Undecided* state and will read from the network. Press the events in this order.

- `n_send.0.1.mk_CS(<leader>, 0)`, means node 0 sends to node 1
- `n_rec.1.0.mk_CS(<leader>, 0)`, means node 1 receive data from node 0
- `n_send.0.2.mk_CS(<leader>, 0)`,node o try to send to node 2
- `n_rec.0.2.10`, node 0 gets a time-out from the transport layer, trying to communicate with node 2

That happens now is that node 1 is a follower and node 0 is still the leader. The follower is now listening to the leader, and the leader is broadcasting its leader claim. The leader it also listening to the network, so the first event options you gets after the communication trances showed above are the leader (node 0) trying to read from the network. So you need to time him out, since nobody else is sending. Do that by clicking tock 4 times.

Now you will get a set of debug events, saying how is the leader. Click the debug event in this order.

- `highest_is.0.1`
- `highest_is.1.0`
- `leaderClaim.0.true`,means node 0 say am the leader
- `leaderClaim.1.false`, means node 1 say am not the leader

So now we have a leader and a follower, we can see node 0 is sending its leadership claim to the network by offing these events

- `n_send.0.1.mk_CS(<leader>, 0)`, means node 0 sends to node 1
- `n_send.0.2.mk_CS(<leader>, 0)`, node o try to send to node 2

We have now completed the steps for *Node joins network which has a leader specification*, so let us do the steps for the *Leader leaves network* specification.

To get the leader to leave the network click on the `off.0` event in the event option window. The follower node 1 will now discover the leader is gone and go to the *Undecided* state. In this state the node will read from the network, to see determine how should be the leader since nobody else is sending data, timeout node 1 by clicking on tock 4 times. You will now get the following debug event options. Click then in the order

- `highest_is.1.0`, mean node 1 is the leader
- `leaderClaim.1.true`

You will now see that Node 1 is broadcasting its leader claim to the network via these events

- `n_send.1.0.mk_CS(<leader>, 0)`
- `n_send.1.2.mk_CS(<leader>, 0)]`

This completed the steps for the *Leader leaves network* specification and whereby this example, close down the SoS by clicking on the *deInit* event. In the model where are some automatic test you can run. The *TestLeaderNode* process is a test case for the *Node joins empty network* specification. The process contains a set of legal trances, witch test the *SoS_election* process. Try running the test process in the simulate mode, and then set a breakpoint in the *sendCS* action. If you now look in the variable view for node 0's call stack. You can see that node 0 has no leaders, meaning node 0 is the leader.

11.3 The CML model

```
-- CML model of the B&O AV Leadership Election
-- Version 1.1
-- Authors: Jeremy Bryans, Klaus Kristensen and Richard Payne
--
-- This version is maintained for the INCOSE 2014 paper

-- modified for version 0.1.4 of toolset
```

```
--
-- node_ids has been changed
-- a minor syntactic error in the AddToQ operation has been fixed.

-- version 0.1.6: works ok

-- circular references in the definitions of NODE_ID
-- and STRENGTH have been fixed,
-- by changing the type of node_ids to a set of nat,
-- and changing llp, ulp to type nat

/*================================================
   THINGS TO DO
   1. Deal with on/off properly:
     The TL should not hear the "on" and "off" events.
   2. get rid of maxset.  This might require composite types

   THINGS TO CONSIDER
   1. Should isleader : bool be a part of summary variables state?
   It would make the Undecided action consistent with the Leader
   and the Follower actions.

   QUESTIONS
   1. Why does the model "synch up", so even if nodes are turned on at
   different times, they all end up reaching the same opint at the same time?
   ANS: because the nodes hear from all other nodes before the 4 second timeout,
   and can get on with the next step without needing to timeout to get there.

============================================================*/

-- ver 0.1.9 snapshot 2013-09-22 ok
-- ver 0.2 ok
-- ver hack by KRT 2013-10-10 time DK 13.05
-- ver further hack JWB 2013-10-14
-- ver further hack JWB 2013-10-23
-- ver by KRT 2013-11-17 time DK 14.31 -- minor changes to the design


types

-- NODE_IDs are natural numbers
  NODE_ID = nat
 inv n == n in set node_ids

  CLAIM = <leader>|<follower>|<undecided>|<off>

  String = seq of char

-- a message from the Transport Layer
```

```
  TL_MSG = nat

  DATA = CS|TL_MSG

-- a claim/strength record
  CS :: c : CLAIM
  s : STRENGTH

-- JWB: what happens if llp > ulp?
   STRENGTH = nat
 inv p == llp <= p and p <= ulp

-- Known only to the Transport Layer

  MSG :: src : NODE_ID
   des : NODE_ID
   pl  : DATA

 values

  node_ids : set of nat = {0,...,2} -- the set of node identifiers

  ulp : nat = 10   --upper limit of strength values

  llp : nat = 0 --lower limit of strength values

  n_timeout : nat = 4  -- the timeout value employed by the Nodes

  tl_timeout : nat = 3

  unreachable : TL_MSG = 10-- timeout message from Transport Layer

functions

  -- this function could easily be made redundant by allowing actions
  -- to use mk_MSG()
  createMSG: NODE_ID * NODE_ID * DATA -> MSG
  createMSG(f,t,p) ==
 mk_MSG(f,t,p)

  -- returns the second parameter if the first is an empty set
  -- returns the second parameter if it is larger than the max of the first
  maxSet: set of nat * nat -> nat
  maxSet(sn,m) ==
  (
   cases sn:
  ({}) -> m,
  others -> (let v = select(sn) in
    (if v > m then maxSet(sn\{v},v)
  else maxSet(sn\{v},m))
    )
 end
  )

  -- helper function for maxSet; selects an arbitrary element of a non-empty set
  select(sn : set of nat) r: nat
  pre sn <> {}
  post r in set sn

  -- selects an arbitrary NODE_ID from a non-empty set
  randomId(ids: set of NODE_ID) r: NODE_ID
  pre ids <> {}
  post r in set ids

  isNodeOn: map NODE_ID to bool * NODE_ID ->bool
  isNodeOn(m,id)== m(id)= true
  pre id in set dom m

channels

   highest_is : NODE_ID * NODE_ID
   leaderClaim : NODE_ID * bool
   init    -- to initialise the LE SoS
   deInit -- to deinitialise the LE SoS
   on, off : NODE_ID   -- to turn nodes off and on


  -- channels for the Node
  -- Observe snd and take have differing payload types CS and DATA
  -- the sending node does not include its own identity:
  -- this is added at the integration stage

  --  --------  snd take   ---------------
  --  -    - ------> ------> - -
  --  - node - - Trans Layer -
  --  -    -  rec give    - -
  --  -    - <----- <------ - -
  --  -------- ---------------


  n_send : NODE_ID * NODE_ID * DATA    -- from * to * payload
  n_rec : NODE_ID * NODE_ID * DATA    -- to * from * payload

process Node = i : nat @
begin
  state
  --  State is divided into immutable, volatile and summary memory.
 id : NODE_ID := i   -- the ID of the node: immutable state

 -- mem is volatile memory
 mem: map NODE_ID to CS :=
   { cid |-> mk_CS(<off>, 0) | cid in set node_ids \ {id} }
   -- the map of node ids to their CSs,
using mk_CS(<off>, 0) instead of nil
  inv dom mem = node_ids  \ {id} and
  -- the domain of mem must not include id (the ID of the node)
   dom mem <> {}
   -- must be more than one node in the network (although this model
          -- should work if there is only one node)

 -- below are summary variables
 highest_strength : STRENGTH := 0
-- the highest strength of leadership claim node is aware of

 highest_strength_id : NODE_ID := 0
-- id of node making the leadership claim with the highest strength
                -- this is allowed to be any node id, including id itself
 inv highest_strength_id in set (dom mem union {id})

 leaders : nat := 0          -- the leader count among my neighbours
 inv leaders <= card dom mem

 myCS : CS := mk_CS(<off>, 0)          -- my claim/strength pair

 myNeighbours : seq of NODE_ID := [i| i in set dom mem @ i <>id]


operations

 Init: () ==> ()
 Init() ==
 (
   id := i;
   -- initialise id with the paramater passed to Node at its instantiation
   flushState() -- flush volatile and summary memory
 )

 flushState: () ==> ()
 flushState() ==
 (
   mem := { cid |-> mk_CS(<off>, 0) | cid in set node_ids \ {id} };
   highest_strength := 0;
   highest_strength_id := if id=0 then 1 else 0;
   -- krt. need to make this generic for all nodes,
   -- the real impl use mac adress values
   leaders := 0;
   myCS := mk_CS(<off>, 0)
 )

 -- used by controller to write a CS to a memory cell


 write: NODE_ID * DATA ==> ()
 write(i,dat) ==
 (
   if is_TL_MSG(dat) then mem(i) := mk_CS(<off>,0) else mem(i) := dat
   -- tmp impl, currently the IDE do not support case statements

   /*cases dat:
    TL_MSG  -> mem(i) := mk_CS(<off>,0),
    -- if we receive a message from the TL we record that the node is off
    CS    -> mem(i) := dat      -- if a claim/strength pair we add it to memory
   end*/
 )
 pre i in set dom mem
 post mem(i) = dat or mem(i).c = <off>

 update:()==>()
 update() ==
 (
   leaders := card{n|n in set dom mem @ mem(n).c = <leader>};
   highest_strength := maxStrength();
   -- highest strength among leadership claims; 0 is no leadership claims.
   highest_strength_id := maxStrengthID()
   -- id of highest strength among the leadership claims,
   -- 0 or 1 if no leadership claims
 )
  post leaders > 0 => mem(highest_strength_id).s = highest_strength

 -- this only returns the maximum strength among the leadership claims
 maxStrength:() ==> nat
 maxStrength() ==
 (
   dcl strs : set of nat := {cs.s|cs in set rng mem @ cs.c = <leader>} @
return maxSet(strs,0)
 )

   -- return the ID of the leader with the max strength;
   -- minId if there are no leaders
 maxStrengthID : () ==> NODE_ID
 maxStrengthID() ==
 (
   dcl minId : NODE_ID, maxStrIds : set of NODE_ID @
  (
   if id = 0
   then minId := 1
   else minId := 0;
   maxStrIds :=
   {n | n in set dom mem @ mem(n).s = highest_strength and mem(n).c = <leader>};
   return maxSet(maxStrIds,minId) -- minId returned if maxStrIds = {}
  )
 )

 -- update my claim
 -- note that we can enforce state transitions through the changeClaim operation
 -- since Claim corresponds directly to the states in the ST diagram
 changeClaim: CLAIM ==> ()
 changeClaim(newc) ==
 (
   dcl currStr : STRENGTH := myCS.s @
   myCS := mk_CS(newc, currStr)
 )
 pre myCS.c = <off> => newc = <undecided> and
  myCS.c = <undecided> => newc = <leader> or newc = <follower> and
  myCS.c = <leader> => newc = <undecided> and
  myCS.c = <follower> => newc = <undecided>

 -- increase the strength of my claim up to the maximum
 -- (upper limit of petitions: ulp)
 incStrength:()==>()
 incStrength() ==
 (
   if myCS.s < ulp
   then myCS := mk_CS(myCS.c, myCS.s+1)
 )
 pre myCS.s < ulp
 post myCS.s = myCS~.s + 1

 amILeader: () ==> bool
 amILeader() ==
 (
   return (leaders = 0)  or
   -- krt: tmp outcomments this, not sure this it correct accordning to the real sp
    highest_strength < myCS.s --or
   -- (highest_strength = myCS.s and highest_strength_id < id)
   -- this fails when a new node join the network,
   -- before a full round for the leader node
 )

 amILeader2: () ==> bool
 amILeader2() ==
 (
   return (leaders = 0)  or
    highest_strength < myCS.s --or
```

```
    --(highest_strength = myCS.s and highest_strength_id < id)
 )

 whoIsHighest: () ==> NODE_ID
 whoIsHighest()==
 (
  return highest_strength_id
 )

actions

 Off = on!id->JoinNetwork --  device is not power on state

 JoinNetwork = Undecided /_\ (off!id -> flushState();Off)
-- the node can be interupted by a power of event

 Controller = ReceiveData;UpdateData;OutputData --update state values

 ReceiveData = (n_rec!id?s?dat ->write(s,dat);ReceiveData) [_ n_timeout _> Skip
 -- receive data from the transport layer, or timeouts

--  UpdateData outputs leader claim
 UpdateData = update(); Skip

 OutputData =
   (dcl isleader:bool @  isleader := amILeader();
  (dcl h2 : NODE_ID @
    h2 := whoIsHighest(); (highest_is!id!h2 -> leaderClaim!id!isleader -> Skip)
   )
   )

   SendCS =  (||| t in set dom mem @ [{}] n_send.id.(t).(myCS) -> Skip)
   -- this insists on sending all messages before it progresses.

   Undecided = changeClaim(<undecided>);Controller;
   -- Undecided state, valid for newly joning nodes, or during elections
      (
    dcl isleader:bool @  isleader := amILeader2() ;
  (
   [isleader]& Leader
      []
      [ not isleader] & Follower
     )
    )

 Leader = changeClaim(<leader>);SendCS;Controller; -- leader state,
    (
     [leaders > 0] & Undecided
     []
     [leaders = 0] & incStrength();Leader
    )

 Follower = changeClaim(<follower>);flushState(); Controller;
 -- krt: a follower do not send status out, followers are
 -- only listing on the leader
      (            -- add logic for hadndling the leader turning of
      -- flush the current state values before updating
    [leaders >1 or leaders=0 ] & Undecided
    []
    [leaders = 1] & Follower
      )

   @ init -> Init();(Off/_\deInit->Skip)
   -- main entry point for a node, need the deInit channel for
   -- testing deadlock/livelock freedom

end

-- process representing the IP transport layer,
-- simulate the multicast logic using in the B&O SoS
process TransportLayer =
begin

state
  nodeOn : map NODE_ID to bool := {n |-> false | n in set node_ids}
   inv dom nodeOn = node_ids and card(dom nodeOn) >= 2

 -- the queue of messages in the TL
 queue : seq of MSG := []

operations

 Init:() ==> ()
 Init() ==
 (
 nodeOn := {n |-> false | n in set node_ids};
   queue := []
 )
 post dom nodeOn = node_ids and
   rng nodeOn = {false} and
   queue = []

 -- JWB: added an inv to say that the rest of the queue hasn't changed
 addToQ: MSG ==> ()
 addToQ(m) ==
 (
   queue := queue ^ [m]
 )
 post len queue = len queue~ + 1 and
   queue(len queue) = m and
   queue~ = queue(1,...,len queue - 1)

 -- JWB: added an inv to say that the rest of the queue hasn't changed
 getNext:() ==> MSG
 getNext() ==
 (
   let h = hd(queue) in
   (
  queue := tl(queue);
  return h
   )
 )
 pre queue <> []
 post len queue = len queue~ - 1 and
  queue = tl(queue~)

 setNodeOff:(NODE_ID) ==> ()
 setNodeOff(nId) ==
 (
   nodeOn(nId) := false
 )
 pre nId in set dom nodeOn
 post nodeOn(nId) = false

 setNodeOn:(NODE_ID) ==> ()
 setNodeOn(nId) ==
 (
   nodeOn(nId) := true
 )
 pre nId in set dom nodeOn
 post nodeOn(nId) = true

 getQueue:() ==> seq of MSG
 getQueue() == (return queue)

actions

 TransportLayerLoop = mu X @ (TransportLayer;X)

 TransportLayer = (Reader [] Writer [] NodeMngt)

 Reader =  n_send?fr?too?payload ->
      (setNodeOn(fr);
       (dcl m:MSG @
        m := createMSG(fr,too,payload); addToQ(m); Skip)
       )
 Writer = [queue <> [] ] &
     (dcl m:MSG, nodeIsOn: bool @ m := getNext();
     nodeIsOn := isNodeOn(nodeOn,m.des);
      (
    [nodeIsOn] &
      (
       (n_rec!(m.des)!(m.src)!(m.pl) -> Skip)
       [_ tl_timeout _> setNodeOff(m.des);
     (if not (m.pl=unreachable) then
     (dcl rep : MSG @ rep := createMSG((m.des),(m.src),unreachable);
     addToQ(rep))); Skip
      )
    []
    [not nodeIsOn] &
       (
     (if not (m.pl=unreachable) then
     (dcl rep : MSG @ rep := createMSG((m.des),(m.src),unreachable);
     addToQ(rep))); Skip
       )
      )
     )


 NodeMngt = on?i -> setNodeOn(i);Skip
      []
      off?i -> setNodeOff(i);Skip

  @  init -> Init();(TransportLayerLoop /_\deInit->Skip)

end

-- process with all nodes
process AllNodes = [|{|init,deInit|}|]
i in set node_ids @  (Node(i))

-- SoS process combining the nodes and the network layer
process SoS_Election =
AllNodes[|{|n_send,n_rec,on,off,init,deInit|}|]TransportLayer

--process for testing the hidding operator
-- currently not working
process ElectionHiddenComms =
AllNodes[|{|n_send,n_rec,on,off,init|}|]TransportLayer
\\ {|n_rec,n_send|}

----------------------------------------------------------------------------------------------------
-- test section

process RunTest = Test [|{|n_send,n_rec,on,off,init|}|] SoS_Election
```

```
-- legal traces for election node validation
process Test =
begin
actions
  Test1 = init -> Stop
  Test2 = init-> on.0-> highest_is.(0).(1)->leaderClaim.(0).(true)->Step2
  Step2 = (dcl c:CS := mk_CS(<leader>, 0)@ n_send.(0).(1).(c)->
  n_send.(0).(2).(c)->off.0->deInit->Skip)
@ Test2
end

-- test SoS for validation of test traces
process TestLeaderNode =
Test [|{|n_send,n_rec,on,off,init,highest_is,leaderClaim,deInit|}|] SoS_Election

-- legal traces for validation of node joining leader network
process TestNodejoinLeaderNetwork =
begin
actions
 FirstNodeTurnOn = init->on.0->highest_is.0.1->
 leaderClaim.(0).(true)->on.1-> n_send.(0).(1).(mk_CS(<leader>, 0))->
 n_send.(0).(2).(mk_CS(<leader>,0))-> n_rec.(1).(0).(mk_CS(<leader>,0))->
 n_rec.(0).(2).(10)-> highest_is.(0).(1)-> highest_is.(1).(0)->
 leaderClaim.(0).(true)-> leaderClaim.(1).(false)->
 n_send.(0).(1).(mk_CS(<leader>, 1))-> n_rec.(1).(0).(mk_CS(<leader>,1))->
 n_send.(0).(2).(mk_CS(<leader>, 1))-> n_rec.(0).(2).(10)->highest_is.(0).(1)->
 highest_is.(1).(0)-> leaderClaim.(0).(true)-> leaderClaim.(1).(false)-> off.0 ->
 highest_is.(1).(0)-> leaderClaim.(1).(true)-> highest_is.(1).(0)->
 leaderClaim.(1).(true)->deInit->Skip
   @FirstNodeTurnOn
end

-- test soS for validation of node joining leader network
process TestNodejoinLeaderNetworkElection =
TestNodejoinLeaderNetwork
[|{|n_send,n_rec,on,off,init,highest_is,leaderClaim,deInit|}|] SoS_Election


process ReplTest = [|{|init,deInit|}|]
i in set node_ids @  Node(i)
```

References

[BPS12] Jörg Brauer, Jan Peleska, and Uwe Schulze. Efficient and Trustworthy Tool Qualification for Model-Based Testing Tools. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems*, Lecture Notes in Computer Science, pages 8–23. Springer Berlin Heidelberg, 2012.

[CK04] Quentin Charatan and Aaron Kans. *From VDM to Java*. Palgrave, New York, 2004.

[HPH$^+$13] Jon Holt, Simon Perry, Finn Overgaard Hansen, Stefan Hallerstede, and Klaus Kristensen. Report on Guidelines for System Integration for SoS. Technical report, COMPASS Deliverable, D21.4, August 2013. Available at http://www.compass-research.eu/.

[Pel02] Jan Peleska. Formal Methods for Test Automation – Hard Real-Time Testing of Controllers for the Airbus Aircraft Family. In *Integrating Design and Process Technology, IDPT-2002*. Society for Design and Process Science, 2002.

[PHP$^+$13] Simon Perry, Jon Holt, Richard Payne, Claire Ingram, Alvaro Miyazawa, Finn Overgaard Hansen, Luís Diogo Couto, Stefan Hallerstede, Anders Kaels Malmos, Juliano Iyoda, Marcio Cornelio, and Jan Peleska. Report on Modelling Patterns for SoS Architectures. Technical report, COMPASS Deliverable, D22.3, February 2013. Available at http://www.compass-research.eu/.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical Monograph PRG-126, Oxford University Computing Laboratory, Jul. 2000.

[Win88] J.M. Wing. A Study of 12 Specifications of the Library Problem. *IEEE Software*, Juli 1988.

[WSC$^+$08] Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The Certification of the Mondex Electronic Purse to ITSEC Level E6. *Formal Aspects of Computing*, 20(1):5–19, 2008.