Analysis of starsense wireless components and their interaction using formal methods

de Vreeze, P.D.

Award date:
2014

Analysis of Starsense Wireless Components and their Interaction Using Formal Methods

Masters Thesis

P.D. de Vreeze
student#: 0766966
p.d.d.vreeze@student.tue.nl

Graduation supervisor
prof.dr.ir. J.F. Groote
j.f.groote@tue.nl

Philips supervisor
ir.ing. L.W.A. van der Wijst
berry.van.der.wijst@philips.com

July 4, 2014

Philips Lighting, Eindhoven
and
Department of Computer Science
Eindhoven University of Technology, Eindhoven

Contents

1 Introduction **5**
- 1.1 Related work . . . 6

2 mCRL2 **7**
- 2.1 mCRL2 Formal Specification Language . . . 7
  - 2.1.1 Processes and actions . . . 7
  - 2.1.2 Data types . . . 8
- 2.2 mCRL2 Toolset . . . 9

3 Introduction to the Starsense Wireless System **10**
- 3.1 Outdoor Luminance Controller . . . 10
- 3.2 Segment Controller . . . 11
- 3.3 CityTouch . . . 11

4 Software Upgrade Components **11**
- 4.1 Implementation . . . 11
  - 4.1.1 Outdoor Luminance Controller Software Upgrade component . . . 11
  - 4.1.2 Segment Controller Software Upgrade component . . . 13
- 4.2 Modeling . . . 18
  - 4.2.1 Simplifications . . . 18
  - 4.2.2 The OLC SWU model . . . 19
  - 4.2.3 The SC SWU model . . . 20
  - 4.2.4 The model of the interacting SC and OLC components . . . 22
- 4.3 Analysis . . . 22
  - 4.3.1 Requirements . . . 23
  - 4.3.2 OLCs not upgraded after message drop . . . 24
  - 4.3.3 Odd behaviour and recommendations . . . 27

5 Commissioning Components **30**
- 5.1 Implementation . . . 30
  - 5.1.1 Segment Controller Database . . . 31
  - 5.1.2 Segment Controller CommissioningStore component . . . 32
  - 5.1.3 Segment Controller Commissioning component . . . 32
  - 5.1.4 Outdoor Luminance Controller Commissioning component . . . 36
- 5.2 Modeling . . . 39
  - 5.2.1 Simplifications . . . 39
  - 5.2.2 The OLC Commissioning model . . . 39
  - 5.2.3 The SC Commissioning model . . . 40
  - 5.2.4 The model of the interacting SC and OLC components . . . 42
- 5.3 Analysis . . . 43
  - 5.3.1 Requirements on the OLC commissioning component . . . 43
  - 5.3.2 Requirements on the interacting commissioning components . . . 44

6 System and Toolkit **49**

7 Conclusion **50**

8 Appendix **52**
A Verify process flowchart 53
B Software Upgrade process mCRL2 specification 54
C Rename file for requirement: A started SWU process must always finish . . . 70
D Combined OLC commissioning component mCRL2 specification 72
E Interacting commissioning components mCRL2 specification 74
F Commissioning Shared mCRL2 specification 77
G Commissioning SC component mCRL2 specification 80
H Commissioning OLC component mCRL2 specification 93
I Rename file for requirement: An OLC may never be in an illegal state 99
J Rename file for requirement: An OLC cannot arrive in a state from which it cannot be commissioned or decommissioned again. 101

Abstract

The Starsense Wireless system consists of luminaire-based Outdoor Luminaire Controllers (OLCs) which all control a single outdoor lighting node, and a Segment Controller (SC) that controls the OLCs. The SC and OLCs are situated in a mesh network, where messages are routed through the network in a hopping fashion. Communication over this network is unreliable and can cause packets to be lost.
Certain components of the SC software are in charge of configuring the OLCs, two of these components are Software Upgrade (SWU) and Commissioning.

In this report, abstract models of these two components and their counterparts in the OLC software are specified in the formal specification language mCRL2. Using the mCRL2 Toolset, these components and the interaction between the SC and the OLCs of these components is analysed and certain properties are verified. During the analysis of the interacting SWU components, rare situations have been found where a single dropped message can cause OLCs not to be upgraded without external interference and rare situations have been found where unnecessary traffic is generated on the network.
The interacting SC and OLC Commissioning components proved not to behave entirely as designed, rare situations have been found where an instruction to commission or decommission an OLC could not always be fulfilled. Further analysis revealed the cause of the problems and the solutions appeared to be straightforward.

1 Introduction

Software in embedded systems is becoming increasingly more complex, the main cause of this complexity is due to the fact that current embedded systems consist of many communicating parallel components. For software architects it is hard to predict the complete behaviour of these systems and for software testers it is almost impossible to write all possible test cases to guarantee the system will always behave according to the specifications.

Philips developed a telemanagement system called Starsense [1] for monitoring, controlling, metering and diagnosing outdoor lighting. The Starsense Wireless system consists of luminaire-based Outdoor Luminaire Controllers (OLCs) which all control a single outdoor lighting node, and a Segment Controller (SC) that controls the OLCs. The SC and OLCs are situated in a mesh network, where messages are routed through the network in a hopping fashion. A network can consist of thousands of OLCs and communication over this network is unreliable and can cause packets to get lost.

The software of the SC and the OLC consists of multiple (parallel) components, where some SC and OLC components communicate with each other. Two of these components are Software Upgrade (SWU) and Commissioning.

The SWU component of the SC has the task to upgrade the software of the OLCs (all or a subset) in the network segment controlled by the SC. An update consists of a large number of packets (typically more than a thousand), where each packet needs to be sent to all OLCs. Since the network is unreliable, causing packets to get lost, some packets need to be sent multiple times. Each packet is sent with some seconds in between, therefore the whole SWU process can take several hours.
The Commissioning component of the SC has the task to contact and configure an uncommissioned OLC that needs to become part of the segment that is under control of the SC. Only a Philips manufactured and pre-configured OLC is allowed to join the segment.

This report describes the analysis of these components and the interaction between the SC and OLC. For each component the source code is studied of both the SC and OLC implementations. The components are written in the C programming language and consist of several thousand lines of code. Abstract models of these components are created in the process specification language mCRL2. Using the mCRL2 toolset requirements on the components are verified and analysis is performed, with the following questions in mind: Can the SWU process duration be reduced? Do situations exist where the software of an OLC cannot be upgraded? Do situations exist where an OLC cannot be commissioned?

In chapter 2 the mCRL2 language and toolset is introduced, chapter 3 introduces the Starsense Wireless system. In chapter 4 the implementation, model and analysis of the SWU components is discussed. Chapter 5 discusses the implementation, model and analysis of the Commissioning components. Finally, chapter 7 concludes this report.

1.1 Related work

The primary goal of this report is to create an abstract model out an already implemented distributed software system and analyse and verify the behavior, to determine whether the system contains errors. A great amount of case studies on the verification of distributed systems have been published. The general conclusion that can be derived from these case studies is that the use of formal methods improves the quality of the software. A small selection of these case studies are discussed below.

In [3] a wireless fire alarm system under design has been verified using UPPAAL and *Spin*. European certification tests have been formalised to serve as requirements. This study revealed previously undiscovered errors that caused the design of the system to be revised. The result of using formal methods during the design phase was that the first prototype already passed all tests, which substantially shortened the test phase. In [4] the use of formal methods during the development of an software bus is described, where mCRL2 is used for modeling and simulation and tested with the model test-tool JTorX. The authors claim that the use of formal methods took 17% of the total development time and that errors have been discovered during the model based testing, which would have been hard to find with conventional methods.

In [5] software development process of the Mars Rover Curiosity is described, specifically, how the risk of errors in the software is reduced to a minimum. The software consists of 350 MLOCs, which is more than all software of the previous mars rovers combined. The model checker *Spin* has been used to verify critical software subsystems. For the Chinese Lunar Rover, formal verification has been successful in discovering undesired behavior in the multitasking system of the control software [6], where the RTOS, the application tasks and the physical environment are modeled as timed automata and model checked using UPPAAL. The usage of formal methods for the Descent Guidance Control program of a Lunar Lander is described in [7], where techniques as simulation, bounded model checking and theorem proving are used.

Formal methods have been extensively used in railway systems. The B formal method has been used for the development of SACEM [8], a fault-tolerant railway signaling system for emergency break activation, speed control and driver signaling, which is used in the Subway of Paris and other cities. The report on the verification of the Paris automatic metro line 14 [9], shows that the B method has been effective in finding errors during specification and no further errors have been found after automatic code generation during the testing phase. A similar case study has been presented in [10], for the automatic shuttle line at the Roissy Charles de Gaulle Airport. No unit tests were performed for the Paris metro line or the Roissy shuttle line projects [11], only global tests were performed which were all successful. The absence of unit tests can result in a significant cost reduction.

In [2] a study has been conducted to determine the quality of software development where formal techniques have been used, measured by the number of errors per KLOC. This shows that, when formal methods are used in the design phase of the software development, then formal techniques could deliver higher quality code, reduction in number of errors and an increase in development productivity.

2 mCRL2

mCRL2 is a formal specification language and using the mCRL2 toolset concurrent systems and protocols can be modelled, validated and verified. For a detailed and extensive book regarding modeling and analysis of concurrent and distributed systems using the mCRL2 specification language refer to [15]. For more information about the mCRL2 language and toolset refer to [13, 14, 16].

2.1 mCRL2 Formal Specification Language

In this section a brief summary of the mCRL2 formal specification language is given, for a more extensive overview of the language refer to [13, 15, 16].

2.1.1 Processes and actions

The most important notions of the language are processes and actions. A process describes the behaviour of a system and is constructed using actions, where an action describes an elementary operation of the system. The following process P describes a system which can perform a single action, namely the action *light_on*.

P = *light_on*;

An action is declared in the following way:
act *a*,*b*;
c:*Bool*;
d:*Nat # Bool*;

Here action *a* and *b* are parameterless and action *c* and *d* contain parameters, where action *c* contains a single boolean parameter and action *b* contains two parameters, a natural number and a boolean.
Basic operators on process expressions are the following:

- Sequential composition: for example; P = *a* . *b*;
  Which states that process P performs an *a* action followed by an *b* action.
- Alternative composition: for example; P = *a* + *b*;
  This operator is also called the choice operator. The expression states that process P non-deterministically chooses to perform either action *a* or *b*.
- Multi-action: for example; P = *a* | *b*
  A multi-action specifies that multiple actions should occur at the same time, where a sequence of actions is separated by bars.
- Parallel composition: for example; P = *a* || *b*; Which is equivalent to P = (*a* . *b*) + (*a* . *b*) + *a* | *b*;
  Stating, process P can perform either *a* followed by *b* or *b* followed by *a* or action *a* and *b* can happen at the same time, indicated by the multiaction operator |.
  The . operator binds stronger than the + operator, so instead of writing P = (*a* . *b*) + (*a* . *b*) + *a* | *b*; we can write P = *a* . *b* + *a* . *b* + *a* | *b*;

- Recursion: for example; P = *a* . P;
  This specifies a process that performs *a* actions endlessly.
- Process with parameters: for example; P(*i*:*Nat*)= *output*(*i*). P(*i*+1);
  Process P initialised with some natural number will 'output' the number indicated by action *output* and add one to parameter *i*. For example, P(3) will present the following behaviour: *output*(3). *output*(4). *output*(5). *output*(6)...
- Conditional operator: for example; P(*b*:*Bool*)= (*b*) $\rightarrow$ *c* $\diamond$ *d*;
  Here the process P has the boolean parameter *b*. This process will perform action *c* if *b* validates to *true* or action *d* if *b* validates to *false*.
- Sum operator: for example; P= sum *b*:*Bool* . *a*(*b*);
  The sum operator instantiates the binding variable with all possible values of the data type. In this example the binding variable *b* of data type *Bool* is instantiated with values *true* and *false*. The sum operator is a generalisation of the choice operator. If the data type is finite then the sum can be unfolded using the choice operator. So this example can be rewritten to P= *a*(*false*) + *a*(*true*);

The initial state of a process is defined using the **init** keyword. The above explained process P(*i*:*Nat*) with parameter *i* can be initialised in the following way: **init** P(3);

2.1.2 Data types

mCRL2 contains a number of built-in data types, such as *Bool*, *Pos*, *Nat*, *Int*, *Real* representing the Booleans, positive numbers, natural numbers, integers and the real numbers, respectively. Other data types are structured types, with structured types the elements of a datatype can be explicitly characterized.

sort *State* = **struct** *Initializing*(*step*:*Pos*)?*IsInitializing* | *Idle*?*IsIdle* | *Busy*?*IsBusy*;

In the above example a structured type *State* has been defined, which contains three elements, *Initializing*, *Idle* and *Busy*. The elements are separated by the | character and each element contains the optional recognizer functions indicated by ?, *e.g.: IsIdle* and *IsBusy*. These functions map the elements to the Boolean type and yields *true* if and only if the recognizer function belongs to the element, *e.g.: IsIdle(Idle) = true* and *IsIdle(Busy) = false*. Each element can hold multiple data types, *e.g.:* element *Initializing* contains the variable *step* to indicate the progress of the initialisation. The variable can be retrieved in the following way: *step(Initializing(n))=n*.

The language also contains *List* type constructors, from which a list of a certain data type can be constructed. The *List*(*State*) defines a list of *State* elements. The empty list is represented as []. With the *head* operator the first element of the list can be retrieved and the *tail* operator yields the list excluding the first element. Operator # yields the length of the list and with the ++ operator two lists can be concatenated. Next to the *List* type constructors the language also contains *Set* and *Bag* type constructors.

2.2 mCRL2 Toolset

In this section a brief summary of the mCRL2 toolset is given, for a more extensive overview of the toolset refer to [14, 16]. The mCRL2 toolset is developed at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven, in collaboration with LaQuSo (Laboratory for Quality Software), CWI (Center for Mathematics and Computer Science) and the University of Twente.

In figure 1 an overview of the mCRL2 toolset is given[1]. The left column shows the main input for the toolset, the mCRL2 specification and modal μ-calculus formula. The mCRL2 specification can be created using the concepts discussed in 2.1 and modal μ-calculus formula is used to verify properties on a model, known as model checking. Both can be described in a plain-text file and can be created using any text editor. The rectangles show the manipulators/tools of the toolset and the ellipses show the input and output objects/files of the tools.

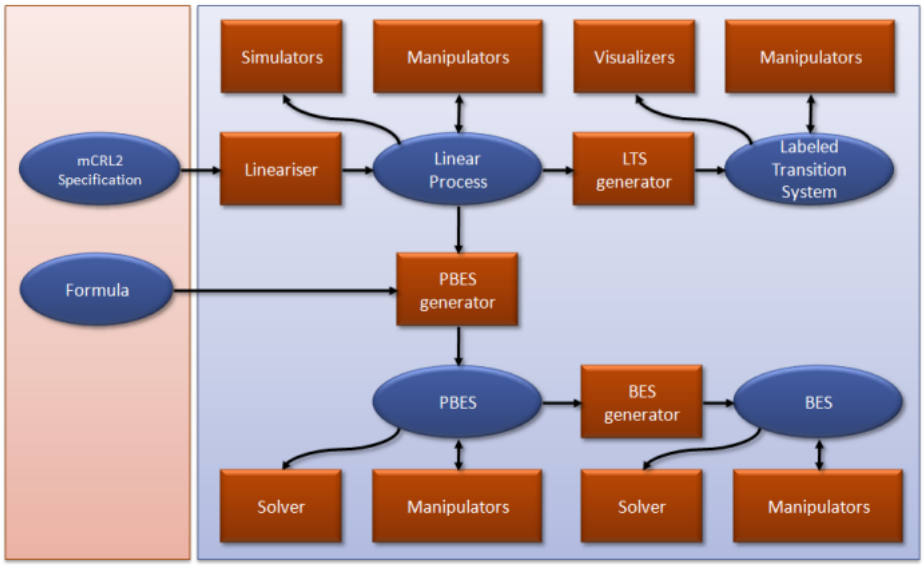


Figure 1: Overview of the mCRL2 toolset

The mCRL2 specification is first linearised before it can be used by other tools, this yields a Linear Process Specification (LPS) and is done by the tool mcrl22lps. The LPS is an mCRL2 specification from which the parallelism has been removed. An LPS can be simulated with the tool lpsxsim, starting from the initial state lpsxsim shows all possible actions that can be performed. The user can select the action to perform after which the process is brought in the next state and lists all possible actions that can be performed from here.
The LPS can be used to generate the state space or Labeled Transition System (LTS) of the mCRL2 specification, this is done by the tool lps2lts. Once the state space is generated it can be viewed using ltsgraph
To verify if a certain property holds on the model via model checking, the LPS is combined with the modal μ-calculus formula and is converted in to a Parameterised Boolean Equation System (PBES) using the tool lps2pbes. The PBES can subsequently be solved using the tool

[1]Figure taken from http://www.mcrl2.org/

pbes2bool which yields either *true* or *false* indicating if the property holds on the model.

The previously discussed tools are the main tools, but the toolset contains many more tools. The mCRL2 toolset is available for Windows, Linux, Apple Mac OS X and FreeBSD. All tools can be executed via a command-line interface and all main tools are accessible via a graphical user interface.

3 Introduction to the Starsense Wireless System

The Starsense Wireless system [12] consists of luminaire-based Outdoor Luminaire Controllers (OLCs) which all control a single outdoor lighting node, a Segment Controller (SC) that controls the OLCs and a managing and monitoring system called CityTouch (figure 2). Communication between the SC and the OLCs is over the 868 MHz frequency band. The SC and OLCs are situated in a mesh network, where messages are routed through the network in a hopping fashion. Communication between the SC and CityTouch is over an Internet connection.

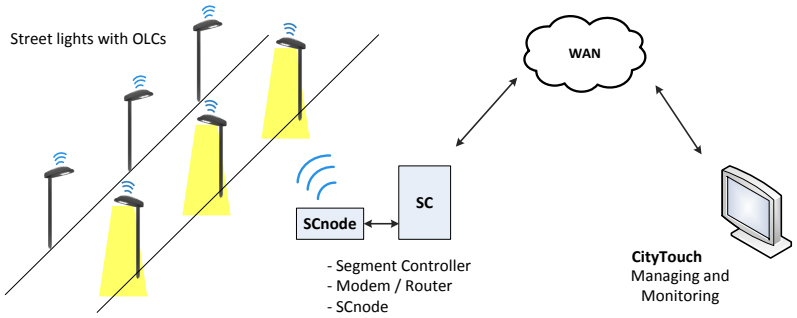


Figure 2: Schematic diagram of Starsense Wireless

3.1 Outdoor Luminance Controller

The OLC can control, log and detect the following functionalities:

- Registration of burning hours, switch-on count.
- Real energy consumption.
- Lamp failure detection.
- System failure detection.
- Switch power on/off.
- Multi-level dim schedules.
- Line voltage registration.
- Stand-alone operation.
- Delayed switching (avoid high inrush current).

As with the SC the OLC also runs multiple applications which all have their own functionality. One of these applications is the software upgrade application.

3.2 Segment Controller

The SC system consists of the SC, which is an embedded Linux board and a SCnode. The SCnode is in essence equal to an OLC node. The SC uses the SCnode to send messages into the network. A SC can handle up to 4000 OLC nodes. Communication between the SC and the OLC is secure.

3.3 CityTouch

With the CityTouch managment system the Starsense Wireless system can be managed and monitored. In the GUI of CityTouch application, every OLC is situated on a street map, which represents the actual location of the OLC. Within this application the real time lighting status is reported, failures are automatically reported and a multi-level dim schedule can be configured.

4 Software Upgrade Components

The Software Upgrade Process is the process in which an SC attempts to upgrade OLCs (all or a subset) in its segment to a new software version.
The SWU process is controlled by four SWU messages, namely: *Reboot*, *Init*, *Data* and *Verify*. These messages are sent by the SC SWU component, following a state machine which controls which message needs to be sent next and how the response needs to be handled. The OLCs that receive the message will handle the message and send a response if asked for.

First the SWU component of the OLC is explained by discussing the function of each message and how the OLC handles these messages. In section 4.1.2 the SWU component of the SC is explained by exhibiting the state machine of the component.

4.1 Implementation

4.1.1 Outdoor Luminance Controller Software Upgrade component

The Outdoor Luminance Controller SWU component is a passive component, in the sense that the component only sends messages as a response to received messages from the SC. When the OLC receives a message that is meant for the SWU component, it will be passed on and handled by SWU component. Depending on the message command it will be handled by a specific function. The messages commands that can be handled by the SWU component are *Reboot*, *Init*, *Data* and *Verify*.

An OLC contains two images, the normal image and the fallback image. Each image can modify the other image but not itself. In order to upgrade an image the OLC must first be booted into the other image. This is executed by the SC by sending a *Reboot* message.

Handling SWU messages
The messages that the SWU component can receive are specified in table 1 and will be further explained below.

Table 1: SWU Messages

Message Type	Parameters	Short Description
Reboot	*TimeDelay*, *OldVersion*, *ForceImage*, *12NCnumber*	Reboot into specified image.
Init	*OldVersion*, *NewVersion*, *NewSize*, *NewChecksum*, *StartAddress*, *12NCnumber*	Initialise for SWU.
Data	*Address*, *Data*	Data chunk for specified address.
Verify	none	Verify if all chunks are received. and checksum matches

Reboot
The reboot message commands the OLC to reboot after *TimeDelay* seconds into the image specified by *ForceImage* if the version number of that image matches *OldVersion*, where *OldVersion* can be a wildcard. Each OLC contains a 12NC number to identify the hardware of the OLC, the 12NC number must be equal to the *12NCnumber* parameter specified in message for the command to be executed.

Init
With this message the OLC is initialised to receive the data chunks of the SWU. Upon receiving this message, the assumption is that the OLC already booted into the opposite image. To initialize the OLC the following conditions must be met:

- The opposite image version must be equal to *OldVersion* (can be a wildcard) and unequal to *NewVersion*.
- *StartAddress* and *NewSize* indicate the address space where the chunks will be written in the flash memory. This must specify an allowed area (*e.g.*: address does not fall within address space of current running image).
- The 12NC hardware identification number of the OLC must be equal to the *12NCnumber* specified in message.

If all conditions are met, then the global flags *isSessionOpened* and *isMy12NC* are set, the flash memory is erased and the parameters *NewChecksum*, *NewVersion*, *StartAddress* and *NewSize* of the SWU are stored for later usage. The parameter *NewChecksum* is used during the verify phase, to verify whether the received image is correct.

Data
This message contains a data chunk to be written at the specified address. The data chunk will only be written if the global flags *isSessionOpened* and *isMy12NC* are true and the specified address indicates an allowed area to write in.

Verify
This message triggers the OLC to verify if there are chunks missing. It does this by using

the *StartAddress* and *NewSize* stored during the execution of the *Init* command and the predefined chunk size, by checking for each chunk location in the flash if it contains data or is erased. Every chunk location that is still erased will be listed and together sent as a response to this message to the SC to inform the SC which chunks need to be resent. When all chunks are received, the image is validated against the previously received checksum. This command will only be executed if the global flags *isSessionOpened* and *isMy12NC* are true.

Response messages
The SC can send the messages of table 1 either as unicast, broadcast with acknowledge or broadcast without acknowledge. When an OLC receives a unicast or broadcast with acknowledge message then a response message is required.

A response message on a received *Reboot* or *Init* message contains as variable the status of the OLC, see table 2 for a list of these statuses. A response message on a received *Verify* message contains, next to the status of the OLC, also a list of chunks that the OLC is missing, the length of this list is limited to 30 chunk IDs. When an OLC is missing more than 30 chunks then these remaining missing chunks will only be reported to SC upon receiving another *Verify* message. Furthermore, the command to which the message is a response is also included.

Table 2: OLC statuses

Status	**Description**
S_OlcAppStatus_Ok	OLC status is OK.
S_OlcAppStatus_SessionClosed	Session is closed, a session is opened after a successful Init message and closed after a handle reboot command error occurs or verify command is successful.
S_OlcAppStatus_NewVersionMatch	The version of the image matches the version of the SWU in the Init message.
S_OlcAppStatus_InvalidAddress	The Init message instructed the OLC to erase the memory of the image that is currently active.

A *Data* message is always sent as broadcast without acknowledge, therefore no response to this message is given.

4.1.2 Segment Controller Software Upgrade component

Multiple components are running on the SC, one of these components is the SWU component. The SC SWU component controls the SWU interaction between the SC and OLC. The SWU component on the SC consists of a state machine for maintaining the states in the update process.
In figure 3 a high level view of the state machine is given. The complete and more detailed state diagram is given in the paragraph **Main state machine** below. The states ***Reboot***, ***Init***, ***Distribute*** and ***Verify*** depicted in figure 3 are the states where the four different types of messages are sent. A short description of the states is given in table 3.

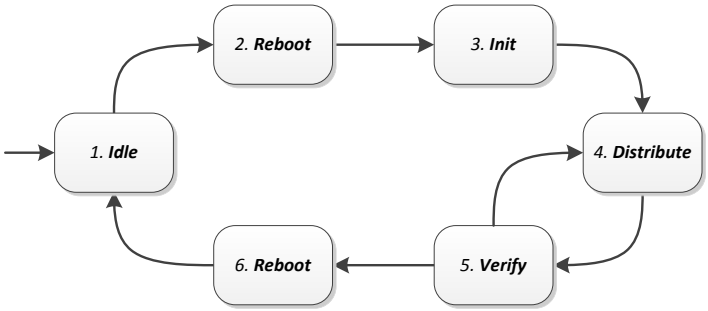


Figure 3: High level SWU state diagram

Table 3: High-level SWU process state machine

State	Activity
1. ***Idle***	Waiting to be instructed to start the SWU process.
2. ***Reboot***	Reboot the OLCs into the image other than the image to be updated.
3. ***Init***	Prepare upgrade, OLCs store version info and erase flash memory of image to be updated.
4. ***Distribute***	Distribute all chunks, OLCs write chunk data into flash.
5. ***Verify***	Instruct the OLCs to verify the received chunks.
6. ***Reboot***	Reboot the OLCs into the newly upgraded image.

The SC SWU component is initiated with a list of OLCs that it needs to upgrade.

Main state machine
The main state machine of the SC SWU component is displayed in figure 4. The six states of figure 3 are now partitioned into sub states for more detail. A **main state** is shown in bold and a *sub state* is shown in italic. The next state of the state machine is determined by the amount of OLCs that need to be upgraded and the status of these OLCs. Each state is explained below. The verify state is more involved, therefore, explained in more detail the paragraph **Verify state machine**.

1. ***Idle***
Idle

Initially the SWU component resides in the idle state, waiting for an external trigger to start the SWU process.

Prepare

In this state the variables needed for the SWU process are initialised and the number of chunks that need to be distributed is calculated. The next state is determined depending on the number of OLCs that need to be upgraded. If the number of OLCs is small (less than 1% of the OLCs in the segment) then the next main state and sub state are ***Reboot*** and *Ucast*, respectively. Else the next main

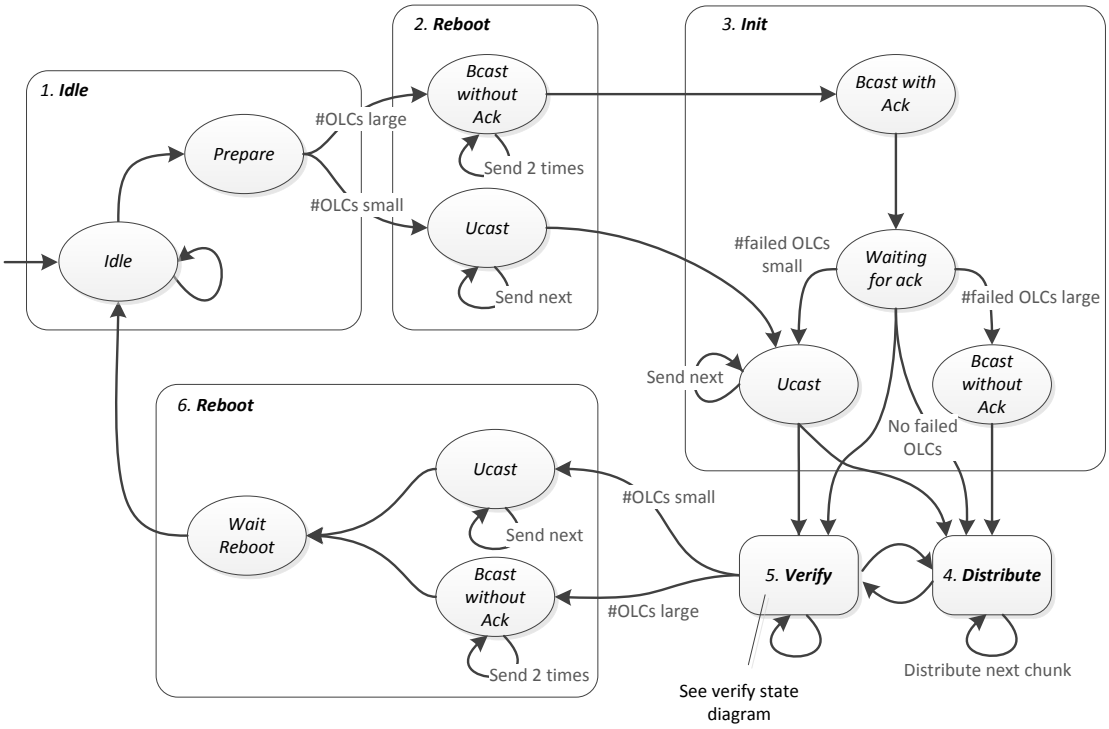


Figure 4: SWU state diagram

state and sub state are ***Reboot*** and *Bcast without ack*, respectively.

2. ***Reboot***
Ucast

A unicast reboot message is prepared and sent to each OLC in the list consecutively. This reboot message is sent to the OLC to make sure that the OLC is not running the image that needs to be upgraded. After each sent unicast message the SC waits for a response from the OLC. If the OLC does not respond before a timeout occurs, then the OLC is flagged with a timeout error. When a unicast message is sent to all OLCs, then the state machine proceeds to the main state and sub state: ***Init*** and *Ucast*, respectively.

Bcast without ack

A reboot message is prepared after which two reboot broadcast messages are sent, with some time in between. In this message it is specified that no response message should be sent back. Subsequently the state machine moves to the main state and sub state: ***Init*** and *Bcast with ack*, respectively.

3. ***Init***
Bcast with ack

An initialisation broadcast message is sent and the SC moves to the state *Waiting for ack*. This initialisation message contains information about the version, the size, checksum and the address of where the update will be written in flash memory.

Waiting for ack

After waiting for the responses on the *Init* broadcast all responses are examined, when:

Not all OLCs responded
If the number of OLCs that did not respond is large the next sub state will be *Bcast without ack* else the next sub state will be *Ucast*.

All OLCs responded
When there are OLCs that have the status OK the next main state will be ***Distribute***, otherwise all OLCs either have an error status or the image of the OLC is equal to the upgrade version, in that case the next main state will be ***Verify***.

Ucast

An initialisation message is sent to each OLC in the list consecutively. The SC waits for a response after sending the message, if a response indicates that the image of the OLC is equal to the upgrade version then the missing chunks counter for this OLC is set to zero and the OLC is flagged as received.
When there are OLCs that have the status OK the next state will be ***Distribute***, otherwise all OLCs either have an error status or the image of the OLC is equal to the upgrade version, in that case the next state will be ***Verify***.

Bcast without ack

After a broadcast *Init*, a large number of OLCs did not respond. In this state another attempt to initialize these OLCs is done by sending another *Init* broadcast message, this time the OLCs are instructed not to send an acknowledge back.

4. ***Distribute***

In this state each chunk that is flagged as missing (initially all chunks are flagged missing) is broadcast to the OLCs and subsequently flagged as not missing. After the verification stage, chunks can be flagged as missing again and the state machine will return to the ***Distribute*** state. In this message is specified that no response message should be send back. When all chunks are distributed, the next state is set to ***Verify***.

5. ***Verify***

The verification process is handled by a different state machine, this state machine is explained in the section below.

6. ***Reboot***
Ucast

A reboot unicast message is sent to each OLC in the list, to reboot the OLC into the new image.

Bcast without ack

As in main state *2.* ***Reboot*** and sub state *Bcast without ack*, a reboot message is prepared after which two reboot broadcast messages are sent, with some time in between. In this message it is specified that no response message should be sent back. Subsequently the state machine moves to the state *Wait Reboot*.

Wait Reboot

The the external system is informed about the result of the SWU process and the state machine proceeds again to the ***Idle*** state.

Verify state machine

After the chunks are distributed, verification must be done to check if an OLC is still missing chunks. This process is shown as the verify state diagram shown in figure 5. The verify state diagram is a sub state of the the main SWU state diagram shown in figure 4. All states are explained below.

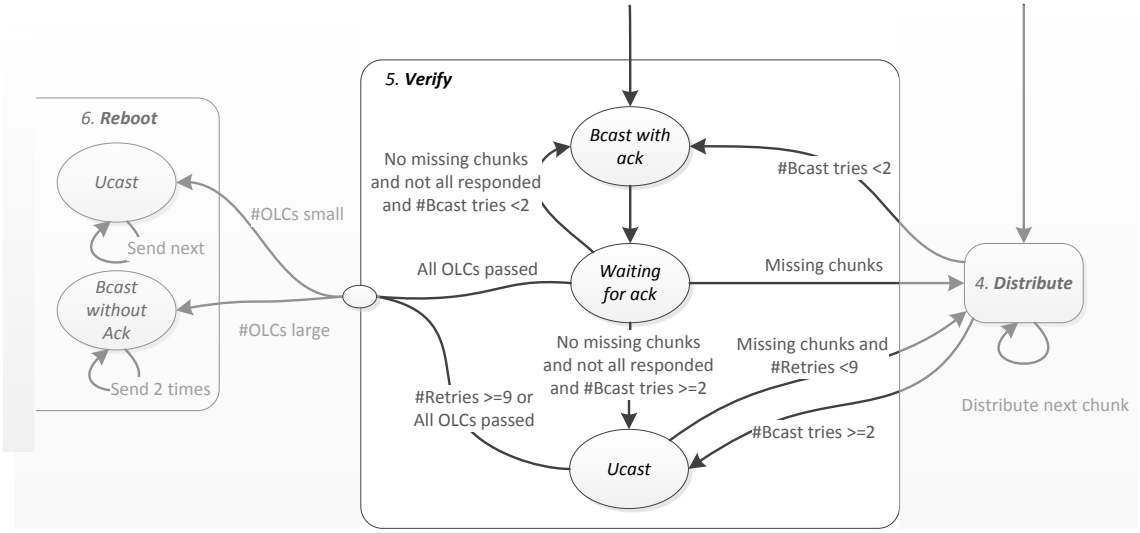


Figure 5: Verify state diagram

Bcast with ack

All OLCs are instructed to verify which chunks they are missing and report back to the SC. A verify broadcast with acknowledge message is sent for this purpose.

Wait for ack

Each OLC in the list is checked whether it responded to the verify message. If it responded, its status will be checked:

- Status OK and no missing chunks, then OLC is successfully upgraded.
- Status OK and missing chunks, then next state will be distribute and the missing chunks will be distributed.
- Error status, OLC is in error status and cannot not be upgraded.

If all OLCs responded and no chunks are missing then the next state is either *Reboot Ucast* or *Reboot Bcast without ack*, depending on the number OLCs in the list.
If not all OLCs responded a second verify broadcast message will be sent. If after this message still not all OLCs responded then the next state is *Ucast* and a verify unicast message will be sent to each non responding OLC.

Ucast

Each OLC that is not yet successfully verified will be addressed with a verify unicast message, if the OLC does not respond after two consecutive times it will be put in an error state and the SC will move to the next OLC. When an OLC does respond it will be checked for its status, as described in the *Wait for ack* state.
Each time an OLC indicates it still has chunks missing the SC will redistribute the chunks, with a maximum of nine times. If after the ninth time the OLC still misses chunks then the SWU for that OLC is considered unsuccessful and the SC will move to the next OLC.

For a detailed flowchart of the verify process appendix A can be consulted.

4.2 Modeling

The SWU process is modeled using the mCRL2 modeling language [13, 14]. In the following section it is explained how the model relates to the implemented SC and OLC SWU components.

4.2.1 Simplifications

In order to be able to automatically analyse the model some complexity is abstracted away. The aim is to include those variables of the C implementation that have influence on the behaviour of the the state machine and which messages will be sent. In this way unwanted behaviour such as unnecessary sent messages can still be observed.
In the implementation for instance three different protocols can be active in the same network at the same time, each message that is sent into the network needs be to sent individually for each active protocol. This does not have influence on the main behaviour of the SWU process and therefore is not included.
Also the underlying network layer is simplified. The fact that a message travels through a

mesh network via different paths is not included. The unreliable network is modeled such that a sent message is either communicated successfully or is dropped.
In the implementation two types of reboot messages can be sent. One reboot message instructs an OLC to do a hard reboot and the other to do a soft reboot. With a soft reboot the OLC is not reset completely, to prevent flickering of the light. In the model only the soft reboot message is considered.
Moreover an SWU in the implementation consists of around 1500 chunks whereas in the model a SWU only consists of two chunks.

4.2.2 The OLC SWU model

The model of the OLC SWU component, abbreviated to the OLC component, can perform the external actions shown in table 4. The only interactions with the OLC is via messages. The OLC can receive a message from the SC and, if asked for, it can send a response to this message. It does not send any messages on its own initiative.

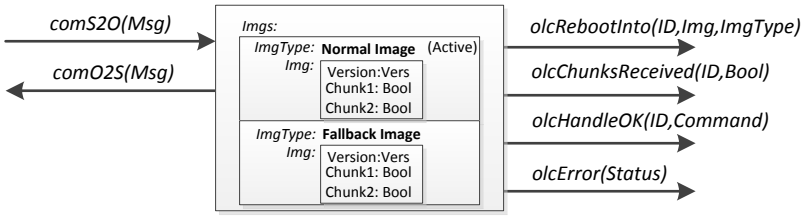


Figure 6: The model of the OLC process with external actions and internal variables

In figure 6 the external actions and the internal *Imgs* variable can be seen. The *Imgs* variable shows information about the two images of the OLC, the normal image and the fallback image, and shows which image is active. Both images contain a version number indicated with *Version* and whether or not the chunks for the image are received. In this model an image consists of two chunks.

Table 4: External actions of the OLC

External Action	**Description**
comS2O(*Msg*)	Receive a *Msg* from the SC.
comO2S(*Msg*)	Send a *Msg* to the SC as a response on a received message.
olcRebootInto(*ID*,*Img*,*ImgType*)	OLC with *ID* reboots into normal or fallback image (*ImgType*) and details of that image are specified in *Img*.
olcChunksReceived(*ID*,*Bool*)	OLC with *ID* reports whether or not it has received all chunks happens after a verify command.
olcHandleOK(*ID*,*Command*)	OLC with *ID* reports that it handled command *Command* successfully, where *Command* is either a *Reboot*, *Init*, *Data* or *Verify* command.
olcError(*ID*,*Status*)	OLC with *ID* reports an error status.

The SC sends a message to the OLC via the external action *comS2O*(*Msg*). The message is passed on to the process APP_SwUpgrade_HandleSwUpgradeCommand, which checks the *MsgType* of the message and determines which process needs to handle the command. The command is either *Reboot*, *Init*, *Data* or *Verify* and is handled by the process HandleRebootCmd, HandleInitCommand, HandleDataCmd or HandleVerifyCmd, respectively.

A message is handled as described in section 4.1.1. When a *Reboot* command is successfully handled then the action *olcRebootInto*(*ID*,*Img*,*ImgType*) is performed, indicating to which image the OLC reboots into and what the current status is of that image.
When a *Verify* command is successfully handled then the action *olcChunksReceived*(*ID*,*Bool*) is performed, indicating whether or not all SWU chunks are received for this OLC. For each command that is handled successfully an *olcHandleOK*(*ID*,*Command*) action is performed, showing which command is handled. When a command is not handled successfully the action *olcError*(*ID*,*Status*) is performed, showing the status of the OLC.

Table 5: Msg structure

Msg	**Description**	**Direction**
MSG(*MsgDes*, *MsgType*)	Send message to *MsgDes* which is either an unicast, broadcast with acknowledge or broadcast without acknowledge.	SC→OLC
MSG_UcastResp(*ID*, *MsgType*)	A response to a *MsgType* unicast message.	OLC→SC
MSG_BcastResp(*ID*, *MsgType*)	A response to a *MsgType* broadcast message.	OLC→SC

The messages communicated using the *comS2O*(*Msg*) and the *comO2S*(*Msg*) actions are explained in table 5. Every *Msg* contains a *MsgType*. A message type specifies which command is sent and what their variables are (table 6).

A message sent from the SC to an OLC contains a *MsgDes*, which specifies whether the message is sent as a unicast, a broadcast with acknowledge or a broadcast without acknowledge. A message sent from an OLC to the SC contains the identifier of the OLC and the response message type, where the response message type is either a verify response or a general response (table 6).

4.2.3 The SC SWU model

The model of the SC SWU component, abbreviated to SC component, is defined by the external actions shown in table 7. An overview of the SC process together with the external actions is depicted in figure 7.

The software upgrade is initiated by the action *startSWU*(*List(OlcInfo)*, *Image*), which supplies the list of OLCs that need to be upgraded to the specified image. Once initiated, the SC process follows the state machine as discussed in section 4.1.2. A message is sent to the OLCs using the action *comS2O*(*Msg*). When a response is expected then either the response is received via action *comO2S*(*Msg*) or a timeout action is performed.
When a broadcast with acknowledge is sent, then from each OLC a response is expected, the process waits for the response of each OLC, or when the message is dropped a timeout is generated.

Table 6: MsgType structure

MsgType	**Description**
Msg_REBOOT(*ImgType*)	*Reboot* message to instruct the OLC to boot into normal or fallback image (*ImgType*)
Msg_INIT(*Vers*,*ImgType*)	*Init* message to instruct the OLC to erase the flash of the normal or fallback image (*ImgType*) and save new version number (*Vers*).
Msg_DATA(*addrFlash*, *Data*)	*Data* message contains the data chunk for address location 0 or 1.
Msg_VFY	*Verify* message instructs OLC to check which chunks are missing.
Msg_VFY_RESP(*Status*, *missingChunks*)	Respond on a verify request with a list of missing chunks.
Msg_GEN_RESP(*Status*, *Command*)	When a response other then verify is requested a general response is sent, stating the *Status* of the OLC and on which command it is a response on (*Command*)

Table 7: External actions of the OLC

External Action	**Description**
startSWU(*List(OlcInfo)*, *Image*)	Start the software upgrade process, update all OLCs in *List(OlcInfo)* to the specified *Image*.
verifiedOlcOK(*ID*, *Bool*)	Shows if OLC with *ID* is verified successfully.
finished	The SWU has finished.
upgradeNotOK	One or more OLCs failed to upgrade.
waitForResponse	Wait for responses from the OLCs.
timeout	A timeout has occurred on waiting for a response message from an OLC.
comS2O(*Msg*)	Send a *Msg* to the OLCs.
comO2S(*Msg*)	Receive a *Msg* response message from an OLC.

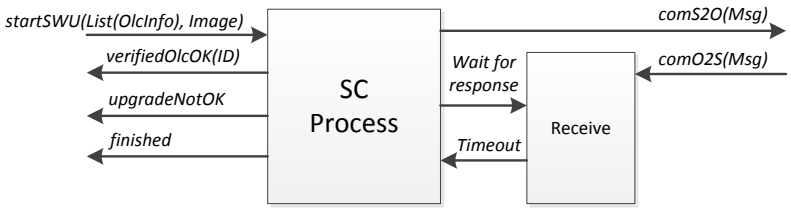


Figure 7: The model of the SC process with external actions

When the state machine is in the verify phase each OLC is verified, to show the result of the verification the action *verifiedOlcOK*(*ID*, *Bool*) is performed for each OLC.
After the state machine enters the *Idle* state again, the action *finished* is performed and the SC process is ready to start a new software upgrade.

4.2.4 The model of the interacting SC and OLC components

The model of the interacting SC and OLC SWU components is created by connecting the model of the SC SWU component with one or more model instances of the model of the OLC SWU component.
The SC and the OLC SWU models communicate with each other via the Net process. The Net process creates the synchronous communication between the SC and OLC processes.

The Net process can be configured to drop a specified number of packages in each direction. The communication actions of the SC and the OLCs including the actions that cause the messages to be dropped are explained in table 8.

Table 8: Communication actions of the SWU process

External Action	**Description**
comS2O(*Msg*)	Forwarding messages from the SC to the OLCs
comS2Odrop(*Msg*)	A message from the SC to the OLCs is dropped before it arrived at any OLC
comO2S(*Msg*)	Forwarding messages from an OLC to the SC
comO2Sdrop(*Msg*)	A message from an OLC to the SC is dropped before it arrived to the SC

The model of the SWU process is depicted in figure 8 here the external actions *comS2O*(*Msg*) and *comO2S*(*Msg*) of the SC and the OLCs are connected. When a message is dropped, it will not arrive at any of the specified receivers.

4.3 Analysis

Analysis on the model is divided in three parts. First requirements are specified and verified. The minimal message drop that can cause an OLC or a segment of OLCs not to be upgraded is investigated and finally odd behaviour that was discovered during analysis is discussed. Analysis on the model is done using the system and toolkit as described in chapter 6.

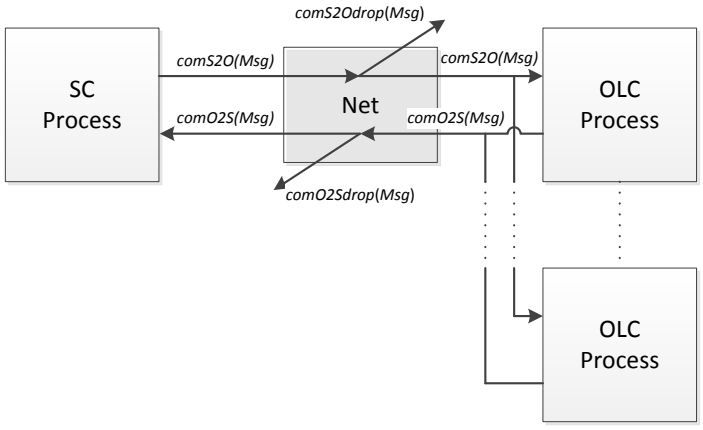


Figure 8: The model of the SWU process

4.3.1 Requirements

The model of the SWU process must comply with the following requirements:

- A started SWU process must always finish, and only after the SWU process is finished it must be able to start again.
- All OLCs must always be successfully upgraded when the communication is reliable.

Verification of these requirements is discussed below.

A started SWU process must always finish
A started SWU process must always finish, and only after the SWU process is finished it must be able to start again. The SWU state machine is started with the action *startSWU* and finished after the *finished* action.

To check this property, the system can be observed by considering *startSWU* and *finished* as the external behaviour and all other actions as internal behaviour. This is done by generating the state space of the model (appendix B) with all actions other than *startSWU* and *finished* renamed to τ actions. Because all internal actions are renamed to the same action name, namely τ, the state space can be reduced. If the stated requirement holds then no internal behaviour will prevent the system from doing a *finished* action always and only after a *startSWU* action, as is shown in figure 9. If the requirement does not hold, then there is a path caused by a τ action that prevents the system from doing the required behaviour.

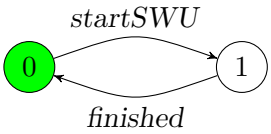


Figure 9: A started SWU process can always finish and be started again

This property has been checked by executing the commands in table 9, which results in the state space shown in figure 9 and thus the requirement holds.

Table 9: The commands used to verify the requirement

	Tool	Arguments
(1)	`mcrl22lps`	`model.mcrl2 model.lps`
(2)	`lpsrename`	`model.lps model_hidden.lps --renamefile=rename/list_of_actions_to_hide.rename`
(3)	`lps2lts`	`model_hidden.lps model_hidden.aut`
(4)	`ltsconvert`	`model_hidden.aut model_red.aut --equivalence=branching-bisim`
(5)	`ltsgraph`	`model_red.aut`

Using (1) the model (appendix B) is linearized. In (2) all actions other than *startSWU* and *finished* are hidden, by supplying a rename file (`list_of_actions_to_hide.rename`, included in appendix C) that contains a list all actions other than *startSWU* and *finished* that need to be renamed to τ actions. Followed by (3) the state space generation and (4) reducing the state space (modulo branching bisimilarity). Finally (5) the state space is visualised.

All OLCs must always be successfully upgraded when the communication is reliable
When the communication between the SC and OLCs is reliable then all OLCs must always be correctly upgraded. This property is checked by hiding all actions apart from *startSWU* and *finished* and *olcRebootInto*, using the method explained in the previous section.
The action *olcRebootInto*(...) is renamed to *rebootOK*(*Vers*,*ImgType*) only if the image contains all chunks.
The model is configured as follows:

- All OLCs are initialised with software version V1.
- The SWU is an Normal Image with software version V2.
- A communication timeout does not occur before all expected messages are received.

The expected behaviour is that after the *startSWU* action all OLCs reboot into the other image (Fallback Image in this case) and after the upgrade is done all OLCs reboot into the Normal Image which now contains the new software version V2.

Figure 10 shows that the behaviour of the model follows the expectation. The figure shows the state space of a model containing a single OLC after reduction. Here, it can be seen that along each path the OLC is upgraded correctly, because a path always ends with the OLC rebooting into the normal image where the image contains both data chunks and has version V2, indicated by action *rebootOK*(*V2, NORMAL_IMAGE*).

4.3.2 OLCs not upgraded after message drop

Minimal message drop can cause an OLC to not get upgraded
An OLC is upgraded by receiving the messages discussed in section 4.1.1, since the communication between the SC and the OLCs is unreliable, messages can get lost. What is the

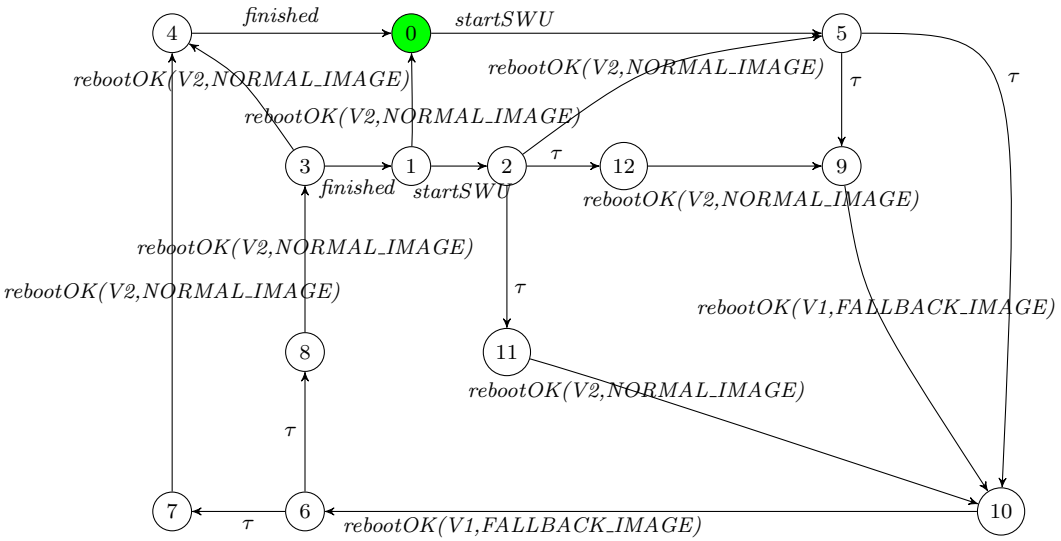


Figure 10: OLCs are always correctly upgraded when the communication is reliable

minimum amount of message drops that can cause an OLC not to get upgraded?

A single message drop can cause an OLC not to be upgraded. In the case that the number of OLCs in the segment is small (less than 1% of the OLCs in the segment) then the *Reboot* and the *Init* messages will be sent via unicast messaging, only once. The following cases describe the situations when dropping a single message will cause an OLC not to be upgraded.

How the problem was discovered
The problem was discovered by configuring the model to be allowed to drop a certain amount of messages. Starting with one, and subsequently configuring the tool lps2lts to detect and report the action *upgradeNotOK* the during state space generation. Configuring the model to drop a singled dropped message already yielded multiple traces to the action *upgradeNotOK*. The analysis of a trace was done by examining first only the communication actions, hiding all other actions in the trace. This yields a trace similar to table 10. With this trace it can easily be observed which message has been dropped. The result of the dropped message can be investigated by adding previously hidden actions to the trace, which leads to the conclusions below.

Message drop in the first reboot phase
When a *Reboot* unicast message from the first reboot phase is dropped and the OLC is currently executing the image that needs to be upgraded, then the OLC will not be rebooted into the other image and cannot be initialised for the software upgrade. When the OLC now receives an *Init* message it will respond with the OLC status *S_OlcAppStatus_InvalidAddress*.

Message drop in the init phase
When the *Init* unicast message is dropped, then the OLC will not be upgraded because the OLC is not correctly initialised, the SWU session is not opened and all distributed chunks will be ignored.

Message drop in the second reboot phase

When the *Reboot* unicast message from the second reboot phase is dropped then the OLC image is upgraded correctly, but the SWU process is not completed correctly because the OLC will not reboot into the newly upgraded image.

Minimal message drop can cause segment to not get upgraded

What is the minimum amount of message drops that can cause a whole segment not to get upgraded?

Two dropped messages can cause a whole segment of OLCs not to be upgraded. When the number of OLCs that need to be upgraded is large then in the first and second reboot phase two *Reboot* broadcast messages without acknowledge will be sent. When in either reboot phase both messages are dropped, then the whole segment will not be upgraded. The following cases describe the situations in more detail.

How the problem was discovered

Same method as described in the paragraph '**Minimal message drop can cause an OLC to not get upgraded**' above has been applied, but this time examining traces where broadcast messages have been dropped.

Two dropped messages in the first reboot phase

The model contains two OLCs both running the normal image. Table 10 shows all communication between the SC and the OLCs, where the messages of the first reboot phase are dropped (denoted by the action *comS2Odrop*). The OLCs in the segment are now not rebooted into the fallback image. As a consequence, the initialisation will not succeed and the segment cannot be upgraded.

After the failed first reboot phase, the SC continues the SWU process generating a lot of pointless traffic on the network. Both the *Init* and the *Verify* message require all OLCs to respond, on which all OLCs respond with the OLC status *S_OlcAppStatus_InvalidAddress* on the *Init* message and *S_OlcAppStatus_SessionClosed* on the *Verify* message.

Table 10: Trace of communication actions: Segment not upgraded when two *Reboot* messages are dropped.

comS2Odrop(MSG(BcastWithoutAck, Msg_REBOOT(FALLBACK_IMAGE)))
comS2Odrop(MSG(BcastWithoutAck, Msg_REBOOT(FALLBACK_IMAGE)))
comS2O(MSG(BcastWithAck, Msg_INIT(V2, NORMAL_IMAGE)))
comO2S(MSG_BcastResp(OLC1, Msg_GEN_RESP(S_OlcAppStatus_InvalidAddress, INIT)))
comO2S(MSG_BcastResp(OLC2, Msg_GEN_RESP(S_OlcAppStatus_InvalidAddress, INIT)))
comS2O(MSG(BcastWithAck, Msg_VFY))
comO2S(MSG_BcastResp(OLC1, Msg_GEN_RESP(S_OlcAppStatus_SessionClosed, VFY)))
comO2S(MSG_BcastResp(OLC2, Msg_GEN_RESP(S_OlcAppStatus_SessionClosed, VFY)))
comS2O(MSG(BcastWithoutAck, Msg_REBOOT(NORMAL_IMAGE)))
comS2O(MSG(BcastWithoutAck, Msg_REBOOT(NORMAL_IMAGE)))

Two dropped messages in the second reboot phase

When in the second reboot phase both reboot messages are dropped, then all OLCs will

be successfully upgraded. But they will not be rebooted into the new image and thus the SWU process is not completed correctly.

4.3.3 Odd behaviour and recommendations

This section discusses cases where unnecessary traffic is generated on the network. For each case the observed behaviour is explained, followed by how the case can be found and finally recommendations are discussed.

Case 1: No response but SC SWU state machine continues
Observed behaviour
In case that no OLC reacts to any messages sent from the first *Reboot* state and the *Init* state in the SC SWU state machine discussed in section 4.1.2, then the state machine still sends the messages from *Verify* and second *Reboot* state.

How the problem was discovered
The model is configured to have reliable communication from the SC to the OLCs and unreliable communication from the OLCs to the SC. All actions apart from *startSWU*, *finished* and *olcChunksReceived* are hidden. By following the same commands as in section 4.3.1 an action *olcChunksReceived*(*OLC1*,*false*) can be found. This action was unexpected, because the communication from the SC to the OLC is reliable and still in the verify step *OLC1* claims that it did not receive all chunks.
To investigate what execution of actions causes this behaviour, a trace to the action *olcChunksReceived*(*OLC1*,*false*) must be found. This is done by specifying in a rename file that this action with exactly those arguments needs to be renamed to a new action, for instance *olcChunksReceived_false*. A trace to this new action can now found during the state space generation using the following commands:

```
mcrl22lps model.mcrl2 model.lps --verbose
lpsrename model.lps model_hidden.lps --renamefile=rename/rename_olcChunksReceived.rename -v
lps2lts --action=olcChunksReceived_false model_hidden.lps --verbose
```

The last command generates all traces to this action, the trace can be inspected by loading it in the simulator tool LpsXsim. During inspection of the trace it became apparent that *OLC1* did not react to any messages from the SC, upon which the SC skipped the distribution of the chunks but continued the state machine, sending *Verify* and *Reboot* messages.

Recommendations
Traffic on the network can be reduced by stopping the SWU process if none of the OLCs respond to any *Init* messages.

Case 2: Unnecessary chunks distribution broadcast
Observed behaviour
All chunks are distributed while no OLC responded to earlier messages, possibly sending around 1500 chunks for no reason.

How the problem was discovered
In case 1, the state machine continues even though nobody responds, but skipping the dis-

tribution state. Interesting question is, whether there is a trace where nobody responds and still all chunks are distributed? This property is checked by configuring the model to drop all messages that are sent from the OLCs to the SC. Then, the tool lps2lts is configured to detect and report the action *distributeChunk* the during state space generation. This yields a trace that shows that there exists a path where packets are distributed while non of the OLCs responded, see table 11 for this trace where all actions are hidden apart from *startSWU*, *distributeChunk*, *finished* and the communication actions.

Table 11: Trace of actions: Distribute chunks while no OLC responds

startSWU
comS2O(MSG(BcastWithoutAck, Msg_REBOOT(FALLBACK_IMAGE)))
comS2O(MSG(BcastWithoutAck, Msg_REBOOT(FALLBACK_IMAGE)))
comS2O(MSG(BcastWithAck, Msg_INIT(V2, NORMAL_IMAGE)))
comS2O(MSG(BcastWithoutAck, Msg_INIT(V2, NORMAL_IMAGE)))
distributeChunk
comS2O(MSG(BcastWithoutAck, Msg_DATA(0, D1)))
distributeChunk
comS2O(MSG(BcastWithoutAck, Msg_DATA(1, D1)))
comS2O(MSG(BcastWithAck, Msg_VFY))
comS2O(MSG(BcastWithAck, Msg_VFY))
comS2O(MSG(BcastWithoutAck, Msg_REBOOT(NORMAL_IMAGE)))
comS2O(MSG(BcastWithoutAck, Msg_REBOOT(NORMAL_IMAGE)))
finished

Recommendations
This shows a case where no OLC responds and so possibly a lot of traffic is generated for no reason. Sending all chunks takes well over an hour and no other actions can be performed by the SC during this time. There is still the possibility that some or all OLC received the messages from the SC and are successfully upgraded, in the case that communication from the SC to the OLC is successful and all communication from the OLCs to the SC is unsuccessful. A possible improvement would be to end the SWU process in the *Init* state when non of the OLCs responded to an *Init* message.

Case 3: Verify broadcast with acknowledge for a single OLC
Observed behaviour
When after the first verify broadcast is sent, all OLCs responded apart from one, a second verify broadcast is sent out, causing all OLCs to respond again. The OLCs that already responded before will respond with the OLC status *S_OlcAppStatus_SessionClosed*.

How the problem was discovered
The model was configured as follows: SC process contains a list with two OLCs, where in the model only one OLC exists, communication is reliable. By now stepping through the state space, manually creating a trace from action *startSWU* to action *finished*, it was observed that a broadcast verify was sent out twice, and the single OLC responded with the status *S_OlcAppStatus_SessionClosed*. In table 12 a fraction of the trace is displayed, where this behaviour can be seen.

Table 12: Fraction of the trace showing two verify broadcast messages and responses

comS2O(MSG(BcastWithAck, Msg_VFY))
olcChunksReceived(OLC1, true)
comO2S(MSG_Resp(OLC1, Msg_VFY_RESP(S_OlcAppStatus_Ok, [])))
timeout
verifiedOlcOk(OLC1,true)
comS2O(MSG(BcastWithAck, Msg_VFY))
olcErr(OLC1,S_OlcAppStatus_SessionClosed)
comO2S(MSG_Resp(OLC1, Msg_GEN_RESP(S_OlcAppStatus_SessionClosed, VFY)))

Recommendations
Before sending a verify broadcast no check is done as to how many OLCs need to be addressed, whereas this is done in other parts. This means that there can be situations where a large part already responded and only a small number of OLCs did not yet respond. When now a verify broadcast message is sent all OLCs will respond, most of them with the OLC status *S_OlcAppStatus_SessionClosed*, causing a lot of unnecessary traffic on the network. An improvement would be to send unicasts instead of a broadcast when the OLCs that need to be addressed is small, as is done in other parts of the state machine.

Case 4: *Reboot* message is sent instead of *Init* message
Observed behaviour
Multiple network protocols can be active in the network at the same time. When three different protocols are active in the network, then messages need to be sent once for each protocol. The message handler for the third protocol sends the wrong message in the function HandleBcastInitwithoutAck. This results in the behaviour that the when the segment is large and three protocols are active, then the OLCs running the third protocol will not be upgraded.

How the problem was discovered
Code inspection. In file A_SWUpgrade.c and function HandleBcastInitwithoutAck, the message sent for the first two protocols is `S_OLCPARM_SWUCOMMAND_SWU_INIT` while the message for the third protocol is `S_OLCPARM_SWUCOMMAND_SWU_REBOOT`.

Recommendations
This error can be corrected by replacing the *Reboot* message for an *Init* message.

5 Commissioning Components

Commissioning is the process in which the SC contacts and configures an uncommissioned OLC that needs to become part of the segment which is under control of the SC. An OLC is only allowed to join the segment if it is manufactured and pre-configured by Philips. To authenticate an OLC, security information needs to be present in the SC and the OLC.

An uncommissioned OLC is addressed via its IEEE address and a commissioned OLC is addressed via its short address. An uncommissioned OLC has a factory configured IEEE address and no short address, this short address is assigned during the commissioning process. Apart from the short address, a PAN ID and network security information is sent to the OLC during commissioning. The PAN ID is used to identify to which segment the OLC belongs, and the network security information is used to provide secure communication after the OLC is commissioned.

Apart from authenticating and adding an OLC to the segment, the SC also configures settings on the OLC for the lamp driver, sends the lighting dim schedules and initiates SWU for the OLCs when the software version of the OLC is lower than the software version of the SCnode.

5.1 Implementation

In figure 11 a schematic overview is given of the main elements in the commissioning interaction.

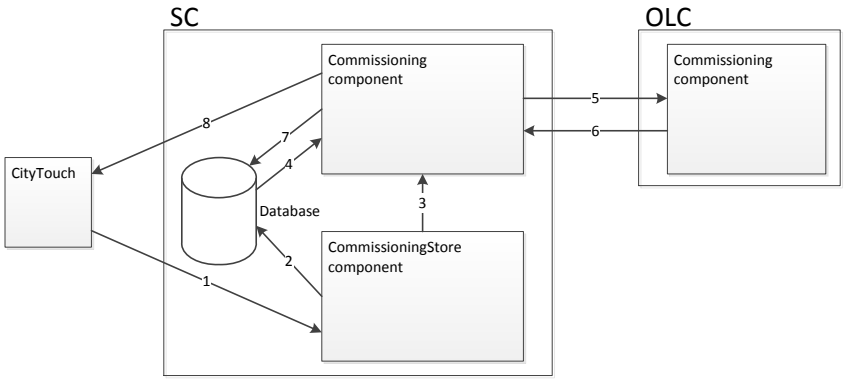


Figure 11: Commissioning components interaction overview

A request to (de)commission an OLC is sent from the CityTouch management system to the SC. This request is parsed and sent to the CommissioningStore component, see arrow (1). This component stores the request from CityTouch into the database (2) and triggers the Commissioning component to handle the pending request by placing a message in the thread queue of the Commissioning component, indicating that the database needs to be checked for OLCs with pending actions (3). Upon handling this message and inspecting the database (4), a single OLC with a pending request will be processed, depending on the request, a message will be sent to the OLC to request the current state of the OLC, followed by messages changing the state of the OLC if necessary (5 and 6). The database and CityTouch are updated with

the success or failure of the request (7 and 8). In case of failure the request will be put into an error state and at a later moment all requests in the error state will be retried.
In the following sections the SC and OLC Commissioning components will be explained in more detail.

5.1.1 Segment Controller Database

For each OLC that has an action a record is stored in the local database of the SC. Such a record contains known information about the OLC such as the OLC type, the IEEE address, a short address, a PAN ID, the *ActionType* and the *CommissioningState*. These last two variables indicate the pending action and the state of the pending action / OLC, see table 13 and table 14, which lists the value space of the *ActionType* and the *CommissioningState* variables.

Table 13: *ActionType* members

ActionType	**Description**
ActionCommissioning	OLC needs to be commissioned.
ActionUpdate	A calender update, lamp replacement or lamp type update needs to be done.
ActionDecommissioning	OLC needs to be decommissioned.
OLCReplacement	The OLC is replaced, needs to be configured and commissioned.
NoAction	No action needs to be done.

Table 14: *CommissioningState* members, stored in the local database of the SC

CommissioningState	**Description**
Update	Operational state depends on *ActionType*.
Decommissioning	Decommissioning process is started.
Decommissioned	Decommissioning process is finished.
ReadyForDeletion	Decommissioning process is finished and the OLC can be removed from the database.
CommissioningError	Commissioning started and failed.
Error_sw_incompatible	OLC software version is newer than SCnode software version.
WaitForReboot	OLC had a reboot command, after a timer the next state is set to *Ready*.
WaitForRebootAndSWU	OLC had a reboot command and needs an SWU, after a timer the next state is set to *ReadyForSWU*.
ReadyForSWU	At least one of the images of the OLC needs an SWU.
ReadyForSWUIncomplete	See *ReadyForSWU*, only now for a whole segment update.
ReadyForSWUIncompleteReady	The main image is upgraded, fallback image still needs to be upgraded. OLC is now operational (only for a whole segment update).
IncompleteSWU	Error during SWU.
Error	Multiple reasons can create this error.
Ready	No action needs to be done.

5.1.2 Segment Controller CommissioningStore component

When an OLC needs to be commissioned or decommissioned, or some other action must be performed, such as OLC replacement, lamp replacement or a calender update then the management system CityTouch is used to communicate this to the SC. The CommissioningStore application processes these requests. When an OLC needs to be commissioned for instance, then the CommissioningStore sets the *ActionType* of that OLC in the local database to *ActionCommissioning* and the *CommissioningState* to *Update*. Subsequently the CommissioningStore executes an external interface function of the Commissioning component to schedule a *Msg_ScanAgain* in the thread message queue.

5.1.3 Segment Controller Commissioning component

The Commissioning component has its own thread, which executes its tasks based on the messages in the thread queue. When a task is finished, the next message is taken from the thread queue. In table 15 the different messages that can be scheduled in the queue are listed. For reasons of clarity, only the messages that are related to the interaction between the SC and OLC are shown.

Scheduling thread messages
A thread message can be scheduled by a call to the external interface of the Commissioning component or as a result of an earlier executed thread message. A thread message can be scheduled with a delay, *e.g.:* the message will be scheduled into the queue after the delay. When a thread message is scheduled then a check is done to see if there is already an instance of this message in the queue, when there is not yet an instance, then it will be scheduled. When there exists an instance then it will only be scheduled if the message contains data (such as the *Msg_Continue_Decommissioning* message), or if the delay of the new message is shorter than the delay of the existing message, then the delay of the existing message will be replaced by the new delay. This means that only one instance of each message can exist in the queue, except for the *Msg_Continue_Decommissioning* message.
In the following paragraph it will be explained how each thread message is handled.

Table 15: Thread queue messages

Thread message	**Description**
Msg_ScanAgain	Scan the database for an OLC with a pending action.
Msg_Start_SWU	Execute an SWU on the segment.
Msg_Continue_Decommissioning(*olc_id*)	Continue with decommissioning of OLC with database ID *olc_id*.
Msg_Finish_Decommissioning	Remove all OLCs from the database that are in state ReadyForDeletion.
Msg_Retry_failed_OLCs	Get an OLC from the database where the *CommissioningState* is an error state.

Thread Messages

Msg_ScanAgain

The local database of the SC is queried to return the first OLC where the *CommissioningState* is equal to *Update* or *CommissioningError*. When no OLC is found, then the thread message *Msg_Start_SWU* will be scheduled.
When an OLC is found then the *ActionType* is examined, see figure 12. If the *ActionType* is equal to *NoAction* then the *CommissioningState* is set to *Ready*, so the OLC will not have a pending action on the next *Msg_ScanAgain* execution. If the *ActionType* is equal to *CalenderUpdate*, then the *ActionType* and *CommissioningState* is set to *NoAction* and *Ready*, respectively. When the *ActionType* is either *OLCReplacement*, *ActionUpdate* or *ActionCommissioning* then the procedure to commission the OLC will be started, and finally if the *ActionType* is *ActionDecommissioning* then the procedure to decommission the OLC will be started. A new thread message *Msg_ScanAgain* will be scheduled to check if the current OLC still has a pending action or else, if another OLC has a pending action.

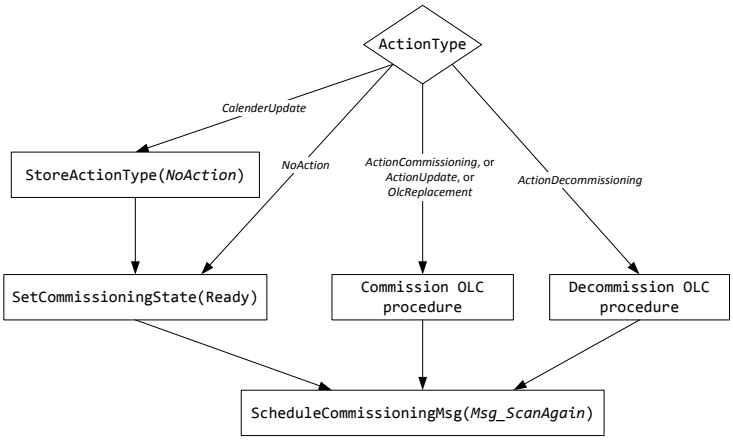


Figure 12: Process OLC with pending action

Commission OLC procedure

Figure 13 shows the commission OLC procedure. For clarity purposes some details have been left out.
A *GetInfo* message is sent to the OLC to determine the software version, short address, PAN ID and *CommState* of the OLC. With the response message of the OLC it is determined whether the software of the OLC should be upgraded and by using the *CommState* it is determined whether the OLC is already commissioned. When the OLC is already commissioned and the software of the OLC needs to be upgraded then *CommissioningState* is set to *ReadyForSWU* and the *ActionType* remains the same. When the software of the OLC is up to date then the commissioning procedure is finished and the *ActionType* is set to *NoAction*.
When the OLC is not commissioned, then messages to configure settings on the

OLC and messages to commission the OLC are sent followed by a message that instructs the OLC to reboot.

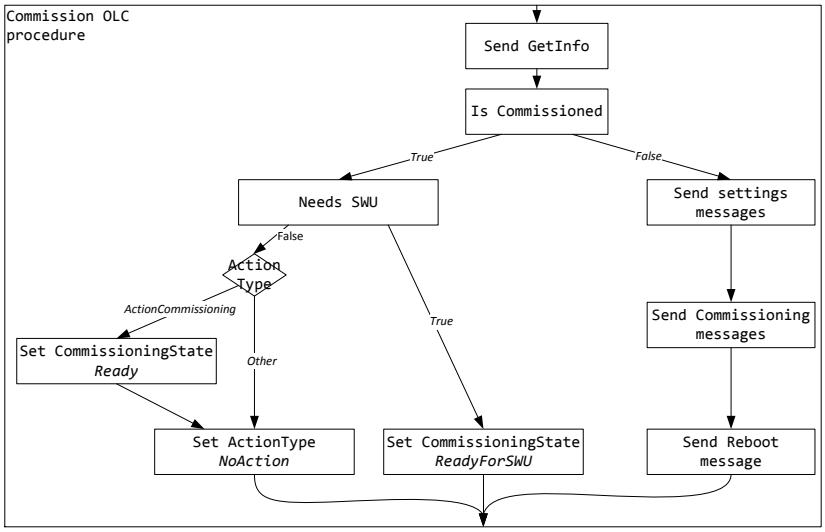


Figure 13: Commission OLC procedure

Decommission OLC procedure

The decommission OLC procedure is shown in figure 14, for clarity purposes some details have been left out.
A *GetInfo* message is sent to the OLC to determine if the OLC is commissioned. When the OLC is commissioned, the *CommissioningState* is set to *Decommissioning* and a separate thread is called to complete the log of the OLC. If after the log completion the *CommissioningState* is still equal to *Decommissioning* then thread message *Msg_Continue_Decommissioning(oldId)* will be scheduled.
When the OLC is not commissioned the *CommissioningState* is set to *Decommissioned* and a separate thread is called to report the log of the OLC. Consequently the *CommissioningState* is set to *ReadyForDeletion* and the thread message *Msg_Finish_Decommissioning* will be scheduled.

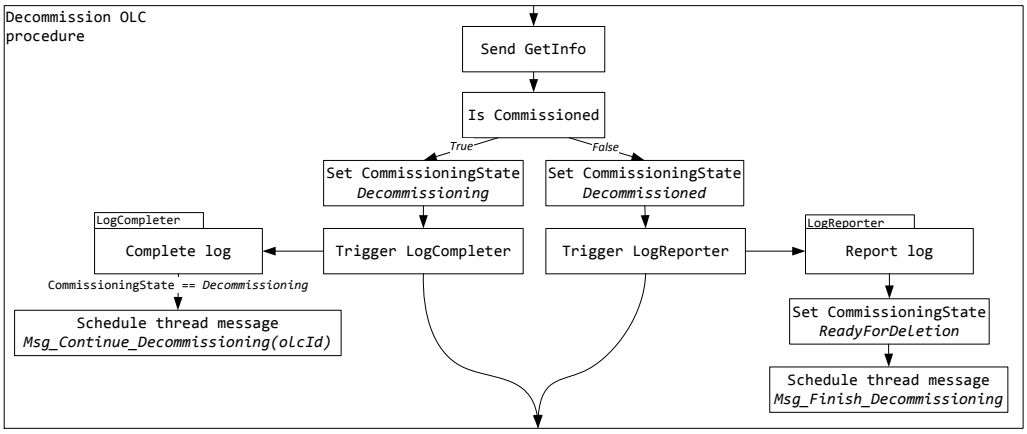


Figure 14: Decommission OLC procedure

Msg_Start_SWU

An OLC has two software images where it can boot from. The following procedure is executed to see if any of the OLCs available in the local database need to perform an SWU, which is subsequently executed when necessary.
The local database of the SC is queried for all OLCs with the *CommissioningState* equal to *ReadyForSWUIncomplete*, which indicates a software upgrade for the whole segment. If the database does not contain any OLC with the specified *CommissioningState*, then the database is queried for all OLCs with the *CommissioningState* equal to *ReadyForSWU*.

First a check is done whether the segment is located in a switched grid configuration (*i.e.,* the OLCs are powerless during the day), if so, then the procedure will wait until after sunset before continuing, because then the grid is powered.

Each OLC in the retrieved database list is prepared for the main image to be updated and a *GetInfo* message is sent to update the information about the OLC in the SC and to see if any of the OLCs in the list are reachable. The SWU component will now be triggered to upgrade all the OLCs in the list, and the procedure will wait until the SWU is finished. The SWU component will be triggered again to update the fallback image of all the OLCs in the list and again the procedure will wait until the SWU is finished. When the main image of the OLC was not correctly upgraded, then again an SWU will be started for the main image. After waiting on the SWU component to finish, each OLC is checked if both image upgrades were successful, when both upgrades were successful, then a schedule distribution is triggered to supply the OLC with the lighting schedules else the *ActionType* of the OLC is set to *ActionCommissioning*, to retry the OLC for SWU.

Msg_Continue_Decommissioning(*olcId*)
When during handling a *Msg_ScanAgain* an OLC needs to be decommissioned, then first the component LogCompleter is triggered to make the logging history of the OLC complete. When the LogCompleter component finishes, it triggers the interface of the Commissioning component to schedule the *Msg_Continue_Decommissioning*.

Here the actual decommissioning of the OLC is done, using the supplied *olcId* the OLC information is retrieved from the database, messages to decommission and reboot the OLC are sent. Consequently the *CommissioningState* is set to *Decommissioned* and the LogReporter thread is triggered, after which the *CommissioningState* is set to *ReadyForDeletion* and the thread message *Msg_Finish_Decommissioning* is scheduled.

Msg_Finish_Decommissioning
Remove all OLCs from the database that are in state *ReadyForDeletion*.

Msg_Retry_failed_OLCs
Set the *CommissioningState* to *Update* of all OLCs that currently have a *CommissioningState* that is considered to be an error state, *i.e.,* the members *Error*, *WaitForReboot*, *WaitForRebootAndSWU*, *ReadyForSWUIncompleteReady*, *ReadyForSWU*, *IncompleteSWU* or *CommissioningError*.

5.1.4 Outdoor Luminance Controller Commissioning component

State variables
The OLC Commissioning component has the following three state variables which denote the state of the Commissioning component: *CommState*, *RFNwk_CommissionFlag* (*CommissionFlag*) and *CommissionEngineState* (*SEC_NCE*).

The *CommState* indicates in which commissioning state the OLC is currently in. The *CommState* structure is present in the OLC and the SC software. The SC can request the *CommState* of an OLC by sending a *GetInfo* message to that OLC. In table 16 the different *CommState* states are explained.

Table 16: *CommState*

State	Description
COMM_STATE_UNCOMMISSIONED	OLC is not commissioned.
COMM_STATE_COMMISSIONED_NEEDS_REBOOT	OLC has been commissioned but needs to be rebooted to finish the process.
COMM_STATE_COMMISSIONED	OLC is commissioned.
COMM_STATE_DECOMMISSIONING	OLC needs to be rebooted to finish decommissioning.

The *CommissionFlag* state is the only commissioning state that is saved in non-volatile memory. This state indicates if the OLC contains valid commissioning information (short address, PAN ID and network security information). After the OLC is successfully commissioned the

CommissionFlag is set to COMMISSION_VALID and after decommissioning the *CommissionFlag* is set to COMMISSION_INVALID. During reboot it is again validated whether the necessary variables are valid. When the OLC has been rebooted while it was busy decommissioning then the state is equal to COMMISSION_BUSY_DECOM (see table 17).

Table 17: *CommissionFlag*

State	Description
COMMISSION_VALID	All necessary variables to be commissioned are valid.
COMMISSION_BUSY_DECOM	Busy with decommissioning.
COMMISSION_INVALID	Not all necessary variables to be commissioned are valid.

The *SEC_NCE* states shown in table 18 are used to indicate the state of the secure commissioning process. The secure commissioning process consists of two messages, when the *SEC_NCE* state is CE_STATE_NOT_COMMISSIONED then the OLC can receive the first secure commissioning message. When the first message has been received, the *SEC_NCE* state is set to CE_STATE_HANDSHAKE_DONE. After the second message has been received and the payload has been sent to be decrypted, the state is set to CE_STATE_DECRYPT_PENDING. Finally after the decryption is done and the commissioning information is stored, then the *SEC_NCE* state is set to CE_STATE_COMMISSIONED.

Table 18: *SEC_NCE*

State	Description
CE_STATE_NOT_COMMISSIONED	OLC is not commissioned.
CE_STATE_HANDSHAKE_DONE	Secure commissioning message 0 has been received.
CE_STATE_DECRYPT_PENDING	Secure commissioning message 1 has been received, waiting for decryption of the payload.
CE_STATE_COMMISSIONED	OLC is commissioned.

Initialisation
The *CommState* and the *SEC_NCE* variables are stored in the volatile memory and thus need to be determined during boot-up. The *CommissionFlag* is stored in non-volatile memory and is used together with the commissioning information to determine the state of the *CommState* and the *SEC_NCE* state variables.

If the commissioning information is incomplete or the *CommissionFlag* is not equal to COMMISSION_VALID, then the OLC is considered uncommissioned, the commissioning information is invalidated and the *CommState* and *SEC_NCE* are set to COMM_STATE_UNCOMMISSIONED and CE_STATE_NOT_COMMISSIONED, respectively. In case the commissioning information is valid and the *CommissionFlag* is equal to COMMISSION_VALID then the *CommState* and *SEC_NCE* are set to COMM_STATE_COMMISSIONED and CE_STATE_COMMISSIONED respectively. In case *CommissionFlag* is equal to COMMISSION_BUSY_DECOM then the OLC was rebooted during decommissioning and decommissioning will continue after initialisation.

Message handling
In table 19 the messages that are handled by the the Commissioning component are shown.

Next to these messages also *GetInfo* and *GetMac* messages are used in the (de)commissioning procedure, these messages are handled by a different software component and are used to query information from the OLC.

Table 19: Commissioning messages

Msg	Description
CommissioningMsg0	First secure commissioning message.
CommissioningMsg1	Second secure commissioning message.
DecommissioningMsg	Decommissioning message.
SecureReboot	Message to reboot the OLC.

CommissioningMsg0
With this message the first step of the commissioning process is executed, the result will be sent back to the SC. When the message is successfully handled, then the *SEC_NCE* will be set to CE_STATE_HANDSHAKE_DONE. The message will only be handled if the OLC is not commissioned.

CommissioningMsg1
This message contains the commissioning information. The payload of the message is encrypted and the OLC will request a separate component to decrypt the message, the *SEC_NCE* is set to CE_STATE_DECRYPT_PENDING. Decrypting is done asynchronously and no response to the SC will yet be sent. Once the decryption is finished, the Commissioning component will be signaled. When decryption was successful then the message contents are retrieved and stored in the non-volatile memory, *SEC_NCE* and *CommState* are set to CE_STATE_COMMISSIONED and the COMM_STATE_COMMISSIONED_NEEDS_REBOOT respectively. The OLC needs to reboot to apply the newly received settings. The *CommissioningMsg1* message is only handled if the OLC is not commissioned and the *SEC_NCE* is equal to CE_STATE_HANDSHAKE_DONE.

DecommissioningMsg
This message is used to decommission the OLC. Upon handling the message, the states *CommissionFlag* and *CommState* are set to COMMISSION_INVALID and COMM_STATE_DECOMMISSIONING respectively. The OLC can still receive messages to its short address as the volatile memory is not reset, therefore a reboot is necessary to finish the decommissioning. The *DecommissioningMsg* message is only handled if the OLC is commissioned.

SecureReboot
This message is used to reboot the OLC after a (de)commissioning procedure, to load the newly stored or erased settings from the non-volatile into the volatile memory. See previous paragraph about initialisation, how commissioning info and state variables are determined.

GetMac
This message is used to query the IEEE address from the OLC. If the OLC is commissioned, then it will only respond if the message is addressed to its short address, and if the OLC is not commissioned then the OLC will only respond if the message is addressed to its IEEE address. This message is therefore also used by the SC to determine whether or not the OLC

is commissioned.

GetInfo
The *GetInfo* message is used to query information about the software version, short address, PAN ID and *CommState* of the OLC. When an OLC receives a *GetInfo* message, it will always reply, independently of whether the message is addressed to its IEEE address or short address and the OLC is commissioned or not.

5.2 Modeling

The SC and OLC Commissioning components are modeled using the mCRL2 modeling language [13, 14]. In the following section it is explained how the model relates to the implemented Commissioning components.

5.2.1 Simplifications

In order to be able to automatically analyse the model some complexity is abstracted away. The aim is to include those variables of the C implementation that have influence on messages sent by the SC and the commissioning state of the OLC. All state variables and messages discussed in section 5.1 are included in the model. In this way it can be investigated whether or not situations can occur such that an OLC cannot be (de)commissioned (see section 5.3).

The underlying network layer is simplified. The fact that a message travels through a mesh network via different paths is not included. The unreliable network is modeled in such a way that a sent message is either communicated successfully or dropped.
The older software versions of the OLC use a different protocol for commissioning, this is not included in the model.

5.2.2 The OLC Commissioning model

The OLC Commissioning model can perform the external actions shown in figure 15 and table 20.

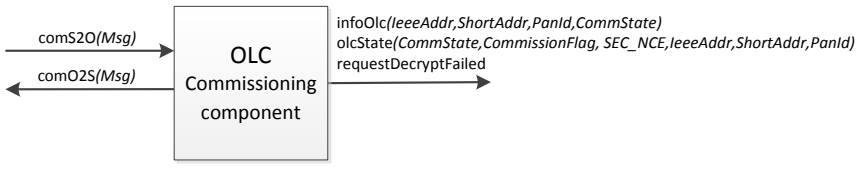


Figure 15: The OLC commissioning model with external actions

The only interactions with the OLC is via messages. The OLC does not send any messages on its own initiative. When it receives a message from the SC it will send a response message after processing the command. When the OLC is not processing a message the action *olcState* can be performed. This is used to verify the requirement stated in section 5.3.1. After boot-up and upon receiving a *GetInfo* message, the action *infoOlc* is performed. When the decryption

Table 20: External actions of the OLC commissioning model

Action	Description
infoOlc(*ieee*,*short*,*pan*,*commState*)	Shows information about the *IeeeAddr*, *ShortAddr*, *PanId* and *CommState* of the OLC, and is executed on boot-up and upon receiving a GetInfo message.
olcState(*commState*,*commissionFlag*, *SEC_NCE*,*ieee*,*short*,*pan*)	Shows the state information of the OLC and can be executed when the OLC is not processing a message.
requestDecryptFailed	*CommissioningMsg1* has been received, decrypting failed.

of *CommissioningMsg1* fails, action *requestDecryptFailed* is performed. These actions are used to verify the requirements stated in section 5.3.2.

5.2.3 The SC Commissioning model

In figure 16 the SC Commissioning model is shown, including the CommissioningStore, Database and Commissioning component. The external actions of these components are explained in tables 21, 22 and 23.

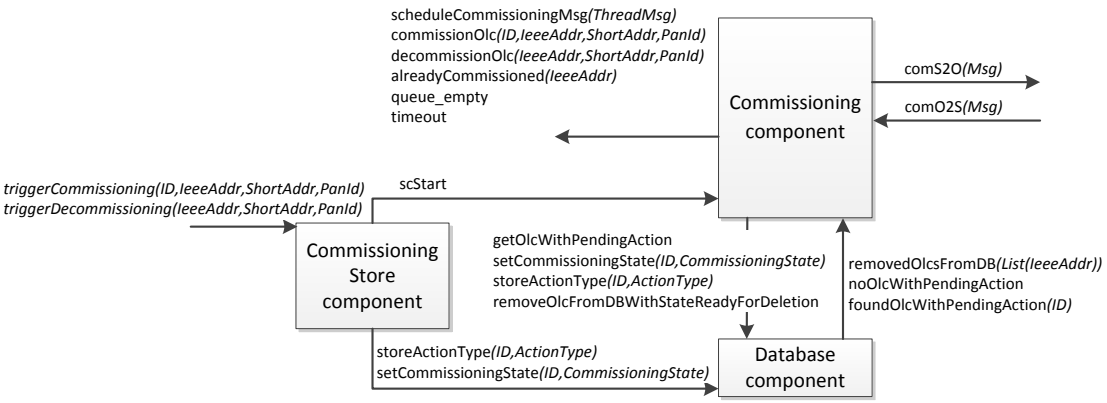


Figure 16: The SC commissioning model with external actions

Table 21: External actions of the SC CommissioningStore component

Action	Description
triggerCommissioning(*id*,*ieee*,*short*,*pan*)	Instruct the SC to commission the OLC with *ID id* and *IeeeAddr ieee* to *ShortAddr short* and *PanId pan*.
triggerDecommissioning(*ieee*,*short*,*pan*)	Instruct the SC to decommission the OLC with *IeeeAddr ieee*, *ShortAddr short* and *PanId pan*.

A request from CityTouch to commission or decommission an OLC is simulated by performing the CommissioningStore actions *triggerCommissioning* and *triggerDecommissioning*, respectively. This will trigger the Database component actions *storeActionType* and *setCommissioningState* to set the appropriate database values for that OLC. Subsequently it will

perform the action *scStart* of the Commissioning component, to schedule the *Msg_ScanAgain ThreadMsg*.

Table 22: External actions of the SC Database component

Action	**Description**
setCommissioningState(*id*,*commState*)	Set the *CommissionState* of the OLC with database identifier *id* to *commState*.
storeActionType(*id*,*actionType*)	Set the *ActionType* of the OLC with database identifier *id* to *actionType*.
removeOlcWStateReadyForDeletion	Instruct the database to remove all OLCs that have the *CommissionState ReadyForDeletion*.
removedOlcsFromDB(*ieees*)	All OLCs with *IeeeAddr* $\in$ *ieees* have been removed from the database.
getOlcWithPendingAction	Request the database for the next OLC that has a pending action.
noOlcWithPendingAction	The database does not contain any OLCs with a pending action.
foundOlcWithPendingAction(*id*)	Found OLC, with database identifier *id*, that has a pending action.

When the Commissioning component starts processing the request it will query for an OLC with a pending request by performing action *getOlcWithPendingAction*. Followed by the actions *commissionOlc* or *decommissionOlc* to indicate that the (de)commissioning has started. During this process the *ActionType* and *CommissioningState* of the OLC can be updated. When an OLC needs to be commissioned, but the OLC already is commissioned, then action *alreadyCommissioned* will be performed.

Table 23: External actions of the SC Commissioning component

Action	**Description**
scStart	Starts the SC commissioning component by scheduling a *Msg_ScanAgain ThreadMsg*.
scheduleCommissioningMsg(*msg*)	Indicates that *ThreadMsg msg* has been scheduled in the thread queue.
commissionOlc(*id*,*ieee*,*short*,*pan*)	Indicates that the commissioning of OLC with database identifier *id* and *IeeeAddr ieee* has started, the OLC will be commissioned to *ShortAddr short* and *PanId pan*.
decommissionOlc(*ieee*,*short*,*pan*)	Indicates that the decommissioning of OLC with *IeeeAddr ieee*, *ShortAddr short* and *PanId pan* has started.
alreadyCommissioned(*ieee*)	OLC with *IeeeAddr ieee* is already commissioned.
queue_empty	Indicates that the thread message queue is empty.
timeout	Indicates that a communication timeout has occurred.

After successful decommissioning an OLC, the OLC will be removed from the database, indicated by actions *removeOlcFromDBWithStateReadyForDeletion* and *removedOlcsFromDB*. When the Commissioning component does not have any *ThreadMsg* to process, it will perform the action *queue_empty*. When the Commissioning component does not receive an expected response message from an OLC then action *timeout* will be performed.

5.2.4 The model of the interacting SC and OLC components

The model of the interacting SC and OLC components is created by connecting the communication actions *comS2O* and *comO2S* of the SC and OLC commissioning models with a Net process, see figure 17. The SC and the OLC commissioning models communicate with each other via the Net process, where the Net process creates the synchronous communication between the SC and OLC processes.

The Net process can either drop a received message or send it to the other party. When a message is dropped then the action *timeout* is performed in the SC Commissioning model. The communication actions of the SC and the OLCs including the actions that cause the messages to be dropped are explained in table 24. The external actions of the SC and OLC commissioning models are explained in the previous sections.

Table 24: Communication actions of the interacting commissioning components

External Action	**Description**
comS2O(*Msg*)	Forwarding messages from the SC to the OLC.
comS2Odrop(*Msg*)	A message from the SC to the OLC is dropped before it arrived the OLC.
comO2S(*Msg*)	Forwarding messages from the OLC to the SC.
comO2Sdrop(*Msg*)	A message from the OLC to the SC is dropped before it arrived to the SC.

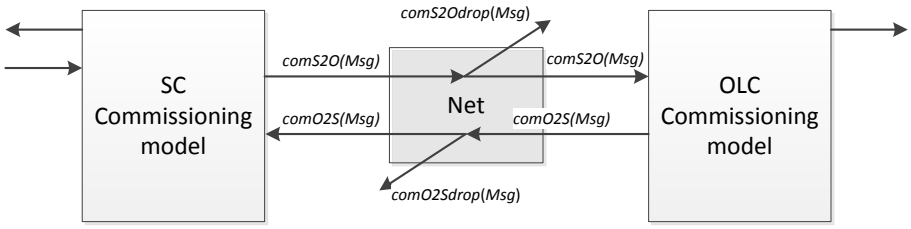


Figure 17: The model of the interacting SC and OLC commissioning components

5.3 Analysis

In this section the requirements that the model OLC commissioning component and the model of the interacting SC and OLC components must comply with are stated in section 5.3.1 and section 5.3.2, respectively. First the requirement is stated, followed by the method of verification and the result.

5.3.1 Requirements on the OLC commissioning component

R01 An OLC should not be able to arrive in a state in which one of the three commissioning states variables indicate that the OLC is commissioned while the *ShortAddr* or *PanId* is invalid:

$$
\begin{aligned}
&(\mathit{CommState} = \textsc{Comm\_State\_Commissioned}\\
&\vee\ \mathit{CommissionFlag} = \textsc{Commission\_Valid}\\
&\vee\ \mathit{SEC\_NCE} = \textsc{Ce\_State\_Commissioned}\\
&) \wedge (\mathit{ShortAddr} = \mathit{ShortNull} \vee \mathit{PanId} = \mathit{PanNull})
\end{aligned}
$$

This is verified by renaming the action *olcState(...)* to *illegal* when the proposition above holds, using a rename file (`rename_illegal_state.rename`, included in appendix I), see below for the rename statement.

$$
\begin{aligned}
&\mathit{rename}\\
&\quad((\mathit{ocs} == \textsc{Comm\_State\_Commissioned}\\
&\quad\ \vee \mathit{cf} == \textsc{Commission\_Valid}\\
&\quad\ \vee \mathit{ces} == \textsc{Ce\_State\_Commissioned}\\
&\quad\ ) \wedge (\mathit{short} == \mathit{ShortNull} \vee \mathit{pan} == \mathit{PanNull}))\\
&\quad\ \ -> \mathit{olcState}(\mathit{ocs}, \mathit{ces}, \mathit{cf}, \mathit{ieee}, \mathit{short}, \mathit{pan}) => \mathit{illegal};
\end{aligned}
$$

Followed by a search for the existence of the action *illegal* in the state space. If the action does not exist in the state space then the requirement holds.
The commands in table 25 are executed to verify this requirement. In (1) the OLC Commissioning component model (appendix D) is linearized. In (2) the action *olcState*(*CommState*, *CommissionFlag*, *SEC_NCE*, *IeeeAddr*, *ShortAddr*, *PanId*) is renamed to action *illegal* when the proposition above holds on the state variables of the action *olcState*(...), by supplying a rename file. Followed by (3) which searches the state space for all occurrences of the action *illegal*. No occurrences of action *illegal* have been found upon executing the commands.

Table 25: The commands used to verify requirement R01

	Tool	Arguments
(1)	`mcrl22lps`	`model.mcrl2 model.lps`
(2)	`lpsrename`	`model.lps model_renamed.lps --renamefile=rename/rename_illegal_state.rename`
(3)	`lps2lts`	`model_renamed.lps --action=illegal --trace`

R02 An OLC cannot arrive in a state from which it cannot be commissioned or decommissioned again.

The action *IsCommissioned*(*Bool*) will indicate whether the OLC is commissioned. To check

this property, the system can be observed by considering *IsCommissioned*(*Bool*) as the external behaviour and all other actions as internal behaviour. This is done by generating the state space of the model (appendix D) with all actions other than *olcIsCommissioned*(...) renamed to τ actions and *olcIsCommissioned*(*ieee*, *short*, *pan*, *b*) renamed to the simplified action *IsCommissioned*(*b*). Because all internal actions are renamed to the same action name, namely τ, the state space can be reduced. If the stated requirement holds then no internal behaviour will prevent the system from doing a *IsCommissioned*(*true*) or *IsCommissioned*(*false*) action, as is shown in figure 18. If the requirement does not hold, then there is a path caused by a τ action that prevents the system from doing the required behaviour.

This property has been checked by executing the commands in table 26, which results in the state space shown in figure 18 and thus the requirement holds.

Table 26: The commands used to verify the requirement

	Tool	**Arguments**
(1)	`mcrl22lps`	`model.mcrl2 model.lps`
(2)	`lpsrename`	`model.lps model_hidden.lps --renamefile=rename/hide_actions_for_R02.rename`
(3)	`lps2lts`	`model_hidden.lps model_hidden.aut`
(4)	`ltsconvert`	`model_hidden.aut model_red.aut --equivalence=branching-bisim`
(5)	`ltsgraph`	`model_red.aut`

Using (1) the model (appendix D) is linearized. In (2) all actions other than *olcIsCommissioned*(...) are hidden and *olcIsCommissioned*(*ieee*, *short*, *pan*, *b*) is renamed to the simplified action *IsCommissioned*(*b*), by supplying a rename file (`list_of_actions_to_hide.rename`, included in appendix J) that contains a list all actions other than *startSWU* and *finished* that need to be renamed to τ actions. Followed by (3) the state space generation and (4) reducing the state space (modulo branching bisimilarity). Finally (5) the state space is visualised.

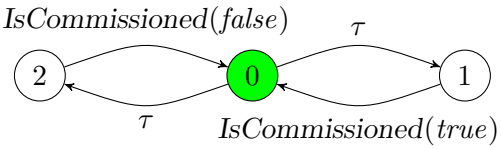


Figure 18: An OLC cannot arrive in a state from which it cannot be (de)commissioned

5.3.2 Requirements on the interacting commissioning components

The requirements which the interacting commissioning components must comply with are stated below. Each requirement is first stated in a way such that a software engineer can easily relate the requirement to the implementation (1), in (2) the requirement is stated in terms of actions and finally the modal formula, which is used to verify the requirement is stated in (3).

R03 An OLC with a pending action should eventually be processed.

(1) Each time the thread message *Msg_ScanAgain* is handled, the database is checked for the first OLC that has a pending action. After processing the OLC, the thread message *Msg_ScanAgain* is scheduled again. If the OLC still has a pending action it will be processed again upon handling the next thread message *Msg_ScanAgain*. It should not be the case that always the same OLC is served, never processing the other OLCs with pending actions.

(2) When a thread message *Msg_ScanAgain* is scheduled by *scheduleCommissioningMsg*(*Msg_ScanAgain*) then eventually no OLCs should be present in the database, indicated by action *noOlcWithPendingAction*.

(3) $[true^\star \cdot scheduleCommissioningMsg(Msg\_ScanAgain)]$
$(\mu X.[\neg noOlcWithPendingAction]X \wedge <true>true)$

The commands in table 27 are executed for each modal formula of the requirements. Using (1) the mCRL2 specification (appendix E) is linearized. In (2) the linearized mCRL2 specification together with the modal formula is converted to a parameterised boolean equation system (PBES), by supplying the formula as a `.mcf` file. Finally (3) the PBES is solved. When the formula holds, (3) will yield *true*. When the formula does not hold, then a counter example (a tree of instantiated variables) can be outputted using argument `--counter`, from which the invalidating trace can be constructed.

Table 27: The commands used to verify the requirements

	Tool	**Arguments**
(1)	`mcrl22lps`	`model.mcrl2 model.lps`
(2)	`lps2pbes`	`model.lps model_formula.pbes --formula=formula.mcf --verbose`
(3)	`pbes2bool`	`model_formula.pbes --counter --verbose`

Requirement R03 does not hold on the model. A trace was found which shows that the same OLC can be processed infinitely often, causing starvation of all other OLCs with pending actions. This can occur when an OLC has *ActionType ActionCommissioning* and *CommissioningState Update*, when the OLC is picked up during processing of the *ThreadMsg Msg_ScanAgain* then commissioning of the OLC will be started. If commissioning fails, then the *CommissioningState* will be set to *CommissioningError* and a *Msg_ScanAgain* is scheduled. Upon processing the *Msg_ScanAgain* message, the same OLC is picked up, because action *getOlcWithPendingAction* will return the first OLC that has either *CommissioningState Update* or *CommissioningError*. When the OLC for some reason always fails to be commissioned then the process will be infinitely repeated, the error would be detected eventually in the CityTouch management system.
Solution: *getOlcWithPendingAction* should only return the first OLC that has *CommissioningState Update* and neglect OLCs with *CommissioningState CommissioningError*.

R04 An OLC with a pending commissioning action should eventually be commissioned.

(1) Each OLC record DBr in the database where DBr.*ActionType* is equal to *ActionCommissioning*, *ActionUpdate* or *OLCReplacement* must eventually have the following variable values, where DBr.*IeeeAddress* = OLC.*IeeeAddress*.

OLC.*CommState* = COMM_STATE_COMMISSIONED
OLC.*ShortAddress* = DBr.*ShortAddress* (where DBr.*ShortAddress*≠ *ShortNull*)
OLC.*PanID* = DBr.*PanID* (where DBr.*PanID*≠ *PanNull*)
DBr.*ActionType* = *NoAction*
DBr.*CommissioningState* = *Ready*

When commissioning fails because of an error, such as a communication timeout or decryption error, then the OLC must be retried for commissioning.

(2) When the CommissioningStore is triggered to commission an OLC indicated by action *triggerCommissioning*(*id*, *ieee*, *short*, *pan*), and the commissioning is not canceled by *triggerDecommissioning*(*ieee*, *short*, *pan*) before commissioning is started, indicated by action *commissionOlc*(*id*, *ieee*, *short*, *pan*), then, if the OLC is not yet commissioned, the actions *setCommissioningState*(*id*,*Ready*), *storeActionType*(*id*, *NoAction*) and *infoOlc*(*ieee*,*short*,*pan*,COMM_STATE_COMMISSIONED) must eventually follow, in any order. If the OLC is already commissioned, then the action *alreadyCommissioned*(*ieee*) must follow eventually. If and only if an error occurs, such as *requestDecryptFailed* or *timeout*, then the OLC must be retried for commissioning, *i.e., commissionOlc*(*id*,*ieee*, *short*,*pan*) must take place. Infinitely many *queue_empty* actions are allowed to happen in between. The variables *b1* to *b7* record whether a particular action has taken place, see table 28.

(3)
$$
\begin{aligned}
&[\mathit{true}^\star]\\
&\forall \mathit{id}{:}\mathit{ID}, \mathit{ieee}{:}\mathit{IeeeAddr}, \mathit{short}{:}\mathit{ShortAddr}, \mathit{pan}{:}\mathit{PanId}.\\
&[\mathit{triggerCommissioning}(\mathit{id}, \mathit{ieee}, \mathit{short}, \mathit{pan}).\\
&\neg \mathit{triggerDecommissioning}(\mathit{ieee}, \mathit{short}, \mathit{pan})^\star.\\
&\mathit{commissionOlc}(\mathit{id}, \mathit{ieee}, \mathit{short}, \mathit{pan})]\\
&\mu X(b1{:}\mathbb{B}{=}\mathit{false}, b2{:}\mathbb{B}{=}\mathit{false}, b3{:}\mathbb{B}{=}\mathit{false}, b4{:}\mathbb{B}{=}\mathit{false},\\
&\quad b5{:}\mathbb{B}{=}\mathit{false}, b6{:}\mathbb{B}{=}\mathit{false}, b7{:}\mathbb{B}{=}\mathit{false}).\nu Y.\\
&(\\
&\quad ((b1 \wedge b2 \wedge b3)\\
&\quad \vee b4\\
&\quad \vee ((b5 \vee b6) \wedge b7)) \vee\\
&\quad ([\mathit{setCommissioningState}(\mathit{id}, \mathit{Ready})]X(\mathit{true}, b2, b3, b4, b5, b6, b7)\\
&\quad \wedge [\mathit{storeActionType}(\mathit{id}, \mathit{NoAction})]X(b1, \mathit{true}, b3, b4, b5, b6, b7)\\
&\quad \wedge [\mathit{infoOlc}(\mathit{ieee}, \mathit{short}, \mathit{pan}, \text{COMM\_STATE\_COMMISSIONED})]X(b1, b2, \mathit{true}, b4, b5, b6, b7)\\
&\quad \wedge [\mathit{alreadyCommissioned}(\mathit{ieee})]X(b1, b2, b3, \mathit{true}, b5, b6, b7)\\
&\quad \wedge [\mathit{requestDecryptFailed}]X(b1, b2, b3, b4, \mathit{true}, b6, b7)\\
&\quad \wedge [\mathit{timeout}]X(b1, b2, b3, b4, b5, \mathit{true}, b7)\\
&\quad \wedge [\mathit{commissionOlc}(\mathit{id}, \mathit{ieee}, \mathit{short}, \mathit{pan})]X(b1, b2, b3, b4, b5, b6, \mathit{true})\\
&\quad \wedge [\neg \mathit{setCommissioningState}(\mathit{id}, \mathit{Ready})\\
&\qquad \cap \neg \mathit{storeActionType}(\mathit{id}, \mathit{NoAction})\\
&\qquad \cap \neg \mathit{infoOlc}(\mathit{ieee}, \mathit{short}, \mathit{pan}, \text{COMM\_STATE\_COMMISSIONED})
\end{aligned}
$$

$$
\begin{aligned}
&\cap \neg alreadyCommissioned(ieee)\\
&\cap \neg requestDecryptFailed\\
&\cap \neg timeout\\
&\cap \neg commissionOlc(id, ieee, short, pan)\\
&\cap \neg triggerCommissioning(id, ieee, short, pan)\\
&\cap \neg triggerDecommissioning(ieee, short, pan)\\
&\cap \neg queue\_empty]X(b1, b2, b3, b4, b5, b6, b7)\\
\wedge\ &[queue\_empty]Y\\
\wedge\ &<\neg queue\_empty>true\\
)\\
)
\end{aligned}
$$

Table 28: Boolean variables record observed actions of R04

Variable	Action
b1	*setCommissioningState*(*id*,*Ready*)
b2	*storeActionType*(*id*,*NoAction*)
b3	*infoOlc*(*ieee*, *short*, *pan*,COMM_STATE_COMMISSIONED)
b4	*alreadyCommissioned*(*ieee*)
b5	*requestDecryptFailed*
b6	*timeout*
b7	*commissionOlc*(*id*, *ieee*, *short*, *pan*)

Requirement R04 does hold on the model.

R05 An OLC with a pending decommissioning action should eventually be decommissioned.

(1) Each OLC record DBr in the database where DBr.*ActionType* is equal to *ActionDecommissioning* must eventually be removed from the database and the OLC variables must be equal to:

OLC.*CommState* = COMM_STATE_DECOMMISSIONED
OLC.*ShortAddress* = *ShortNull*
OLC.*PanID* = *PanNull*

Unless the OLC is commissioned with an OLC.*ShortAddress* or OLC.*PanID* different from the database record DBr.*ShortAddress* and DBr.*PanID*.

(2) When the CommissioningStore is triggered to decommission an OLC indicated by action *triggerDecommissioning*(*ieee*, *short*, *pan*), and the decommissioning is not canceled by *triggerCommissioning*(*id*, *ieee*, *short*, *pan*) before decommissioning is started, indicated by action *decommissionOlc*(ieee,short,pan). Then, if the OLC is commissioned with *ShortAddr* = *short* $\wedge$ *PanId* = *pan* or the OLC is not commissioned, *removedOlcsFromDB*(*ieees*) and *infoOlc*(*ieee*,*ShortNull*,*PanNull*,COMM_STATE_UNCOMMISSIONED) should eventually follow, in any order. If the OLC is commissioned with *ShortAddr* $\neq$ *short* $\vee$ *PanId* $\neq$ *pan*, then action *infoOlc*(*ieee*,*s*,*p*,COMM_STATE_COMMISSIONED) should eventually follow, where $s \neq short$ and $p \neq pan$. Infinitely many

queue_empty actions are allowed to happen in between. The variables *b1* to *b5* record whether a particular action has taken place, see table 29.

(3) $[true^\star]$
$\forall ieee{:}IeeeAddr, short{:}ShortAddr, pan{:}PanId.$
$[triggerDecommissioning(ieee, short, pan).$
$\neg\exists id{:}ID.triggerCommissioning(id, ieee, short, pan)^\star$
$decommissionOlc(ieee, short, pan)]$
$\mu X(b1{:}\mathbb{B} = false, b2{:}\mathbb{B} = false, b3{:}\mathbb{B} = false, b4{:}\mathbb{B} = false, b5{:}\mathbb{B} = false).\nu Y.$
(
$((b1 \wedge b2)$
$\vee\, b3$
$\vee\, (b4 \wedge b5)) \vee$
$([\exists ieees{:}List(IeeeAddr).(removedOlcsFromDB(ieees) \wedge ieee \in ieees)]X(true, b2, b3, b4, b5)$
$\wedge\, [infoOlc(ieee, ShortNull, PanNull, \textsc{Comm\_State\_Uncommissioned})]X(b1, true, b3, b4, b5)$
$\wedge\, [\exists s{:}ShortAddr, p{:}PanId.(short \neq s \vee pan \neq p)$
$\wedge\, infoOlc(ieee, s, p, \textsc{Comm\_State\_Commissioned})]X(b1, b2, true, b4, b5)$
$\wedge\, [timeout]X(b1, b2, b3, true, b5)$
$\wedge\, [decommissionOlc(ieee, short, pan)]X(b1, b2, b3, b4, true)$
$\wedge\, [\neg(\exists ieees{:}List(IeeeAddr).removedOlcsFromDB(ieees) \wedge ieee \in ieees)$
$\cap\, \neg infoOlc(ieee, ShortNull, PanNull, \textsc{Comm\_State\_Uncommissioned})$
$\cap\, \neg(\exists s{:}ShortAddr, p{:}PanId.(short \neq s \vee pan \neq p)$
$\wedge\, infoOlc(ieee, s, p, \textsc{Comm\_State\_Commissioned}))$
$\cap\, \neg timeout$
$\cap\, \neg decommissionOlc(ieee, short, pan)$
$\cap\, \neg triggerDecommissioning(ieee, short, pan)$
$\cap\, \neg\exists id{:}ID.triggerCommissioning(id, ieee, short, pan)$
$\cap\, \neg(queue\_empty]X(b1, b2, b3, b4, b5)$
$\wedge\, [queue\_empty]Y$
$\wedge\, <\neg queue\_empty>true$
)
)

Table 29: Boolean variables record observed actions of R05

Variable	**Action**
b1	$removedOlcsFromDB(ieees) \wedge ieee \in ieees$
b2	$infoOlc(ieee, ShortNull, PanNull, \textsc{Comm\_State\_Uncommissioned})$
b3	$infoOlc(ieee, s, p, \textsc{Comm\_State\_Commissioned}) \wedge (short \neq s \vee pan \neq p)$
b4	*timeout*
b5	$decommissionOlc(ieee, short1, pan1)$

Requirement R05 does not hold on the model. Table 30 shows the trace which invalidates the formula, where the last action can be done infinitely (indicated by $\star$). In this trace a communication timeout occurred, causing the OLC not to be decommissioned and the thread message queue is empty and thus decommissioning of the OLC is not retried. Retry of decommissioning will be done only after an external interface of the commissioning component

schedules a new thread message.
Solution: Reschedule *ThreadMsg Msg_Continue_Decommissioning* when decommissioning failed, so that decommissioning will be retried.

Table 30: Trace not complying to R05

Action
infoOlc(*Ieee1, Short1, Pan1,* COMM_STATE_COMMISSIONED)
scheduleCommissioningMsg(*Msg_ScanAgain*)
getOlcWithPendingAction
foundOlcWithPendingAction(*OLC1*)
decommissionOlc(*Ieee1, Short1, Pan1*)
infoOlc(*Ieee1, Short1, Pan1*, COMM_STATE_COMMISSIONED)
setCommissioningState(*OLC1, Decommissioning*)
scheduleCommissioningMsg(*Msg_ScanAgain*)
getOlcWithPendingAction
noOlcWithPendingAction
scheduleCommissioningMsg(*Msg_Start_SWU*)
scheduleCommissioningMsg(*Msg_Continue_Decommissioning(OLC1)*)
infoOlc(*Ieee1, Short1, Pan1*, COMM_STATE_COMMISSIONED)
timeout
infoOlc(*Ieee1, Short1, Pan1*, COMM_STATE_COMMISSIONED)
queue_empty*

The requirements have been verified on a model with a single OLC and where the CommissionStore is only allowed to do two *triggerCommissioning*/*triggerDecommissioning* actions. The CommissionStore can set all possible combinations of the variables *ActionType* and *CommissioningState* in the OLC database record.
Verification of the requirements is done using the system and toolkit as described in chapter 6.

6 System and Toolkit

The system and toolkit used is described below:

System:

CPU	: 56x Intel(R) Xeon(R) CPU E5520 2.27GHz
RAM	: 935GB RAM (aggregated)
Operating System	: Fedora release 12 (Constantine)
Linux kernel	: Linux version 2.6.27.44-6.vSMP

Toolkit:

mCRL2 toolset	: 201310.0.12197M (Release)

7 Conclusion

In this report the implementation, modeling and analysis of two interacting SC and OLC components have been discussed. The SWU components, where the SC is in charge of upgrading the software of the OLCs which are part of its segment. And the Commissioning components, where the SC needs to contact and configure an uncommissioned OLC that needs to become part of the network segment. The C implementation of the components have been studied and abstract models have been created of that implementation. With these models, the questions posed in the introduction have been answered.

During the analysis of the SWU models it has been verified that when the communication is reliable then all OLCs will always be successfully upgraded. In case communication is unreliable, then situations can occur in which a single dropped message can cause the whole segment of OLCs fail to be upgraded. Also situations have been found in which the SC causes a great deal of unnecessary traffic to be generated. For instance, there exists a situation where the SC receives no response from any OLC during the initialisation phase and still continues the SWU process, sending all 1500 SWU packages into the network. This is interesting because the SWU process takes a long time, so reducing the amount of messages generated is an improvement for the system.

During the analysis of the Commissioning models it has been discovered that a starvation situation can occur, where continuously the same (failing) OLC is being served causing other OLCs not to be (de)commissioned. Also a situation has been found in which an OLC that fails to decommission is not automatically retried for decommissioning. Further analysis revealed that the cause of these problems and their solutions appeared to be straightforward.

Although the process creating abstract models out of existing software systems is a slow and tedious process, it does provide a better understanding of the system and with it we have been successful in gaining better insights and finding possible improvements in the implemented software components which were not discovered using conventional testing methods. When formal methods are used in the design phase of the software development instead, then formal techniques could deliver higher quality code, reduction in number of errors and an increase in development productivity [2].

References

[1] Starsense. (n.d.). Philips Lighting. Retrieved June 16, 2014, from http://www.lighting.philips.com/main/products/controls/outdoor/starsense/

[2] Osaiweran, A.A.H., Groote, J.F., Schuts, M.T.W., Hooman, J.J.M. & Rijnsoever, B.J. van (2012). Evaluating the effect of formal techniques in industry. (External Report, Computer Science Report, No. 12-13). Eindhoven: Technische Universiteit Eindhoven, 21 pp.

[3] Feo-Arenis, S., Westphal, B., Dietsch, D., Muiz, M., Andisha, S. The Wireless Fire Alarm System: Ensuring Conformance to Industrial Standards through Formal Verification. FM 2014.

[4] Sijtema, Marten and Stoelinga, Marille and Belinfante, Axel and Marinelli, Lawrence (2011) Experiences with formal engineering: model-based specification, implementation and testing of a software bus at Neopost. In: 16th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2011, 29-30 August 2011, Trento, Italy (pp. pp. 117-133).

[5] G.J. Holzmann (2014). Mars Code. Communications of the ACM, Vol. 57, No. 2, Feb. 2014, pp. 64-73.

[6] L. Shan, Y. Wang, N. Fu, X. Zhou, L. Zhao, L. Wan, L. Qiao, J. Chen (2014). Formal Verification of Lunar Rover Control Software Using UPPAAL. FM 2014, pages 718-732.

[7] H. Zhao, M. Yang, N. Zhan, B. Gu, L. Zou, Y. Chen (2014). Formal Verification of a Descent Guidance Control Program of a Lunar Lander. FM 2014, pages 733-748.

[8] C. DaSilva, B. Dehbonei, F. Mejia. Formal specification in the development of industrial applications: subway speed control system. In: Proceedings 5th IFIP Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE92), Perros-Guirec, North-Holland, pp. 199213 (1993)

[9] P. Behm, P. Benoit, A. Faivre, J.-M. Meynadier (1999). METEOR: a successful application of B in a large project. FM 1999. LNCS, vol. 1708, pp. 369387. Springer, Heidelberg (1999)

[10] F. Badeau. Using B as a high level programming language in an industrial project: Roissy val. In Proceedings of ZB05, 2005

[11] J. R. Abrial. Formal methods in industry: achievements, problems, future. ICSE '06: Proceeding of the 28th international conference on Software engineering, page 761-768. New York, NY, USA, ACM Press, (2006)

[12] Wijst, Berry (2011). Starsense Wireless. Internal document. Starsense Wireless For External.PPT

[13] J.F. Groote, A. Mathijssen, M.A. Reniers, Y.S. Usenko, M.J. van Weerdenburg. Analysis of distributed systems with mCRL2. In M. Alexander, W. Gardner (Eds.), Process Algebra for Parallel and Distributed Processing, pages 99-128. Chapman and Hall, 2008.

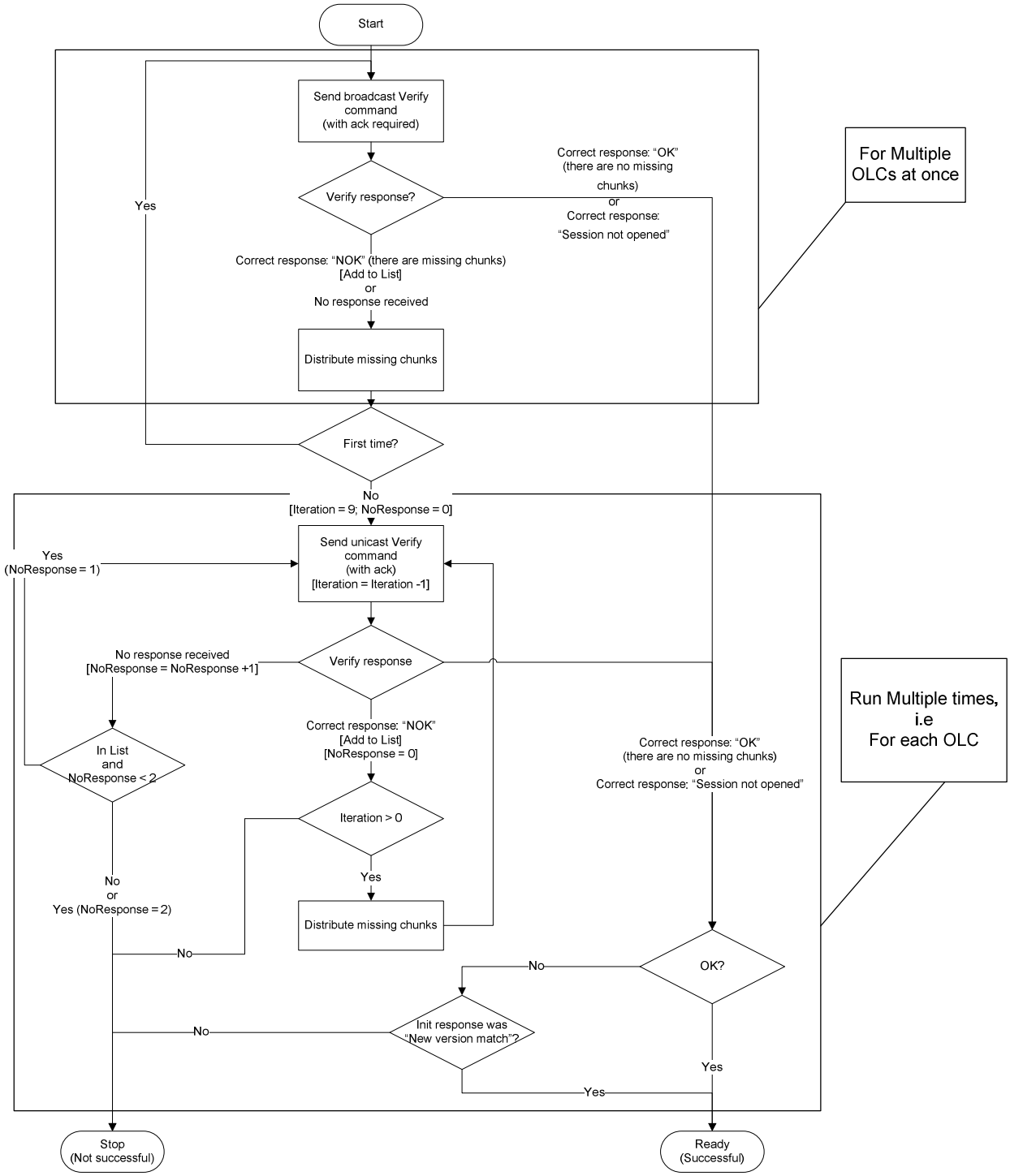
[14] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, J. W. Wesselink, T.A.C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. TACAS 2013, pages 199-213.

[15] J.F. Groote, M.R. Mousavi. Modeling and Analysis of Communicating Systems. MIT-press. 2014.

[16] mCRL2 Homepage. Retrieved June 26, 2014, from http://www.mcrl2.org/

8 Appendix

A Verify process flowchart



B Software Upgrade process mCRL2 specification

Filename: `A_SWUpgrade.mcrl2`
Description: This file contains the mCRL2 specification of the Software Upgrade components, both the Segment Controller Software Upgrade Component and the Outdoor Luminair Controller. These models are combined by a network model, which can lose messages.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Model of the Software Upgrade Components                                                                     %
% Model includes comments taken from the C implementation                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% The Distribution Info struct DstrInfo has 2 elements, getChunkId and chunks
% getChunkId is pointing to the last checked chunk that is missing.
% The chunks  element is a list of ChunkInfo structs
% a ChunkInfo consists of a data element and a boolean missing, indicating if it needs to be distributed.
% Edit elements of DstrInfo struct
map getNextMissingChunkId:   DstrInfo -> Nat;
    getNextMissingChunk:     DstrInfo -> Data;
    curChunkDistributed:     DstrInfo -> DstrInfo;
    setMissingChunks:        List(Nat) # DstrInfo -> DstrInfo;
    setMissingChunk:         Nat # DstrInfo -> DstrInfo;
    setAllChunksMissing:     DstrInfo -> DstrInfo;
    setAllChunksMissing':    Nat # DstrInfo -> DstrInfo;
    editDstrInfo_getChunkId: Nat # DstrInfo -> DstrInfo;
var di:DstrInfo,i:Nat,l:List(Nat);
eqn % Get check all chunks in the list until there is one missing, return that id
    (getChunkId(di) >= #chunks(di))  -> getNextMissingChunkId(di) =  #chunks(di);
    !(getChunkId(di) >= #chunks(di)) -> getNextMissingChunkId(di)
                                     = if(missing(chunks(di).getChunkId(di)), getChunkId(di),
                                          getNextMissingChunkId(Dstr_Info(getChunkId(di)+1, chunks(di))));

    % Get check all chunks in the list until there is one missing, return the data of that chunk
    (getChunkId(di) >= #chunks(di))  -> getNextMissingChunk(di) = D1;
    !(getChunkId(di) >= #chunks(di)) -> getNextMissingChunk(di)
                                     = if(missing(chunks(di).getChunkId(di)),
                                          data(chunks(di).getChunkId(di)),
                                          getNextMissingChunk(Dstr_Info(getChunkId(di)+1, chunks(di))));

    % Flag the current chunk as NON missing (CHUNK_OK = false)
    curChunkDistributed(di) = Dstr_Info(getChunkId(di)+1,setMissingInList(false, getChunkId(di),chunks(di)));

    % Set chunk with id = i as missing (CHUNK_MISSING = true)
    (l==[])  -> setMissingChunks(l, di) = di;
    !(l==[]) -> setMissingChunks(l, di) = setMissingChunks(tail(l), setMissingChunk(head(l), di));
    setMissingChunk(i, di) =  Dstr_Info(getChunkId(di), setMissingInList(true, i, chunks(di)));

    % All chunks missing (CHUNK_MISSING = true)
    setAllChunksMissing(di) = setAllChunksMissing'(0,di);
    (i >= #chunks(di))  -> setAllChunksMissing'(i,di) = di;
    !(i >= #chunks(di)) -> setAllChunksMissing'(i,di)
                        = setAllChunksMissing'(i+1,Dstr_Info(getChunkId(di),
                                                             setMissingInList(true, i,chunks(di))));

    % Edit the getChunkId element
    editDstrInfo_getChunkId(i,di) = Dstr_Info(i, chunks(di));

% Flag missing in ChunkInfo list
map setMissingInList: Bool#Nat#List(ChunkInfo) -> List(ChunkInfo);
var lci:List(ChunkInfo),ci:ChunkInfo,b:Bool,i:Nat;
eqn i == 0 -> setMissingInList(b,i,lci)     = Chunk_Info(data(head(lci)),b) |> tail(lci);
    i > 0  -> setMissingInList(b,i,ci|>lci) = ci |> setMissingInList(b,Int2Nat(i-1),lci);

% Edit elements of VfyInfo struct
map editVfyInfo_timesVfyBcast: Nat # VfyInfo -> VfyInfo;
    editVfyInfo_olcIndex: Nat # VfyInfo -> VfyInfo;
    editVfyInfo_noResponse: Nat # VfyInfo -> VfyInfo;
    editVfyInfo_iterationUcast: Nat # VfyInfo -> VfyInfo;
var vi:VfyInfo, i:Nat;
eqn editVfyInfo_timesVfyBcast(i,vi) = Vfy_Info(i, iterationUcast(vi), noResponse(vi), olcIndex(vi));
    editVfyInfo_iterationUcast(i,vi) = Vfy_Info(timesVfyBcast(vi), i, noResponse(vi), olcIndex(vi));
    editVfyInfo_noResponse(i,vi) = Vfy_Info(timesVfyBcast(vi), iterationUcast(vi), i, olcIndex(vi));
    editVfyInfo_olcIndex(i,vi) = Vfy_Info(timesVfyBcast(vi), iterationUcast(vi), noResponse(vi), i);

% Edit elements of ReqInfo struct
map editReqInfo_rebootDelay: BcastRebootDelay # ReqInfo -> ReqInfo;
```

```
    editReqInfo_bulkOutTimeOut: Bool # ReqInfo -> ReqInfo;
    editReqInfo_nrchunks: Nat # ReqInfo -> ReqInfo;
    editReqInfo_nrchunks': Nat # Nat # ReqInfo -> ReqInfo;
    editReqInfo_UpdateOLC_RSI_Status: Status # ID # ReqInfo -> ReqInfo;
    editReqInfo_UpdateOLC_LOO_Status: Status # ID # ReqInfo -> ReqInfo;
    editReqInfo_UpdateOLCreceived: Bool # ID # ReqInfo -> ReqInfo;
    editReqInfo_UpdateOLCnrchunks: Nat # ID # ReqInfo -> ReqInfo;
    allNewStatus: ReqInfo -> Bool;                                    % listOfOlcs[j].status
    S_OlcAppStatus_OkExists: ReqInfo -> Bool;                         % listOfOlcs[j].status
    getIndexOfOLC:  ID # ReqInfo -> Nat;
    resetReceivedFlag: Bool # ReqInfo -> ReqInfo;
var brd:BcastRebootDelay, ri:ReqInfo, b:Bool, c,i:Nat, s:Status,id:ID;      % listOfOlcs[j].status
eqn editReqInfo_rebootDelay(brd,ri) = Req_Info(brd, forceUpgrade(ri), oldVer(ri), newVer(ri),
                                               isAutomatic(ri), OLCs(ri), bulkOutTimeOut(ri),
                                               bulkOutResponsedNumOLC(ri));
    editReqInfo_bulkOutTimeOut(b,ri) = Req_Info(rebootDelay(ri), forceUpgrade(ri), oldVer(ri),
                                                newVer(ri), isAutomatic(ri), OLCs(ri), b,
                                                bulkOutResponsedNumOLC(ri));
    % This functions sets in the ReqInfo struct for every OLC in the OLCs list the number of chunks
    editReqInfo_nrchunks(c,ri) = editReqInfo_nrchunks'(c,0,ri);
    (i >= #OLCs(ri))  -> editReqInfo_nrchunks'(c,i,ri) = ri;
    !(i >= #OLCs(ri)) -> editReqInfo_nrchunks'(c,i,ri)
                         = editReqInfo_nrchunks'(c,i+1,Req_Info(rebootDelay(ri), forceUpgrade(ri),
                                                  oldVer(ri), newVer(ri), isAutomatic(ri),
                                                  insert_nrchunks(i,c,OLCs(ri)), bulkOutTimeOut(ri),
                                                  bulkOutResponsedNumOLC(ri)));
    editReqInfo_UpdateOLC_RSI_Status(s,id,ri)
     = Req_Info(rebootDelay(ri), forceUpgrade(ri), oldVer(ri), newVer(ri),
                isAutomatic(ri), updateOLC_RSI_Status(s,id,OLCs(ri)), bulkOutTimeOut(ri),
                bulkOutResponsedNumOLC(ri));

    editReqInfo_UpdateOLC_LOO_Status(s,id,ri)
     = Req_Info(rebootDelay(ri), forceUpgrade(ri), oldVer(ri), newVer(ri),
                isAutomatic(ri), updateOLC_LOO_Status(s,id,OLCs(ri)), bulkOutTimeOut(ri),
                bulkOutResponsedNumOLC(ri));

    editReqInfo_UpdateOLCreceived(b,id,ri)
     = Req_Info(rebootDelay(ri), forceUpgrade(ri), oldVer(ri), newVer(ri),
                isAutomatic(ri), updateOLCreceived(b,id,OLCs(ri)), bulkOutTimeOut(ri),
                bulkOutResponsedNumOLC(ri));
    editReqInfo_UpdateOLCnrchunks(c,id,ri)
     = Req_Info(rebootDelay(ri), forceUpgrade(ri), oldVer(ri), newVer(ri),
                isAutomatic(ri), updateOLCnrchunks(c,id,OLCs(ri)), bulkOutTimeOut(ri),
                bulkOutResponsedNumOLC(ri));


    % Is there an OLC that responded, and has status S_OlcAppStatus_Ok? Then allNewStatus == false
    allNewStatus(ri) = allNewStatusOI(OLCs(ri),true);
    S_OlcAppStatus_OkExists(ri) = S_OlcAppStatus_OkExistsOI(OLCs(ri),false);
    getIndexOfOLC(id,ri) = getIndexOfOLCOI(id,OLCs(ri),0);
    resetReceivedFlag(b,ri) = Req_Info(rebootDelay(ri), forceUpgrade(ri), oldVer(ri), newVer(ri),
                               isAutomatic(ri), resetReceivedFlagOI(#OLCs(ri),b,OLCs(ri)), bulkOutTimeOut(ri),
                               bulkOutResponsedNumOLC(ri));

% set all nrchunks (missing) in OlcInfo list
map insert_nrchunks: Nat#Nat#List(OlcInfo) -> List(OlcInfo);
    updateOLC_LOO_Status: Status#ID#List(OlcInfo) -> List(OlcInfo);
    updateOLC_RSI_Status: Status#ID#List(OlcInfo) -> List(OlcInfo);
    updateOLCreceived: Bool#ID#List(OlcInfo) -> List(OlcInfo);
    updateOLCnrchunks: Nat#ID#List(OlcInfo) -> List(OlcInfo);
    allNewStatusOI: List(OlcInfo) # Bool -> Bool;
    S_OlcAppStatus_OkExistsOI: List(OlcInfo) # Bool -> Bool;
    getIndexOfOLCOI:  ID # List(OlcInfo) # Nat -> Nat;
    resetReceivedFlagOI: Nat # Bool # List(OlcInfo) -> List(OlcInfo);
var loi:List(OlcInfo),oi:OlcInfo,i:Nat,nrchunks:Nat,s:Status,id:ID, received:Bool, b:Bool;
eqn (i==0)-> insert_nrchunks(i,nrchunks,loi)
             = Olc_Info(id(head(loi)), LOOstatus(head(loi)), received(head(loi)), nrchunks,
                        RSIstatus(head(loi))) |> tail(loi);
    (i>0) -> insert_nrchunks(i,nrchunks,oi|>loi) = oi |> insert_nrchunks(Int2Nat(i-1),nrchunks,loi);
```

```
    updateOLC_LOO_Status(s,id,oi|>loi)
     = if(id(oi)==id, Olc_Info(id(oi), s, received(oi), nrchunks(oi), RSIstatus(oi)) |> loi,
                      oi|>updateOLC_LOO_Status(s,id,loi));

    updateOLC_RSI_Status(s,id,oi|>loi)
     = if(id(oi)==id, Olc_Info(id(oi), LOOstatus(oi), received(oi), nrchunks(oi), s) |> loi,
                      oi|>updateOLC_RSI_Status(s,id,loi));

    updateOLCreceived(received,id,oi|>loi)
     = if(id(oi)==id, Olc_Info(id(oi), LOOstatus(oi), received, nrchunks(oi), RSIstatus(oi)) |> loi,
                      oi|>updateOLCreceived(received,id,loi));
    updateOLCnrchunks(nrchunks,id,oi|>loi)
     = if(id(oi)==id, Olc_Info(id(oi), LOOstatus(oi), received(oi), nrchunks, RSIstatus(oi)) |> loi,
                      oi|>updateOLCnrchunks(nrchunks,id,loi));

    (loi==[])->  allNewStatusOI(loi,b) = b;
    !(loi==[])-> allNewStatusOI(loi,b) = if((received(head(loi)) && LOOstatus(head(loi))==S_OlcAppStatus_Ok),
                                        false, allNewStatusOI(tail(loi),b));
    (loi==[])-> S_OlcAppStatus_OkExistsOI(loi,b) = b;
    !(loi==[])-> S_OlcAppStatus_OkExistsOI(loi,b) = if((LOOstatus(head(loi))==S_OlcAppStatus_Ok),
                                                  true, S_OlcAppStatus_OkExistsOI(tail(loi),b));

    (id(loi.i)==id) -> getIndexOfOLCOI(id,loi,i) = i;
    !(id(loi.i)==id) -> getIndexOfOLCOI(id,loi,i) = getIndexOfOLCOI(id,loi,i+1);

    (i==0)-> resetReceivedFlagOI(i,b,loi) = [];
    (i>0)-> resetReceivedFlagOI(i,b,oi|>loi) = Olc_Info(id(oi), LOOstatus(oi), b, nrchunks(oi),
                                           RSIstatus(oi)) |> resetReceivedFlagOI(Int2Nat(i-1),b,loi);


% Get the oposite image type
map otherImg: ImgType -> ImgType;
var i:ImgType;
eqn otherImg(i) = if(IsNORMAL_IMAGE(i), FALLBACK_IMAGE, NORMAL_IMAGE);

sort
    State =
    struct  SWU_IDLE                                          ?IsSWU_IDLE
         |  SWU_PREPARE                                       ?IsSWU_PREPARE
         |  SWU_BCAST_REBOOT_WITHOUT_ACK                      ?IsSWU_BCAST_REBOOT_WITHOUT_ACK
         |  SWU_BCAST_REBOOT_WITHOUT_ACK_VERIFICATION_DONE    ?IsSWU_BCAST_REBOOT_WITHOUT_ACK_VERIFICATION_DONE
         |  SWU_UNICAST_REBOOT                                ?IsSWU_UNICAST_REBOOT
         |  SWU_UNICAST_REBOOT_VERIFICATION_DONE              ?IsSWU_UNICAST_REBOOT_VERIFICATION_DONE
         |  SWU_BCAST_INIT_WITH_ACK                           ?IsSWU_BCAST_INIT_WITH_ACK
         |  SWU_WAITING_FOR_INIT_ACK                          ?IsSWU_WAITING_FOR_INIT_ACK
         |  SWU_BCAST_INIT_WITHOUT_ACK                        ?IsSWU_BCAST_INIT_WITHOUT_ACK
         |  SWU_UNICAST_INIT                                  ?IsSWU_UNICAST_INIT
         |  SWU_DISTRIBUTE                                    ?IsSWU_DISTRIBUTE
         |  SWU_VERIFY                                        ?IsSWU_VERIFY
         |  SWU_WAIT_REBOOT                                   ?IsSWU_WAIT_REBOOT;

    VerifyState = struct  VERIFY_BROADCAST           ?IsVERIFY_BROADCAST
                        |  VERIFY_WAITFORBULKOUT      ?IsVERIFY_WAITFORBULKOUT
                        |  VERIFY_UNICAST             ?IsVERIFY_UNICAST;

    % Different types of messages
    MsgType = struct Msg_REBOOT(forceImg:ImgType, code12NC:Hw12Nc)?IsMsg_REBOOT
                   | Msg_INIT(newVers:Vers, code12NC:Hw12Nc, startAddr:ImgType)?IsMsg_INIT
                   | Msg_DATA(addrFlash:Nat, data:Data)?IsMsg_DATA
                   | Msg_VFY?IsMsg_VFY
                   | Msg_VFY_RESP(status:Status, missingChunks:List(Nat))?IsMsg_VFY_RESP
                   | Msg_GEN_RESP(status:Status, cmd:Command)?IsMsg_GEN_RESP;   %general response
    Command = struct REBOOT?IsREBOOT | INIT?IsINIT | DATA?IsDATA | VFY?IsVFY;

    % Broadcast and Unicast messages
    Msg     = struct MSG(des:MsgDes, msg:MsgType)?IsMSG
                   | MSG_UcastResp(src:ID, msg:MsgType)?IsMSG_UcastResp
                   | MSG_BcastResp(src:ID, msg:MsgType)?IsMSG_BcastResp;

    MsgDes = struct Ucast(id: ID)?IsUcast
```

```
                | BcastWithAck?IsBcastWithAck
                | BcastWithoutAck?IsBcastWithoutAck;

    ID  = struct OLC1 | OLC2 | OLC3;

    % Images, a node has 2 images, a NORMAL and FALLBACK
    ImgType = struct NORMAL_IMAGE?IsNORMAL_IMAGE
                | FALLBACK_IMAGE?IsFALLBACK_IMAGE;
    Hw12Nc  = struct hw12nc1;
    Vers    = struct V1 | V2 | V3;
    Data    = struct D1; % | D2;
    Status  = struct S_OlcAppStatus_Ok
                | S_OlcAppStatus_InvalidAddress
                | S_OlcAppStatus_12NcMismatch
                | S_OlcAppStatus_SessionClosed
                | ERR_MSG_TIMEOUT
                | S_OlcAppStatus_NewVersionMatch;

                ChunkInfo = struct Chunk_Info(data:Data, missing:Bool);

    % Note, nrchunks is added in here from the implementation struct listOfOlcs[]
    OlcInfo = struct Olc_Info(id:ID, LOOstatus:Status, received:Bool, nrchunks:Nat, RSIstatus:Status);

    % requestedSWUInfo struct
    ReqInfo = struct Req_Info_Null
                | Req_Info(rebootDelay:BcastRebootDelay, forceUpgrade:ImgType, oldVer:Vers,
                           newVer:Vers, isAutomatic:Bool, OLCs:List(OlcInfo), bulkOutTimeOut:Bool,
                           bulkOutResponsedNumOLC:Nat);
    % VerifyInfo struct
    VfyInfo = struct Vfy_Info(timesVfyBcast:Nat, iterationUcast:Nat, noResponse:Nat, olcIndex:Nat);

    % Distribute struct, actual chunks are also added to this struct. If a chunk is missing is added to this
    % struct. RequestedSWUInfo.DistributeInfo.totalChunks same as #chunks
    DstrInfo = struct Dstr_Info(getChunkId:Nat, chunks:List(ChunkInfo));

    % VerifyResponse, missing chunks is included in ChunkInfo, in DstrInfo struct
    VfyRsp = struct Vfy_Rsp(nrofchunkids:Nat);

    % Update Image struct, this is the image we update to
    UpdImg  = struct Upd_Img_Null | Upd_Img(imgType:ImgType, hw12nc:Hw12Nc);

    % The two reboot delays, first=5min second=1min, then responding OLCs should reboot around same time
    BcastRebootDelay = struct Delay_First | Delay_Second;

act
    receiveResp: Msg;
    st:State;
    stv: VerifyState;
    handlePrepareState, handleBcastRebootwithoutAck, handleUnicastReboot, handleBcastInitwithAck;
    handleBcastInitwithoutAck, handleUnicastInit, distributeChunk, handleVerifySubStateMachine;
    reportStatus: List(OlcInfo);
    waitingForInitAck, verifyFinished, numOlcIsSmall, numOlcIsLarge, chunksMissing;
    sendingUcastDone, stopBulkOutForInit, chunksDistributed;
    ucastRebootVfyDone, stopBulkOutForVfy, nowVerifyUcast, againVerifyBcast, nextOLC;
    verifyOlcOK: ID # Bool;
    timeout, timeout_sc, olcReceivedAndStatusOKexists, olcReceivedAndStatusOKdoesNotExists,olcStatusOKexists,
    olcStatusOKdoesNotExists, finished, upgradeNotOK, case4, startThread, idle, startSWU;
    scUpgradeOK,upgradeOK: ID # Bool;

proc
    % Init, create SWUpgrade thread
    A_SWUpgrade_Init
    = startThread.
      P(SWU_IDLE,VERIFY_BROADCAST,Req_Info_Null,Upd_Img_Null,Vfy_Info(0,0,0,0),0,Dstr_Info(0,[]),Vfy_Rsp(0));

    A_SWUpgrade_StartSoftwareUpgrade
    = startSWU.
      P(SWU_PREPARE,
        VERIFY_BROADCAST,
        Req_Info(Delay_First,NORMAL_IMAGE, V1, V2, true,
```

```
       %NOTE: LOO status and RSI status for an OLC have to be the same,
       % this is done in the prepare state in the implementation
      [Olc_Info(OLC1,S_OlcAppStatus_Ok,false,0,S_OlcAppStatus_Ok),
       Olc_Info(OLC2,S_OlcAppStatus_Ok,false,0,S_OlcAppStatus_Ok)],false,0),
    Upd_Img(NORMAL_IMAGE, hw12nc1),
    Vfy_Info(0,0,0,0),
    0,
    % This contains the update (chunks) and initially missing
    Dstr_Info(0,[Chunk_Info(D1,true),Chunk_Info(D1,true)]),
    Vfy_Rsp(0)
    )
;

P(s:State, vs:VerifyState, info:ReqInfo, upd_img:UpdImg, vfy_info:VfyInfo,
  ucastIterator:Nat, dstr:DstrInfo, vfy_rsp:VfyRsp)
=
( st(s). P()

+ IsSWU_IDLE(s)
  -> ( idle. P()
     + A_SWUpgrade_StartSoftwareUpgrade
     )

% 1. Prepare
+ IsSWU_PREPARE(s)
  -> handlePrepareState.
     % Initially set to true all chunks are missing.
     % Init all OLCs with all chunks missing: editReqInfo_nrchunks
     ( numOlcIsLarge.
       P(s=SWU_BCAST_REBOOT_WITHOUT_ACK,vs=VERIFY_BROADCAST,info=editReqInfo_nrchunks(#chunks(dstr),info),
         vfy_info=editVfyInfo_timesVfyBcast(0,vfy_info), dstr=setAllChunksMissing(dstr),
         vfy_rsp=Vfy_Rsp(#chunks(dstr)))
     + numOlcIsSmall.
       P(s=SWU_UNICAST_REBOOT,vs=VERIFY_BROADCAST,
         info=editReqInfo_rebootDelay(Delay_First,editReqInfo_nrchunks(#chunks(dstr),info)),
         vfy_info=editVfyInfo_timesVfyBcast(0,vfy_info), dstr=setAllChunksMissing(dstr),
         vfy_rsp=Vfy_Rsp(#chunks(dstr)))
     )

% 2. reboot OLCs BROADCAST
+ IsSWU_BCAST_REBOOT_WITHOUT_ACK(s)
  -> % Get reboot parameters
     % Broadcast reboot
     handleBcastRebootwithoutAck.
     %requestedSWUInfo.bulkOutTimeOut = false;
     ScSend(MSG(BcastWithoutAck,Msg_REBOOT(otherImg(imgType(upd_img)), hw12nc(upd_img)))).
     ( (rebootDelay(info) == Delay_First)
        -> P(info=editReqInfo_rebootDelay(Delay_Second,editReqInfo_bulkOutTimeOut(false,info)))
        <> P(s=SWU_BCAST_INIT_WITH_ACK, info=editReqInfo_bulkOutTimeOut(false,info))
     )
     % TimerThread Wait %d sec for all reboot

% 2. reboot OLCs UNICAST
+ IsSWU_UNICAST_REBOOT(s)
  ->(% send to all OLCs, when done, go to next state
     (ucastIterator==#(OLCs(info)))
       -> sendingUcastDone.
          % TimerThread Wait %d sec for unicast reboots + NBtable fill
          P(s=SWU_UNICAST_INIT,ucastIterator=0, info=editReqInfo_bulkOutTimeOut(false,info))
       <> handleUnicastReboot.
          ScSend(MSG(Ucast(id(OLCs(info).ucastIterator)),
                 Msg_REBOOT(otherImg(imgType(upd_img)), hw12nc(upd_img)))).
          ( timeout_sc.
            P(ucastIterator=ucastIterator+1,
              info=editReqInfo_UpdateOLC_RSI_Status(ERR_MSG_TIMEOUT,id(OLCs(info).ucastIterator),info))
          + % We are waiting for a general response from the OLC that we send the unicast message to
            sum m:Msg. (IsMSG_UcastResp(m) && src(m)==id(OLCs(info).ucastIterator)
                        && IsMsg_GEN_RESP(msg(m)) && IsREBOOT(cmd(msg(m))))
             -> receiveResp(m).
                (status(msg(m)) != S_OlcAppStatus_Ok)
```

```
                -> P(ucastIterator=ucastIterator+1,
                     info=editReqInfo_UpdateOLC_RSI_Status(status(msg(m)),id(OLCs(info).ucastIterator),
                                                           info))
                <> P(ucastIterator=ucastIterator+1)
          )
    )

% 3. init OLCs
+ IsSWU_BCAST_INIT_WITH_ACK(s)
  -> handleBcastInitwithAck.
     % start bulkout
     % Get Init Parameters
     ScSend(MSG(BcastWithAck,Msg_INIT(newVer(info), hw12nc(upd_img), imgType(upd_img)))).
     % An acknowlegde must be send back (and the response handled) done in WAITFORBULKOUT
     % timeOutInfo.bulkOutTimeOut = TRUE;
     % Start timer to set a max time to wait for callbacks
     % then wait for either:
     % 1. callback from service layer on bcast ack
     % 2. callback from timer thread indicating timeout
     % wait for response (Timer thread) Wait %d sec for broadcast init acks
     P(s=SWU_WAITING_FOR_INIT_ACK, info=editReqInfo_bulkOutTimeOut(true,resetReceivedFlag(false,info)))

% 3. wait for the response
+ IsSWU_WAITING_FOR_INIT_ACK(s)
  % Start loop of unicast to un-ack'ed OLCs to init
  % Determine callback of ack or timeout timer expired and act on it)
  ->( waitingForInitAck.
     % Stop the bulkout
     (bulkOutTimeOut(info))
      -> stopBulkOutForInit.
         % Bulkout is really stopped in callback func A_SWUpgrade_OlcMsgCallBack
         % Different process for the bulkout handling
         WAITFORBULKOUT(s,vs,editReqInfo_bulkOutTimeOut(false,info),upd_img,vfy_info,ucastIterator,dstr,
                        vfy_rsp,INIT,0)
      <> ( %Not all OLCs responded
           (#(OLCs(info)) != bulkOutResponsedNumOLC(info))
            -> ( % calculate number of failed OLCs, (NOTE: No depence on amount of OLCs)
                 numOlcIsLarge.
                 P(s=SWU_BCAST_INIT_WITHOUT_ACK)
               + numOlcIsSmall.
                 P(s=SWU_UNICAST_INIT)
               )
            <> ( % All the olcs that responded, if
                 % requestedSWUInfo.DistributeInfo.getChunkId = 0;
                 % Real question is, is there an OLC that responded, and has status S_OlcAppStatus_Ok
                 % otherwise there is no point in distributing packages, the OLC either has current
                 % version or is in error state
                 (allNewStatus(info) == false)
                  -> olcReceivedAndStatusOKexists.
                     P(s=SWU_DISTRIBUTE, dstr=editDstrInfo_getChunkId(0,dstr))
                  <> % Of the ones that responded..
                     olcReceivedAndStatusOKdoesNotExists.
                     P(s=SWU_VERIFY)
               )
        )
    )

% 3. init OLCs BROADCAST, Do once more broadcast INIT, but without ack
+ IsSWU_BCAST_INIT_WITHOUT_ACK(s)
  -> handleBcastInitwithoutAck.
     % get init parameters and send broadcast
     ScSend(MSG(BcastWithoutAck,Msg_INIT(newVer(info), hw12nc(upd_img), imgType(upd_img)))).
     % NO acknowlegde is expected to be send back
     % requestedSWUInfo.DistributeInfo.getChunkId = 0;
     % wait for response (timer thread)
     P(s=SWU_DISTRIBUTE, dstr=editDstrInfo_getChunkId(0,dstr))

% 3. init OLCs UNICAST
+ IsSWU_UNICAST_INIT(s)
  -> (ucastIterator==#(OLCs(info)))
```

```
      -> ( % If OLC exists with an S_OlcAppStatus_OK (received=true is not checked here?)
           (S_OlcAppStatus_OkExists(info))
            -> olcStatusOKexists.
               P(s=SWU_DISTRIBUTE, ucastIterator=0, dstr=editDstrInfo_getChunkId(0,dstr))
            <> olcStatusOKdoesNotExist.
               P(s=SWU_VERIFY, ucastIterator=0)
         )
      <> ( handleUnicastInit.
           % Get init parameters
           (received(OLCs(info).ucastIterator) == false)
            -> ScSend(MSG(Ucast(id(OLCs(info).ucastIterator)),Msg_INIT(newVer(info), hw12nc(upd_img),
                                                                       imgType(upd_img)))).
               % either a timeout comes, or a message arrives
               ( timeout_sc.
                 P(ucastIterator=ucastIterator+1,
                   info=editReqInfo_UpdateOLC_RSI_Status(ERR_MSG_TIMEOUT,id(OLCs(info).ucastIterator),
                        editReqInfo_UpdateOLC_LOO_Status(S_OlcAppStatus_SessionClosed,
                                                         id(OLCs(info).ucastIterator),info)))
               + % We are waiting for a general response from the OLC that we send the unicast message to
                 % Here also the S_OlcAppStatus_NewVersionMatch response is handled
                 (sum m:Msg.  (IsMSG_UcastResp(m) && src(m)==id(OLCs(info).ucastIterator)
                               && IsMsg_GEN_RESP(msg(m)) && IsINIT(cmd(msg(m))))
                  -> receiveResp(m).
                     (status(msg(m)) != S_OlcAppStatus_Ok)
                      -> (status(msg(m)) == S_OlcAppStatus_NewVersionMatch)
                         % Received assigned TRUE and nrchunks assigned Zero to avoid SWU verification
                         -> P(ucastIterator=ucastIterator+1,
                              info=editReqInfo_UpdateOLCreceived(true,id(OLCs(info).ucastIterator),
                                   editReqInfo_UpdateOLCnrchunks(0,id(OLCs(info).ucastIterator),
                                   editReqInfo_UpdateOLC_LOO_Status(status(msg(m)),
                                                                    id(OLCs(info).ucastIterator),info))))
                         % Else just update OLC status
                         <> P(ucastIterator=ucastIterator+1,
                              info=editReqInfo_UpdateOLC_RSI_Status(status(msg(m)),
                                                                    id(OLCs(info).ucastIterator),info))
                      <> P(ucastIterator=ucastIterator+1) )
               )
            <> P(ucastIterator=ucastIterator+1)
         )

% 4. distribute image
+ IsSWU_DISTRIBUTE(s)
  -> % Get the next missing chunk id from VerifyResponse.chunkids[]
     % if we still have missing chunk ids
     (getNextMissingChunkId(dstr) < #(chunks(dstr)))
      -> distributeChunk.
         ScSend(MSG(BcastWithoutAck,Msg_DATA(getNextMissingChunkId(dstr),getNextMissingChunk(dstr)))).
         % wait 3 seconds before distributing the next chunk (timer thread)
         P(dstr=curChunkDistributed(dstr),info=editReqInfo_bulkOutTimeOut(false,info))
      <> chunksDistributed.
         P(s=SWU_VERIFY,info=editReqInfo_bulkOutTimeOut(false,info))

% 5. verify
% HandleVerifySubStateMachine
+ IsSWU_VERIFY(s)
  -> handleVerifySubStateMachine.
     VERIFY(s, vs, info, upd_img, vfy_info, ucastIterator, dstr, vfy_rsp, false)

% 6. Reboot to new image UNICAST
+ IsSWU_UNICAST_REBOOT_VERIFICATION_DONE(s)
  -> ((ucastIterator==#(OLCs(info)))
      -> ucastRebootVfyDone.
         % Timer thread Wait %d sec for unicast reboots + NBtable fill
         P(s=SWU_WAIT_REBOOT,ucastIterator=0,info=editReqInfo_bulkOutTimeOut(false,info))
      <> handleUnicastReboot.
         % Get init parameters
         ScSend(MSG(Ucast(id(OLCs(info).ucastIterator)),Msg_REBOOT(imgType(upd_img), hw12nc(upd_img)))).
         % either a timeout comes, or a message arrives
         ( timeout_sc.
           P(ucastIterator=ucastIterator+1,
```

```
            info=editReqInfo_UpdateOLC_RSI_Status(ERR_MSG_TIMEOUT,id(OLCs(info).ucastIterator),info))
        + % We are waiting for an general response from the OLC that we send the unicast message to
          sum m:Msg.  (IsMSG_UcastResp(m) && src(m)==id(OLCs(info).ucastIterator)
                       && IsMsg_GEN_RESP(msg(m)) && IsREBOOT(cmd(msg(m))))
           -> receiveResp(m).
            (status(msg(m)) != S_OlcAppStatus_Ok)
             -> P(ucastIterator=ucastIterator+1,
                  info=editReqInfo_UpdateOLC_RSI_Status(status(msg(m)),id(OLCs(info).ucastIterator),
                                                        info))
              <> P(ucastIterator=ucastIterator+1)
        )
    )

% 6. Reboot to new image BROADCAST
+ IsSWU_BCAST_REBOOT_WITHOUT_ACK_VERIFICATION_DONE(s)
  -> handleBcastRebootwithoutAck.
     ScSend(MSG(BcastWithoutAck,Msg_REBOOT(imgType(upd_img), hw12nc(upd_img)))).
     % requestedSWUInfo.bulkOutTimeOut = false;
     ( (rebootDelay(info) == Delay_First)
        -> % wait for reboot (Timer thread)
           P(info=editReqInfo_rebootDelay(Delay_Second,editReqInfo_bulkOutTimeOut(false,info)))
        <> % wait for reboot (Timer thread)
           P(s=SWU_WAIT_REBOOT,info=editReqInfo_bulkOutTimeOut(false,info))
     )

% 7. Wait for reboot to finish and report status
+ IsSWU_WAIT_REBOOT(s)
  -> %Do info dist immediately to set the clock in the rebooted OLC's
     reportStatus(OLCs(info)).
     finished.
     sum b:Bool. scUpgradeOK(OLC1,b).
     (b==false)
      -> upgradeNotOK.
         F
      <> F
);

F = a. F;

% 5. verify
% HandleVerifySubStateMachine
VERIFY(s:State, vs:VerifyState, info:ReqInfo, upd_img:UpdImg, vfy_info:VfyInfo,
  ucastIterator:Nat, dstr:DstrInfo, vfy_rsp:VfyRsp, notAllOlcsResponded:Bool)
=
( stv(vs). VERIFY()
+ IsVERIFY_BROADCAST(vs)
  -> % upgradeOp.type = S_OlcParm_SwUpgradeOperationType_Start;
     % verifyResponse.nrofchunkids = 0;
     ScSend(MSG(BcastWithAck,Msg_VFY)).
     % An acknowlegde must be send back and response is handled in WAITFORBULKOUT
     P(s,VERIFY_WAITFORBULKOUT,
       editReqInfo_bulkOutTimeOut(true,resetReceivedFlag(false,info)),upd_img,vfy_info,ucastIterator,dstr,
       Vfy_Rsp(0))

+ IsVERIFY_WAITFORBULKOUT(vs)
  -> (bulkOutTimeOut(info))
      -> stopBulkOutForVfy.
         % Bulkout is really stopped in callback func A_SWUpgrade_OlcMsgCallBack
         % Different process for the bulkout handling, (if not used can be removed all together)
         WAITFORBULKOUT(s,vs,editReqInfo_bulkOutTimeOut(false,info),upd_img,vfy_info,ucastIterator,dstr,
                        vfy_rsp,VFY,0)
         %P(s,vs,editReqInfo_bulkOutTimeOut(false,info),upd_img,vfy_info,ucastIterator,dstr,vfy_rsp)
      <> ((olcIndex(vfy_info) < #OLCs(info))
          -> ( % notAllOlcsResponded = FALSE;
               % Step 2 : Verify the OLC response from Bulk out list
               % if OLC received,
               (received(OLCs(info).olcIndex(vfy_info)))
                % Step 9 & Step 10 : Check OLC's AppStatus is S_OlcAppStatus_NewVersionMatch
                % or S_OlcAppStatus_Ok & missing chunks is zero for Successfull Upgraded OLC
                -> (LOOstatus(OLCs(info).olcIndex(vfy_info)) == S_OlcAppStatus_NewVersionMatch
```

```
                    || ((LOOstatus(OLCs(info).olcIndex(vfy_info)) == S_OlcAppStatus_Ok)
                        && nrchunks(OLCs(info).olcIndex(vfy_info)) == 0))
                     -> % Successful Upgraded OLC
                        verifyOlcOK(id(OLCs(info).olcIndex(vfy_info)), true).
                        nextOLC.
                        VERIFY(vfy_info=editVfyInfo_olcIndex(olcIndex(vfy_info)+1,vfy_info),
                               info=editReqInfo_UpdateOLC_RSI_Status(LOOstatus(OLCs(info).
                                                                               olcIndex(vfy_info)),
                                                                     id(OLCs(info).olcIndex(vfy_info)),
                                                                     info))
                     <> ((LOOstatus(OLCs(info).olcIndex(vfy_info)) == S_OlcAppStatus_Ok)
                         && nrchunks(OLCs(info).olcIndex(vfy_info)) > 0)
                         -> % Status OK but missing chunks, distribute
                            % Step 3 : Distribute the missing chunks
                            % TODO: Add more info? which OLC, which chunks
                            chunksMissing.
                            nextOLC.
                            VERIFY(s=SWU_DISTRIBUTE, dstr=editDstrInfo_getChunkId(0,dstr),
                                   vfy_info=editVfyInfo_olcIndex(olcIndex(vfy_info)+1,vfy_info))
                         <> % Step 11: Not successful in upgrading OLCs
                            % TODO: add S_OlcAppStatus_* status `s?
                            % NOTE: order of if else clauses is different
                            verifyOlcOK(id(OLCs(info).olcIndex(vfy_info)), false).
                            nextOLC.
                            VERIFY(vfy_info=editVfyInfo_olcIndex(olcIndex(vfy_info)+1,vfy_info),
                                   info=editReqInfo_UpdateOLC_RSI_Status(LOOstatus(OLCs(info).
                                                                                   olcIndex(vfy_info)),
                                                                         id(OLCs(info).olcIndex(vfy_info)),
                                                                         info))
                  <> % notAllOlcsResponded = TRUE;
                     nextOLC.
                     VERIFY(vfy_info=editVfyInfo_olcIndex(olcIndex(vfy_info)+1,vfy_info),
                            notAllOlcsResponded=true)
               )

            % Step 2 : Verify the OLC response from Bulk out list
            % If All the OLC have responsed and Missing chunk is zero
            % Successfully Upgraded.
            <> ((nrofchunkids(vfy_rsp) == 0 && notAllOlcsResponded == false)
                -> verifyFinished.
                ( numOlcIsSmall.
                  P(SWU_UNICAST_REBOOT_VERIFICATION_DONE,vs,editReqInfo_bulkOutTimeOut(false,info),upd_img,
                    vfy_info,ucastIterator,dstr,vfy_rsp)
                + numOlcIsLarge.
                  P(SWU_BCAST_REBOOT_WITHOUT_ACK_VERIFICATION_DONE,vs,
                    editReqInfo_rebootDelay(Delay_First,editReqInfo_bulkOutTimeOut(false,info)),upd_img,
                    vfy_info,ucastIterator,dstr,vfy_rsp)
                )
                <> % Step 4 : Is First time sent broadcast verify?
                   ((timesVfyBcast(vfy_info)+1)>=2)
                    -> % prepare Unicast verify
                       % requestedSWUInfo.VerifyInfo.olcIndex = 0;
                       % requestedSWUInfo.VerifyInfo.noResponse = 0;
                       % requestedSWUInfo.VerifyInfo.iterationUnicast = 0;
                       nowVerifyUcast.
                       P(s,VERIFY_UNICAST,info,upd_img,
                         editVfyInfo_iterationUcast(0,editVfyInfo_noResponse(0,
                                                    editVfyInfo_timesVfyBcast(timesVfyBcast(vfy_info)+1,
                                                    editVfyInfo_olcIndex(0,vfy_info)))),
                         ucastIterator,dstr,vfy_rsp)
                    <> againVerifyBcast.
                       P(s,VERIFY_BROADCAST,info,upd_img,
                         editVfyInfo_timesVfyBcast(timesVfyBcast(vfy_info)+1,vfy_info),
                         ucastIterator,dstr,vfy_rsp)
               )
          )
+ IsVERIFY_UNICAST(vs)
  -> (olcIndex(vfy_info) < #OLCs(info))
      -> % Unicast verify only for OLC which didnt respond to Broadcast verify
         % and OLCs which responded to Broadcast verify with more missing chunks
```

```
(((LOOstatus(OLCs(info).olcIndex(vfy_info)) == S_OlcAppStatus_Ok)
  && nrchunks(OLCs(info).olcIndex(vfy_info)) > 0)
 || (received(OLCs(info).olcIndex(vfy_info)) == false))
 -> % Step 7 : check number of Iteration per OLC
    (iterationUcast(vfy_info) < 10) % 10 = MAX_VERIFY_RETRIES
     -> % TODO: verifyResponse.nrofchunkids = 0; is updated after the response is received...
        % Step 5 : Send unicast verify command
        ScSend(MSG(Ucast(id(OLCs(info).olcIndex(vfy_info))),Msg_VFY)).
        % Step 6a: Verify Response
        ( % Step 6b : If Olc has not responsed for 2 times, olc upgrade is Not successful
          timeout_sc.
          (noResponse(vfy_info) < 1)
           -> % Step 5 : Send unicast verify command with ack
              P(SWU_VERIFY,vs,
                editReqInfo_bulkOutTimeOut(false,
                editReqInfo_UpdateOLC_RSI_Status(LOOstatus(OLCs(info).olcIndex(vfy_info)),
                                                 id(OLCs(info).olcIndex(vfy_info)),info)),
                upd_img,editVfyInfo_noResponse(noResponse(vfy_info)+1,vfy_info),ucastIterator,
                dstr,Vfy_Rsp(0))
           <> % Step 11 : no response for 2 times for unicast verify, so move olc to ERROR
              % Not successful
              verifyOlcOK(id(OLCs(info).olcIndex(vfy_info)),false).
              P(SWU_VERIFY,vs,
                editReqInfo_bulkOutTimeOut(false,
                editReqInfo_UpdateOLC_RSI_Status(ERR_MSG_TIMEOUT,id(OLCs(info).
                                                                    olcIndex(vfy_info)),info)),
                upd_img,
                editVfyInfo_olcIndex(olcIndex(vfy_info)+1,
                editVfyInfo_iterationUcast(0,
                editVfyInfo_noResponse(0,vfy_info))),ucastIterator,dstr,Vfy_Rsp(0))
        + % There was a response, reset counter (noResponse)
          % We are waiting for a vfy response message from the OLC that we send the unicast to
          sum m:Msg. (IsMSG_UcastResp(m) && src(m)==id(OLCs(info).olcIndex(vfy_info))
                   && (IsMsg_VFY_RESP(msg(m)) || (IsMsg_GEN_RESP(msg(m)) && IsVFY(cmd(msg(m))))))
           -> receiveResp(m).
              (status(msg(m)) == S_OlcAppStatus_Ok)
               -> % Step 9: Check OLC's AppStatus is S_OlcAppStatus_Ok
                  (#missingChunks(msg(m)) == 0)
                   -> % Step 12 Check OLC's AppStatus is S_OlcAppStatus_Ok & missing chunks is
                      % zero for Successfull Upgraded OLC
                      % successfull
                      verifyOlcOK(id(OLCs(info).olcIndex(vfy_info)), true).
                      P(SWU_VERIFY,vs,
                        editReqInfo_bulkOutTimeOut(false,
                        editReqInfo_UpdateOLC_RSI_Status(S_OlcAppStatus_Ok,src(m),info)),
                        upd_img,
                        editVfyInfo_olcIndex(olcIndex(vfy_info)+1,
                        editVfyInfo_iterationUcast(0,
                        editVfyInfo_noResponse(0,vfy_info))),ucastIterator,dstr,Vfy_Rsp(0))
                   <> % Step 8 : If OLC is responsed with S_OlcAppStatus_Ok and missing chunks,
                      % Distribute the missing chunks
                      P(SWU_DISTRIBUTE,vs,
                        editReqInfo_bulkOutTimeOut(false,
                        editReqInfo_UpdateOLC_RSI_Status(S_OlcAppStatus_Ok,src(m),info)),upd_img,
                        editVfyInfo_iterationUcast(iterationUcast(vfy_info)+1,
                        editVfyInfo_noResponse(0,vfy_info)),ucastIterator,
                        editDstrInfo_getChunkId(0,setMissingChunks(missingChunks(msg(m)),dstr)),
                        Vfy_Rsp(#missingChunks(msg(m))))
              <> % Step 10 Check OLC's AppStatus is not S_OlcAppStatus_Ok, then set error as
                 % S_OlcAppStatus_SessionClosed.
                 P(SWU_VERIFY,vs,
                   editReqInfo_UpdateOLC_RSI_Status(S_OlcAppStatus_SessionClosed,src(m),info),
                   upd_img,editVfyInfo_olcIndex(olcIndex(vfy_info)+1,
                   editVfyInfo_iterationUcast(0,
                   editVfyInfo_noResponse(0,vfy_info))),ucastIterator,dstr,Vfy_Rsp(0))
        )
     <> % Step 11 : Ten Iteration is not enough to upgrade a OLC, so move to ERR_MSG_TIMEOUT
        % requestedSWUInfo.OLCs[requestedSWUInfo.VerifyInfo.olcIndex].status = ERR_MSG_TIMEOUT;
        % requestedSWUInfo.VerifyInfo.noResponse       = 0;
        % requestedSWUInfo.VerifyInfo.iterationUnicast = 0;
```

```
                    % requestedSWUInfo.VerifyInfo.olcIndex++;
                    verifyOlcOK(id(OLCs(info).olcIndex(vfy_info)),false).
                    P(SWU_VERIFY,vs,
                      editReqInfo_UpdateOLC_RSI_Status(ERR_MSG_TIMEOUT,id(OLCs(info).
                                                                    olcIndex(vfy_info)),info),upd_img,
                      editVfyInfo_olcIndex(olcIndex(vfy_info)+1,
                      editVfyInfo_iterationUcast(0,editVfyInfo_noResponse(0,vfy_info))),ucastIterator,dstr,
                      Vfy_Rsp(0))
               <> % Move to next OLC if status is not S_OlcAppStatus_Ok or Received status is false
                  % requestedSWUInfo.VerifyInfo.noResponse       = 0;
                  % requestedSWUInfo.VerifyInfo.iterationUnicast = 0;
                  % requestedSWUInfo.VerifyInfo.olcIndex++;
                  P(SWU_VERIFY,vs,
                    editReqInfo_UpdateOLC_RSI_Status(LOOstatus(OLCs(info).olcIndex(vfy_info)),
                                                     id(OLCs(info).olcIndex(vfy_info)),info),upd_img,
                    editVfyInfo_olcIndex(olcIndex(vfy_info)+1,
                    editVfyInfo_iterationUcast(0,
                    editVfyInfo_noResponse(0,vfy_info))),ucastIterator,dstr,Vfy_Rsp(0))
          <> verifyFinished.
             ( numOlcIsSmall.
               P(SWU_UNICAST_REBOOT_VERIFICATION_DONE,vs,
                 editReqInfo_bulkOutTimeOut(false,info),upd_img,vfy_info,ucastIterator,dstr,vfy_rsp)
             + numOlcIsLarge.
               P(SWU_BCAST_REBOOT_WITHOUT_ACK_VERIFICATION_DONE,vs,
                 editReqInfo_rebootDelay(Delay_First,
                 editReqInfo_bulkOutTimeOut(false,info)),upd_img,vfy_info,ucastIterator,dstr,vfy_rsp)
             )
    );

act scSend, scSend_drop:Msg;
proc
    ScSend(m:Msg) = scSend(m) + scSend_drop(m);

act bulktimeout,timeout_sc_all,timeout_all;
proc
    WAITFORBULKOUT(s:State, vs:VerifyState, info:ReqInfo, upd_img:UpdImg, vfy_info:VfyInfo,
      ucastIterator:Nat, dstr:DstrInfo, vfy_rsp:VfyRsp, cmd:Command, NrOfRecRsp:Nat)
      % In S_OlcMessage_SwUpgrade_ReceiveResponse a distinction is made between a Ucast response
      % and a Bcast response, thats why here seperate channels are made for Bcast and Ucast response
      % Update: received = true, nrofchunkids (that are missing), OLC status, set the chunks as missing
    = sum m:Msg. (IsMSG_BcastResp(m) && IsMsg_VFY_RESP(msg(m)))
       % Is verify response
       -> receiveResp(m).
          WAITFORBULKOUT(info=editReqInfo_UpdateOLCreceived(true,src(m),
                              editReqInfo_UpdateOLCnrchunks(#missingChunks(msg(m)),src(m),
                              editReqInfo_UpdateOLC_LOO_Status(status(msg(m)),src(m),info))),
                         dstr=setMissingChunks(missingChunks(msg(m)),dstr),
                         vfy_rsp=Vfy_Rsp(nrofchunkids(vfy_rsp)+#missingChunks(msg(m))),NrOfRecRsp=NrOfRecRsp+
                         1)
       % Else is general response
    + sum m:Msg. (IsMSG_BcastResp(m) && IsMsg_GEN_RESP(msg(m)) && cmd(msg(m))==cmd)
       -> receiveResp(m).
          (
            (cmd(msg(m))!=VFY)
             -> WAITFORBULKOUT(info=editReqInfo_UpdateOLCreceived(true,src(m),
                                    editReqInfo_UpdateOLCnrchunks(0,src(m),
                                    editReqInfo_UpdateOLC_LOO_Status(status(msg(m)),src(m),info))),
                               NrOfRecRsp=NrOfRecRsp+1)
          + ((cmd(msg(m))==VFY
              && LOOstatus(OLCs(info).getIndexOfOLC(src(m),info)) != S_OlcAppStatus_12NcMismatch
              && LOOstatus(OLCs(info).getIndexOfOLC(src(m),info)) != S_OlcAppStatus_NewVersionMatch
              && LOOstatus(OLCs(info).getIndexOfOLC(src(m),info)) != S_OlcAppStatus_InvalidAddress))
             -> % This case cannot occur, because of the special message type VFY which is handled above
                WAITFORBULKOUT(info=editReqInfo_UpdateOLCreceived(true,src(m),
                                    editReqInfo_UpdateOLCnrchunks(0,src(m),
                                    editReqInfo_UpdateOLC_LOO_Status(status(msg(m)),src(m),info))),
                                    NrOfRecRsp=NrOfRecRsp+1)
             <> % Retain the unicast/broadcast init command response for verify
                WAITFORBULKOUT(info=editReqInfo_UpdateOLCreceived(true,src(m),
                                    editReqInfo_UpdateOLCnrchunks(0,src(m),info)), NrOfRecRsp=NrOfRecRsp+1)
```

```
        )
    + (#OLCsInModel == NrOfRecRsp)
       -> bulktimeout.
          P(s,vs,info,upd_img,vfy_info,ucastIterator,dstr,vfy_rsp)
    + timeout_sc. WAITFORBULKOUT(NrOfRecRsp=NrOfRecRsp+1)
    + timeout_sc_all. WAITFORBULKOUT(NrOfRecRsp=#OLCsInModel);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%                                    Network                                     %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act netR_scSend,netR_scSend_drop,netS_scSend:Msg;
    netR_olcSendResp,netR_olcSendResp_drop,netS_olcSendResp:Msg;
    %communication actions
    comO2S,comS2O,comS2Odrop,comO2Sdrop:Msg;
    timeout_net,timeout_net_all;
sort DroppedMsg = struct Dropped_Msg(mgs:Msg, messageCount:Nat);
proc
    % to find multiple traces to failed OLC upgrades
    NET(dropS2O:Nat,dropO2S:Nat,droppedMessages:List(DroppedMsg),messageCount:Nat)
    = sum m:Msg. netR_scSend(m)|netS_scSend(m).
      (IsUcast(des(m)) && !ForceWaitForOLCUcastResp(id(des(m))))
       -> timeout_net.
          NET(messageCount=messageCount+1)
       <> NET(messageCount=messageCount+1)
    + (dropS2O>0)
       -> sum m:Msg. netR_scSend_drop(m).
          % Only messages that need a response need a timeout_net when message is dropped
          IsBcastWithoutAck(des(m))
           -> NET(dropS2O=Int2Nat(dropS2O-1), droppedMessages=Dropped_Msg(m,messageCount)|>droppedMessages,
                  messageCount=messageCount+1)
           <> (IsBcastWithAck(des(m)))
               -> timeout_net_all.
                  % Tell the WAITFORBULKOUT that non of the responses will arrive
                  NET(dropS2O=Int2Nat(dropS2O-1),droppedMessages=Dropped_Msg(m,messageCount)|>droppedMessages,
                      messageCount=messageCount+1)
               <> timeout_net.
                  NET(dropS2O=Int2Nat(dropS2O-1),droppedMessages=Dropped_Msg(m,messageCount)|>droppedMessages
                  ,
                      messageCount=messageCount+1)

    + sum m:Msg. netR_olcSendResp(m)|netS_olcSendResp(m). NET(messageCount=messageCount+1)
    + (dropO2S>0)
       -> sum m:Msg. netR_olcSendResp_drop(m).
          timeout_net.
          NET(dropO2S=Int2Nat(dropO2S-1), droppedMessages=Dropped_Msg(m,messageCount)|>droppedMessages,
          messageCount=messageCount+1)
    ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%                                      OLC                                       %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort
    BBuf = List(Bool);
map store:                  Nat#BBuf -> BBuf;
    chunksRec:              BBuf->Bool;
    oldVersIsforceImVers:   Vers#Imgs->Bool;
    nonActiveVer:           Imgs->Vers;
    activeVer:              Imgs->Vers;
    nonActiveChunks:        Imgs->BBuf;
    getMissingChunkIds:     BBuf->List(Nat);
    getMissingChunkIds':    BBuf#Nat->List(Nat);
    setVersInNonActive:     Vers#Imgs->Imgs;
    eraseNonActive:         Imgs->Imgs;
    storeInNonActive:       Nat#Imgs->Imgs;
    getImageInfo:           ImgType#Imgs->Img;
var i: Nat; c,c':Bool; b:BBuf;
    v:Vers; im:Imgs, imt:ImgType;
eqn i == 0 -> store(i,b) = true |> tail(b);
    i > 0 -> store(i,c'|>b) = c' |> store(Int2Nat(i-1),b);

    chunksRec(b) = !(false in b);
```

```
    oldVersIsforceImVers(v,im) =if(IsNORMAL_IMAGE(active(im)), vers(norm_img(im))==v, vers(fallb_img(im))==v);
    nonActiveVer(im) = if(IsNORMAL_IMAGE(active(im)),vers(fallb_img(im)), vers(norm_img(im)));
    activeVer(im) = if(IsNORMAL_IMAGE(active(im)), vers(norm_img(im)),vers(fallb_img(im)));
    nonActiveChunks(im) = if(IsNORMAL_IMAGE(active(im)),chunks(fallb_img(im)),chunks(norm_img(im)));
    getMissingChunkIds(b) = getMissingChunkIds'(b,0);
    setVersInNonActive(v,im) = if(IsNORMAL_IMAGE(active(im)),
                                IMGs(norm_img(im), IMG(v,chunks(fallb_img(im))), active(im)),
                                IMGs(IMG(v, chunks(norm_img(im))), fallb_img(im), active(im)));
    eraseNonActive(im) = if(IsNORMAL_IMAGE(active(im)),
                          IMGs(norm_img(im), IMG(vers(fallb_img(im)), [false,false]), active(im)),
                          IMGs(IMG(vers(norm_img(im)), [false,false]), fallb_img(im), active(im)));
    storeInNonActive(i,im)
      = if(IsNORMAL_IMAGE(active(im)),
           IMGs(norm_img(im),IMG(vers(fallb_img(im)), store(i,chunks(fallb_img(im)))),active(im)),
           IMGs(IMG(vers(norm_img(im)), store(i,chunks(norm_img(im)))),fallb_img(im),active(im)));
    getImageInfo(imt,im) = if(IsNORMAL_IMAGE(imt), norm_img(im), fallb_img(im));

    (i<#b) ->getMissingChunkIds'(b,i) = if(b.i==false,
                                        i|>getMissingChunkIds'(b,i+1),
                                        getMissingChunkIds'(b,i+1));
    !(i<#b)->getMissingChunkIds'(b,i) = [];
sort
    Imgs = struct IMGs(norm_img:Img, fallb_img:Img, active:ImgType);
    Img = struct IMG(vers:Vers, chunks:BBuf);

act
    a;
    olcReceive:Msg;
    comS,comResp:Msg;
    olcChunksReceived: ID # Bool;
    olcHandleOK: ID # Command;
    olcRebootInto: ID # Img # ImgType;
    olcErr: ID # Status;
    olcUpgradeOK: ID # Bool;

proc
    APP_SwUpgrade(id:ID, isSessionOpenend:Bool, isMy12NC:Bool, my12NC:Hw12Nc, imgs:Imgs)
    = sum m:Msg.
      (IsBcastWithAck(des(m)) || (IsBcastWithoutAck(des(m)) || IsUcast(des(m)) && id(des(m)) == id))
       -> olcReceive(m).
          APP_SwUpgrade_HandleSwUpgradeCommand(msg(m), des(m), id, isSessionOpenend,isMy12NC,my12NC,imgs)
       %else consume message
       <> olcReceive(m).
          APP_SwUpgrade(id,isSessionOpenend,isMy12NC,my12NC,imgs)
    + olcUpgradeOK(id, (active(imgs) == NORMAL_IMAGE
                        && activeVer(imgs) == V2
                        && #getMissingChunkIds(chunks(norm_img(imgs))) == 0 ))
    ;

    APP_SwUpgrade_HandleSwUpgradeCommand(m:MsgType, msg_des:MsgDes, id:ID, isSessionOpenend:Bool,
                                         isMy12NC:Bool, my12NC:Hw12Nc, imgs:Imgs)
    = IsMsg_REBOOT(m)
       -> HandleRebootCmd(forceImg(m),code12NC(m),msg_des,id,isSessionOpenend,isMy12NC,my12NC,imgs)
    + IsMsg_INIT(m)
       -> HandleInitCommand(newVers(m),code12NC(m),startAddr(m),msg_des,id,isSessionOpenend,isMy12NC,my12NC,
                            imgs)
    + IsMsg_DATA(m)
       -> HandleDataCmd(addrFlash(m),data(m),msg_des,id,isSessionOpenend,isMy12NC,my12NC,imgs)
    + IsMsg_VFY(m)
       -> HandleVerifyCmd(msg_des,id,isSessionOpenend,isMy12NC,my12NC,imgs)
    ;

    HandleInitCommand(newVers:Vers,code12NC:Hw12Nc,startAddr:ImgType, msg_des:MsgDes, id:ID,
                      isSessionOpenend:Bool, isMy12NC:Bool, my12NC:Hw12Nc, imgs:Imgs)
    % if isMy12NC (check if this update is destined for this hardware type)
    %  if theImageToUpgrade has the expected old version (theImageToUpgrade is always the non-active version)
    %    if new version is not equal to current version (of theImageToUpgrade)
    %      if location to write is ok (not in model)
    %        open session, erase flash, write {size, checksum, version, start (address)}
    = (my12NC==code12NC)
```

```
      -> (activeVer(imgs)!=newVers)
          -> (startAddr != active(imgs))
              -> olcHandleOK(id,INIT).
                 SendGenResponse(S_OlcAppStatus_Ok,msg_des,id,INIT).
                 APP_SwUpgrade(id,true,true,my12NC,setVersInNonActive(newVers,eraseNonActive(imgs)))
              <> olcErr(id,S_OlcAppStatus_InvalidAddress).
                 SendGenResponse(S_OlcAppStatus_InvalidAddress,msg_des,id,INIT).
                 APP_SwUpgrade(id,false,isMy12NC,my12NC,imgs)
          <> olcErr(id,S_OlcAppStatus_NewVersionMatch).
             SendGenResponse(S_OlcAppStatus_NewVersionMatch,msg_des,id,INIT).
             APP_SwUpgrade(id,false,isMy12NC,my12NC,imgs)
      <> olcErr(id,S_OlcAppStatus_12NcMismatch).
         SendGenResponse(S_OlcAppStatus_12NcMismatch,msg_des,id,INIT).
         APP_SwUpgrade(id,false,isMy12NC,my12NC,imgs);

    HandleDataCmd(addrFlash:Nat, data:Data, msg_des:MsgDes, id:ID, isSessionOpenend:Bool, isMy12NC:Bool,
                  my12NC:Hw12Nc, imgs:Imgs)
    % if isSessionOpened and isMy12NC and address to flash is in range (not in model)
    %   write data packet into flash, set time out for 12h (not in model)
    = (isSessionOpenend)
       -> (isMy12NC)
           -> olcHandleOK(id,DATA).
              %SendGenResponse(S_OlcAppStatus_Ok,msg_des,id,DATA).
              APP_SwUpgrade(id,isSessionOpenend,isMy12NC, my12NC,storeInNonActive(addrFlash,imgs))
           <> olcErr(id,S_OlcAppStatus_12NcMismatch).
              %SendGenResponse(S_OlcAppStatus_12NcMismatch,msg_des,id,DATA).
              APP_SwUpgrade(id,isSessionOpenend,isMy12NC,my12NC,imgs)
       <> olcErr(id,S_OlcAppStatus_SessionClosed).
          %SendGenResponse(S_OlcAppStatus_SessionClosed,msg_des,id,DATA).
          APP_SwUpgrade(id,isSessionOpenend,isMy12NC,my12NC,imgs);

    HandleVerifyCmd(msg_des:MsgDes, id:ID, isSessionOpenend:Bool, isMy12NC:Bool, my12NC:Hw12Nc, imgs:Imgs)
    %check for all chunks if it is received and written to flash (by checking that non of the memory
    positions
    % are erased)
    % when a chunk is missing, add to rpyBuf (replyBuffer?)
    = (isSessionOpenend)
        -> ((isMy12NC)
            -> olcHandleOK(id,VFY).
               olcChunksReceived(id,chunksRec(nonActiveChunks(imgs))).
               SendVFYResponse(S_OlcAppStatus_Ok,getMissingChunkIds(nonActiveChunks(imgs)),msg_des,id).
               (chunksRec(nonActiveChunks(imgs)))
                -> APP_SwUpgrade(id,false,isMy12NC,my12NC,imgs)
                <> APP_SwUpgrade(id,isSessionOpenend,isMy12NC,my12NC,imgs)
            <> olcErr(id,S_OlcAppStatus_12NcMismatch).
               SendGenResponse(S_OlcAppStatus_12NcMismatch,msg_des,id,VFY).
               APP_SwUpgrade(id,isSessionOpenend,isMy12NC,my12NC,imgs))
        <> olcErr(id,S_OlcAppStatus_SessionClosed).
           SendGenResponse(S_OlcAppStatus_SessionClosed,msg_des,id,VFY).
           APP_SwUpgrade(id,isSessionOpenend,isMy12NC,my12NC,imgs);

    HandleRebootCmd(forceImg:ImgType, code12NC:Hw12Nc, msg_des:MsgDes, id:ID, isSessionOpenend:Bool,
                    isMy12NC:Bool, my12NC:Hw12Nc, imgs:Imgs)
    % if check12NC is (check if this command is destined for this hardware type)
    %   if oldVers matches forceIm version
    %    set reboot delay           (not in model)
    %    set active image = forceIm
    % else isSessionOpened=false
    = (code12NC==my12NC)
      -> olcHandleOK(id,REBOOT).
         olcRebootInto(id,getImageInfo(forceImg,imgs),forceImg).
         SendGenResponse(S_OlcAppStatus_Ok,msg_des,id,REBOOT).
         APP_SwUpgrade(id,isSessionOpenend,isMy12NC,my12NC,IMGs(norm_img(imgs),fallb_img(imgs),forceImg))
      <> olcErr(id,S_OlcAppStatus_12NcMismatch).
         SendGenResponse(S_OlcAppStatus_12NcMismatch,msg_des,id,REBOOT).
         APP_SwUpgrade(id,false,isMy12NC,my12NC,imgs);

act olcSendResp,olcSendResp_drop:Msg;
    noRespNeeded;
proc
```

```
    SendGenResponse(status:Status, msg_des:MsgDes, id:ID, cmd:Command)
    = (IsUcast(msg_des))
       -> ( olcSendResp(MSG_UcastResp(id,Msg_GEN_RESP(status,cmd)))
          + olcSendResp_drop(MSG_UcastResp(id,Msg_GEN_RESP(status,cmd))) )
    + (IsBcastWithAck(msg_des))
       -> ( olcSendResp(MSG_BcastResp(id,Msg_GEN_RESP(status,cmd)))
          + olcSendResp_drop(MSG_BcastResp(id,Msg_GEN_RESP(status,cmd))) )
    + (IsBcastWithoutAck(msg_des))-> noRespNeeded;

    SendVFYResponse(status:Status, missingChunks:List(Nat), msg_des:MsgDes, id:ID)
    = (IsUcast(msg_des))
       -> ( olcSendResp(MSG_UcastResp(id,Msg_VFY_RESP(status,missingChunks)))
          + olcSendResp_drop(MSG_UcastResp(id,Msg_VFY_RESP(status,missingChunks))) )
    + (IsBcastWithAck(msg_des))
       -> ( olcSendResp(MSG_BcastResp(id,Msg_VFY_RESP(status,missingChunks)))
          + olcSendResp_drop(MSG_BcastResp(id,Msg_VFY_RESP(status,missingChunks))) )
    + (IsBcastWithoutAck(msg_des))-> noRespNeeded;

% needed test model with reliable connection
map OLCsInModel: List(ID);
%eqn OLCsInModel = [OLC1];          % 1 OLC
eqn OLCsInModel = [OLC1,OLC2];      % 2 OLCs

map ForceWaitForOLCUcastResp: ID -> Bool;
var id:ID;
eqn ForceWaitForOLCUcastResp(id) = (id in OLCsInModel);


init
  allow({
    % SC
    st, stv, startThread, idle,startSWU,
    handlePrepareState, handleBcastRebootwithoutAck, handleUnicastReboot, handleBcastInitwithAck,
    handleBcastInitwithoutAck, handleUnicastInit, distributeChunk, handleVerifySubStateMachine,
    reportStatus, waitingForInitAck, verifyFinished, numOlcIsSmall, numOlcIsLarge, chunksMissing,
    sendingUcastDone, stopBulkOutForInit, chunksDistributed,
    ucastRebootVfyDone, stopBulkOutForVfy, nowVerifyUcast, againVerifyBcast, nextOLC,
    verifyOlcOK, timeout, olcStatusOKexists, olcStatusOKdoesNotExists,
    olcReceivedAndStatusOKexists, olcReceivedAndStatusOKdoesNotExists,
    finished,
    %Bulkout
    bulktimeout,
    %Net
    comO2S,comS2O,comS2Odrop,comO2Sdrop,
    %OLC
    a, olcChunksReceived, olcHandleOK, noRespNeeded,olcRebootInto, olcErr,
    upgradeNotOK, upgradeOK, timeout_all, case4
 },
    comm({
          timeout_sc|timeout_net->timeout,
          timeout_sc_all|timeout_net_all->timeout_all,
          olcUpgradeOK|scUpgradeOK->upgradeOK,
          %scSend|netR_scSend|netS_scSend|olcReceive->comS2O,              % 1 OLC, also change OLCsInModel
          scSend|netR_scSend|netS_scSend|olcReceive|olcReceive->comS2O,    % 2 OLCs, also OLCsInModel
          scSend_drop|netR_scSend_drop->comS2Odrop,
          olcSendResp|netR_olcSendResp|netS_olcSendResp|receiveResp->comO2S,
          olcSendResp_drop|netR_olcSendResp_drop->comO2Sdrop

         },
        A_SWUpgrade_Init
     || NET(43,15,[],0) % 43,15 max that has influence
     || APP_SwUpgrade(OLC1,false,false,hw12nc1,IMGs(IMG(V1,[true,true]), IMG(V1,[true,true]), NORMAL_IMAGE))
     || APP_SwUpgrade(OLC2,false,false,hw12nc1,IMGs(IMG(V1,[true,true]), IMG(V1,[true,true]), NORMAL_IMAGE))
  )
 )
;
```

C Rename file for requirement: A started SWU process must always finish

Filename: `hide_all_excluding_some.rename`
Description: This file is used to rename all actions apart from *startSWU* and *finish*, to τ actions, to verify the requirement: A started SWU process must always finish.

```
var
 b:Bool;
 s:State;
 m:Msg;
 vs:VerifyState;
 id:ID;
 loi:List(OlcInfo);
 im:Img;
 imt:ImgType;
 status: Status;
 cmd:Command;

rename
 % SC
 st(s) => tau;
 stv(vs) => tau;
 startThread=> tau;
 idle=> tau;
 %startSWU => tau;
 handlePrepareState => tau;
 handleBcastRebootwithoutAck => tau;
 handleUnicastReboot => tau;
 handleBcastInitwithAck => tau;
 handleBcastInitwithoutAck => tau;
 handleUnicastInit => tau;
 distributeChunk => tau;
 handleVerifySubStateMachine => tau;
 reportStatus(loi) => tau;
 %finished => tau;
 waitingForInitAck => tau;
 verifyFinished => tau;
 numOlcIsSmall => tau;
 numOlcIsLarge => tau;
 chunksMissing => tau;
 sendingUcastDone => tau;
 stopBulkOutForInit => tau;
 chunksDistributed => tau;
 ucastRebootVfyDone => tau;
 stopBulkOutForVfy => tau;
 nowVerifyUcast => tau;
 againVerifyBcast => tau;
 nextOLC => tau;
 verifyOlcOK(id,b) => tau;
 timeout => tau;
 bulktimeout => tau;
 olcReceivedAndStatusOKexists => tau;
 olcReceivedAndStatusOKdoesNotExists => tau;
 olcStatusOKexists => tau;
 olcStatusOKdoesNotExists => tau;
 a => tau;

 % Communicatie
 comO2S(m) => tau;
 comS2O(m) => tau;
 comS2Odrop(m) => tau;
 comO2Sdrop(m) => tau;

 % OLC
 olcErr(id,status) => tau;
 olcChunksReceived(id,b) => tau;
 olcHandleOK(id,cmd) => tau;
 noRespNeeded => tau;
 olcRebootInto(id,im,imt) => tau;
 upgradeOK(id,b) => tau;
```

D Combined OLC commissioning component mCRL2 specification

Filename: `Combine_Commissioning_OLC_App.mcrl2`
Description: This file combines the model of the OLC commissioning component with a process that sends commissioning messages to the OLC commissioning model. The initialisation of the combined model is also specified in this file.
This file includes the files:`Commissioning_Shared.mcrl2` and `Commissioning_OLC_App.mcrl2`, included in appendices F and H, respectively.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OLC commissioning component combined with shared structures                                                    %
% Model of the OLC Commissioning component combined with shared structures                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#INCLUDE(Commissioning_Shared.mcrl2)
#INCLUDE(Commissioning_OLC_App.mcrl2)
act
    send,com:Msg#Bool;

proc
    % Process to send the messages to the OLC
    Sender
    = sum m:Msg. ((!(short(des(m))==ShortNull && ieee(des(m))==IeeeNull))
                  && (short(des(m))==ShortNull || ieee(des(m))==IeeeNull)
                  && IsMSG(m))
       -> ((IsMsg_COMM_MSG1(msg(m)) && short(msg(m)) != ShortNull && panId(msg(m)) != PanNull)
           || (!IsMsg_COMM_MSG1(msg(m))))
           -> send(m,true). Sender
    ;

init
  allow({
    olcHandleMessage, olcHandleRebootMessage, olcDoDecrypt, olcAppInit, olcHandleCommissionMsg0,
    olcHandleCommissionMsg1,olcHandleDecryptDone, olcCommEngInit, olcHandleDeCommissionMessage, com,
    olcSendResp,olcHandled, s_no_response_olc, armForReboot, olcReboot,olcCmdInvalid, msgNotFor,
    olcCommissionMsg1Handled,olcStateNotOK, requestDecryptFailed,requestDecryptOK,olcDecryptError,
    setConfig,olcState,skip, c_registerAddresses, c_unregisterShortAddress, c_addressIsActive, infoOlc,
    olcIsCommissioned
  },
    comm({send|olcReceive->com,
          % Active Addresses
          s_registerAddresses|r_registerAddresses->c_registerAddresses,
          s_unregisterShortAddress|r_unregisterShortAddress->c_unregisterShortAddress,
          s_addressIsActive|r_addressIsActive->c_addressIsActive},
      InitOLCModel ||
      Sender ||
      ActiveAddresses({},{})
  )
 )
;
```

E Interacting commissioning components mCRL2 specification

Filename: `Combine_CommissioningProcess.mcrl2`
Description: This file combines the model of the SC commissioning component with the OLC commissioning component. The initialisation of the combined model is also specified in this file.
This file includes the files:`Commissioning_Shared.mcrl2`, `Commissioning_SC_App.mcrl2` and `Commissioning_OLC_App.mcrl2`, included in appendices F, G and H, respectively.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Interacting commissioning components                                                                   %
% Model of the OLC application combined with the model of the SC application                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#INCLUDE(Commissioning_Shared.mcrl2)
#INCLUDE(Commissioning_SC_App.mcrl2)
#INCLUDE(Commissioning_OLC_App.mcrl2)

    InitDatabase
    = P_DB([DB_vars(OLC1,NoAction,Ready,Ieee1,Short1,Pan1)])
    ;

init
  allow({
    c_scheduleCommissioningMsg,c_clearCommissioningMsg,c_queue_pop, process,
    foundOlcWithErroneousStatus, noOlcWithErroneousStatus, executeOlcAction, noOlcWithPendingAction,
    foundOlcWithPendingAction, hasActionType, noOLCsInDeletionState, OLCsInDeletionState,
    comS2O,comO2S,comS2Odrop,comO2Sdrop,c_timeout,c_no_response_olc,
    fastRebootOlc, sendAddressingInfo, configOlc, resetRxCounters,
    commOlcStateIEEEMisMatch,commOlcStateNotCommissioned,commOlcStateCommissioned,commissioningDone,
    c_removeOlcFromDBWithStateReadyForDeletion,c_getOLClist_in_State,c_getOlcWithPendingAction,
    c_getOLCidWithErroneousStatus,c_setCommissioningState,c_storeActionType, commOlcStateShortMisMatch,
    commOlcStateUnreachable,olcNeedsSWU,tmp,
    c_registerAddresses,c_unregisterShortAddress,c_addressIsActive,
    c_setErrorOlcsToUpdate, secureCommissioningFailed,
    c_setLogReporter, c_setLogCompleter,c_getOlcByID,c_queue_empty,c_getOLClist_ReadyForUpgrade,
    % Acties van APP_CommissioningEngine.mclr2
    olcHandleMessage, olcHandleRebootMessage, olcDoDecrypt, olcAppInit,
    olcHandleCommissionMsg0,olcHandleCommissionMsg1,olcHandleDecryptDone,
    olcCommEngInit, olcHandleDeCommissionMessage,
    olcHandled, armForReboot,
    olcReboot,olcCmdInvalid, msgNotFor, olcCommissionMsg1Handled,olcStateNotOK,
    requestDecryptFailed,requestDecryptOK,olcDecryptError,setConfig,skip,findError,
    %olcState,
    olcIsCommissioned,decommissionOlc,removedOlcsFromDB,infoOlc,commissionOlc,alreadyCommissioned,
    scStart,triggerDecommissioning,triggerCommissioning,triggerLampTypeUpdate_LampReplacement,
    event_OlcDecommissionResolved,
    processingSWU, bothImgsUpgraded, atleastOneImgUpgradeFailed,processingSWUdone
  },
    comm({
        % Database
        r_removeOlcFromDBWithStateReadyForDeletion|s_removeOlcFromDBWithStateReadyForDeletion
         ->c_removeOlcFromDBWithStateReadyForDeletion,
        r_getOLClist_in_State|s_getOLClist_in_State->c_getOLClist_in_State,
        r_getOlcWithPendingAction|s_getOlcWithPendingAction->c_getOlcWithPendingAction,
        r_getOLCidWithErroneousStatus|s_getOLCidWithErroneousStatus->c_getOLCidWithErroneousStatus,
        r_setCommissioningState|s_setCommissioningState->c_setCommissioningState,
        r_storeActionType|s_storeActionType->c_storeActionType,
        r_setErrorOlcsToUpdate|s_setErrorOlcsToUpdate->c_setErrorOlcsToUpdate,
        r_getOLClist_ReadyForUpgrade|s_getOLClist_ReadyForUpgrade->c_getOLClist_ReadyForUpgrade,
        % Thread Msg Queue
        s_scheduleCommissioningMsg|r_scheduleCommissioningMsg->c_scheduleCommissioningMsg,
        s_clearCommissioningMsg|r_clearCommissioningMsg->c_clearCommissioningMsg,
        s_queue_pop|r_queue_pop->c_queue_pop,
        s_queue_empty|r_queue_empty->c_queue_empty,
        % Msg communication
        scSend|netR_scSend|s_addressIsActive|r_addressIsActive|netS_scSend|olcReceive->comS2O,
        scSend_drop|netR_scSend_drop->comS2Odrop,
        olcSendResp|netR_olcSendResp|netS_olcSendResp|receiveResp->comO2S,
        olcSendResp_drop|netR_olcSendResp_drop->comO2Sdrop,
        s_no_response_olc|r_no_response_olc->c_no_response_olc,
        timeout_sc|timeout_net->c_timeout,
        % Active Addresses
        s_registerAddresses|r_registerAddresses->c_registerAddresses,
        s_unregisterShortAddress|r_unregisterShortAddress->c_unregisterShortAddress,
        % Callback
        s_setLogReporter|r_setLogReporter->c_setLogReporter,
        s_setLogCompleter|r_setLogCompleter->c_setLogCompleter,
        s_getOlcByID|r_getOlcByID->c_getOlcByID
```

```
    },
    % Every OLC in the database is assigned an short address and a pan number.
    CommStore(0,false,false) ||
    InitDatabase ||
    ThreadMsgQueue([],true) ||
    CommThread(Sc_Node(V2)) ||
    NET(1,1) ||
    InitOLCModel ||
    ActiveAddresses({},{}) ||
    Callback([],[])
  )
)
;
```

F Commissioning Shared mCRL2 specification

Filename: `Commissioning_Shared.mcrl2`
Description: This file contains the shared structures of the SC commissioning model and the OLC commissioning model.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SHARED                                                                                                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sort
    % Original struct name: commStateGetInfo_t
    OLCCommState
     = struct COMM_STATE_UNCOMMISSIONED              ?IsCOMM_STATE_UNCOMMISSIONED
            | COMM_STATE_COMMISSIONED_NEEDS_REBOOT   ?IsCOMM_STATE_COMMISSIONED_NEEDS_REBOOT
            | COMM_STATE_COMMISSIONED                ?IsCOMM_STATE_COMMISSIONED
            | COMM_STATE_DECOMMISSIONING             ?IsCOMM_STATE_DECOMMISSIONING
            ;

    ID  = struct OLC1;

    IeeeAddr    = struct IeeeNull | Ieee1;               % IeeeNull is the uninitialized ieee address
    ShortAddr   = struct ShortNull | Short1;             % ShortNull is the uninitialized short address
    PanId       = struct PanNull | Pan1;                 % PanNull is the uninitialized PAN number
    Vers        = struct V1 | V2 | V3;

    Imgs = struct IMGs(norm_img:Img, fallb_img:Img, active:ImgType);
    Img = struct IMG(vers:Vers);
    % Images, a node has 2 images, a NORMAL and FALLBACK
    ImgType = struct NORMAL_IMAGE?IsNORMAL_IMAGE
                   | FALLBACK_IMAGE?IsFALLBACK_IMAGE;

    % Different types of messages
    MsgTypeS
     = struct Msg_GET_INFO                                          ?IsMsg_GET_INFO
            | Msg_GET_MAC                                           ?IsMsg_GET_MAC
            | Msg_COMM_MSG0                                         ?IsMsg_COMM_MSG0
            | Msg_COMM_MSG1(short:ShortAddr, panId:PanId)           ?IsMsg_COMM_MSG1
            | Msg_DECOMM                                            ?IsMsg_DECOMM
            | Msg_SEC_REBOOT                                        ?IsMsg_SEC_REBOOT
            ;

    MsgTypeR
     = struct Msg_GET_INFO_RESP(olcInfo:OlcInfo)                    ?IsMsg_GET_INFO_RESP
            | Msg_GET_MAC_RESP(ieee:IeeeAddr)                       ?IsMsg_GET_MAC_RESP
            | Msg_CMD_INVALID(msg:MsgTypeS)                         ?IsMsg_CMD_INVALID
            | Msg_ERR_OK(msg:MsgTypeS)                              ?IsMsg_ERR_OK
            ;

    Msg
     = struct MSG(des:MsgDes, msg:MsgTypeS)?IsMSG
            | MSG_UcastResp(msgr:MsgTypeR)?IsMSG_UcastResp
            ;

    MsgDes
     = struct Ucast(ieee:IeeeAddr, short:ShortAddr)?IsUcast
            ;

    OlcInfo
     = struct Olc_Info(imgNormVers:Vers, imgFallVers:Vers, activeImg:ImgType, ieee:IeeeAddr,
                       short:ShortAddr,panId:PanId, commState:OLCCommState, isExtended:Bool);

map versToNat: Vers -> Nat;
var v:Vers;
eqn (v == V1)-> versToNat(v) = 1;
    (v == V2)-> versToNat(v) = 2;
    (v == V3)-> versToNat(v) = 3;

map isAddressActive: Msg # Set(ShortAddr) # Set(IeeeAddr)-> Bool;
var m:Msg, shorts:Set(ShortAddr), ieees:Set(IeeeAddr);
eqn isAddressActive(m,shorts,ieees)
     = ((ieee(des(m)) != IeeeNull && (ieee(des(m)) in ieees))
         || (short(des(m)) != ShortNull && (short(des(m)) in shorts)));

    % This process keeps track of the addresses that are active in the network
    % When ever an OLC is commissioned or decommissioned, addresses are removed or added.
```

```
    % This is only used for modeling, to force reliable communication. Such that an OLC does not
    % respond ONLY when the message is dropped.
act s_registerAddresses,r_registerAddresses,c_registerAddresses: IeeeAddr # ShortAddr;
    s_unregisterShortAddress,r_unregisterShortAddress,c_unregisterShortAddress: ShortAddr;
    s_addressIsActive,r_addressIsActive,c_addressIsActive: Msg # Bool;
proc
    ActiveAddresses(ieees:Set(IeeeAddr),shorts:Set(ShortAddr))
    = sum ieee:IeeeAddr,short:ShortAddr. r_registerAddresses(ieee,short).
      ActiveAddresses(ieees=ieees+{ieee},shorts=shorts+{short})
    + sum short:ShortAddr. r_unregisterShortAddress(short). ActiveAddresses(shorts=shorts-{short})
    + sum m:Msg. r_addressIsActive(m,isAddressActive(m,shorts,ieees)). ActiveAddresses()
    ;

act infoOlc: IeeeAddr # ShortAddr # PanId # OLCCommState;

% END SHARED
```

G Commissioning SC component mCRL2 specification

Filename: `Commissioning_SC_App.mcrl2`
Description: This file contains the mCRL2 specification of the SC commissioning component.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SC Commissioning component                                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% NOTE:
% - The model is an abstraction from the latest version of the implementation (as of april 2014)
% - From 2.0 and onwards the response data is extended with panid, short address and commstate.
% - From 2.0 and onwards AssignShortAddr has become obsolete, this is done via secure commissioning.
% - Only commissioning and SWU of OLCs is considered, not of the SCnode itself.
% - ActionType Error is renamed to ActionError, to distinguish from CommissioningState Error
% - The CommissioningState variable of the database is named DBCommState
% - The CommissioningState commStateGetInfo is named OLCCommState
sort
    ThreadMsg
     = struct
           % scan DB for uncommissioned OLC, nr=1                 no data            no delay/5min
             Msg_ScanAgain(nr:Pos,wait:Bool)                      ?IsMsg_ScanAgain
           % OLCs in error state shall try again, nr=2            no data            1hour
           | Msg_Retry_failed_OLCs(nr:Pos,wait:Bool)              ?IsMsg_Retry_failed_OLCs
           % See if Commissioning-SWU is needed, nr=3             no data            3.5min
           | Msg_Start_SWU(nr:Pos,wait:Bool)                      ?IsMsg_Start_SWU
           % Decommissioning step, nr=4                           data OLC id        no delay
           | Msg_Continue_Decommissioning(nr:Pos,id:ID,wait:Bool) ?IsMsg_Continue_Decommissioning
           % Decommissioning step, nr=5                           no data            no delay/30sec
           | Msg_Finish_Decommissioning(nr:Pos,wait:Bool)         ?IsMsg_Finish_Decommissioning
           ;

    ThreadMsgQ = List(ThreadMsg);

% ThreadMsgQueue: only add message if ThreadMsg does not yet exists in ThreadMsgQ
map schedCommMsg: ThreadMsg#ThreadMsgQ -> ThreadMsgQ;
    thrdMsgExists: ThreadMsg#ThreadMsgQ -> Bool;
    thrdMsgExists': ThreadMsg#ThreadMsgQ#Bool -> Bool;
    clearCommMsg: ThreadMsg#ThreadMsgQ -> ThreadMsgQ;
var tm:ThreadMsg,tmq:ThreadMsgQ,b:Bool;
eqn % schedule a commissionig thread msg, only if not exists
    schedCommMsg(tm,tmq) = if(thrdMsgExists(tm,tmq),tmq, tmq<|tm);
    % check if the commissionig thread msg exists
    thrdMsgExists(tm,tmq) = thrdMsgExists'(tm,tmq,false);
    (tmq==[])-> thrdMsgExists'(tm,tmq,b) = b;
    !(tmq==[])-> thrdMsgExists'(tm,tmq,b) = if(nr(head(tmq))==nr(tm), true, thrdMsgExists'(tm,tail(tmq),b));
    % clearCommissioningMsg
    (tmq==[])-> clearCommMsg(tm,tmq) = tmq;
    !(tmq==[])-> clearCommMsg(tm,tmq)
       = if(nr(head(tmq))==nr(tm), clearCommMsg(tm,tail(tmq)), head(tmq) |> clearCommMsg(tm,tail(tmq)));

map eqnSetDBCommState:  ID # DBCommState # DB -> DB;
    eqnSetDBActionType: ID # ActionType  # DB -> DB;

    %"select olc_id from OlcConfig_Actual
    % where CommissioningState is 'update' or CommissioningState is 'commissioningError'
    % and shortAddress is not 1
    % limit 1;"
    eqnGetOlcWithPendingAction: DB -> DBvars;

    % s_pS_DbWrapper->GetOLCidWithErroneousStatus(&olcId);
    %"select olc_id from OlcConfig_actual
    % where CommissioningState in ('error', 'waitForReboot', 'waitForReboot&SWU', readyForSWUIncompleteReady',
    %                             'readyForSWU', 'incompleteSWU', 'commissioningError')
    % and shortAddress is not 1
    % limit 1;"
    eqnGetOLCidWithErroneousStatus: DB -> DBvars;
    eqnGetOLClist_in_State: DB # DBCommState -> List(DBvars);
    eqnDeleteOlcFromDBWithStateReadyForDeletion: DB -> DB;
    eqnSetErrorOlcsToUpdate: DB -> DB;
    eqnGetOlcByID: ID # DB -> DBvars;
    eqnGetIEEElistOfOLCsToBeDeleted: DB -> List(IeeeAddr);
var db:DB, dbv:DBvars, id:ID, dcs:DBCommState, at:ActionType;
eqn
    (db==[])-> eqnSetDBCommState(id,dcs,db) = [];
```

```
    (db!=[])-> eqnSetDBCommState(id,dcs,db)
                = if(id(head(db))==id,
                     DB_vars(id(head(db)), actionType(head(db)), dcs, ieee(head(db)),
                             short(head(db)), panId(head(db))) |> tail(db),
                     head(db)|>eqnSetDBCommState(id,dcs,tail(db)));

    (db==[])-> eqnSetDBActionType(id,at,db) = [];
    (db!=[])-> eqnSetDBActionType(id,at,db)
                = if(id(head(db))==id,
                     DB_vars(id(head(db)), at, commState(head(db)), ieee(head(db)),
                             short(head(db)), panId(head(db))) |> tail(db),
                     head(db)|>eqnSetDBActionType(id,at,tail(db)));

    % GetOlcWithPendingAction: finds the first OLC with DBCommState Update or CommissioningError
    % NOTE: Does not depend on the ActionType
    (db==[])-> eqnGetOlcWithPendingAction(db) = DBOlcNotFound;
    (db!=[])-> eqnGetOlcWithPendingAction(db)
               = if((IsUpdate(commState(head(db))) || IsCommissioningError(commState(head(db)))),
                    head(db), eqnGetOlcWithPendingAction(tail(db)));

    (db==[])-> eqnGetOLCidWithErroneousStatus(db) = DBOlcNotFound;
    (db!=[])-> eqnGetOLCidWithErroneousStatus(db)
               = if((IsError(commState(head(db))) || IsWaitForReboot(commState(head(db)))
                    || IsWaitForRebootAndSWU(commState(head(db)))
                    || IsReadyForSWUIncompleteReady(commState(head(db)))
                    || IsReadyForSWU(commState(head(db)))
                    || IsIncompleteSWU(commState(head(db)))
                    || IsCommissioningError(commState(head(db)))),
                    head(db), eqnGetOLCidWithErroneousStatus(tail(db)));

    (db==[])-> eqnGetOLClist_in_State(db,dcs) = [];
    (db!=[])-> eqnGetOLClist_in_State(db,dcs)
               = if(commState(head(db)) == dcs,
                    head(db)|>eqnGetOLClist_in_State(tail(db),dcs),
                    eqnGetOLClist_in_State(tail(db),dcs));

    (db==[])-> eqnGetIEEElistOfOLCsToBeDeleted(db) = [];
    (db!=[])-> eqnGetIEEElistOfOLCsToBeDeleted(db)
               = if(commState(head(db)) == ReadyForDeletion,
                    ieee(head(db))|>eqnGetIEEElistOfOLCsToBeDeleted(tail(db)),
                    eqnGetIEEElistOfOLCsToBeDeleted(tail(db)));

    (db==[])-> eqnDeleteOlcFromDBWithStateReadyForDeletion(db) = [];
    (db!=[])-> eqnDeleteOlcFromDBWithStateReadyForDeletion(db)
               = if(commState(head(db)) != ReadyForDeletion,
                    head(db)|>eqnDeleteOlcFromDBWithStateReadyForDeletion(tail(db)),
                    eqnDeleteOlcFromDBWithStateReadyForDeletion(tail(db)));

    (db==[])-> eqnSetErrorOlcsToUpdate(db) = [];
    (db!=[])-> eqnSetErrorOlcsToUpdate(db)
               = if((IsError(commState(head(db))) || IsWaitForReboot(commState(head(db)))
                     || IsWaitForRebootAndSWU(commState(head(db)))
                     || IsReadyForSWUIncompleteReady(commState(head(db)))
                     || IsReadyForSWU(commState(head(db)))
                     || IsIncompleteSWU(commState(head(db)))
                     || IsCommissioningError(commState(head(db)))),
                    DB_vars(id(head(db)),actionType(head(db)),Update,ieee(head(db)),short(head(db)),
                            panId(head(db)))|>eqnSetErrorOlcsToUpdate(tail(db)),
                    head(db)|>eqnSetErrorOlcsToUpdate(tail(db)));

    (db==[])-> eqnGetOlcByID(id,db) = DBOlcNotFound;
    (db!=[])-> eqnGetOlcByID(id,db)
               = if(id(head(db)) == id, head(db), eqnGetOlcByID(id,tail(db)));

map checkNeedsSWU: OlcInfo # ScNode -> Bool;
    swuIncompatible: OlcInfo # ScNode -> Bool;
    getActiveVerOI: OlcInfo -> Vers;
var oi:OlcInfo, sn:ScNode;
eqn checkNeedsSWU(oi,sn) = (versToNat(imgNormVers(oi)) < versToNat(imgVers(sn))
                           || versToNat(imgFallVers(oi)) < versToNat(imgVers(sn)));
```

```
    swuIncompatible(oi,sn) = (versToNat(imgNormVers(oi)) > versToNat(imgVers(sn))
                            || versToNat(imgFallVers(oi)) > versToNat(imgVers(sn)));
    getActiveVerOI(oi)
     = if(IsNORMAL_IMAGE(activeImg(oi)), imgNormVers(oi), imgFallVers(oi));

sort
    ActionType
     = struct ActionCommissioning      ?IsActionCommissioning
            | ActionUpdate             ?IsActionUpdate
            | ActionDecommissioning    ?IsActionDecommissioning
            | ActionError              ?IsActionError           % Used as uninitialised value of an ActionType
            | NoAction                 ?IsNoAction
            | UpdateCalendar           ?IsUpdateCalendar
            | OLCReplacement           ?IsOLCReplacement
            ;

    DBCommState
     = struct
            % Operational state depends on action type
              Update                        ?IsUpdate
            % Decommissioning process is started
            | Decommissioning               ?IsDecommissioning
            % Decommissioning process is finished.
            | Decommissioned                ?IsDecommissioned
            % Decommissioning process is finished and the OLC can be removed from the database.
            | ReadyForDeletion              ?IsReadyForDeletion
            % Commissioning started and failed.
            | CommissioningError            ?IsCommissioningError
            % OLC software version is newer than Scnode software version.
            | Error_sw_incompatible         ?IsError_sw_incompatible
            % OLC had a reboot commando, after a timer the next state is set to Ready.
            | WaitForReboot                 ?IsWaitForReboot
            % OLC had a reboot commando and needs a SWU, after a timer the next state is set to ReadyForSWU.
            | WaitForRebootAndSWU           ?IsWaitForRebootAndSWU
            % At least one of the images of the OLC needs a SWU.
            | ReadyForSWU                   ?IsReadyForSWU
            % See ReadyForSWU, only now for a whole segment update.
            | ReadyForSWUIncomplete         ?IsReadyForSWUIncomplete
            % The main image is upgraded, fallback image still needs to be upgraded. OLC is now
            % operational (only for a whole segment update).
            | ReadyForSWUIncompleteReady    ?IsReadyForSWUIncompleteReady
            % Error during SWU.
            | IncompleteSWU                 ?IsIncompleteSWU
            % Multiple reasons can create this error
            | Error                         ?IsError
            % No action needs to be done
            | Ready                         ?IsReady
            ;

    DBvars
     = struct DB_vars(id:ID, actionType:ActionType, commState:DBCommState, ieee:IeeeAddr,
                    short:ShortAddr, panId:PanId)
            | DBOlcNotFound?IsDBOlcNotFound
            ;

    DB = List(DBvars);

    ScNode
     = struct Sc_Node(imgVers:Vers);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%                                       SC                                          %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act
    findError: Msg;
    tmp:  OlcInfo # ScNode;
    c_no_response_olc,c_timeout;

act
    scStart;
```

```
    triggerDecommissioning, triggerLampTypeUpdate_LampReplacement: IeeeAddr # ShortAddr # PanId;
    triggerCommissioning: ID # IeeeAddr # ShortAddr # PanId;
proc
    CommStore(i:Nat, com:Bool, decom:Bool)
    = (i<2)
      -> sum id:ID, dbv:DBvars.
         s_getOlcByID(id,dbv).
         (!IsDBOlcNotFound(dbv))
          -> (
               (!com)
                -> triggerCommissioning(id(dbv),ieee(dbv),short(dbv),panId(dbv)).
                   (s_storeActionType(id,ActionCommissioning) + s_storeActionType(id,OLCReplacement)).
                   s_setCommissioningState(id,Update).
                   scStart.
                   s_scheduleCommissioningMsg(Msg_ScanAgain(1,false)).
                   CommStore(i=i+1, com=true)

             + (!decom)
                -> triggerDecommissioning(ieee(dbv),short(dbv),panId(dbv)).
                   s_storeActionType(id,ActionDecommissioning).
                   s_setCommissioningState(id,Update).
                   scStart.
                   s_scheduleCommissioningMsg(Msg_ScanAgain(1,false)).
                   CommStore(i=i+1, decom=true)
             )
    ;

    % Database
act r_removeOlcFromDBWithStateReadyForDeletion,s_removeOlcFromDBWithStateReadyForDeletion;
    c_removeOlcFromDBWithStateReadyForDeletion;
    r_getOLClist_in_State,s_getOLClist_in_State,c_getOLClist_in_State: DBCommState # List(DBvars);
    r_getOlcWithPendingAction,s_getOlcWithPendingAction,c_getOlcWithPendingAction: DBvars;
    r_getOLCidWithErroneousStatus,s_getOLCidWithErroneousStatus,c_getOLCidWithErroneousStatus: DBvars;
    r_setCommissioningState,s_setCommissioningState,c_setCommissioningState: ID # DBCommState;
    r_storeActionType,s_storeActionType,c_storeActionType: ID # ActionType;
    r_setErrorOlcsToUpdate,s_setErrorOlcsToUpdate,c_setErrorOlcsToUpdate;
    r_getOlcByID,s_getOlcByID,c_getOlcByID: ID # DBvars;
    r_getOLClist_ReadyForUpgrade,s_getOLClist_ReadyForUpgrade,c_getOLClist_ReadyForUpgrade: List(DBvars);
    removedOlcsFromDB: List(IeeeAddr);
proc
    P_DB(db:DB)
    = r_removeOlcFromDBWithStateReadyForDeletion.
      removedOlcsFromDB(eqnGetIEEElistOfOLCsToBeDeleted(db)).
      P_DB(eqnDeleteOlcFromDBWithStateReadyForDeletion(db))
    + sum dbcs:DBCommState. r_getOLClist_in_State(dbcs,eqnGetOLClist_in_State(db,dbcs)).
      P_DB()
    + r_getOlcWithPendingAction(eqnGetOlcWithPendingAction(db)).
      P_DB()
    + r_getOLCidWithErroneousStatus(eqnGetOLCidWithErroneousStatus(db)).
      P_DB()
    + sum dbcs:DBCommState,id:ID. r_setCommissioningState(id,dbcs).
      P_DB(eqnSetDBCommState(id,dbcs,db))
    + sum at:ActionType,id:ID. r_storeActionType(id,at).
      P_DB(eqnSetDBActionType(id,at,db))
    + r_setErrorOlcsToUpdate.
      P_DB(eqnSetErrorOlcsToUpdate(db))
    + sum id:ID. r_getOlcByID(id,eqnGetOlcByID(id,db)).
      P_DB()
    + r_getOLClist_ReadyForUpgrade(eqnGetOLClist_in_State(db,ReadyForSWU)
                                   ++eqnGetOLClist_in_State(db,ReadyForSWUIncomplete)).
      P_DB()
    ;

    % Thread Msg Queue
act s_scheduleCommissioningMsg,r_scheduleCommissioningMsg,c_scheduleCommissioningMsg:ThreadMsg;
    s_clearCommissioningMsg,r_clearCommissioningMsg,c_clearCommissioningMsg:ThreadMsg;
    s_queue_pop,r_queue_pop,c_queue_pop:ThreadMsg;
    s_queue_empty,r_queue_empty,c_queue_empty;
proc
    ThreadMsgQueue(thrd_msgs:ThreadMsgQ, withDelay:Bool)
```

```
    = (!(#thrd_msgs == 0))
       -> (#thrd_msgs>1&&withDelay)
           -> ( s_queue_pop(thrd_msgs.0).
                ThreadMsgQueue(thrd_msgs=clearCommMsg(thrd_msgs.0,thrd_msgs))
              + s_queue_pop(thrd_msgs.1).
                ThreadMsgQueue(thrd_msgs=clearCommMsg(thrd_msgs.1,thrd_msgs))
              )
           <> s_queue_pop(thrd_msgs.0).
              ThreadMsgQueue(thrd_msgs=clearCommMsg(thrd_msgs.0,thrd_msgs))
       <> s_queue_empty. ThreadMsgQueue()
    + (#thrd_msgs < 6)
       -> sum tm:ThreadMsg. r_scheduleCommissioningMsg(tm).
          ThreadMsgQueue(thrd_msgs=schedCommMsg(tm,thrd_msgs))
    + sum tm:ThreadMsg. r_clearCommissioningMsg(tm). ThreadMsgQueue(thrd_msgs=clearCommMsg(tm,thrd_msgs));

    % Commissioning Thread
act process:ThreadMsg;
    foundOlcWithPendingAction, foundOlcWithErroneousStatus:ID;
    noOlcWithErroneousStatus, executeOlcAction, noOlcWithPendingAction, noOLCsInDeletionState;
    hasActionType: ActionType;
    OLCsInDeletionState: List(DBvars);
    receiveResp:Msg;
    commOlcStateCommissioned,commOlcStateNotCommissioned,commOlcStateIEEEMisMatch,
    commOlcStateShortMisMatch,commOlcStateUnreachable;
    timeout_sc;
    decommissionOlc: IeeeAddr # ShortAddr # PanId;
    commissionOlc: ID # IeeeAddr # ShortAddr # PanId;
    alreadyCommissioned: IeeeAddr;
    event_OlcDecommissionResolved;
    processingSWU, bothImgsUpgraded, atleastOneImgUpgradeFailed,processingSWUdone;
proc
    CommThread(sc_node:ScNode)
    = sum tm:ThreadMsg. r_queue_pop(tm). Processrequest(tm,sc_node)
    + r_queue_empty. CommThread()
    ;

    Processrequest(thrd_msg:ThreadMsg,sc_node:ScNode)
    = (IsMsg_Continue_Decommissioning(thrd_msg))
       -> sum dbv:DBvars. s_getOlcByID(id(thrd_msg),dbv).
          (!IsDBOlcNotFound(dbv))
           -> ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_GET_INFO)).
              ( % No Response
                timeout_sc
              + sum m_gi:Msg. (IsMSG_UcastResp(m_gi) && IsMsg_GET_INFO_RESP(msgr(m_gi)))
                 -> receiveResp(m_gi)
              ).
              % still continue, even if request GetInfo failed
              ScSend(MSG(Ucast(IeeeNull,short(dbv)),Msg_DECOMM)).
              ( timeout_sc.
                Commissioning_DecommissioningFailed(id(dbv)).
                % also send reboot when decommissioning message was not received
                ScSend(MSG(Ucast(IeeeNull,short(dbv)),Msg_SEC_REBOOT)).
                ( % No Response
                  timeout_sc
                + sum m:Msg. (IsMSG_UcastResp(m) && IsMsg_SEC_REBOOT(msg(msgr(m))))
                   -> receiveResp(m)
                )
              + sum m:Msg. (IsMSG_UcastResp(m) && IsMsg_DECOMM(msg(msgr(m))))
                 -> receiveResp(m).
                    (IsMsg_ERR_OK(msgr(m)))
                     -> ScSend(MSG(Ucast(IeeeNull,short(dbv)),Msg_SEC_REBOOT)).
                        ( % No Response
                          timeout_sc.
                          Commissioning_DecommissioningFailed(id(dbv))
                        + sum m:Msg.
                          (IsMSG_UcastResp(m) && IsMsg_SEC_REBOOT(msg(msgr(m))))
                           -> receiveResp(m).
                              (IsMsg_ERR_OK(msgr(m)))
                               -> % A_Commissioning_DecommissioningDone, both decommissioning messages are OK
                                  % 1. Set decommissioned olc state to decommissioned
```

```
                              s_setCommissioningState(id(dbv),Decommissioned).
                              % 2. Send the Pending Logs if any for the decommissioned olc and wait
                              % callback_SendLogReportForDecommissionedOLCs(olcInfo->id);
                              s_setLogReporter(id(dbv))
                           <> % function:A_Commissioning_DecommissioningFailed
                              % not sure what has happened to the OLC, let the CMS try again
                              s_setCommissioningState(id(dbv),Error)
                     )
                  <> % also send reboot when decommissioning message caused an error
                     ScSend(MSG(Ucast(IeeeNull,short(dbv)),Msg_SEC_REBOOT)).
                     ( % No Response
                       timeout_sc
                       + sum m:Msg. (IsMSG_UcastResp(m) && IsMsg_SEC_REBOOT(msg(msgr(m))))
                          -> receiveResp(m)
                     ).
                     Commissioning_DecommissioningFailed(id(dbv))
          ).
          CommThread(sc_node)
       <> % Failed to retrieve OLC %s from DB
          % NOTE: In the implementation the
          CommThread(sc_node)
+ (IsMsg_Finish_Decommissioning(thrd_msg))
   -> sum lstReadyForDel:List(DBvars). s_getOLClist_in_State(ReadyForDeletion,lstReadyForDel).
      (lstReadyForDel==[])
       -> noOLCsInDeletionState.
          CommThread(sc_node)
       <> OLCsInDeletionState(lstReadyForDel).
          s_scheduleCommissioningMsg(Msg_Finish_Decommissioning(5,true)).
          event_OlcDecommissionResolved.
          % NOTE: Remove OLCs from database only if they have no pending events other
          % than decommissioning
          s_removeOlcFromDBWithStateReadyForDeletion.
          CommThread(sc_node)
+ (IsMsg_Retry_failed_OLCs(thrd_msg))
   -> % Msg_Retry_failed_OLCs, function: A_Commissioning_SetErrorOlcsToUpdate()
      s_setErrorOlcsToUpdate.
      s_scheduleCommissioningMsg(Msg_ScanAgain(1,false)).
      CommThread(sc_node)
+ (IsMsg_ScanAgain(thrd_msg))
   -> s_clearCommissioningMsg(Msg_Start_SWU(3,false)).
      sum dbv:DBvars. s_getOlcWithPendingAction(dbv).
      % A_Commissioning_CommissioningOlcAllowed()
      ( (IsDBOlcNotFound(dbv))
         -> % no OLC found
            noOlcWithPendingAction.
            sum dbv:DBvars. s_getOLCidWithErroneousStatus(dbv).
            ((IsDBOlcNotFound(dbv))
             -> noOlcWithErroneousStatus
             <> foundOlcWithErroneousStatus(id(dbv)).
                s_scheduleCommissioningMsg(Msg_Retry_failed_OLCs(2,true))
            ).
            s_scheduleCommissioningMsg(Msg_Start_SWU(3,true))

         <> % OLC found
            foundOlcWithPendingAction(id(dbv)).
            % A_Commissioning_ExecuteOlcAction
            ExecuteOlcAction(sc_node,dbv).
            % post a new signal for a next OLC
            s_scheduleCommissioningMsg(Msg_ScanAgain(1,false))
      ).
      CommThread(sc_node)
+ (IsMsg_Start_SWU(thrd_msg))
   -> % ListCleanupSuccessfulOLCs
      processingSWU.
      sum ldbv:List(DBvars). s_getOLClist_ReadyForUpgrade(ldbv).
      (((ldbv) != [])
       -> SWUResult(ldbv)
       <> processingSWUdone.
          CompleteDecommissioning
      ).
```

```
      CommThread(sc_node)
;

SWUResult(ldbv:List(DBvars))
= ( bothImgsUpgraded.
    CommissioningDone(head(ldbv))
  + atleastOneImgUpgradeFailed.
    s_setCommissioningState(id(head(ldbv)),IncompleteSWU).
    s_scheduleCommissioningMsg(Msg_ScanAgain(1,false))
  ).
  ((tail(ldbv) != [])
   -> SWUResult(tail(ldbv))
   <> processingSWUdone)
;

ResolveOlcsInStateDecommissioned
= sum lstDBvars:List(DBvars).  s_getOLClist_in_State(Decommissioned,lstDBvars).
  ResolveOlcsInStateDecommissionedLoop(lstDBvars)
;
ResolveOlcsInStateDecommissionedLoop(lstDBvars:List(DBvars))
= (lstDBvars!=[])
   -> s_setCommissioningState(id(head(lstDBvars)),Decommissioned).
      % 2. Send the Pending Logs if any for the decommissioned olc and wait
      s_setLogReporter(id(head(lstDBvars))).
      ResolveOlcsInStateDecommissionedLoop(tail(lstDBvars))
   <> tau
;

ExecuteOlcAction(sc_node:ScNode, dbv:DBvars)
= executeOlcAction.
  hasActionType(actionType(dbv)).
  (
    (IsActionError(actionType(dbv)) || IsNoAction(actionType(dbv)))
     -> % ActionError or NoAction
        % post a new signal for a next OLC
        s_setCommissioningState(id(dbv),Ready)

  + (IsUpdateCalendar(actionType(dbv)))
     -> % Action UpdateCalender
        s_setCommissioningState(id(dbv),Ready).
        s_storeActionType(id(dbv),NoAction)

  + (IsActionCommissioning(actionType(dbv)) || IsActionUpdate(actionType(dbv))
     || IsOLCReplacement(actionType(dbv)))
     -> %ActionCommissioning or ActionUpdate or OLCReplacement
        commissionOlc(id(dbv),ieee(dbv),short(dbv),panId(dbv)).
        %NOTE: (OLCReplacement first does ResetRxCounters)
        % GetVersionInfo
        ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_GET_INFO)).
        ( % No Response
          timeout_sc.
          % function: A_Commissioning_CommissioningFailed
          s_setCommissioningState(id(dbv),Error)
        + % Response on GetInfo message
          sum m_gi:Msg. (IsMSG_UcastResp(m_gi) && IsMsg_GET_INFO_RESP(msgr(m_gi)))
           -> receiveResp(m_gi).
              ((isExtended(olcInfo(msgr(m_gi))))
               -> ( (IsCOMM_STATE_UNCOMMISSIONED(commState(olcInfo(msgr(m_gi))))
                     || IsCOMM_STATE_COMMISSIONED_NEEDS_REBOOT(commState(olcInfo(msgr(m_gi)))))
                    -> CommOlcStateNotCommissioned(sc_node,dbv,olcInfo(msgr(m_gi)))
                  + (IsCOMM_STATE_COMMISSIONED(commState(olcInfo(msgr(m_gi)))))
                    -> alreadyCommissioned(ieee(dbv)).
                       (short(dbv)==short(olcInfo(msgr(m_gi)))
                        && panId(dbv)==panId(olcInfo(msgr(m_gi))))
                        -> CommOlcStateCommissioned(sc_node,olcInfo(msgr(m_gi)),dbv)

                        <> commOlcStateShortMisMatch.
                           % function: A_Commissioning_CommissioningFailed
                           ((swuIncompatible(olcInfo(msgr(m_gi)), sc_node))
                            -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
```

```
                        <> s_setCommissioningState(id(dbv),Error))

                + (IsCOMM_STATE_DECOMMISSIONING(commState(olcInfo(msgr(m_gi)))))
                   -> % NOTE: 3184    //shall never occur...: But it does. When
                      % commOlcStateUnreachable is set in the function:
                      % A_Commissioning_CheckCommissioningNeeded, then there is no case for it
                      commOlcStateUnreachable.
                      % function: A_Commissioning_CommissioningFailed
                      ((swuIncompatible(olcInfo(msgr(m_gi)), sc_node))
                        -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                        <> s_setCommissioningState(id(dbv),Error))
                )
             <> ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_GET_MAC)).
                % either a timeout comes, or a message arrives
                ( % No response, either because the OLC is commmisssioned, or message is
                  % dropped
                  timeout_sc.
                  ScSend(MSG(Ucast(IeeeNull,short(dbv)),Msg_GET_MAC)).
                  ( % No Response
                    timeout_sc.
                    CommOlcStateNotCommissioned(sc_node,dbv,olcInfo(msgr(m_gi)))
                  + % Response on Get MAC message addressed to short address
                    sum m:Msg. (IsMsg_GET_MAC_RESP(msgr(m)))
                     -> receiveResp(m).
                        (ieee(dbv) == ieee(msgr(m)))
                         -> CommOlcStateCommissioned(sc_node,olcInfo(msgr(m_gi)),dbv)
                         <> commOlcStateIEEEMisMatch.
                            ((swuIncompatible(olcInfo(msgr(m_gi)), sc_node))
                              -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                              <> s_setCommissioningState(id(dbv),Error))
                  )
                + % Response on Get MAC message addressed to ieee address
                  sum m:Msg. (IsMsg_GET_MAC_RESP(msgr(m)))
                   -> receiveResp(m).
                      CommOlcStateNotCommissioned(sc_node,dbv,olcInfo(msgr(m_gi)))
                )
            )
      )

+ (IsActionDecommissioning(actionType(dbv)))
   -> % Action Decommissioning
      decommissionOlc(ieee(dbv),short(dbv),panId(dbv)).
      ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_GET_INFO)).
      ( % No Response
        timeout_sc.
        s_setCommissioningState(id(dbv),Error)
      + % Response on GetInfo message
        sum m_gi:Msg. (IsMSG_UcastResp(m_gi) && IsMsg_GET_INFO_RESP(msgr(m_gi)))
         -> receiveResp(m_gi).
            (isExtended(olcInfo(msgr(m_gi))) && short(olcInfo(msgr(m_gi)))==ShortNull
             && panId(olcInfo(msgr(m_gi)))==PanNull)
             -> commOlcStateNotCommissioned.
                % If we think we need to decommission (entry in database), but the OLC is not
                % commissioned, the decommission entry in the database needs to go away ...
                % function: A_Commissioning_DecommissioningDone
                s_setCommissioningState(id(dbv),Decommissioned).
                % 2. Send the Pending Logs if any for the decommissioned olc and wait
                s_setLogReporter(id(dbv))
             <> % function: A_Commissioning_CheckDecommissioningNeeded
                ScSend(MSG(Ucast(IeeeNull,short(dbv)),Msg_GET_MAC)).
                ( % No Response
                  timeout_sc.
                  % function: A_Commissioning_CheckOlcIsInDecommissionedState
                  ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_GET_MAC)).
                  % either a timeout comes, or a message arrives
                  ( % No response
                    timeout_sc.
                    commOlcStateUnreachable.
                    % function: A_Commissioning_DecommissioningFailed
                    s_setCommissioningState(id(dbv),Error)
```

```
                    + sum m:Msg. (IsMsg_GET_MAC_RESP(msgr(m)))
                       -> receiveResp(m).
                          commOlcStateNotCommissioned.
                          % function: A_Commissioning_DecommissioningDone
                          s_setCommissioningState(id(dbv),Decommissioned).
                          % 2. Send the Pending Logs if any for the decommissioned olc and wait
                          s_setLogReporter(id(dbv))
                    )
                  + % Response on Get MAC message, OLC is reached on its short address.
                    sum m:Msg. (IsMsg_GET_MAC_RESP(msgr(m)))
                     -> receiveResp(m).
                        commOlcStateCommissioned.
                        s_setCommissioningState(id(dbv),Decommissioning).
                        % Complete the missing gaps if any for the olc via logcompleter
                        % Since we are going to decommission this olc, try only once
                        % s_pA_LogCompleter->Trigger(olcInfo.id, true);
                        % After LogCompleter finished, a callback is triggered?
                        s_setLogCompleter(id(dbv))
                  )
            )
      )
    ;


proc
    Commissioning_DecommissioningFailed(id:ID)
    = % function:A_Commissioning_DecommissioningFailed
      s_setCommissioningState(id,Error).
      %NOTE: This line below is added, to prevent a single dropped message from never decommissining
      %  the OLC, thus failing the requirement.
      s_scheduleCommissioningMsg(Msg_Retry_failed_OLCs(2,false))
    ;

proc
    CommOlcStateCommissioned(sc_node:ScNode, olc_info:OlcInfo, dbv:DBvars)
    = commOlcStateCommissioned.
      tmp(olc_info, sc_node).
      NeedsSWU(sc_node, olc_info, dbv)
    ;

act olcNeedsSWU: OlcInfo # Bool;
proc
    NeedsSWU(sc_node:ScNode, olc_info:OlcInfo, dbv:DBvars)
    = (checkNeedsSWU(olc_info, sc_node))
       -> olcNeedsSWU(olc_info, true).
          s_setCommissioningState(id(dbv),ReadyForSWU)
       <> olcNeedsSWU(olc_info, false).
          CommissioningDone(dbv)
    ;

act commissioningDone;
proc
    CommissioningDone(dbv:DBvars)
    = commissioningDone.
      sum dbv':DBvars. s_getOlcByID(id(dbv),dbv').
      (!IsDBOlcNotFound(dbv'))
       -> ( (IsActionCommissioning(actionType(dbv')))
             -> s_setCommissioningState(id(dbv'),Ready).
                s_storeActionType(id(dbv'),NoAction)
          + (IsOLCReplacement(actionType(dbv')))
             -> s_storeActionType(id(dbv'),NoAction)
          + (!IsActionCommissioning(actionType(dbv'))
             && !IsOLCReplacement(actionType(dbv')))
             -> s_storeActionType(id(dbv'),NoAction)
          )
    ;

proc
   LastPartFastReboot(sc_node:ScNode, olc_info:OlcInfo, dbv:DBvars)
   = ((checkNeedsSWU(olc_info, sc_node))
```

```
      -> s_setCommissioningState(id(dbv),WaitForRebootAndSWU)
      <> s_setCommissioningState(id(dbv),WaitForReboot)
      ).
      NeedsSWU(sc_node, olc_info, dbv)
   ;


act fastRebootOlc, sendAddressingInfo, configOlc, resetRxCounters;
    secureCommissioningFailed: IeeeAddr;
proc
    CommOlcStateNotCommissioned(sc_node:ScNode, dbv:DBvars, olc_info:OlcInfo)
    = commOlcStateNotCommissioned.
      ( ((isExtended(olc_info) && (IsCOMM_STATE_UNCOMMISSIONED(commState(olc_info))))
        || !isExtended(olc_info))
          -> configOlc.
             sendAddressingInfo.
             % When old version (<V2 then AssignShortAddress), else secure commissioning.
             (versToNat(getActiveVerOI(olc_info)) >= versToNat(V2))
              -> % Send first secure commissioning message
                 ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_COMM_MSG0)).
                 ( timeout_sc.
                   ((swuIncompatible(olc_info, sc_node))
                    -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                    <> s_setCommissioningState(id(dbv),CommissioningError)
                   ).
                   secureCommissioningFailed(ieee(dbv))
                 + sum m:Msg. (IsMSG_UcastResp(m) && IsMsg_COMM_MSG0(msg(msgr(m))))
                    -> receiveResp(m).
                       (IsMsg_ERR_OK(msgr(m)))
                        -> % Send second secure commissioning message
                           ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_COMM_MSG1(short(dbv), panId(dbv)))).
                           ( timeout_sc.
                             ((swuIncompatible(olc_info, sc_node))
                              -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                              <> s_setCommissioningState(id(dbv),CommissioningError)
                             ).
                             secureCommissioningFailed(ieee(dbv))
                           + sum m:Msg. (IsMSG_UcastResp(m) && IsMsg_COMM_MSG1(msg(msgr(m))))
                              -> receiveResp(m).
                                 (IsMsg_ERR_OK(msgr(m)))
                                  -> fastRebootOlc.
                                     ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_SEC_REBOOT)).
                                     ( timeout_sc.
                                       ((swuIncompatible(olc_info, sc_node))
                                        -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                                        <> s_setCommissioningState(id(dbv),CommissioningError)
                                       ).
                                       secureCommissioningFailed(ieee(dbv))
                                     + sum m:Msg. (IsMSG_UcastResp(m) && IsMsg_SEC_REBOOT(msg(msgr(m))))
                                        -> receiveResp(m).
                                           (IsMsg_ERR_OK(msgr(m)))
                                            -> LastPartFastReboot(sc_node, olc_info, dbv).
                                               resetRxCounters
                                            <> ((swuIncompatible(olc_info, sc_node))
                                                -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                                                <> s_setCommissioningState(id(dbv),CommissioningError)
                                               ).
                                               secureCommissioningFailed(ieee(dbv))
                                     )
                                   <> ((swuIncompatible(olc_info, sc_node))
                                       -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                                       <> s_setCommissioningState(id(dbv),CommissioningError)
                                     ).
                                     secureCommissioningFailed(ieee(dbv))
                           )
                         <> ((swuIncompatible(olc_info, sc_node))
                             -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                             <> s_setCommissioningState(id(dbv),CommissioningError)
                           ).
                           secureCommissioningFailed(ieee(dbv))
```

```
              )
      + (isExtended(olc_info) && (IsCOMM_STATE_COMMISSIONED_NEEDS_REBOOT(commState(olc_info))))
         -> fastRebootOlc.
            ScSend(MSG(Ucast(ieee(dbv),ShortNull),Msg_SEC_REBOOT)).
            ( timeout_sc
            + sum m:Msg. (IsMSG_UcastResp(m) && IsMsg_SEC_REBOOT(msg(msgr(m))))
               -> receiveResp(m).
                  (IsMsg_ERR_OK(msgr(m)))
                   -> LastPartFastReboot(sc_node, olc_info, dbv).
                      resetRxCounters
                   <> ((swuIncompatible(olc_info, sc_node))
                       -> s_setCommissioningState(id(dbv),Error_sw_incompatible)
                       <> s_setCommissioningState(id(dbv),CommissioningError)
                      ).
                      secureCommissioningFailed(ieee(dbv))
            ).
            resetRxCounters
      )
    ;

act scSend:Msg # Bool;
    scSend_drop:Msg;
proc
    ScSend(m:Msg) = sum addrIsActive:Bool. scSend(m,addrIsActive)
                  + scSend_drop(m)
                  ;


act s_setLogReporter,r_setLogReporter,c_setLogReporter: ID;
    s_setLogCompleter,r_setLogCompleter,c_setLogCompleter: ID;
proc Callback(logCompleter:List(ID), logReporter:List(ID))
     = sum id:ID. r_setLogCompleter(id). Callback(logCompleter=logCompleter<|id)
     + sum id:ID. r_setLogReporter(id). Callback(logReporter=logReporter<|id)
     + (#logCompleter>0)
        -> % This is called by logcompleter, and calls the fuction: continueDecommissioning
           sum dbv:DBvars. s_getOlcByID(head(logCompleter),dbv).
           (!IsDBOlcNotFound(dbv) && IsDecommissioning(commState(dbv)))
            -> s_scheduleCommissioningMsg(Msg_Continue_Decommissioning(4,head(logCompleter),false)).
               Callback(logCompleter=tail(logCompleter))
            <> Callback(logCompleter=tail(logCompleter))
     + (#logReporter>0)
        -> % This is called by A_Commissioning_DecommissioningDone
           % setCommissioningState is set in LogReporter.c
           s_setCommissioningState(head(logReporter),ReadyForDeletion).
           % next is executed in Commissioning.c, function:CompleteDecommissioning
           CompleteDecommissioning.
           Callback(logReporter=tail(logReporter))
     ;

proc
    CompleteDecommissioning
    = % next is executed in Commissioning.c, function:CompleteDecommissioning
      sum lstReadyForDel:List(DBvars). s_getOLClist_in_State(ReadyForDeletion,lstReadyForDel).
      ((lstReadyForDel==[])
       -> noOLCsInDeletionState
       <> s_scheduleCommissioningMsg(Msg_Finish_Decommissioning(5,false))
      )
    ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%                                         Network                                        %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act netR_scSend,netS_scSend,comS2O:Msg # Bool;
    netR_olcSendResp,netS_olcSendResp,netR_scSend_drop,netR_olcSendResp_drop:Msg;
    comO2S,comS2Odrop,comO2Sdrop:Msg;
    r_no_response_olc,timeout_net;
proc
    NET(dropS2O:Nat,dropO2S:Nat)
    = sum m:Msg,addrIsActive:Bool.
      netR_scSend(m,addrIsActive)|netS_scSend(m,addrIsActive)|s_addressIsActive(m,addrIsActive).
```

```
  (!addrIsActive)
   -> timeout_net. NET()
   <> NET()
+ (dropS2O>0)
   -> sum m:Msg. netR_scSend_drop(m).
      timeout_net.
      %NET(dropS2O=Int2Nat(dropS2O-1))
      NET()

+ sum m:Msg. netR_olcSendResp(m)|netS_olcSendResp(m). NET()
+ (dropO2S>0)
   -> sum m:Msg. netR_olcSendResp_drop(m).
      timeout_net.
      %NET(dropO2S=Int2Nat(dropO2S-1))
      NET()

+ % OLC does not send a response because it is not allowed to do
  r_no_response_olc.
  timeout_net.
  NET()
;
```

H Commissioning OLC component mCRL2 specification

Filename: `Commissioning_OLC_App.mcrl2`
Description: This file contains the mCRL2 specification of the OLC commissioning component.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OLC Commissioning component                                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% NOTE:
%  - The model is an abstraction from the latest version of the implementation (as of april 2014)
%  - cfgShort:ShortAddr and cfgPanId:PanId are the PanId and Short address that after a succesfull
%    commissioning is saved, to be loaded as the configured PanId and Short address. This is
%    this is in the implementation saved into the eemprom.
sort
    % Original struct name: BootUpData_t, element RFNwk_CommissionFlag
    RFNwk_CommissionFlag
     = struct COMMISSION_VALID                      ?IsCOMMISSION_VALID
            | COMMISSION_BUSY_DECOM                 ?IsCOMMISSION_BUSY_DECOM
            | COMMISSION_INVALID                    ?IsCOMMISSION_INVALID
            ;

    % Original struct name: commissionEngineState_t
    CEState
     = struct CE_STATE_NOT_COMMISSIONED             ?IsCE_STATE_NOT_COMMISSIONED
            | CE_STATE_HANDSHAKE_DONE               ?IsCE_STATE_HANDSHAKE_DONE
            | CE_STATE_DECRYPT_PENDING              ?IsCE_STATE_DECRYPT_PENDING
            | CE_STATE_COMMISSIONED                 ?IsCE_STATE_COMMISSIONED
            ;

    DoDecrypt
     = struct DecryptEmpty                          ?IsDecryptEmpty
            | Do_Decrypt(cyphermsg:MsgTypeS)        ?IsDo_Decrypt
            ;

    OlcVars
     = struct Olc_Vars(ieee:IeeeAddr, short:ShortAddr, panId:PanId, imgs:Imgs,
                   commState:OLCCommState, cestate:CEState, comm_flag:RFNwk_CommissionFlag,
                   isUncommissioned:Bool, armForReboot:Bool, doDecrypt:DoDecrypt,
                   cfgShort:ShortAddr, cfgPanId:PanId);

    % Msg is either addressed to short address or ieee address.
    % When an OLC is commissioned, it will only listen to messages sent to its short address, unless
    % the sent message is a GetInfo message.
map isSentToIeee:              Msg -> Bool;
    isSentToShort:             Msg -> Bool;
    % When the OLC is assigned an short address, it is considered to be commissioned.
    isCommissioned:            ShortAddr -> Bool;
    setCfgShortAddr:              ShortAddr # OlcVars -> OlcVars;
    setCfgPanId:                  PanId # OlcVars -> OlcVars;
    setCeState:                CEState # OlcVars -> OlcVars;
    setCommState:              OLCCommState  # OlcVars -> OlcVars;
    setRFNwk_CommissionFlag:   RFNwk_CommissionFlag # OlcVars -> OlcVars;
    setArmForReboot:           Bool # OlcVars -> OlcVars;
    setDoDecrypt:              DoDecrypt # OlcVars -> OlcVars;
var m:Msg, b:Bool, sa:ShortAddr, pi:PanId, ov:OlcVars, cs:CEState, cf:RFNwk_CommissionFlag,
    ocs:OLCCommState, dd:DoDecrypt;
eqn isSentToIeee(m) = (short(des(m)) == ShortNull && ieee(des(m)) != IeeeNull);
    isSentToShort(m) = (short(des(m)) != ShortNull && ieee(des(m)) == IeeeNull);
    isCommissioned(sa) = (sa != ShortNull);
    setCfgShortAddr(sa, ov)
     = Olc_Vars(ieee(ov), short(ov), panId(ov), imgs(ov), commState(ov), cestate(ov), comm_flag(ov),
            isUncommissioned(ov), armForReboot(ov), doDecrypt(ov), sa, cfgPanId(ov));
    setCfgPanId(pi, ov)
     =  Olc_Vars(ieee(ov), short(ov), panId(ov), imgs(ov), commState(ov), cestate(ov), comm_flag(ov),
             isUncommissioned(ov), armForReboot(ov), doDecrypt(ov), cfgShort(ov), pi);
    setCeState(cs, ov)
     =  Olc_Vars(ieee(ov), short(ov), panId(ov), imgs(ov), commState(ov), cs, comm_flag(ov),
             isUncommissioned(ov), armForReboot(ov), doDecrypt(ov), cfgShort(ov), cfgPanId(ov));
    setCommState(ocs, ov)
     =  Olc_Vars(ieee(ov), short(ov), panId(ov), imgs(ov), ocs, cestate(ov), comm_flag(ov),
             isUncommissioned(ov), armForReboot(ov), doDecrypt(ov), cfgShort(ov), cfgPanId(ov));
    setRFNwk_CommissionFlag(cf, ov)
     =  Olc_Vars(ieee(ov), short(ov), panId(ov), imgs(ov), commState(ov), cestate(ov), cf,
             isUncommissioned(ov), armForReboot(ov), doDecrypt(ov), cfgShort(ov), cfgPanId(ov));
```

```
    setArmForReboot(b,ov)
     =  Olc_Vars(ieee(ov), short(ov), panId(ov), imgs(ov), commState(ov), cestate(ov), comm_flag(ov),
                 isUncommissioned(ov), b, doDecrypt(ov), cfgShort(ov), cfgPanId(ov));
    setDoDecrypt(dd,ov)
     =  Olc_Vars(ieee(ov), short(ov), panId(ov), imgs(ov), commState(ov), cestate(ov), comm_flag(ov),
                 isUncommissioned(ov), armForReboot(ov), dd, cfgShort(ov), cfgPanId(ov));

act
    olcHandleMessage, olcHandleRebootMessage, olcDoDecrypt, olcAppInit, olcHandleCommissionMsg0;
    olcHandleCommissionMsg1,olcHandleDecryptDone, olcCommEngInit, olcHandleDeCommissionMessage;
    olcState: OLCCommState # CEState # RFNwk_CommissionFlag #IeeeAddr # ShortAddr # PanId;
    olcReceive:Msg # Bool;
    olcIsCommissioned: IeeeAddr # ShortAddr # PanId # Bool;
    s_no_response_olc;
    msgNotFor: IeeeAddr # ShortAddr;
    olcHandled, armForReboot,olcReboot,olcCmdInvalid,olcCommissionMsg1Handled,olcStateNotOK;
    requestDecryptFailed,requestDecryptOK,olcDecryptError,setConfig,skip;
proc
    % from file: APP_Init.c
    % Original function name: APP_Init
    APP_Init(ieee:IeeeAddr, short:ShortAddr, pan:PanId, imgs:Imgs, comm_flag:RFNwk_CommissionFlag)
    = olcAppInit.
      s_registerAddresses(ieee,short).
      (short==ShortNull || pan==PanNull || comm_flag != COMMISSION_VALID)
       -> APP_CommissioningEngine_Init(ieee,ShortNull,PanNull,imgs,COMMISSION_INVALID,true)
       <> APP_CommissioningEngine_Init(ieee,short,pan,imgs,comm_flag,false)
    ;

    % from file: APP_CommissioningEngine.c
    % Original function name: APP_CommissioningEngine_Init
    % Sets CEState, OLCCommState
    APP_CommissioningEngine_Init(ieee:IeeeAddr, short:ShortAddr, panId:PanId, imgs:Imgs,
                                 comm_flag:RFNwk_CommissionFlag, isUncommissioned:Bool)
    = olcCommEngInit.
      olcIsCommissioned(ieee,short,panId,!isUncommissioned).
      (isUncommissioned)
       -> infoOlc(ieee,short,panId,COMM_STATE_UNCOMMISSIONED).
          (comm_flag == COMMISSION_BUSY_DECOM)
           -> HandleDeCommissionMessage(Olc_Vars(ieee,short,panId,imgs, COMM_STATE_UNCOMMISSIONED,
                       CE_STATE_NOT_COMMISSIONED, comm_flag, isUncommissioned,false,DecryptEmpty,
                       short,panId),[])
           <> OLC(Olc_Vars(ieee,short,panId,imgs, COMM_STATE_UNCOMMISSIONED, CE_STATE_NOT_COMMISSIONED,
                       comm_flag, isUncommissioned,false,DecryptEmpty,short,panId))
       <> infoOlc(ieee,short,panId,COMM_STATE_COMMISSIONED).
          OLC(Olc_Vars(ieee,short,panId,imgs, COMM_STATE_COMMISSIONED, CE_STATE_COMMISSIONED,
                       comm_flag, isUncommissioned,false,DecryptEmpty,short,panId))
    ;

    % The main process of the commissioning engine on the OLC
    OLC(vars:OlcVars)
    = ( sum m:Msg,addrIsActive:Bool.
        olcReceive(m,addrIsActive).
        % Is the messsage for us?
        (IsUcast(des(m)) && (isSentToIeee(m) && ieee(des(m)) == ieee(vars)
                             || isSentToShort(m) && short(des(m)) == short(vars)))
         -> (
              (IsMsg_GET_INFO(msg(m)))
               -> infoOlc(ieee(vars),short(vars),panId(vars),commState(vars)).
                  SendResponse(MSG_UcastResp(Msg_GET_INFO_RESP(
                                  Olc_Info(vers(norm_img(imgs(vars))), vers(fallb_img(imgs(vars))),
                                           active(imgs(vars)), ieee(vars), short(vars),
                                           panId(vars), commState(vars), true)))).
                  OLC()
             + % Only respond to GET MAC if (sent to IEEE && !commissioned OR sent to Short and
               % commissioned)
               (IsMsg_GET_MAC(msg(m)))
               -> ((isCommissioned(short(vars)) && isSentToShort(m))
                  || (!isCommissioned(short(vars)) && isSentToIeee(m)))
                   -> SendResponse(MSG_UcastResp(Msg_GET_MAC_RESP(ieee(vars)))).
                      OLC()
```

```
              <> s_no_response_olc.
                 OLC()
       + (IsMsg_COMM_MSG0(msg(m)) || IsMsg_COMM_MSG1(msg(m))
          || IsMsg_DECOMM(msg(m)) || IsMsg_SEC_REBOOT(msg(m)))
          -> HandleMessage(vars, msg(m))
       )
    <> msgNotFor(ieee(vars),short(vars)).
       OLC()
  + (armForReboot(vars))
    -> olcReboot.
       % if we are decommissioning, short address needs to be cleared for the new reboot.
       (IsCOMM_STATE_DECOMMISSIONING(commState(vars)))
        -> s_unregisterShortAddress(short(vars)).
           APP_Init(ieee(vars),ShortNull,PanNull,
                    IMGs(IMG(vers(norm_img(imgs(vars)))), IMG(vers(fallb_img(imgs(vars)))),
                         active(imgs(vars))),
                    comm_flag(vars))
        <> APP_Init(ieee(vars),cfgShort(vars),cfgPanId(vars),
                    IMGs(IMG(vers(norm_img(imgs(vars)))), IMG(vers(fallb_img(imgs(vars)))),
                         active(imgs(vars))),
                    comm_flag(vars))
  + (IsDo_Decrypt(doDecrypt(vars)))
    -> olcDoDecrypt.
       HandleDecryptDone(vars,true)
  + olcState(commState(vars),cestate(vars),comm_flag(vars),ieee(vars),short(vars),panId(vars)).
    OLC()
  )
;

% By default if the message can not be handled, the reply command invalid (CMD_INVALID) will be sent
% Original function name: APP_CommissioningEngine_HandleMessage
HandleMessage(vars:OlcVars, msg:MsgTypeS)
= olcHandleMessage.
  (isUncommissioned(vars))
   -> ( (IsMsg_COMM_MSG0(msg))
         -> HandleCommissionMsg0(vars,msg)
      + (IsMsg_COMM_MSG1(msg))
         -> HandleCommissionMsg1(vars,msg)
      + (IsMsg_SEC_REBOOT(msg))
         -> HandleRebootMessage(vars,msg)
      + (!IsMsg_COMM_MSG0(msg) && !IsMsg_COMM_MSG1(msg) && !IsMsg_SEC_REBOOT(msg))
         -> olcCmdInvalid.
            SendResponse(MSG_UcastResp(Msg_CMD_INVALID(msg))).
            OLC(vars)
      )
   <> ( (IsMsg_DECOMM(msg))
         -> HandleDeCommissionMessage(vars,[msg])
      + (IsMsg_SEC_REBOOT(msg))
         -> HandleRebootMessage(vars,msg)
      + (!IsMsg_DECOMM(msg) && !IsMsg_SEC_REBOOT(msg))
         -> olcCmdInvalid.
            SendResponse(MSG_UcastResp(Msg_CMD_INVALID(msg))).
            OLC(vars)
      )
;

% Original function name: SEC_NCE_HandleCommissionMsg0
HandleCommissionMsg0(vars:OlcVars, msg:MsgTypeS)
= olcHandleCommissionMsg0.
  (IsCE_STATE_COMMISSIONED(cestate(vars)))
   -> olcStateNotOK.
      SendResponse(MSG_UcastResp(Msg_CMD_INVALID(msg))).
      OLC(vars)
   <> olcHandled.
      SendResponse(MSG_UcastResp(Msg_ERR_OK(msg))).
      OLC(setCeState(CE_STATE_HANDSHAKE_DONE,vars))
;

% Original function name: SEC_NCE_HandleCommissionMsg1
HandleCommissionMsg1(vars:OlcVars, msg:MsgTypeS)
```

```
    = olcHandleCommissionMsg1.
      (cestate(vars) != CE_STATE_HANDSHAKE_DONE)
       -> olcStateNotOK.
          SendResponse(MSG_UcastResp(Msg_CMD_INVALID(msg))).
          OLC(vars)
       <> olcCommissionMsg1Handled.
          % Decrypt
          % a decrypt job is posted, and will.....
          ( requestDecryptOK.
            % We do not want to send a reply NOW! Decrypting the message is done asynchronously!
            % Send the reply later
            OLC(vars=setCeState(CE_STATE_DECRYPT_PENDING, setDoDecrypt(Do_Decrypt(msg),vars)))
          + % Could not post a decrypt job
            requestDecryptFailed.
            SendResponse(MSG_UcastResp(Msg_CMD_INVALID(msg))).
            OLC(vars=setCeState(CE_STATE_NOT_COMMISSIONED, vars))
          )
    ;

    % Original function name: HandleRebootMessage
    HandleRebootMessage(vars:OlcVars, msg:MsgTypeS)
    = olcHandleRebootMessage.
      SendResponse(MSG_UcastResp(Msg_ERR_OK(msg))).
      OLC(vars=setArmForReboot(true,vars))
    ;

    % Original function name: SEC_NCE_HandleDeCommissionMessage
    % Upon decomissioning de eprom is reinitialised, and short addres and keys are then cleared.
    % But RAM is not cleared, only upon start up the 'new' empty short address is loaded.
    HandleDeCommissionMessage(vars:OlcVars, msg:List(MsgTypeS))
    = olcHandleDeCommissionMessage.
      (cestate(vars) != CE_STATE_COMMISSIONED)
       -> olcStateNotOK.
          ((#msg>0)-> SendResponse(MSG_UcastResp(Msg_CMD_INVALID(msg.0))) <> skip).
          OLC(vars)
       <> ( armForReboot.
            ((#msg>0)-> SendResponse(MSG_UcastResp(Msg_ERR_OK(msg.0))) <> skip).
            OLC(vars=setRFNwk_CommissionFlag(COMMISSION_INVALID,
                     setCommState(COMM_STATE_DECOMMISSIONING,vars)))
          )
    ;

    % Original function name: APP_CommissioningEngine_HandleDecryptDone
    HandleDecryptDone(vars:OlcVars,isOk:Bool)
    = (IsCE_STATE_DECRYPT_PENDING(cestate(vars)))
       -> (isOk)
           -> (short(cyphermsg(doDecrypt(vars))) != ShortNull
               && panId(cyphermsg(doDecrypt(vars))) != PanNull)
               -> olcHandleDecryptDone.
                  setConfig.
                  SendResponse(MSG_UcastResp(Msg_ERR_OK(cyphermsg(doDecrypt(vars))))).
                  OLC(setCfgShortAddr(short(cyphermsg(doDecrypt(vars))),
                      setCfgPanId(panId(cyphermsg(doDecrypt(vars))),
                      setDoDecrypt(DecryptEmpty,
                      setCeState(CE_STATE_COMMISSIONED,
                      setCommState(COMM_STATE_COMMISSIONED_NEEDS_REBOOT,
                      setRFNwk_CommissionFlag(COMMISSION_VALID,vars)))))))
               <> olcDecryptError.
                  SendResponse(MSG_UcastResp(Msg_CMD_INVALID(cyphermsg(doDecrypt(vars))))).
                  OLC(setCeState(CE_STATE_NOT_COMMISSIONED,setDoDecrypt(DecryptEmpty,vars)))
           <> olcDecryptError.
              SendResponse(MSG_UcastResp(Msg_CMD_INVALID(cyphermsg(doDecrypt(vars))))).
              OLC(setCeState(CE_STATE_NOT_COMMISSIONED,setDoDecrypt(DecryptEmpty,vars)))
       <> olcStateNotOK.
          SendResponse(MSG_UcastResp(Msg_CMD_INVALID(cyphermsg(doDecrypt(vars))))).
          OLC(setDoDecrypt(DecryptEmpty,vars))
    ;

act olcSendResp,olcSendResp_drop:Msg;
proc
```

```
SendResponse(m:Msg)
= olcSendResp(m)
+ olcSendResp_drop(m)
;

%NOTE: V1 OLCs worden niet ondersteund
InitOLCModel
= sum cf:RFNwk_CommissionFlag, short:ShortAddr, pan:PanId, ieee:IeeeAddr.
  (ieee != IeeeNull)
   -> APP_Init(ieee,short,pan,IMGs(IMG(V2), IMG(V2), NORMAL_IMAGE),cf)
;
```

I Rename file for requirement: An OLC may never be in an illegal state

Filename: `rename_illegal_state.rename`
Description: This file is used to rename an illegal action to action *illegal*. Such that the state space can be searched to prove the absence of the illegal situation.

```
act illegal;
var
 ieee:IeeeAddr;
 short:ShortAddr;
 ocs:OLCCommState;
 ces:CEState;
 cf:RFNwk_CommissionFlag;
 pan:PanId;

rename
 ((ocs==COMM_STATE_COMMISSIONED || cf==COMMISSION_VALID || ces==CE_STATE_COMMISSIONED)
  && (short==ShortNull || pan==PanNull))
  -> olcState(ocs,ces,cf,ieee,short,pan) => illegal;
```

J Rename file for requirement: An OLC cannot arrive in a state from which it cannot be commissioned or decommissioned again.

Filename: `hide_actions_for_R02.rename`
Description: This file is used to rename all actions to τ, apart from the action *olcIsCommissioned*(*ieee,short,pan,b*), which is renamed to the shorter variant *IsCommissioned*(b). With this, the requirement: 'An OLC cannot arrive in a state from which it cannot be commissioned or decommissioned' can be verified.

```
act IsCommissioned: Bool;
var
 m:Msg;
 ieee:IeeeAddr;
 short:ShortAddr;
 b:Bool;
 ocs:OLCCommState;
 ces:CEState;
 cf:RFNwk_CommissionFlag;
 pan:PanId;
 oi:OlcInfo;

rename
 armForReboot => tau;
 olcDoDecrypt => tau;
 olcAppInit => tau;
 olcHandleMessage => tau;
 olcHandleRebootMessage => tau;
 olcHandleCommissionMsg0 => tau;
 olcHandleCommissionMsg1 => tau;
 olcHandleDeCommissionMessage => tau;
 olcHandleDecryptDone => tau;
 olcIsCommissioned(ieee,short,pan,b) => IsCommissioned(b);
 olcCommEngInit => tau;
 olcHandled => tau;
 olcReboot => tau;
 olcCmdInvalid => tau;
 msgNotFor(ieee,short) => tau;
 olcCommissionMsg1Handled => tau;
 olcStateNotOK  => tau;
 requestDecryptOK => tau;
 olcDecryptError => tau;
 setConfig => tau;
 olcState(ocs,ces,cf,ieee,short,pan) => tau;
 skip => tau;
 olcSendResp(m) => tau;
 com(m,b) => tau;
 c_registerAddresses(ieee,short) => tau;
 s_no_response_olc => tau;
 infoOlc(ieee,short,pan,ocs) => tau;
 c_unregisterShortAddress(short) => tau;
 requestDecryptFailed => tau;
```