Formal Modelling in the Concept Phase of Product Development

Mathijs Schuts[1], **and Jozef Hooman**[2,3]
[1]Philips HealthTech, Best, The Netherlands
[2]Embedded Systems Innovation (ESI) by TNO, Eindhoven, The Netherlands
[3]Radboud University, Nijmegen, The Netherlands

Abstract— *The traditional process framework for product realisation in industry often leads to a long and difficult integration phase. An important reason is that in the concept phase only informal descriptions are made about the required product, its decomposition, and the interfaces between components. We propose a formal modelling approach for the concept phase, using a new light-weight modelling tool to formalize system behaviour, decomposition and interfaces. The confidence in the product concept is increased by simulation, both manual and automatic with random system characteristics. By means of a dedicated graphical user interface, communication with different stakeholders is improved. We discuss the application of the proposed approach at Philips HealthTech.*

Keywords: Formal models; software engineering; concept definition; simulation

1. Introduction

We propose a method to improve the concept phase of product realisation by means of formal techniques. A traditional development process from concept to a validated product is depicted in Figure 1, see for instance [1]. It describes six distinct phases between concept and product. During the concept phase an informal document is being created with a high level description of the concept. This document is reviewed and agreed upon by all stakeholders. The document consists of a decomposition of the developed product, the different hardware and software components it consists of, the responsibilities per component, and the interaction between the components, possibly with an informal interface description. From the concept description, different development groups concurrently start developing the component they are responsible for. This may also include 3rd party components developed by other companies.

Such a process framework provides a structured way to come from concept to product and allows the concept to be decomposed into different components such that multiple development groups can concurrently work on the different components. A frequently occurring problem in industry, however, is that the integration and validation phase takes a large amount of time and is rather uncontrollable because many problems are detected in this phase and might require a redesign of components.

An important reason for these problems is the informal nature of the concept phase. Clearly, this leads to ambiguities and inconsistencies. Moreover, only a part of the complete behaviour is described in an informal document, often only a part of the basic functional behaviour without taking errors or non-functional aspects into account. The complete behaviour is defined during the implementation phase of the different components. Hence, a large part of system behaviour is implicitly defined during the implementation phase. If multiple development groups work in parallel in realizing the concept, the integration phase can take a lot of time because the independently developed components do not work together seamlessly. Another problem is that during the integration phase sometimes issues are found in which hardware is involved. Then it is usually too late to change the hardware and a workaround in software has to be found.

To prevent these types of problems, we propose the use of formal modelling techniques in the concepts phase, because it is early in the process and all consecutive phases can benefit from an improved unambiguous concept description. Moreover, errors made in this phase are very costly to repair in a later phase [2], [3].

By making a formal model of the system in the concept phase, ambiguities, contradictions and errors are removed from the informal concept description. During modelling one is forced to think about the exceptional behaviour early in the development process. Many questions needs to be answered which would be implicitly defined during the implementation phase otherwise. Moreover, by formalizing interface descriptions, less problems during the integration phase are expected. Figure 2 depicts a graphical representation of the proposed extension of the product realisation framework.

The formal model is developed incrementally to allow updates after aligning with stakeholders and to incorporate new insights frequently. Before choosing a formal method, we first list the aspects that are important in the concept phase:

- The definition of complete system behaviour, including error scenarios.
- A clear and unambiguous definition of interfaces and concepts to support parallel development in subsequent phases.
- The possibilities to explore concepts and design decisions fast.
- Communication with stakeholders to obtain agreement on the concepts and externally visible behaviour of the product.
- The possibility to deal with a combination of hardware and software components.

Furthermore, the formal method should be easy to use by industrial engineers and scalable to large and complex systems. Based on earlier experiences, see, e.g., [4], we decided not to aim for exhaustive model checking. Since our applications consist of many asynchronous components with queues and also timing aspects are important, one almost immediately runs into state-space explosion problems.

As an alternative to increase the confidence in the system model, we will use simulation. Formal models are expressed using the Parallel Object Oriented Specification Language (POOSL). The language is supported by a simulator and a new Eclipse Integrated Development Environment (IDE). The tooling can easily be combined with a dedicated graphical user interface to support communication with all stakeholders.

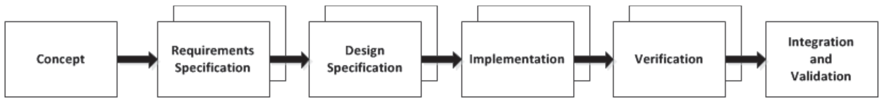


Fig. 1: Traditional process framework

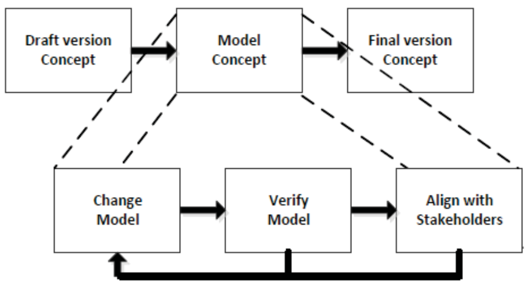


Fig. 2: Model-based concept phase

The use of formal techniques in the concept phase of hardware development has been proposed in [5]. The approach uses ACL2 logic [6] for the specification of the communication structure of a system on chip. Formal proofs of desirable properties, e.g., messages reach their destination, show the correctness of the specifications.

The application of formal methods early in the development process was already described in [7]. It describes the application of tools such as PVS [8] to requirements modelling for spacecraft fault protection systems. Although the specification language of PVS appears to be easy understandable by engineers, the interactive proof of properties is far from trivial. Hence, the conclusion of [7] proposes a rapid prototyping approach, where prototypes are tested against high level objectives.

The difficulty to use formal methods early in the development process, when there are many uncertainties and information changes rapidly is also observed in [9]. They investigated the use of formal simulations based on rewriting logic, namely Maude executable specifications [10]. The approach has been applied to the design of a new security protocol.

The paper is organised as follows. More details about POOSL and tool support can be found in Section 2. Section 3 describes the application at Philips HealthTech where the proposed method has been used. The models made for this application are presented in Section 4. Section 5 contains our concluding remarks.

2. POOSL

The long-term goal of the POOSL tooling is to shorten the development time of complex high-tech systems by providing a light-weight modelling and simulation approach. It is targeted at the early phases of system development, where requirements might not yet be very clear and many decisions have to be taken about the structure of the system, the responsibilities and behaviour of the components, and their interaction.

The approach fills a gap between expensive commercial modelling tools (like MATLAB [11] and Rational Rhapsody [12]) that require detailed modelling, often close to the level of code, and drawing tools (such as Visio and UML drawing tools) that do not allow simulation. More related to the POOSL approach is the OMG specification called the Semantics of a Foundational Subset for Executable UML Models (fUML) [13] with, e.g., the Cameo Simulation Toolkit [14].

In Section 2.1 we introduce the POOSL modelling language and describe the available tool support in Section 2.2.

2.1 POOSL modelling language

POOSL is a modelling language for systems that include both software and digital hardware. It is not intended for continuous aspects, e.g., modelling physical processes by differential equations is not possible. POOSL is an object-oriented modelling language with the following aspects:

- *Concurrent parallel processes* A system consists of a number of parallel processes. A process is an instance of a process class which describes the behaviour of the process by means of an imperative language. A process has a number of ports for message-based communication with its environment.
- *Hierarchical structure* A number of processes can be grouped into a cluster. A cluster is an instance of a cluster class which has a number of external ports and specifies how the ports of its processes are connected.
- *System definition* A system is defined by a number of instances of processes and clusters and the connections between the ports of its instances.
- *Synchronization* Processes communicate by synchronous message passing along ports, similar to CSP [15] and CCS [16]. That is, both sender and receiver of a message have to wait until a corresponding communication statement is ready to execute. A process may contain parallel statements which communicate by shared memory.
- *Timing* Progress of time can be represented by statements of the form *delay(d)*. It postpones the execution of the process by *d* time units. All other statements do not take time. Delay statements are only executed if no other statement can be executed.
- *Object-oriented data structures* Processes may use data objects that are instances of data classes. Data objects are passive sequential entities which can be created dynamically. A number of structures are predefined, such as set, queue, stack, array, matrix, etc.
- *Stochastic behaviour* The language supports stochastic distribution functions; a large number of standard distribution functions are predefined, such as DiscreteUniform, Exponential, Normal, and Weibull.

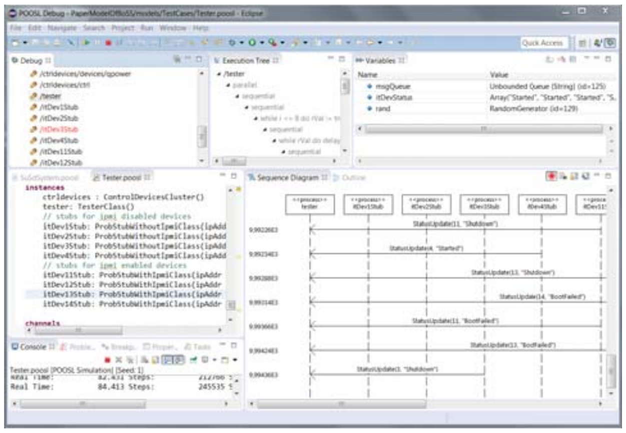
The formal semantics of POOSL has been defined in [17] by means of a probabilistic structural operational semantics for the process layer and a probabilistic denotational semantics for the data layer.

2.2 POOSL tooling

As explained in [17], the operational semantics of POOSL has been implemented in a high-speed simulation engine called Rotalumis. It supports the Software/Hardware Engineering (SHE) methodology [18]. The tool SHESim [19] is intended for editing POOSL models and validating them by interactive simulation. Recently, a modern Eclipse IDE has been developed on top of an improved Rotalumis simulation engine. The combination of the last two tools have been used for the application described in this paper.

The Eclipse IDE is free available [20] and supports advanced textual editing with early validation and extensive model debugging possibilities. It is easy to use for industrial users and scalable to large systems; it is possible to define and simulate systems with hundreds of components. The tool contains on-line explanation and documentation.

Model validation is convenient to detect modelling errors early, before they appear during simulation. It includes checks on undeclared variables and ports, types, unconnected ports, and mismatches between send and receive statements. The debugging view shown below allows step-wise execution of models, inspection of variables, setting of breakpoints, and a running sequence diagram during simulation.



```
instances
    ctrldevices : ControlDevicesCluster()
    tester: TesterClass()
    // stubs for ipmi disabled devices
    itDev1Stub: ProbStubWithoutIpmiClass(ipAdd
    itDev2Stub: ProbStubWithoutIpmiClass(ipAdd
    itDev3Stub: ProbStubWithoutIpmiClass(ipAdd
    itDev4Stub: ProbStubWithoutIpmiClass(ipAdd
    // stubs for ipmi enabled devices
    itDev11Stub: ProbStubWithIpmiClass(ipAddr
    itDev12Stub: ProbStubWithIpmiClass(ipAddr
    itDev13Stub: ProbStubWithIpmiClass(ipAddr
    itDev14Stub: ProbStubWithIpmiClass(ipAddr

channels
```

```
Tester.poosl [POOSL Simulation] [Seed: 1]
Real Time:        82.431 Steps:        212766
Real Time:        84.413 Steps:        245535
```

3. Application at Philips

The proposed approach has been applied at Philips HealthTech, in the context of the innovation of interventional X-ray systems. These systems are intended for minimally invasive treatment of mainly cardiac and vascular diseases. The system provides advanced X-ray images to guide the physician through the arteries of the patient to the point of interest and to execute a certain medical procedure, such as placing a stent. For a new product release, we have created a new concept for starting up and shutting down the system. This section briefly describes the informal concepts of the new start-up/shut-down (SU/SD) behaviour.

An interventional X-ray system contains a number of IT devices such as computers and touch screen modules. All IT devices can communicate with each other via an internal Ethernet control network. The IT devices are configured in such a way that they immediately start-up once they are powered. There is a central SU/SD controller which coordinates SU/SD scenarios. A user of the system can initiate a SU/SD scenario by pressing a button on the User Interface (UI). The SU/SD controller will then instruct the power distribution component to switch power taps on or off and send notification messages to the various IT devices over the internal Ethernet control network. Another scenario can be initiated by the Uninterruptable Power Supply (UPS), for instance, when mains power source fails or when mains power recovers.

The system is partitioned into two segments: A and B (for reasons of confidentiality, some aspects have been renamed). This partitioning is mainly used in the case of a power failure. When all segments are powered and the mains power is lost, the UPS takes over. Once this happens, the A segment is shut down in a controlled way, leaving the B segment powered by the battery of the UPS. If the battery energy level of the UPS becomes critical, also the B segment is shut down in a controlled way. Usually, the diesel generator of the hospital will provide power before this happens. An IT device is part of either the A segment or the B segment.

The new SU/SD concept uses the Intelligent Platform Management Interface (IPMI) [21], a standard interface to manage and monitor IT devices in a network. The IT devices in our system are either IPMI enabled or IPMI disabled.

- IPMI disabled IT devices are started and stopped directly by switching the power tap on or off.
- IPMI enabled IT devices are on a power tap that is continuously powered. To start-up these IT devices, the SU/SD controller sends a command via IPMI to them.

Combined with the two types of segments, this leads to four types of IT devices, as depicted in Figure 3.

This figure also shows that there are several communication mechanisms between the components

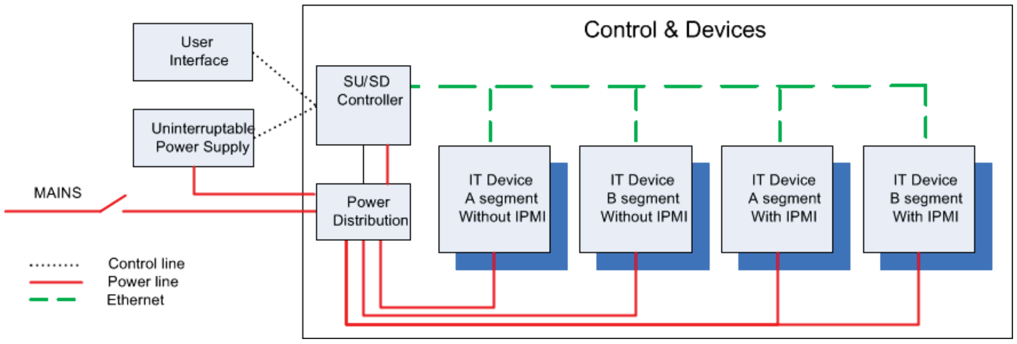
- Power lines for turning the power on and off.
- Control lines to connect the controller to the UI and the UPS.
- The internal Ethernet network, which is used for different purposes:
  - By the IT devices, to request the SU/SD state of the SU/SD controller and to receive SU/SD notification messages from this controller.
  - By the SU/SD controller, to ping the Operating System (OS) of an IPMI disabled IT device to observe its shut down.
  - By the SU/SD controller, to turn on an IPMI enabled IT device and to observe the shut down of the device.

A mains disconnector switch (MDS) can be used to power the complete system. An example of a SU/SD scenario is the shut-down scenario. When all segments are powered and the SU/SD controller detects that the AllSegmentOff button is pressed by the user, it will send an AllSegmentOff-pressed notification to all registered IT devices. Next all IT devices go through the following shut-down phases:

- The applications and services running on the IT device are stopped.
- The IPMI disabled IT devices will register themselves and ask the SU/SD controller to observe their shut-down. This is needed because the controller does not know which IPMI disabled devices are connected to a power tap. The IPMI enabled devices are known to the controller by configuration.
- Once the applications and services are stopped, the OS will be shut down.

The scenario ends when the SU/SD controller has detected that all IT devices are shut down. IPMI disabled IT devices are pinged to observe that they are shut down and IPMI enabled IT devices are requested for their state via IPMI to detect that they are shut down. Next the SU/SD controller will instruct the power distribution component to turn off the switchable power taps with which the IPMI disabled IT devices are powered. The IT tap that powers the IPMI enabled IT devices remains powered while these devices are in the standby state.

In the past, an abstract model of the current start-up and shut-down concept for a simpler version of the system has been made for three model checkers: mCRL2 [22], FDR2 [23] and CADP [24]. For reasons of comparison, exactly the same model was made for all three tools, leading to 78,088,550 states and 122,354,296 transitions. Model checking such a model easily takes hours. The new concept described here is far more complex because of the many asynchronous IT devices that all exhibit different behaviour. For example, the IT devices can sometimes fail to start-up or shut down. Also the timing and order in which they start-up and shut down might be different. Hence, the new concept is too complex to model check. Consequently, we decided to model the system in POOSL and used simulation to increase the confidence in the concepts.



Control & Devices
User Interface
Uninterruptable Power Supply
SU/SD Controller
MAINS
Power Distribution
IT Device A segment Without IPMI
IT Device B segment Without IPMI
IT Device A segment With IPMI
IT Device B segment With IPMI
Control line
Power line
Ethernet

Fig. 3: System overview

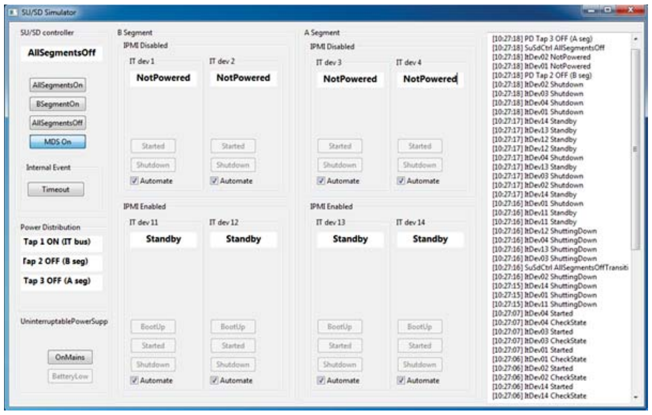
4. Modeling the SU/SD Concept in POOSL

This section describes the incremental approach to model the SU/SD concepts in POOSL. The scope of the model and the simulation environment is described in Section 4.1. Section 4.2 contains the modelling steps. A few details of the POOSL models can be found in Section 4.3. Our approach to test models automatically is presented in Section 4.4.

4.1 Modelling Scope and Simulator

The aim was to model the Control & Devices part of Figure 3 in POOSL. Besides the SU/SD Controller and the Power Distribution, the model should contain all four types of IT devices, i.e., all combinations of segments (A and B) and IPMI support. Moreover, to capture as much as possible of the timing and ordering behaviour, we decided to include two instances of each type.

To be able to discuss the main concepts to stakeholders, we connect the POOSL model to a simulation of the environment of the Control & Devices part. We created a Simulator in Java with the use of WindowBuilder in Eclipse to allow the manual execution of scenarios. It allows sending commands from the User Interface and power components to the model and displaying information received from the model. Additionally, one can observe the status of IT devices and even influence the behaviour of these devices, e.g., to validate scenarios in which one or more IT devices do not start-up or shut down properly. The next figure shows a screenshot of the SU/SD simulator.



SU/SD controller: AllSegmentsOff

Power Distribution: Tap 1 ON (IT bus) · Tap 2 OFF (B seg) · Tap 3 OFF (A seg)

	IT dev 1	IT dev 2	IT dev 3	IT dev 4
IPMI Disabled	NotPowered	NotPowered	NotPowered	NotPowered

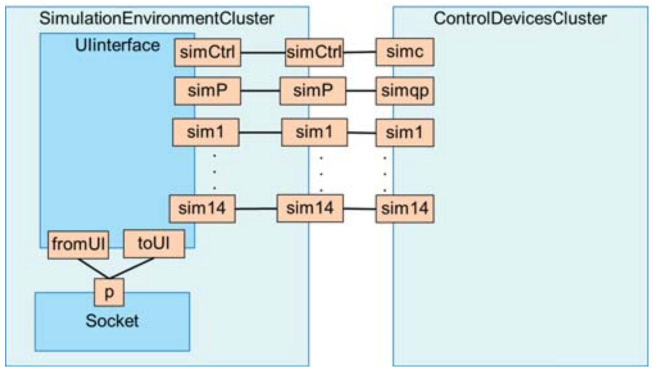
	IT dev 11	IT dev 12	IT dev 13	IT dev 14
IPMI Enabled	Standby	Standby	Standby	Standby

```
[10:27:18] PD Tap 3 OFF (A seg)
[10:27:18] SuSdCtrl AllSegmentsOff
[10:27:18] ItDev02 NotPowered
[10:27:18] ItDev01 NotPowered
[10:27:18] PD Tap 2 OFF (B seg)
[10:27:18] ItDev02 Shutdown
[10:27:18] ItDev03 Shutdown
[10:27:18] ItDev04 Shutdown
[10:27:18] ItDev01 Shutdown
[10:27:17] ItDev14 Standby
[10:27:17] ItDev13 Standby
[10:27:17] ItDev12 Standby
[10:27:17] ItDev12 Standby
[10:27:17] ItDev04 Shutdown
[10:27:17] ItDev13 Standby
[10:27:17] ItDev03 Shutdown
[10:27:17] ItDev02 Shutdown
[10:27:17] ItDev14 Standby
[10:27:16] ItDev01 Shutdown
[10:27:16] ItDev11 Standby
[10:27:16] ItDev11 Standby
[10:27:16] ItDev12 ShuttingDown
[10:27:16] ItDev04 ShuttingDown
[10:27:16] ItDev13 ShuttingDown
[10:27:16] ItDev03 ShuttingDown
[10:27:16] SuSdCtrl AllSegmentsOffTransiti
[10:27:16] ItDev02 ShuttingDown
[10:27:15] ItDev14 ShuttingDown
[10:27:15] ItDev01 ShuttingDown
[10:27:15] ItDev11 ShuttingDown
[10:27:07] ItDev04 Started
[10:27:07] ItDev04 CheckState
[10:27:07] ItDev03 Started
[10:27:07] ItDev03 CheckState
[10:27:07] ItDev01 Started
[10:27:06] ItDev01 CheckState
[10:27:06] ItDev02 Started
[10:27:06] ItDev02 CheckState
[10:27:06] ItDev14 Started
[10:27:06] ItDev14 CheckState
```

There are three main columns:

- The left column contains three parts:
  - On the top, the state and the UI buttons to control the SU/SD controller are displayed.
  - In the middle, the tap state of the segments is displayed.
  - On the bottom, the UPS triggers are displayed.
- The middle part contains a column for the B segment and one for the A segment; each contains a row for the IPMI disabled IT devices and one for the IPMI enabled IT devices. For each IT device the state is displayed. The start-up and shut-down behaviour of an IT device can be simulated automatically or it can be set to manual to simulate error scenarios, where the system might fall into a Timeout (see the Internal Event in the column of the SU/SD controller).
- In the right column, the status updates of the model are displayed.

The Java simulation is connected to POOSL by means of a socket. The structure of the POOSL system model is depicted in the next figure.



The system part to be modelled (the Control & Devices part) is represented by cluster ControlDevicesCluster. It has 10 external ports, one to communicate with the SU/SD controller (simc), one for power commands (simqp) and 8 for the IT devices: sim1, sim2, sim3, and sim4 for IPMI disabled devices; sim11, sim12, sim13, sim14 for IPMI enabled devices. These ports are connected to corresponding ports of the SimulationEnvironmentCluster. This cluster contains an instance of the standard Socket class provided by the POOSL library. Class UIinterface is responsible for the translation between strings of the socket interface and the SU/SD system interface.

4.2 Modelling steps

After the simulator was build, the ControlDevicesCluster has gradually been defined in POOSL. The proposed framework defines

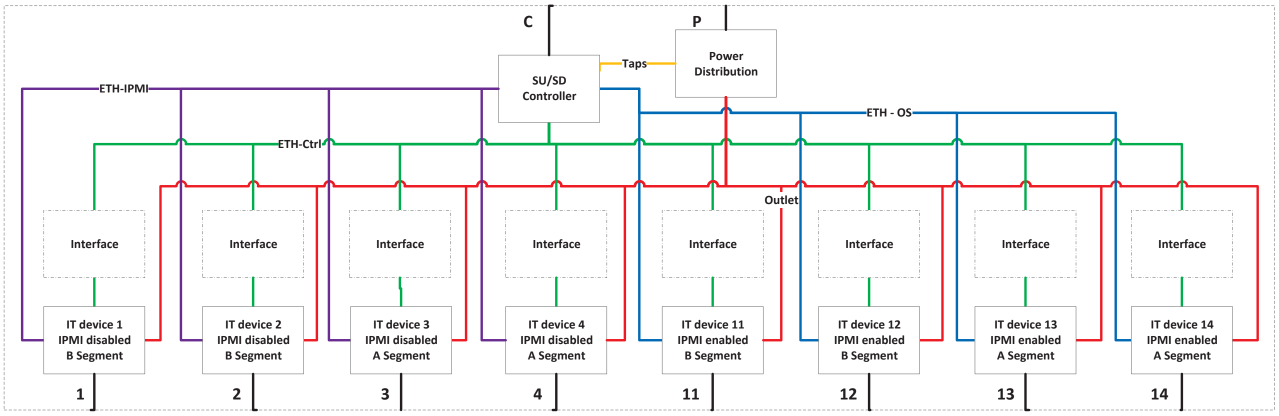

C
P
SU/SD Controller
Power Distribution
Taps
ETH-IPMI
ETH-Ctrl
ETH - OS
Outlet
Interface
IT device 1 IPMI disabled B Segment
IT device 2 IPMI disabled B Segment
IT device 3 IPMI disabled A Segment
IT device 4 IPMI disabled A Segment
IT device 11 IPMI enabled B Segment
IT device 12 IPMI enabled B Segment
IT device 13 IPMI enabled A Segment
IT device 14 IPMI enabled A Segment
1
2
3
4
11
12
13
14

Fig. 4: Structure of the POOSL model of the ControlDevicesCluster

an incremental approach to build the model of the concept. We have used the simulator to validate the intermediate models and align the behaviour with internal stakeholders.

We started with a model of an IPMI disabled IT device and a model of the SU/SD controller for shutting down these IT devices of the A segment. In this model there were two instantiations of IPMI disabled IT devices. Note that POOSL supports a partial model where not all ports are used.

This model has been extended gradually to a model where all 8 instances of IT devices are present. Next, the SU/SD controller was extended with error behaviour to verify, for instance, that the system is always in a defined state after shut-down, which is an important requirement.

Finally, we added a model of the interface between the IT device and the SU/SD controller, because these two components will be developed concurrently. Hence, it is important to specify this contract formally and to verify it. Every IT device has an instance of the same interface model, which is implemented in such a way that the system will deadlock if the formal interface is violated. Hence, interface compliance is verified continuously during simulation.

The structure of the resulting model of this incremental approach is depicted in Figure 4.

4.3 Modelling Devices and Control

This section provides some details of the POOSL models. The first part of the model of an IT device with IPMI is shown below. It imports a library which, e.g., defines queues. Next the process class is defined, including two parameters for the IP address and the segment. All IT devices have an IP address to be able to connect them to the same network. Subsequently, the ports, the messages (only one is shown here), the variables and the initial method are defined. Note that the variables define two queues.

In the initial method *init()()*, the queues are initialized, which are FIFO by default. Next the method defines three parallel activities. The first activity defines a state machine, where the states are represented by methods. It starts the state machine by calling the initial state *ItDevNotPowered()()*.

```
import "../libraries/structures.poosl"
process class ItDevWithIpmiClass(ipAddr : Integer,
                                 segment: String)
ports
    outlet, con, ipmi, sim
messages
    outlet ? On,…
variables
    msgQueue : Queue,
    ipmiQueue: Queue
init
    init()()
methods
init()() /* initial method */
    msgQueue := new(Queue);
    ipmiQueue := new(Queue);
    par
        ItDevNotPowered()()
    and
        MsgReceiveBuffer()()
    and
        IpmiReceiveBuffer()()
    rap
```

Below we show a typical definition of a state, in this case state *ItDevShuttingDown()()*.

```
ItDevShuttingDown()() | m : String |
sel
    [!(ipmiQueue isEmpty())] m := ipmiQueue remove();
    if m = "status" then ipmi ! On(ipAddr) fi;
    ItDevShuttingDown()()
or sim ? Shutdown; ItDevPowered()()
or outlet ? On; ItDevShuttingDown()()
or outlet ? Off; ItDevNotPowered()()
or sim ? Started; ItDevShuttingDown()()
or sim ? BootUp; ItDevShuttingDown()()
les
```

The state is defined as a method with local variable *m*. It selects the next state based on the contents of the *ipmiQueue* or the receipt (indicated by "*?*") of a particular message on one of its ports. Since switching a power tap on or off is instantaneous and cannot be refused by a process, all states allow the receipt of messages *On* and *Off* via port *outlet*.

The other two parallel activities of the *init()()* method are used to model the asynchronous nature of the Ethernet communication. Method *MsgReceiveBuffer* receives messages on port *con* and stores them in queue *msgQueue*.

```
MsgReceiveBuffer()() | m : String, ip : Integer |
    con ? RecvEvent(m, ip | ip = ipAddr);
    msgQueue add(m);
    delay(1);
    MsgReceiveBuffer()()
```

Note that POOSL allows a condition on the receive statement to express that only messages with the corresponding IP address are received. Similarly, method *IpmiReceiveBuffer* stores messages in *ipmiQueue*.

4.4 Extensive Model Testing

The simulator has been used to align the behaviour with internal stakeholders and to get confidence in the correctness of the behaviour. To increase the confidence without the need of many manual mouse clicks, we created a separate test environment in POOSL. Therefore, a stub is connected to every IT device. A stub is a process which randomizes the start-up and shut-down timing of an IT device. In addition, a stub randomly decides if a device fails to start-up or shut-down. Also in these random cases the system has to respond well and it needs to be forced into defined states. The next POOSL fragment depicts how the random timing and random behaviour is implemented in the Stub.

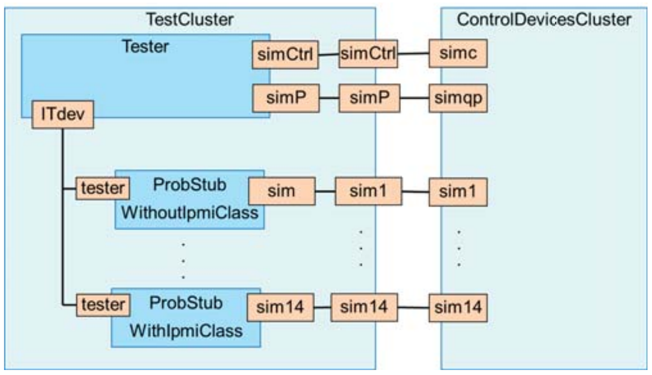
```
process class ProbStubWithoutIpmiClass
                        (ipAddr      : Integer,
                         StartUpProp : Real,
                         ShutDownProp: Real)
ports
    sim, tester
    ...

Loop()() | message : String |
[!(msgQueue isEmpty())] message := msgQueue remove();
if message = "Booting" then
    delay(rand random * 5.0);
    if rand random <= StartUpProp then
        sim ! Started;
        tester ! StatusUpdate(ipAddr, "Started")
    else
        tester ! StatusUpdate(ipAddr, "StartFailed")
    fi
fi;
```

The stubs are configured such that they fail to start-up or shut-down in 10% of the cases.

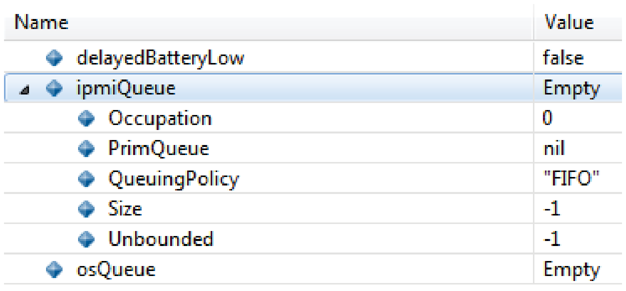
```
// stubs for ipmi disabled devices
itDev1Stub: ProbStubWithoutIpmiClass(ipAddr := 1,
                                     StartUpProp := 0.9,
                                     ShutDownProp := 0.9)
itDev2Stub: ProbStubWithoutIpmiClass(ipAddr := 2, □
itDev3Stub: ProbStubWithoutIpmiClass(ipAddr := 3, □
itDev4Stub: ProbStubWithoutIpmiClass(ipAddr := 4, □
// stubs for ipmi enabled devices
itDev11Stub: ProbStubWithIpmiClass(ipAddr := 11,
                                   StartUpProp := 0.9,
                                   ShutDownProp := 0.9)
itDev12Stub: ProbStubWithIpmiClass(ipAddr := 12, □
itDev13Stub: ProbStubWithIpmiClass(ipAddr := 13, □
itDev14Stub: ProbStubWithIpmiClass(ipAddr := 14, □
```

In reality the IT devices are quite reliable, but to reduce testing time it is more convenient to make the IT devices less reliable. Moreover, we are interested in the error handling behaviour of the system and not in the statistical behaviour.

For the execution of scenarios initiated by a user and the UPS, a Tester process has been created to automatically drive the system. Every stub has a feedback channel to the Tester to report the status of an IT device. The next figure depicts how the Tester and Stubs are connected to the system.



The definition of the Tester is such that it leads to a deadlock when the SU/SD controller or the IT devices do not behave as intended. Already during the first simulation run we experienced such a deadlock. The cause of the problem was found using the debug possibilities of the new POOSL IDE. We simulated the model in debug mode and inspected the sequence diagram when the deadlock occurred. In this sequence diagram we saw a problem with a message about the IPMI status of an IT device. Next we inspected the variables window shown below.



Name	Value
delayedBatteryLow	false
ipmiQueue	Empty
Occupation	0
PrimQueue	nil
QueuingPolicy	"FIFO"
Size	-1
Unbounded	-1
osQueue	Empty

It revealed that the ipmiQueue was empty, which was not expected at this point in the execution. When checking the code that handles the IPMI queue, we found that the queue was emptied after the IPMI status request has been send. The race condition was fixed by changing the order; first empty the queue and then send the IPMI status request. After fixing the race condition, the model has been executed 100 000 random start-up and shut-down cycles without experiencing a single deadlock.

5. Concluding Remarks

In the concept phase of product definition, we have used a formal system description in POOSL in combination with a graphical user interface to align stakeholders and get confidence in the behaviour of the system. We have added a model with a formal interface description between two important components of the system that will be developed concurrently. To increase the confidence in the concept, we created an automated test driver for the system with stubs that exhibit random behaviour and random timing.

While modelling, we found several issues that were not foreseen in the draft concept. We had to address issues that would otherwise have been postponed to the implementation phase and which might easily lead to integration problems. We observed that the definition of a formal executable model of the SU/SD system required a number of design choices. We give two examples of such choices.

- If all segments are on and the UPS indicates that the mains power input fails, then the system will shut down the A segment. If, however, during this transition one or more of the IPMI enabled IT devices fail to shut down, then the SU/SD controller has no way to force these IT devices into the right state. This could be solved by an additional tap, but given the costs of an extra tap and the small chance that this will happen (both mains power and shut down of an IT device should fail), we have decided to leave it this way. If the user experiences unexpected behaviour of the system, the user can always recover the system by turning it off and on again.
- An early version of the SU/SD controller did not track if an IPMI enabled IT device did in fact start up. However, if something is wrong with the start-up or shut-down of an IPMI enabled IT device, we want to toggle the power during shut-down in the hope that a reset will solve the issue. Once we found the described issue with the simulator, we extended the model of the SU/SD controller with a storage of the start-up status of an IPMI enabled IT device.

In addition, the model triggered many discussions about the combined behaviour of the hardware and software involved in start-up and shut-down. This resulted in a clear description of responsibilities in the final concept. Also the exceptional system behaviour when errors occur has been elaborated much more compared to the traditional approach. Note that the modelling approach required a relatively small investment. The main POOSL model and the Java simulator were made in 40 hours; the tester and the stubs required another 10 hours.

The application of exhaustive model-checking techniques to the full model is not feasible, give the large number of concurrent processes and the use of queues for asynchronous communication. Scalability problems are, for instance, reported in [25], where a transformation of POOSL models to Uppaal [26], a model-checker for timed systems, is applied to an industrial application. However, it might be possible to apply these techniques to verify certain aspects on an abstraction of the model.

In the future, we want to use the test driver for the model to validate the behaviour of the SU/SD controller by means of model-based testing. Since the interface between the test driver and the model is equal to the interface between test driver and the real implementation, we might also use our test approach for the realized system when it become available. The idea is to use the test driver and a thin manually written adapter that makes an Ethernet connection between the test driver and the real implementation.

References

[1] S. R. Koo, H. S. Son, and P. H. Seong, “Nusee: Nuclear software engineering environment,” in *Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems*, ser. Springer Series in Reliability Engineering. Springer London, 2009, pp. 121–135.

[2] B. Boehm and V. Basili, “Software defect reduction top 10 list,” *IEEE Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[3] J. Westland, “The cost of errors in software development: evidence from industry,” *The Journal of Systems and Software*, vol. 62, pp. 1–9, 2002.

[4] J. Groote, A. Osaiweran, M. Schuts, and J. Wesselius, “Investigating the effects of designing industrial control software using push and poll strategies,” Eindhoven University of Technology, the Netherlands, Computer Science Report 11/16, 2011.

[5] J. Schmaltz and D. Borrione, “A functional approach to the formal specification of networks on chip,” in *Formal Methods in Computer-Aided Design*, ser. LNCS, no. 3312. Springer–Verlag, 2004, pp. 52–66.

[6] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[7] S. M. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, “Experiences using lightweight formal methods for requirements modeling,” *IEEE Trans. Software Eng.*, vol. 24, no. 1, pp. 4–14, 1998.

[8] S. Owre, J. Rushby, N. Shankar, and F. von Henke, “Formal verification for fault-tolerant architectures: prolegomena to the design of pvs,” *Software Engineering, IEEE Transactions on*, vol. 21, no. 2, pp. 107–125, Feb 1995.

[9] A. Goodloe, C. A. Gunter, and M.-O. Stehr, “Formal prototyping in early stages of protocol design,” in *Proc. of the 2005 Workshop on Issues in the Theory of Security*, ser. WITS ’05. ACM, 2005, pp. 67–80.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, “Maude: specification and programming in rewriting logic,” *Theoretical Computer Science*, vol. 285, no. 2, pp. 187 – 243, 2002.

[11] MathWorks, “Matlab and Simulink,” 2015. [Online]. Available: www.mathworks.com

[12] IBM, “Rational Rhapsody,” 2015. [Online]. Available: www.ibm.com/software/products/en/ratirhapfami

[13] OMG, “Semantics of a foundational subset for executable UML models (fUML),” 2015. [Online]. Available: http://www.omg.org/spec/FUML/

[14] MagicDraw, “Cameo simulation toolkit,” 2015. [Online]. Available: http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html

[15] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, “A theory of communicating sequential processes,” *J. ACM*, vol. 31, pp. 560–599, 1984.

[16] R. Milner, *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[17] L. van Bokhoven, “Constructive tool design for formal languages; from semantics to executing models,” Eindhoven University of Technology, the Netherlands,” PhD thesis, 2004.

[18] SHE, “System-level design with the SHE methodology,” 2015. [Online]. Available: www.es.ele.tue.nl/she/

[19] M. Geilen, “Formal techniques for verification of complex real-time systems,” Eindhoven University of Technology, the Netherlands,” PhD thesis, 2002.

[20] POOSL, “Parallel Object-Oriented Specification Language,” 2015. [Online]. Available: poosl.esi.nl

[21] Intel, “Intelligent Platform Management Interface (IPMI) - specifications,” 2015. [Online]. Available: www.intel.com/content/www/us/en/servers/ipmi/ipmi-specifications.html

[22] J. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. v. Weerdenburg, W. Wesselink, T. Willemse, and J. v. d. Wulp, “The mCRL2 toolset,” in *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008.

[23] F. Systems, “Failures-Divergences Refinement (FDR),” 2015. [Online]. Available: www.fsel.com

[24] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, “CADP 2011: a toolbox for the construction and analysis of distributed processes,” *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 2, pp. 89–107, 2013.

[25] J. Xing, B. Theelen, R. Langerak, J. van de Pol, J. Tretmans, and J. Voeten, “From POOSL to UPPAAL: Transformation and quantitative analysis,” in *10th Int. Conf. on Application of Concurrency to System Design (ACSD)*, 2010, pp. 47–56.

[26] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on Uppaal,” in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, no. 3185. Springer–Verlag, 2004, pp. 200–236.