Translating SysML models into Symbolic Transition Systems for Model-Based Testing

Tobias Bachmann
contact@tobiasbachmann.ch
13028642

September 18, 2021, 81 pages

Academic supervisor: Ana Oprescu, a.m.oprescu@uva.nl
Daily supervisor: Machiel van der Bijl, Axini B.V., vdbijl@axini.com
Host organisation: ProRail B.V., www.prorail.nl
Contact host organisation: Daan van der Meij, daan.vandermeij@prorail.nl

Universiteit van Amsterdam
Faculty of Science
Master Software Engineering
www.uva.nl/en/programmes/masters/software-engineering/software-engineering.html

Abstract

A railway interlocking system consists of various subsystems. With a shift from monolithic systems containing proprietary interfaces towards modularized solutions, interoperability between different suppliers is achieved and the lifecycle costs reduced. The European initiative, the EULYNX Consortium combines 13 infrastructure managers concerning itself with the specification and standardization of interfaces that is necessary for the modularization and the separation across multiple suppliers.

The conformance of an implementation is proven based on the EULYNX requirement specifications which pose high demands on the quality of the specifications. For that reason, EULYNX applies the Systems Modeling Language (SysML) as semi-formal method to specify and verify system requirements. SysML allows creating high quality specifications that are also understandable for people without a strong mathematical background.

In the railway domain, safety is one of the most important aspects that is embedded in the entire system. This makes testing one of the most important and expensive stages in the development process. To assess the conformance of an implementation we utilize formal methods. In this project, we research the feasibility to translate the semi-formal SysML models into a formal model to facilitate effective Model-Based Testing (MBT). The goal of our research is to determine to what extend a SysML model can be translated into a Symbolic Transition System (STS) for Model-Based Testing and what the remaining challenges are when translating the EULYNX SysML specifications.

In our work, we first present a translation framework to translate SysML models into STS using the Axini Modeling Language (AML) as additional abstraction layer. Then, as a proof of concept, we translate parts of the EULYNX SysML specifications of the subsystem point into AML models. Next, we carry out a qualitative experiment to validate our translation. We evaluate our findings to improve our translation and we identify several non-conforming behaviours in our System Under Test (SUT). As a result of this we are able to provide a validated translation framework which we use to perform a quantitative experiment where we discover additional non-conformances in our SUT. Finally, we demonstrate that specification models rather than implementation models benefit the applicability of MBT using AML with the given EULYNX SysML specifications.

Contents

Chapter 1

Introduction

The responsibility of a railway interlocking system is to ensure nonconflicting train movements. Trains move along routes that are set and locked. A route consists of field elements, such as light signals and points. Signalling subsystems manage these field elements. The interlocking system commands the field elements into a certain state and evaluates the reported monitoring information of the physical trackside elements. The interlocking system operates as the safety layer between the traffic control system and the field elements.
The interfaces between interlocking systems and their signalling subsystems vary fundamentally among different railway infrastructure operators [1]. National regulations and legacy systems pose a challenge for interoperability [2, 3]. To address this issue, 13 railway infrastructure managers (IM) founded EULYNX[1] as a European standard for modular interface specifications. With a standardisation of these interfaces, EULYNX aims for a significant reduction of lifecycle costs of signalling systems. Costly proprietary implementations that are only valid within limited geographical dimensions are no longer necessary because signalling subsystems that are compliant with a certain version of the EULYNX standard can be used interchangeably. To ensure interoperability a verification of the implementation against the specification is essential.

EULYNX applies Model-Based System Engineering (MBSE) whereby *Use Cases*, provided by the participating infrastructure managers, are facilitated as the input for the interface specifications. The specifications are provided in documents with mainly textual information. Modelling experts and signalling system engineering experts then convert these specifications into semi-formal SysML models.

1.1 Problem statement

Testing is an essential part of the verification and certification process of software systems and consumes 30-50% of the total software development costs [4]. Nowadays, modern interlocking systems including their subsystems are complex software systems. These systems must be compliant with CEI EN50126 [5], CENELEC EN50128 [6], and BSI EN50129 [7] and therefore adhere to the highest Safety Integrity Level 4 (SIL4). With a resulting *Probability of dangerous failure per hour* of 10^{-8} to 10^{-9}, testing becomes even more important. Introducing Model-Based Testing (MBT) to this field promises to improve quality and effectiveness of testing while reducing the costs [4].

MBT aims to automatically generate numerous test cases from a formal model, thus complementing automatic test execution. It provides automatic, faster, and less error-prone test generation [8]. MBT is automated black-box functionality testing that is based on a formal model of the Implementation Under Test (IUT) [4]. The basic concept is, that if the IUT and its model are put into black boxes and we would perform all possible test cases, then it is not possible to distinguish between the IUT and its model in view of the observed behaviour [8]. The model is a formal specification of the IUT's behaviour. UML specifications (where SysML is based on) are usually translated into a formal notation [9] to facilitate MBT.

[1] https://eulynx.eu/

Among others, the infrastructure managers *ProRail B.V.* and *DB Netz AG* are members of the EULYNX Consortium and intend to implement this standard in the future. Together with the *University of Twente* (UT) and *Eindhoven University of Technology* (TUE), these two infrastructure managers have formed the **FormaSig project**. The goal of FormaSig is the verification of EULYNX interface specifications with the use of formal methods. With model-checking, a verification whether the original designs are compliant with the requirements is achieved.
Djurre van der Wal and Mark Bouwman of the FormaSig project have constructed a formal model for model-checking [10–12]. This model is described in the specification language mCRL2[2]. The mCRL2 toolset provides primarily instruments for validation and verification of concurrent systems. It is mainly used for model-checking. With JTorX [13], a tool for model-driven test derivation and execution that was developed at the *University of Twente*, the authors obtained test cases which they used against a simulation of the point subsystem. This simulation is provided by a third-party company and did not provide an API or command line interface. Recently, the authors extended this simulation to allow a programmatical integration using an adapter. Due to limitations in the test suite generation and the simulation, this set-up has an experimental character. Additionally, JTorX is mainly targeting on educational purposes and is not further developed. Improving JTorX to overcome these issues as well as implementing MBT with its successor *TorXakis*[3] are parts of the FormaSig project and are being researched by the authors in the future.
Shubhangi Salunkhe created an automated transformation of SysML models into Event-B[4] models [14]. Event-B is a formal method for system-level modelling and analysis. The models are verified with the Rodin platform [14].

1.1.1 Research questions

There is no formal model of the EULYNX specifications yet that enables efficient MBT. Previously created models focus on model-checking rather than MBT. In this project we closely collaborate with the company Axini[5]. The Axini Modeling Platform (AMP) provides a powerful solution for MBT that automates the entire testing process. With this platform and a formal model based upon Symbolic Transition Systems (STS), we facilitate MBT. We use the Axini Modeling Language (AML) as part of AMP to formalize our model. This model is then converted into an STS and used for MBT. We therefore formulate our **main research question**:

RQ) To what extent can a SysML model be translated into a Symbolic Transition System (STS) for Model-Based Testing (MBT)?

Based on findings of our main research and the encountered challenges, we formulate recommendations for further improvements of the application of MBT. We provide recommendations on both sides, the EULYNX specifications and the formal modelling. Additionally, we briefly compare the proposed formalism to mCRL2, as this is used in the mentioned related FormaSig project. We address our findings in a concise discussion where we highlight the fundamental characteristics and their applicability for MBT.

SQ 1) What are the remaining challenges when translating EULYNX SysML specifications into AML?
SQ 2) How does our approach compare to the approach with a formal model in mCRL2 and JTorX for MBT?

1.1.2 Research method

To research our main research question **RQ 1**, we carry out a case study. In detail, we define translation patterns from EULYNX SysML specifications into AML. We limit ourselves to the specifications of the point subsystem because only for this subsystem a simulation is available which we need for our validation. For the validation, we focus on the subsystem specific parts of the specifications. As a first step, we translate all specifications of each specified logical component into independent formal models. Subsequently, we perform a parallel composition to combine the models.

[2]https://www.mcrl2.org/web/user_manual/index.html
[3]https://github.com/TorXakis/TorXakis
[4]http://www.event-b.org/
[5]https://www.axini.com/

Our verification is twofold. First, we take the independent formal models and explore the derived STS manually while comparing against the specifications. Next, we use the composed model and use the simulation of the point subsystem as System Under Test (SUT). With an adapter, we then run generated test suites against this simulation.

For the first sub-question **SQ 1.1** we look at the identified challenges from our main research. We investigate the reasons behind them and how they can be addressed. In an interview with the authors of the SysML specifications, we gain insight into made design decisions. We provide recommendations towards the EULYNX Consortium as well as towards Axini. We present our findings in a discussion that takes the derived results into account.

For our second sub-question **SQ 1.2** we look deeper into the formal mCRL2 model. We provide a non-formal comparison of the two approaches and how they address the identified challenges. Additionally, we carry out several interviews with the researchers of the FormaSig project. We briefly compare their fundamental principles and results.

1.2 Contributions

Our research makes the following contributions:

1. We provide a translation pattern for SysML specifications into a formal model in AML. As part of the SysML specifications, this includes a translation of the Atego Structured Action Language (ASAL).
2. As a proof of concept, we translate the SysML specifications of the point subsystem into AML models.
3. We develop a plug-in adapter for AMP on Windows that connects the simulation as SUT to AMP running on a Linux environment.
4. We carry out a qualitative and a quantitative experiment with our point simulation and present our found non-conforming behaviour.
5. We provide a discussion about the differences to the approach with mCRL2.
6. We contribute recommendations to improve the application of MBT to the EULYNX SysML specifications.

1.3 Outline

In Chapter 2 we describe the background of this thesis. Chapter 3 contains our research with the proposed translation framework to transform semi-formal SysML models into Symbolic Transition Systems. In Chapter 4 we explain how we implement our experiments and discuss the system parts and our architecture. The results retrieved from our experiments are then presented in Chapter 5 and discussed in Chapter 6. Chapter 7 contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 8 together with future work.

Chapter 2

Background

In this chapter, we will lay out the background of this research. First, we provide background on Model-Based Testing (MBT). We explain how MBT works and how it relies on a formal model which in turn is based on mathematical principles. Second, we look into the Axini Modeling Platform (AMP) which is an MBT implementation built upon Symbolic Transition Systems (STS). Next, we briefly explain what EULYNX is and how their Model-Based Systems Engineering Specification Framework (MBSE SF) works. Last, we provide background on an alternative modelling language to be able to have a concise discussion about the differences to our approach.

2.1 Model-Based Testing (MBT)

Testing is the main technique for validating software systems. Software testing includes the execution of a program on a set of test cases and subsequently comparing the actual results with the expected results [15]. A *test case* is a finite structure of input and expected output. A *test suite* consists of a finite set of test cases [16]. Test cases are defined based on the specification of the software system as a basis. Model-Based Testing (MBT) aims for an automation of this testing process. It encompasses the processes and techniques for the automatic derivation of test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases [16]. The mentioned abstract models serve as source for all test cases and consequently for the entire test suite. Since the test cases are derived in an automated way, this poses requirements to the model. Thus, the model needs to implement a formalism that allows its algorithmic evaluation.
The underlying assumption in the model-based approach is the existence of a precise formal model of the system being developed [17]. The formal model describes the system that is being tested. This system is widely referred to as *System Under Test* (SUT) or *Implementation Under Test* (IUT). Internal implementation details of the SUT are hereby not in the scope of MBT, only the observable behaviour is. This approach is known as *black-box testing*. Therefore, the model is a further abstraction of the SUT, reduced to its externally observable interactions.
For MBT, Mark Timmer, et al. [18] define three major stages:

1. formally modelling the requirements
2. generating test cases from the model
3. running these test cases against an actual system under test and evaluating the results

The first stage requires to define a formalism for the model that implements the requirements. Jan Peleska and Wen-ling Huang state that all standards related to safety-critical systems verification emphasise such a requirements-based testing as the main focus [19]. In the second stage the actual test cases are derived in an automatic way whereas the last stage requires to establish a definition for conformance.

2.1.1 Test case generation

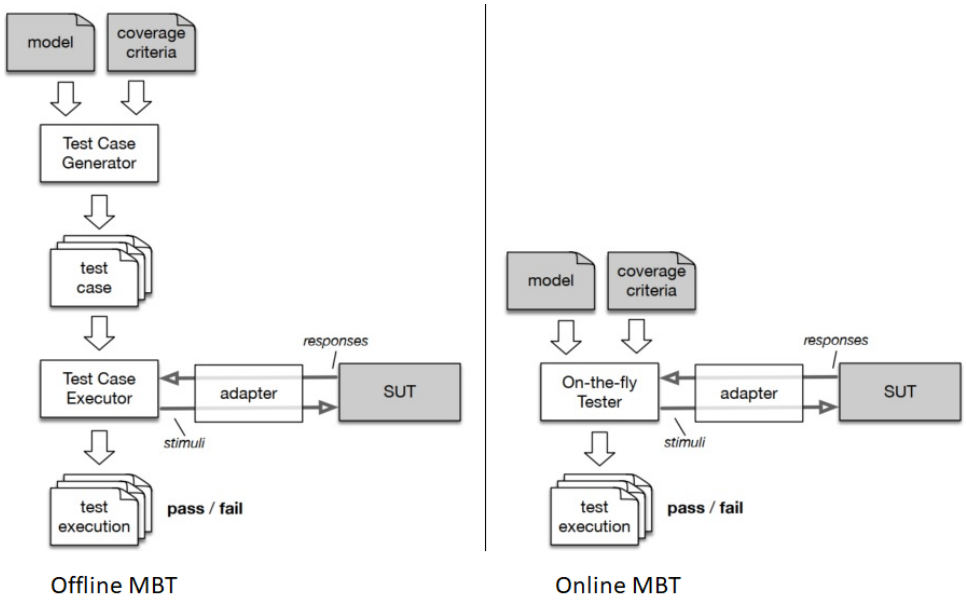
To achieve automated testing with MBT, test cases are derived algorithmically by a test generator from the formal model. This generator offers the possibility to influence the generation of the test cases. Typically, a selection of one or more coverage criteria is possible [20]. The test generator then automatically generates test cases that adhere to the selected coverage criteria. Test cases are a finite set of *labels*. Labels can either be a *stimulus* for the SUT or a *response* that is expected from the SUT.

This finite set of concrete labels forms a *trace*.
The test generator produces a test suite containing test cases based on the given model. The test executor then attempts to execute each test case on the SUT [21]:

- If the label in the trace is a stimulus, the adapter transforms the stimulus to a physical label and offers the physical label to the SUT.
- If the label in the trace is a response, the test executor waits until it receives the logical response from the adapter.

If the SUT returns a response which is not expected in the test case, it concludes that the test has **failed**. If the test executor can execute a complete trace of the test case against the SUT, it concludes that the test has **passed** [21].

Figure 2.1: Offline and online MBT [21]



This generation of test cases can be done in two ways, offline and online (see Figure 2.1). In the offline method, all test cases are generated before the actual testing takes place. In online testing, a so called *on-the-fly* tester computes the test cases continuously. Test cases are generated on-the-fly and partly while testing the SUT. In our project, test cases are based on an STS. Thus, the on-the-fly tester walks over the partly generated STS of all processes of the model, offering stimuli to the adapter and observing the responses of the SUT. The on-the-fly tester decides on the next test-step after each stimulus or response (as opposed to generating all test-steps before testing) [20].

2.1.2 Formal model

The formal model describes the interface between SUT and testing environment. It specifies the SUTs behaviour as far as visible on its interface [19]. Jan Peleska and Wen-ling Huang state that an essential feature of the model is its possibility for a top-down decomposition to express the overall SUT functionality by a set of concurrent sub-components using internal communication [19]. Another essential feature stated by Mark Timmer, et al. is to have unambiguous models in order to ensure an unambiguous test process [18].
To achieve these goals, the use of different formalisms is proposed. The supposedly most commonly used formalisms are finite state machines (FSM) [16, 22] and labelled transition systems (LTS) [4, 23]. FSM and LTS consist of a set of states in which the system can be and describe transitions between these states. The latter describes the intended behaviour of a system by specifying the allowed interactions between the system and its environment. The transitions are used in such a way that correct behaviour simply corresponds to paths through these models [18]. FSM and LTS are basic mathematical structures. Mark Timmer, et al. [18] additionally mention the specification languages VDM [24], Z [25], and PROMELA [26].

In this thesis, we will translate semi-formal SysML models into Symbolic Transition Systems (STS) for MBT. STS augment LTS by adding the concepts of location variables and data, which can be communicated over gates. Both influence the flow of control, thereby allowing us to specify data dependent behaviour [27]. STS (and therefore AML, see Section 2.2.1) are used to describe the behaviour of reactive systems. A reactive system is a (or part of a) message-oriented system that reacts with observable outputs to inputs received from the environment [21].
Jan Peleska and Wen-ling Huang use Kripke structures over LTS in their publication about MBT of Safety-Critical Systems [19]. They base their decision on the observation that many interfaces occurring in the embedded systems world follow the shared variable paradigm, so that the concepts of atomic events and synchronous communication are considered as optional higher-level abstractions [19].

2.1.3 Conformance testing

With the generation of complete test suites, it is possible to show conformance of the model and the actual implementation [17]. This leads us to the definition of the term *conformance* that relates implementations to specifications. A formal definition of conformance is required [4]. The notions of conformance and of test execution have to be linked in such a way that from test execution an indication about conformance can be obtained [4].
Tretmans defines the *input-output conformance*, called *ioco* [4]. *ioco* is based on the assumption that the implementation can be represented by input-output labels whereas the implementation is *ioco* conform to a model, iff [20]

- the implementation produces an output x after trace σ, then the model can produce the same output x after σ
- the implementation cannot produce any output after trace σ, then the model cannot produce any output after σ

A trace σ is defined as a set of labels after a state p of an LTS. We will use the definition of *ioco* as the formal implementation relation to define conformance between an SUT and its specification. Although *ioco* was originally defined for LTS, the *ioco* theory has been lifted to STS [27].

2.2 Axini Modeling Platform (AMP)

The Axini Modeling Platform (AMP) implements the mathematical concepts of STS for MBT. The proprietary Axini Modeling Language (AML) is used to describe formal models and leveraged to generate STS. AMP consists of the following components [28]:

- **Tester:** The Tester is the component that drives the test execution. It manages the simulator, adapters, and strategy.
- **Explorer:** The Explorer is a component and feature that allows the user to step though a model. To achieve that, it uses a simulator.
- **Simulator:** The Simulator is the component that holds the model's state when exploring or testing. This simulator state contains the state vectors and the clock.
- **Adapters:** An Adapter is a component that facilitates communication with the SUT. Adapters are responsible for maintaining the connection to the SUT, observing responses, and converting labels into messages and vice versa.
- **Strategies:** A Strategy is an algorithm that decides what action to perform while testing. This is essentially the same as a user does when using the Explorer. Different strategies are available in AMP.

In the following we will provide additional background information about AML, the Adapter, and the SUT, since they are essential parts of this work.

2.2.1 Axini Modeling Language (AML)

AML is the proprietary modelling language and used to model the behaviour of the SUT. As mentioned in Section 2.1, MBT is black-box testing. Therefore, the internal implementation details of the SUT are not in scope of the model, only the externally observable behaviour is. As elaborated before, the concrete test cases are derived from the underlying STS. STS (and LTS) tend to grow extremely large very quickly and are therefore tedious and error prone to construct and maintain by hand [21].

AML provides a higher abstraction level that addresses these issues. An AML model describes a reactive system and specifies the inputs that it is intended to accept and the expected outputs. With a formal translation from this high-level modelling language to a state-transition system, the STS for test execution is retrieved. The high-level modelling language has thus a formal semantics: for every construct of the language, it is formally defined to what state-transition model this is mapped [21]. The semantics of AML models is defined upon STS.
An essential concept of AML is the support of concurrent *processes*. Each process defined in an AML model is translated to a complete STS, embodying the behaviour of the process [21]. Multiple processes are considered to be parallel and can be advanced upon independently [28]. Executions of the parallel processes are interleaved with each other in a non-deterministic way, but they can be synchronized using internal synchronization.

2.2.2 Adapter

As depicted in Figure 2.1, the adapter is an essential part of the MBT approach and its development can be expensive, forming a considerable part of the entire MBT effort [21]. The adapter connects the MBT platform (the test case executor, to be more precise) with the SUT and comes in two variations: as traditional adapter and as plug-in adapter. We will only focus on the latter one as the first type of adapter is not relevant for this project.
The main responsibility of the adapter is the translation of logical labels of the model to physical labels of the SUT and vice versa [21]. A physical label is the stimulus or response that interacts with the interface of the SUT. The characteristics of the physical label depends on the characteristics of the SUT's interface.
A plug-in adapter decouples the MBT platform from the environment of the SUT. The adapter is detached from the AMP and connects to it using a communication protocol. Plug-in adapters should run as closely to the SUT as possible but can run in any environment that is able to connect to AMP and the SUT [28]. Therefore, plug-in adapters can be implemented in virtually any programming language that is able to implement the *plug-in adapter protocol*. The basic concept of this protocol encompasses the connection of the plug-in adapter to AMP using *websockets* as communication protocol. Over the websocket, both AMP and the plug-in adapter communicate using the plug-in adapter protocol. The messages of this protocol are defined using Google's Protocol Buffers[1] *protobuf*. The official protobuf tools can generate code for many programming languages from schema files [28]. To use protobuf, the use of a supported programming language is required.
For a detailed description of the plug-in adapter protocol, we refer the reader to *Effective AML* [28].

2.2.3 System Under Test (SUT)

As the name suggests, the SUT is the system that is being tested using MBT. Sometimes it is also referred to as Implementation Under Test (IUT) which indicates that the system is actually an implementation of the specification against which the validation is made. It is considered to be a black-box. The SUT can theoretically be any system, from a low-level embedded system to an abstract service. To make it eligible for MBT it needs to be a reactive system that comprises of an interface where it can be connected to the adapter. The interface can be anything that is programmatically accessible such as I/O interfaces, CLI, API calls, and even GUIs.

2.3 EULYNX

In railway signalling, an interlocking is a system composed by a set of signal apparatus that prevents trains from conflicting movements through only allowing trains to receive authority to proceed, when routes have been set, locked, and detected in safe combinations [29]. An interlocking system (ILS) comprises an electronic interlocking (EIL), a command control system (CCS) and field elements such as points, signals, and so forth [30]. In recent years a shift from monolithic systems with proprietary interfaces to modularized solutions was observable among railway infrastructure managers (IM). These modular systems aim for a solution where the various parts are supplied separately.
The modularization requires the specification of interfaces and the separation across multiple suppliers requires the standardisation of them. EULYNX as a European initiative by 13 IMs develops standardised interfaces between the elements of signalling systems. These standards enable different suppliers to offer

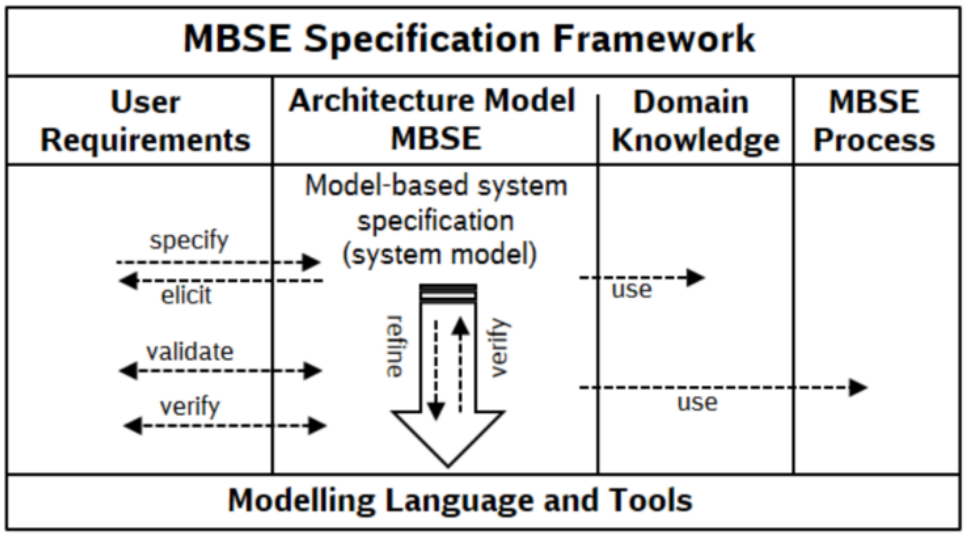
[1] https://developers.google.com/protocol-buffers

compatible products. The compatibility is proven based on the requirement specifications. This poses high demands on the quality of the specifications as suppliers will work with them when developing their components and IMs will use them for system integration tests.
It is an important issue among IMs, the railway industry, and researchers to find appropriate forms to specify the architectures of complex component systems right up to huge systems of systems (SoS) [30]. Used methodologies in the past have raised a number of criticisms. The specifications were made using different forms like natural language and graphical representations of system requirements [30] but have lacked a formalism. The *EULYNX Modelling Standard* highlights problems with capturing requirements and refers to studies that show how this step is one of the most decisive and critical in system development [30]. To address aforementioned issues, EULYNX applies semi-formal methods to specify and verify system requirements. As semi-formal method, EULYNX chose to use the Systems Modeling Language (SysML), because it allows creating high quality specifications that are also understandable for people without a strong mathematical background [30].
In the railway domain, safety is one of the most important aspects that is embedded in the entire system. This creates a need for safety to be part of the standardised interfaces and requires highly detailed system specifications [30]. Prove that the system specifications meet safety critical requirements is achieved by modelling and simulating in a Model-Based Systems Engineering (MBSE) approach.
Based on these observations, EULYNX developed an engineering-oriented model-based methodology for the stepwise specification of digital Command Control and Signalling (CCS) systems using the Systems Modeling Language (SysML) [30]. This supports different professionals such as railway engineers. Using a semi-formal specification, it facilitates the validation and verification of the corresponding system requirements.

2.3.1 Model-Based Systems Engineering Specification Framework (MBSE SF)

The identified key success for addressing the aforementioned challenges are suitable concepts for abstraction and structuring at different levels of granularity [30]. Based on the *SPES (Software Platform Embedded Systems) Modeling Framework* proposed in Model-Based Engineering of Embedded Systems [31], EULYNX introduces the *MBSE Specification Framework* (MBSE SF) as depicted in Figure 2.2. This framework satisfies the basic requirements identified in the *SPES Modeling Framework* of a clear notion that is formalized by a comprehensive modelling theory. The modelling framework provides appropriate models and description techniques for modelling the different aspects and artefacts of system development [30].

Figure 2.2: Model-Based Systems Engineering Specification Framework [30]



The MBSE SF consists of five areas [30]:

- **User Requirements:** The user requirements specify the system model based on design decisions. Special about these design decisions is that they are derived from the problem domain (problem definition). Different stakeholders define what is expected from the system under development (SUD). User requirements are the main source of design decisions for the system model. The design decisions lead to a refinement of the requirements which facilitates eliciting new user requirements.
  The user requirements are subsequently used to validate the system model and ensure that the stakeholders' intentions are reflected completely and correctly. The goal is to retrieve a formal verification (proof) of the fulfillment of the user requirements from the system model (model-checking).
- **Architecture Model MBSE:** This model in the solution domain reflects the system as model-based system specification. It is created using the MBSE process and domain knowledge. The system requirements are based on user requirements. The system model is validated and verified with these user requirements.
  The system model itself is successively refined where each refinement step is verified against the previous one. Each abstraction level represents design decisions about the refined or decomposed implementation of its predecessor. After the last refinement step, the goal is to have a system model that can be used as input for a specific modelling language while maintaining the verification with user requirements.
- **Domain Knowledge:** Domain knowledge is used by the system model as basis for specification, verification, and validation.
- **MBSE Process:** The MBSE process is described in the MBSE SF that specifies the relation between various artifacts of the system model. It defines the processes for model refinement, verification, and analysis. Process activities are carried out subsequently whereas the result of one activity serves as the input of the next one. The process is ensured using pre- and post-conditions of each activity.
- **Modelling Language and Tools**: The result of the MBSE process is a system model that can be used as input for a modelling language.

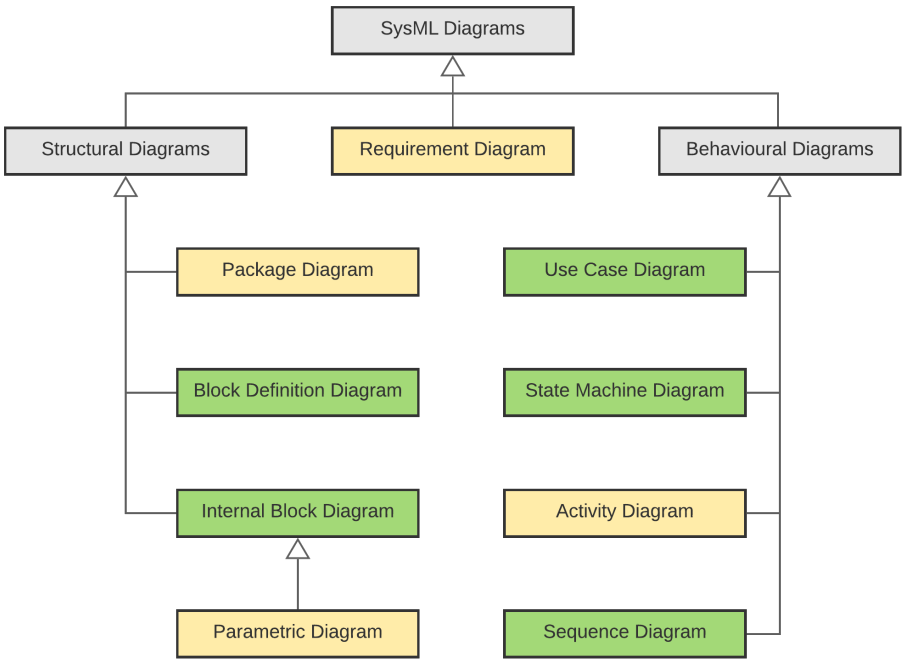
2.3.2 Systems Modeling Language (SysML) and Atego Structured Action Language (ASAL)

SysML reuses a subset of UML 2 and provides additional extensions to satisfy the System Engineering's needs. SysML is designed to provide simple but powerful constructs for modelling a wide range of systems engineering problems [32].
Nine SysML diagram types as depicted in Figure 2.3 define a concrete syntax that describes how SysML concepts are visualized graphically or textually. Each diagram represents a specific view of the model of the system under development (SUD) [30]. Of the defined nine diagram types, EULYNX uses the following five:

- **State machines (STDs):** Specifies the externally observable behaviour of a specific component.
- **Block definition diagrams (BDDs):** Specifies the existence of components and their relation.
- **Internal block diagrams (IBDs):** Specifies the exact connections between components.
- **Use case diagrams (UCDs)**
- **Sequence diagrams (SDs)**

The UCDs and SDs specify how systems must behave in specific scenarios but do not offer a complete picture of the specified behaviour. Thus, they are not relevant for our translation [10].
To specify the necessary executable behaviour, SysML is enriched with the Atego Structured Action Language (ASAL). ASAL is a simple programming language and used in the specifications to define computational behaviour. ASAL supports simple data types, function, statements, and expressions. It is suitable to specify executable algorithms in a target language independent way [30]. ASAL is used in various parts of a SysML diagram to describe behaviour, such as states (entry- and exit-behaviour) and transitions (guards and effects).
An extensive description of the features of ASAL can be found in *Appendix C* of the EULYNX Modelling Standard [30].

Figure 2.3: SysML diagram types (green = used by EULYNX)



2.4 mCRL2

To compare our approach and results to the ones from the related FormaSig project, we need to introduce the fundamentals of mCRL2[2]. mCRL2 (micro Common Representation Language 2) provides a toolset to support the analysis of complex distributed systems. The toolset consists over 60 tools that allows it to analyse complex system designs which are formally described [33]. The mCRL2 language itself consists of three different sublanguages [33]:

- Data language
- Process language
- Property language

The **data language** supports built-in data types but additionally allows complex types to be constructed [33]. Users can also define operations on them [34].
The behaviour of a concurrent system is described using the **process language**. mCRL2 is based on the Algebra of Communicating Processes (ACP) where a fundamental concept in mCRL2 is the process [35]. Processes are composed from a set of defined actions and operators [33]. Processes can be sequential whereby multiple sequential processes can be composed sequentially [34]. Processes can also be parallelly composed. mCRL2 supports additional process algebraic operators such as multi-action composition [33] to describe a set of actions that can happen simultaneously. Data can be used to influence the flow of control permitting the modelling of systems whose behaviour crucially depends on data [33].
Using the **property language**, the specified behaviour can be analysed in order to determine whether it satisfies certain requirements. These requirements are specified in the first-order modal μ-calculus [34]. The problem of deciding whether a given property expressed in the μ-calculus holds for a given process specification is automatically encoded in a Parameterised Boolean Equation System (PBES) [33].
The mCRL2 specifications are automatically transformed into the Linear Process Specification (LPS) which is another mCRL2 specification with a restricted grammar [33] where all parallelism has been removed. These linear processes can be simulated, and their state space can be explicitly generated and stored. For a more detailed explanation we refer the reader to *Linearization in parallel pCRL* [36].

[2]https://www.mcrl2.org/

Chapter 3

Translation of SysML models into Symbolic Transition Systems

To answer the research question to what extent a SysML model can be translated into a Symbolic Transition System (STS), we define a translation framework from the SysML modelling inventory onto an STS. For this translation we take parts of the EULYNX interface specifications, provided as SysML models, and define a mapping onto the language features of AML. This mapping describes an exemplary conversion and serves as a basis for discussions about the identified challenges. It should help to reproduce the translation in future projects. Additionally, we intend that the provided translation framework can be adopted in the future to achieve an automatic translation.
As basis for our research, we have used the Modelling Standard (Eu.Doc.30) [30], the generic interface and subsystem requirements specification (Eu.Doc.20) [37], and the requirements specification for the point subsystem (Eu.Doc.36) [38]. We used all documents of the baseline 3.0 (0.A).
First, we provide an overview over the different types of logical components and interfaces. Next, we research the translation of the used action language ASAL into AML. Then, we provide mapping patterns for the extensive inventory of SysML, including ports, states, and transitions. Finally, we look at the parallel composition of congruent processes which is needed for the decomposed SysML implementation models.

3.1 Logical Components

The subsystem is further decomposed into logical components (LCps) of two types, *TypeA* and *TypeB*.

3.1.1 Logical Component Type A (LCp-TypeA)

The structure of a System Under Development (SUD) is arranged in architectural entities. The LCp-TypeA represents these entities aiming at a logical system architecture that is agnostic of technological constraints. LCps of TypeA are again further decomposed into sub-components. The stimulus-response behaviour of the LCp-TypeA is represented by the interaction of these contained sub-components and additionally by the interaction with the interfaces of the SUD. A special type of LCps are *atomic* LCp-TypeA. Atomic LCp-TypeA are logical components which are not required to be further decomposed. They represent therefore the most refined type and encapsulate a part of the entire externally visible stimulus-response behaviour of the SUD [30]. This behaviour may be modularised by LCps-TypeB.

3.1.2 Logical Component Type B (LCp-TypeB)

LCps of TypeB modularise the stimulus-response behaviour of an atomic LCp-TypeA (which is an architectural entity of the system) and represent behavioural entities [30]. An LCp-TypeB is not further decomposable and can describe two interfaces:

- **Syntactic interface:** This interface is described using the SysML diagram type IBD. It defines primarily the *signatures* of the in- and out-ports. If relevant, the signatures of block properties and block operations (functions) are also included.

- **Semantic interface:** This interface is described using the SysML diagram type STD. It specifies the stimulus-response behaviour.

In our mapping, we mostly use LCp-TypeB since they are used to describe the essential states as defined at abstraction level *AL1 System Definition* as well as the stimulus-response behaviour defined at abstraction level *AL2 System Requirements* of the MBSE SF.

3.2 Syntactic interface

The syntactic interface is described in the partial view *Events* and defines the signatures of the in- and out-ports. These ports define the interface of the LCp through which the LCp interacts with its environment using stimuli and responses. The stimuli and responses are represented by atomic SysML flow ports owned by a SysML block and depicted in a SysML diagram of type IBD [30]. Besides other properties, the ports can be of three different types which deserve special attention for the mapping: Trigger port (T), Data Trigger port (DT), and Data port (D). We will discuss them further in Section 3.4.2.
Additionally to the port signatures, *block properties* and *block operations* are also defined in the syntactic interface. Block properties are variables to cache values or define (configuration-) data, while the latter are functions that operate on the cached values. Both are defined using unique signatures whereby the scope is limited to the related LCp.

3.3 Semantic interface

The behaviour of an LCp-TypeB is described in the partial view *Behaviour*. The behaviour of an LCp-TypeB specifying the functional system requirements of a reactive system is not only a direct consequence of its stimuli, but it also depends on its preceding state, i.e., its history [30]. This history defines a *state* of the LCp-TypeB which in turn results in the current behaviour that is either providing a response or consuming a stimulus. The stimulus-response behaviour of an LCp-TypeB is described using a SysML diagram of type STD. To model the behaviour of each component, we need to define a mapping from the STD into an STS using AML.

3.4 Mapping patterns

3.4.1 Atego Structured Action Language (ASAL)

The Atego Structured Action Language (ASAL) is a proprietary UML Action Language from PTC[1] that is suitable for specifying executable algorithms in a target language independent way [39]. It is used in various SysML models, especially in STDs, to describe low-level computational behaviour.
In order to translate the described computational behaviour, we map ASAL to AML. Additionally to *constraints* and *expressions*, AML supports *functions* and *macros* where valid Ruby code is supported. We therefore focus on the translation of ASAL into Ruby. We have the hypothesis that it is possible to translate behaviour defined in ASAL into our models using either native AML expressions or Ruby.
In the following we analyse the inventory of ASAL and define a mapping of the different language functionalities.

Data types

The used data types in the EULYNX specifications are limited to the provided data types of ASAL. They are referred to as *SySim value types*. Other data types are not supported. In fact, the actually used data types are only a small subset of the theoretically supported ones and we can limit our mapping to those. We provide a mapping between SySim value types and AML data types in Table 3.1.
Data types serving as a container (like Lists, Sets, ...) can be simulated using Strings and Operations[2] [30]. AML provides generic list functionalities. These are, however, currently not used in the specifications.
We conclude that we are able to map all SySim value types onto AML data types without expensive conversion.

[1] https://www.ptc.com/
[2] In SysML, operations have the semantics of functions

SySim value type	*AML data type*	*Comment*
Boolean	Boolean	Boolean values are directly translatable without further conversion.
String	String	The used Strings in the EULYNX specifications are used as Enumerable. String values are only assigned and evaluated, no String transformations or String operations are performed. The AML data type *String* provides more functionality which is not needed in our case. Thus, we use *String* in AML without further conversions.
Single, Decimal, Double	Decimal	These data types are currently not used in the EULYNX specifications. If needed in the future, the AML datatype *Decimal* can be used, which provides arbitrary-precision floating point decimal arithmetic.
Integer, Long	Integer	*Long* is currently not used in the EULYNX specifications. We therefore limit our mapping to *Integer*. The SySim value type *Integer* is signed and represented using 4-bytes. Therefore, we to not need a conversion.
PulseIn	Generic stimuli	These two redefined Boolean types are special types used to represent Trigger ports. We discuss them separately in Section *Trigger in/out port*
PulseOut	Generic response	

Table 3.1: ASAL mapping pattern: Data types

Logical operators

In the EULYNX specifications, only the equality operators (=, <>) and logical operators (NOT, AND, OR) are used [10]. However, we do not restrict our mapping to those operators because this might be subject to change in future versions. ASAL supports logical operators and Boolean operators according to Table 3.2. We map them to the corresponding AML operators as shown in the same table. AML supports the use of logical operators in constraints and update statements. As mentioned above, Ruby operators are also supported in functions and macros.
We can map all ASAL operators directly to an AML operator, except for XOR. AML and the underlying Ruby does not have an exclusive disjunction operator[3], but we can transform XOR ($\oplus$) into a supported Boolean statement using Boolean algebra:

$$p \oplus q \equiv (p \vee q) \wedge \neg(p \wedge q)$$

[3]Note that the operator ^ is a binary and not Boolean XOR operator.

	Operator	*ASAL operator*	*AML operator*
Equations	Greater than	>	>
	Greater than or equal	>=	>=
	Less than	<	<
	Less than or equal	<=	<=
	Equal	=	==
	Not equal	<>	!=
Boolean operators	Conjunction	*AND*	&&
	Disjunction	*OR*	||
	Negation	*NOT*	!
	Exclusive disjunction	*XOR*	

Table 3.2: ASAL mapping pattern: Logical operators

Basic statements

AML provides functionality to specify executable algorithms, such as state variables and loops. Additionally, the use of functions and macros is supported in which valid Ruby code is accepted. We have the hypothesis that we can map all ASAL statements to either native AML functionalities or at least to Ruby functions.
To validate our hypothesis, we analyse all basic ASAL statements as introduced in the Modelling Standard [30] and provide a mapping to AML. The following list of ASAL language constructs provides the results of our analysis. For the ASAL syntax definitions, we refer the reader to the Modelling Standard [30].

Reading the value of a port

```
<A> := <port>
```

A value change of a port is equivalent with receiving a stimulus in an STS. If the port is of type *Trigger in*, it can only be of the values *true* or *false*. Only the change to *true* is evaluated which is identical to receiving a stimulus without parameters. If the port is of type *Data Trigger in* or *Data in*, reading its value corresponds to receiving a stimulus with a parameter whereby the value of the port is contained in the parameter of the stimulus. We store the value of the received parameter in a local struct `DT` of type `dataPorts__LCp` using AMLs *update* construct (see Section *Data in/out port*). Only ports with the direction *in* are readable. With the use of stimuli, it is impossible to read a value of a port with the direction *out* since they are defined as responses.

Assigning the value of a port

```
<port> := <value>;
```

Analogous to reading the value, assigning the value of a port is equivalent to sending a response in an STS. It depends on the type of the port as well. If the port is of type *Trigger out*, assigning its value to *true* is simply sending the corresponding response without any parameters. A *Trigger out* port can be thought of as an impulse that is sent. Thus, we do not need to reset it to *false* explicitly. If the data type of the port is *Data Trigger out* or *Data out*, assigning the value corresponds to sending a response with a parameter whereby the value of the port is assigned to the parameter of the response. We send the response and set the parameter value using AMLs *constraint* construct. Only ports with the direction *out* are writable. With the use of responses, it is impossible to set a value to a port with the direction *in* since they are defined as stimuli.

Calling an operation

```
<operation> ([<parameters>]);
```

Operation calls in ASAL are comparable to function calls. ASAL operations accept a list of parameters and return a value. AML supports functions that work in a similar way. They also accept multiple parameters and may return a value. Additionally, AML macros provide a similar functionality. However, there are some differences between AML functions and macros. While macros are evaluated on parse-time, functions are evaluated only at run-time, when the expression they appear in is evaluated [28]. Functions operate on AML values, constants, and variables and can consequently only be used within AML update or constraint constructs [28]. Macros on the other hand are supported in any part of the AML model. They also accept parameters but do not return a value. If we need a return value, we achieve this with an intermediate local state variable. In both, functions and macros, valid Ruby code is supported. We argue that we can translate any ASAL operation to an AML function or macro.
In the considered specifications of this research, the used operations were limited to operations that either returned a Boolean value or void. The first type is used as conditions in guards, the latter one to encapsulate certain behaviour specifications.

Declaring variables

```
declare <variable list> : <type> ;
```

We use AML state variables. ASAL variables have a corresponding SySim value type which we map to an AML data type as explained in Section *Data types*. Additionally, we may assign an initial value according to the specification. In AML, the scope of state variables is limited to the place they are defined.

Assigning values to variables

```
<variable> := <expression> ;
```

The value of local state variables can be updated within their definition scope. We use AMLs *update* construct for this. We can also assign new values within functions and macros using Ruby.

Conditional execution of code

```
if <condition> then
    ... //code to execute
elseif <condition> then
    ... //code to execute
else
    ... //code to execute
end if
```

We have different options to represent this statement in AML. As mentioned before, in functions and macros we can use Ruby code directly. Therefore, we use the Ruby if-else conditions. Outside of a function, we use the *constraints* construct of AML for conditional execution. This construct does not provide an else clause. We model the else case with an additional *constraint* construct that has the conjunct conditions of all previous if-statements negated.
The conditions are Boolean expressions. As a general best practice, we outsource the conditions to functions that return a Boolean value. This gives us the possibility to simply use the negated function as the constraint for the else clause.
AML also provides an *_if <expression>, - _then - _else* construct for modelling deterministic choices. The *_else* clause is optional which allows for modelling the conditional execution in a more concise way.

While loops

```
while <condition>
    ... //code to execute
end while
```

Within functions and macros, we directly use Ruby while loops. They provide exactly the same semantics as ASAL while loops. In native AML, we construct while loops using the *repeat* statement in combination with the *constraint* construct and local state variables.
However, in the considered specifications of this research, while-loops are not used. Comparing with a newer version (Baseline: 3.1 (2.A)) of the *Generic interface and subsystem requirements*[4], there are indications that it might be used in the LCp *F_SMI_EfeS_SR*.

Case selection

```
select case <condition>
    case <condition>:
        ... //code to execute
    case else:
        ... //code to execute
end select
```

While we can directly use Ruby's *Case* expression in functions and macros, there is no such expression in native AML. However, this is not an issue since case statements can easily be transformed into if-else statements. We therefore have the same approach to model these statements as described under *Conditional execution of code*.

Return statement

```
return <expression> ;
```

As described above, functions in AML support a return value as well. Macros do not support them but assigning the value to a state variable and evaluate it afterwards gives us the same functionality. We assume that return statements are only used inside operation bodies.

Comments

```
// <text>
```

Comments are supported throughout AML.

Conclusion

We have listed an exhaustive inventory of the ASAL language and provided a translation either into native AML or Ruby. We have demonstrated that it is almost entirely possible to translate everything directly into AML. Whenever AML does not provide a certain functionality, we can always fall back to use functions or macros where we have access to the full potential of Ruby. Furthermore, we have shown that in some cases we have different options for the translation. Wherever it makes sense in terms of reusability or readability we will opt for the translation to a function or macro.
We conclude that we are able to fully translate ASAL from SysML models into AML. If the ASAL inventory will be extended in future EULYNX versions, we expect to be able to translate these additions too.

3.4.2 Ports

Each LCp has multiple ports specified. These ports are defined in the SysML *Internal Block Diagram* (IBD) of the syntactic interface. SysML distinguishes between three different types of ports, each of them either in the direction *in* (from the environment into the LCp) or *out* (from the LCp to the environment):

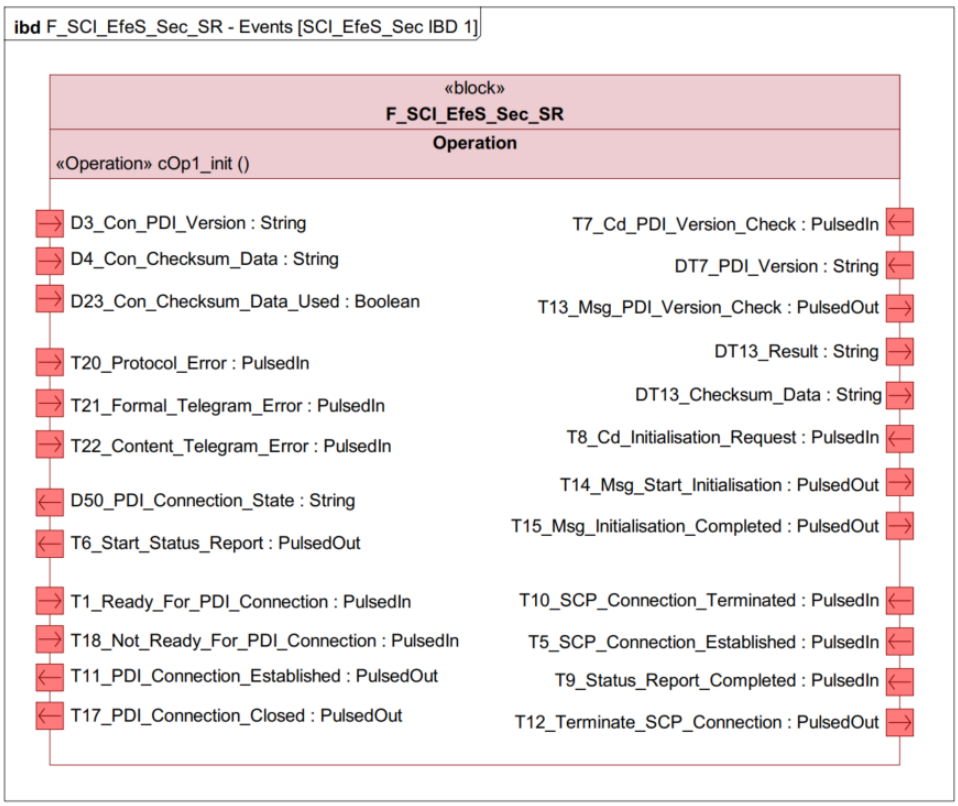
- Trigger in/out port
- Data Trigger in/out port
- Data in/out port

In Figure 3.1 is an example of an IBD that depicts the different port types. The arrows of *in*-ports point from the environment into the LCp, the ones of *out*-ports in the other direction. Trigger ports start with the letter *T*. For example, *T7_Cd_PDI_Version_Check* is a Trigger in port, *T13_Msg_PDI_Version_Check*

[4] Eu.Doc.20

a Trigger out port. Data Trigger ports start with the letters *DT* followed by the number of a corresponding Trigger port. For example, *DT7_PDI_Version* is a Data Trigger in port that corresponds to the aforementioned Trigger in port; *DT13_Result* and *DT13_Checksum_Data* are Data Trigger out ports that correspond to the aforementioned Trigger out port. Data ports start with the letter *D*. In our example, *D3_Con_PDI_Version* is a Data in port, *D50_PDI_Connection_State* is a Data out port.

Figure 3.1: IBD of LCp *F_SCI_EfeS_Sec_SR* from EULYNX Eu.Doc.20 [37]



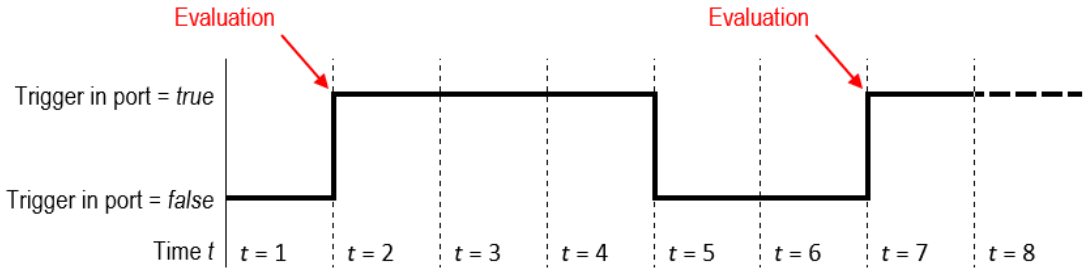
In the following, we further research the semantics of each port type and provide a translation from SysML into AML. Additionally to the port types, we also look at block properties.

Trigger in/out port

Trigger in/out ports are identified by the prefix of the letter *T* followed by a consecutive number. Appended is a descriptive name about the semantics of this port. The datatype of all Trigger in ports is *PulsedIn*; for Trigger out ports *PulsedOut*.
The *PulsedIn* and *PulsedOut* data types are redefined Boolean types and can therefore take the values *true* or *false*, whereby *false* the default value is. Whenever the value changes from *false* to *true*, it will automatically recede to *false* after a defined time.

Trigger in port The value of a Trigger in port is only valid during the evaluation phase when it changes from *false* to *true* [30]. In electrotechnical terminology, this means that only the raising edge is evaluated while the falling edge of the Trigger port is ignored. As long as the port value remains *true*, no further evaluation is allowed. The evaluation of Trigger in ports is depicted in Figure 3.2.

Figure 3.2: Evaluation of Trigger in port



Conclusion The semantics of Trigger in ports corresponds in context of the LCps stimulus-response behaviour to external or internal (between two LCps) stimuli without parameters [40].

Definition Following the EULYNX interpretation rules for model-based requirements [40], we define each Trigger in port as stimulus on the corresponding channel. We preserve the names of the ports unchanged and assign no parameters.
If we define a stimulus on an internal channel, we are required to have a corresponding response defined on the same channel in the model part that represents the other LCp. In the EULYNX specifications, the names of connected ports might vary between the LCps. To achieve readability and preserve the link back to the specifications, we concatenate both names whereby the name of the response (Trigger out port) comes first followed by two underscores and the name of the stimulus (Trigger in port).

Initialisation Trigger in ports do not need to be initialized. They are also not initialized in the EULYNX specifications. On start up, they are always regarded as *false* and only change to *true* if received during execution time.

Updating According to the EULYNX specification, whenever a Trigger in port can lead to a transition activation in a certain state, we implement the receiving of this stimulus with the corresponding behaviour. If a Trigger in port is not specified as an event at a certain state, we do not model it as well. There is, however, an exception to this. To avoid blocking processes in combination with internal synchronization, we model the receiving of all internal stimuli in every state (see Section 3.5).
Since a region is not allowed to stay in a pseudostate, the transition from one state to another is regarded as atomically (run to completion semantics, see Section 3.4.4). This makes the receiving of Trigger in stimuli in pseudostates not valid. Thus, we only accept Trigger in stimuli in a stable state (after a transition has completed). This also applies to entry- and exit-behaviour: When entering a state, we execute the entry-behaviour **before** accepting any new Trigger in stimuli. Similarly, when leaving a state, we execute the exit-behaviour followed by the entire behaviour of the transition before accepting any new Trigger in stimuli.

Trigger out port Trigger out ports are used to send event-driven responses without additional parameters [30]. To send such a response, the corresponding Trigger out port is set from *false* to *true*. Following the definition of the *PulsedOut* data type, this denotes an impulse being sent out. This impulse is reset to *false* automatically after a defined time. In electrotechnical terminology, this describes an electrical impulse being sent over a physical interface where the impulse de-energizes after some time (see also Figure 3.2).

Conclusion	The semantics of Trigger out ports corresponds in context of the LCps stimulus-response behaviour to external or internal (between two LCps) responses without parameters [40].
Definition	Following the EULYNX interpretation rules for model-based requirements [40], we define each Trigger out port as response on the corresponding channel. We preserve the names of the ports unchanged and assign no parameters. If we define a response on an internal channel, we must concatenate the name of the response with the corresponding stimulus (see Section *Trigger in port* for details).
Initialisation	In the EULYNX specifications, all Trigger out ports are initialized to *false*. This is done in the initialization operation of each LCp. We can omit this from the initialization operation because we do not store the values of Trigger out ports in our model and only use responses.
Updating	When a Trigger is sent as part of an effect, the value *true* is assigned to the corresponding Trigger out port. Because we translate Trigger out ports directly to responses without parameters, we can simply send the corresponding response. Note that multiple responses may be sent in a transition or behaviour.

Data Trigger in/out port and block property

Data Trigger in/out ports describe parameters that are allocated to a Trigger in/out port and may be part of the interface of an LCp with its environment. These ports carry complementary data for the corresponding Trigger port. Data Trigger in/out ports are identified by the prefix of the letters *DT* followed by the number of the corresponding Trigger in/out port. Appended is a descriptive name about the semantics of the information this port transmits (may be the same name as the corresponding Trigger port). Each Data Trigger in/out port has a datatype specified. The data type can be of any supported SySim value type (except *PulsedIn* and *PulsedOut*) but in the considered specifications of this research only the following data types are used:

- Boolean
- String

Data Trigger in/out ports have the semantics of a parameter assigned to a Trigger in/out port. In the EULYNX specification, however, they are specified as independent ports of the LCp. The association to the Trigger in/out port is exclusively made by the number contained in the ports name. Data Trigger in/out ports are allowed to change the value at any time, solely the *evaluation* is only allowed during a transition that was activated by the corresponding Trigger in/out port. Following the run to completion semantics (see Section 3.4.4) we assume that Data Trigger in/out ports must change the value *before* the change event from an associated Trigger in/out port is activated.
To validate this assumption, we look at the various EULYNX specifications. We see that values of Data Trigger out ports are always assigned *before* the corresponding Trigger out port is set to *true*. Thus, we conclude that on the recipient side the Data Trigger in port carries valid data whenever we receive the Trigger in event. This does not mean that the data of these ports must change before the corresponding Trigger in port creates another change event. The data might remain unchanged. If the value of the Data Trigger in port does not change, we simply operate with the last received value which is still valid in this case.

Block property The validity of the data of a Data Trigger in ports is restricted to the evaluation time of the corresponding Trigger in port. Only when a change event by the associated Trigger in port occurs, the data is regarded as valid. After the triggered transition is completed, the port loses its validity and is not to be evaluated. With block properties, values can also be evaluated at various points in time (defined by the behaviour of the respective Trigger in port) [30].
If data was made available from a Data Trigger in port that is supposed to be used at a later point in time, outside of the evaluation phase of the change event, and outside the transition caused by the corresponding Trigger in port, they must be stored temporarily in a corresponding *block property* [30]. This makes a data structure for block properties necessary. Thus, we define the following struct of type `blockProperty__LCp`, where `LCp` is the name of the corresponding LCp the block property is located in.

For example, the block property of the LCp *F_SCI_P_SR*:

```
def blockProperty__F_SCI_P_SR
  {
    'Mem_Move_Point' => :string,
    'Mem_Point_Position' => :string
  }
end
```

This struct is instantiated with the name `BP` in each process of the LCp separately and updated between different processes, as described in Sections 3.4.3 and 3.5:

```
var 'BP', blockProperty__F_SCI_P_SR, :void
```

Whenever a new value is assigned to a block property in the EULYNX specifications, we update the corresponding value in this struct.

Data Trigger in port As discussed in Section *Block property*, the values of Data Trigger in ports are only valid when the corresponding Trigger in port creates a change event. Thus, a Data Trigger in port is not to be used as argument of a Boolean expression of a change event [30]. However, during the transition, values of Data Trigger in ports can be used in guard conditions.
An activated transition can pass one ore more pseudostates. The initial activation of the transition to a pseudostate might be made by a change event of a Trigger in port. At this (or a later) pseudostate which is part of the same transition, a Boolean expression in a guard condition that evaluates an assigned Data Trigger in port may be present. This results in a temporal decoupling from the arise of the Trigger in change event and the evaluation of an assigned Data Trigger in port. Thus, it requires us to store the value of Data Trigger in ports, even if they are not part of a block property. To store these values, we use the struct `DT` of type `dataPorts__LCp` in the same way as for Data in/out ports (see Section *Data in/out port*).

Conclusion Data Trigger in ports could be modelled as parameters of Trigger in ports but following the previous conclusion in Section *Data Trigger in/out port and block property*, these ports obtain more the semantics of stimuli that occur punctual. We decided against an aggregation of Data Trigger in ports with their corresponding Trigger in port to keep the model as close to the EULYNX specifications as possible.

Definition We define each Data Trigger in port as stimulus on the corresponding channel with exactly one parameter assigned. This parameter represents the value of the Data Trigger in port and has the AML data type corresponding to the SySim value type of the port (see Section *Data types*). We preserve the names of the ports unchanged. The name of the parameter is equal to the name of the Data Trigger in port (and therefore of the stimulus itself) with an underscore prepended.
If we define a stimulus on an internal channel, we must concatenate the name of the stimulus with the corresponding response (see Section *Trigger in port* for details).

Initialisation As described above, we always store the last received value of the Data Trigger in port. Whenever we evaluate a Boolean expression containing the value of this port, we use the stored one. Data Trigger in ports are not initialized in the EULYNX specifications. We do not initialize them as well. Thus, after start-up, we might not have received a value on a Data Trigger in port yet and any evaluation would fail. Since an evaluation is only allowed when the associated Trigger in port triggers a change event, we can assume that whenever this happens, we have received the Data Trigger in port containing the value beforehand.

Updating The value of the Data Trigger in ports can change anytime during execution. Therefore, it is needed that the model accepts a stimulus to update the value in any state. As a result, we add a *receive* statement for every Data Trigger in port to every state that the model can be in. To follow the DRY[5] concept, we outsource this to an AML macro. When the model receives a stimulus with the new value, we update the value in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*). This does not activate a transition to another state nor an internal transition and thus has no further effect.
If an LCp is modelled with multiple processes (e.g., orthogonal regions), this responsibility is given to only one of the processes. This process updates the value as described. Subsequently, it sends the updated `DT` struct to all other processes to distribute the new values.
One exception to this rule are pseudostates. Since a region is not allowed to stay in a pseudostate, the transition from one state to another is regarded as atomically (run to completion semantics, see Section 3.4.4). This makes the receiving of Data Trigger in stimuli in pseudostates also not valid as it would influence subsequent evaluations of guards of the same transition. A transition can consist of one or more pseudostates where multiple guards may apply. These guards always operate with the Data Trigger in port values that were present when the transition got activated by a change event (for Data Trigger in ports the transition got activated by a change event from the associated Trigger in port). Thus, we only accept Data Trigger in stimuli in a stable state (after a transition has completed).
This also applies to entry- and exit-behaviour: When entering a state, we execute the entry-behaviour **before** accepting any new Data Trigger in stimuli. Similarly, when leaving a state, we execute the exit-behaviour followed by the entire behaviour of the transition before accepting any new Data Trigger in stimuli.

Data Trigger out port As discussed in Section *Block property*, the values of Data Trigger out ports are only valid when the corresponding Trigger out port creates a change event. However, the values of Data Trigger out ports may be changed at any time during execution. In fact, if we arise a change event of a Trigger out port, we first send the updated values of all associated Data Trigger out ports as responses. This is also according to the specifications where a new value is first assigned to the Data Trigger out ports. For the sake of completeness, we update these values in our struct `DT` of type `dataPorts__LCp` in the same way as for Data in/out ports (see Section *Data in/out port*).

Conclusion Data Trigger out ports could be modelled as parameters of Trigger out ports but following the previous conclusion in Section *Data Trigger in-/out port and block property*, these ports obtain more the semantics of responses that occur punctual. We decided against an aggregation of Data Trigger out ports with their corresponding Trigger out port to keep the model as close to the EULYNX specifications as possible.

Definition We define each Data Trigger out port as response on the corresponding channel with exactly one parameter assigned. This parameter represents the value of the Data Trigger out port and has the AML data type corresponding to the SySim value type of the port (see Section *Data types*). We preserve the names of the ports unchanged. The name of the parameter is equal to the name of the Data Trigger out port (and therefore of the response itself) with an underscore prepended.
If we define a response on an internal channel, we must concatenate the name of the response with the corresponding stimulus (see Section *Trigger in port* for details).

[5] Don't Repeat Yourself

Initialisation In the EULYNX specifications, Data Trigger out ports are initialized with a default value in the initialization operation. We therefore also initialize the values in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*).
From an electrotechnical view where these ports are physical interfaces connected to an adjacent LCp, the new values of these ports would be received by the adjacent LCp immediately after initialization. We therefore assume that the LCp shares the initialized values with its environment. Thus, after assigning the initial value to a Data Trigger out port, we send the response containing the newly initialized value.

Updating As part of an effect, the value of Data Trigger out ports can be updated. We update the value in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*) according to the EULYNX specification. This can occur as effect of any transition, internal and external.
Whenever the Data Trigger out value is updated, it is expected that the LCp shares the newly assigned value with its environment (e.g., with the adjacent LCp). Thus, similarly as during the initialization, we send the new value as part of a response immediately after updating the value. As concluded above (see Section *Data Trigger in/out port and block property*), for Data Trigger out ports this must be done immediately before the associated Trigger out port is set to *true*.

Data in/out port

Data in/out ports may be part of the interface of an LCp with its environment. They are used to pass information to and from components. Unlike Data Trigger in/out ports, Data in/out ports are not associated with any Trigger in/out port (or any port in that matter). Data in/out ports are identified by the prefix of the letter *D* followed by a consecutive number. Appended is a descriptive name about the semantics of the information this port transmits. Each Data in/out port has a datatype specified. The data type can be of any supported SySim value type (except *PulsedIn* and *PulsedOut*) but in the considered specifications of this research only the following data types are used:

- Boolean
- String
- Integer

Unlike Data Trigger ports where it is only allowed to evaluate the data at certain times, the value of Data ports is regarded as valid at any given time [30]. Some of the Data ports may change the transmitted value during execution time. According to the Modelling Standard, we therefore separate between [30]:

- **constant information** that is supposed not to change the value during system operation
- **alterable information** that may change the value during system operation

However, in the EULYNX specifications, it is not mentioned if a Data port carries constant or alterable information. This distinction is only made in the semantics of the information the ports provide and sometimes given in informal port descriptions. Therefore, we can not reliably distinguish between constant and alterable information. Nevertheless, it does not matter for our purpose since we can simply treat all Data ports as alterable.
The semantics of Data ports differs from the stimulus-response behaviour because the information these ports are carrying is regarded as *always* valid. From an electrotechnical view, these ports can be regarded as physical interfaces where a constant current is present, modulating the information. Thus, we need to map these semantics onto the stimulus-response behaviour of an STS since AML as additional abstraction layer does also not allow to model this functionality directly.

Finding 1: By definition, AML and the underlying STS do not support reading or sending a continuous flow of data that would allow for the evaluation of a port (or label) at any time. Only stimuli and responses are supported.

In our AML model, we send changed values of Data out ports as a response that contains the new information in a parameter. Similar to that, we receive changed values as a stimulus containing the new information in a parameter. The stimulus is received whenever the information changes. We can store

the last received value of these ports to allow for a later evaluation. This makes a data structure for Data ports necessary. We therefore define the following struct of type `dataPorts__LCp`, where `LCp` is the name of the corresponding LCp the Data port is located in.
For example, the Data port struct of the LCp *F_SCI_EfeS_Sec_SR*:

```
def dataPorts__F_SCI_EfeS_Sec_SR
  {
    'in' => {'DT7_PDI_Version' => :string,
      'D3_Con_PDI_Version' => :string,
      'D4_Con_Checksum_Data' => :string,
      'D23_Con_Checksum_Data_Used' => :boolean
    },
    'out' => {'D50_PDI_Connection_State' => :string,
      'DT13_Result' => :string,
      'DT13_Checksum_Data' => :string
    }
  }
end
```

This struct is instantiated with the name `DT` in each process of the LCp separately and updated between different processes, as described in Sections 3.4.3 and 3.5:

```
var 'DT', dataPorts__F_SCI_EfeS_Sec_SR, :void
```

Whenever a value of a Data port changes, we update the corresponding value in this struct.

Data in port As discussed before, the values of Data in ports are permanently regarded as valid [30]. The LCp receives continuously data on these ports during operation. The received data is not regarded as locally stored within the LCp but evaluated directly at the port itself. This makes an initialization of the values redundant. During operation, alterable information can change spontaneously. Whenever this happens, it is expected that the LCp operates immediately with the new values.

Conclusion Since STS does not support reading a continuous flow of data, we will translate Data in ports to stimuli and store the lastly received values. If the SUT sends the values as a continuous data flow, the adapter ignores unchanged values and only generates a stimulus if the value changes.

Definition The definition follows the definition of a Data Trigger in port.

Initialisation As described above, we always store the last received value of the Data in port. Whenever we evaluate a Boolean expression containing the value of this port, we use the stored one. Data in ports are not initialized in the EULYNX specifications because they are not assumed to be stored in the LCps scope. When these ports are evaluated, it is expected that they are providing valid data.
Since we do not have the environment of the real system, we are missing values of Data in ports. Thus, after start-up, we might not have received a value on a Data in port yet and any evaluation fails. We therefore either have to make sure that we send a stimulus with a value before a Data in port is evaluated or we have to initialize these ports during the initialization operation. We decided for the latter one.
We need to assume the initialized default values of these ports because the EULYNX specifications do not provide this information.

Updating The updating process follows the updating process of a Data Trigger in port with the exception that a transition to another state or an internal transition might be activated (a change event is generated).

Finding 2: Without the real environment, some Data in ports remain uninitialized. Any evaluation fails.

Data out port The value of the Data out ports is shared with the environment of an LCp in form of a continuous flow. On system start, these ports are initialized with a default value and uninterruptedly sent out. The values of Data out ports may be changed at any time during execution. Whenever we change the value of a Data out port, the LCp is expected to distribute the new value to its environment. We do this with sending a response containing the new values. For the sake of completeness, we update these values in our struct `DT` of type `dataPorts__LCp`.

Conclusion	Since STS does not support sending a continuous flow of data, we will translate Data out ports to responses and store the lastly sent values. If the SUT expects the values as a continuous data flow, the adapter can generate a continuous data flow if needed.
Definition	The definition follows the definition of a Data Trigger out port.
Initialisation	In the EULYNX specifications, Data out ports are initialized with a default value in the initialization operation. We therefore also initialize the values in our struct `DT` of type `dataPorts__LCp`. We then send the newly initialized values to the environment of the LCp according to the description of Data Trigger out port.
Updating	The updating process follows the updating process of a Data Trigger out port.

3.4.3 States

States are a central language element of the SysML STDs. Each state is contained within a **region**. Each STD has at least one region which serves as a container for a number of states, including pseudostates, and the transitions between them. During execution, each region has one single active state that determines the transitions that are currently viable in that region [30]. When entering a region, its initial pseudostate is activated. Each region must therefore contain such an initial pseudostate. Additionally, a final state that corresponds to the completion of the region may be present. Multiple regions can describe **concurrent behaviour** within the STD where the regions are defined.
A **state** models a situation in the execution during which some invariant condition holds [41]. The invariant condition is fulfilled as long as the state remains active and represents a static situation or a *stable state*. An active state may be left with the receiving of a change event (stimulus) while effects may be processed according to its defined behaviour. The Modelling Standard defines a state as the representation of a *between stimuli* condition [30]. States are defined by a unique name which is descriptive about the invariant condition encapsulated by the state.
AML supports states directly. AML states serve a similar purpose as the states used in SysML although they are rather a "marker" in the model and lack some needed semantics. For example, there is no explicit support of entry- and exit-behaviour and only one type of state is known. Other types of states, such as composite and orthogonal regions, are not available in AML. In SysML and the underlying UML, different kinds of states are defined where some of them are used in the EULYNX specifications.
We have the hypothesis for our research that we can translate these types of states with the combination of different AML functionalities and map them into an STS. We therefore analyse the translation of each type of state in the following where we give special attention to the modelling of pseudostates, composite states and orthogonal regions.

Simple states

The definition of a simple state is a state that does not contain any other states (including pseudostates). It is defined by a unique name that is descriptive about the invariant condition encapsulated in this particular state. The entry of a state is defined by its activation caused by a received change event which enabled a transition into this state. Similarly, the exit is the deactivation of the state which is also triggered by a received change event that enabled a transition out of this state.
A simple state can contain behaviour that are executed on entry or exit of the state. These behaviours are called **entry-behaviour** and **exit-behaviour** respectively. And entry-behaviour is always the first behaviour that is executed when a state is activated, before any other behaviour is considered. Analogously the exit-behaviour is always the first behaviour that is executed when a state is deactivated.
Additionally, a state can contain a **do-behaviour**. Do-behaviour are executed immediately after the entry-behaviour has completed and is continuously executed until the state is deactivated. In the un-

derlying methodology of the Modelling Standard, do-behaviour are not to be used [30]. We therefore do not consider this type of behaviour in our research.
Simple states may also contain **internal transitions**. These transitions are called *internal* because they do not lead to the deactivation of a state. They have the same source and destination state. Since the same state remains active, entry- and exit-behaviour are not executed. However, internal transitions have an **effect** that is executed.
In contrast to internal transitions, **external transitions** are used to deactivate a state and activate another state. This type of transition has the currently active state as source and another state as destination. The destination can either be another simple state but also any other type of state, except an initial pseudostate. When the transition is activated, the currently active state will be left and therefore the exit-behaviour is executed. Like internal transitions, external transitions can have an assigned transition effect. This effect is executed after the exit-behaviour of the state is completed. We will discuss external transitions separately in Section 3.4.4.
Concluding the characteristics of simple states, we define a translation for following properties:

- Signature
- Entry-behaviour
- Exit-behaviour
- Internal transitions

Signature Each state is defined within the containing region uniquely by its name. We adopt the name of states unchanged from the EULYNX specification. This makes it easier to relate the AML model back to the EULYNX specifications and allows for easier navigation throughout the model. For simple states, we use the *state* construct of AML which encapsulates the entire state and demarcates the scope. We model all subsequent model parts that belong to a particular state after the *state* declaration.

Entry-behaviour As previously explained, the entry-behaviour is executed as the first behaviour on state activation. Therefore, we model it directly after the *state* declaration. The behaviour can include the sending of responses and the processing of data updates. The receiving of stimuli can not be part of an entry-behaviour. Moreover, the entry-behaviour is always deterministic and therefore no *choice* nor *repeat* structures are allowed. States not always have an entry-behaviour specified. If it is not specified, we can simply omit it in our AML model as well.
Most entry-behaviour are similar across multiple states within a region, a composite state, or even within an entire STD. For example, an entry-behaviour often contains updating the value of the same Data out ports. This is done by means of assigning a string value that describes the currently active state of the LCp to a Data out port which is then exposed to its environment. In this manner the LCp propagates its currently active state to its environment. To simplify the model, we outsource frequently used entry-behaviour into functions with appropriate parameters. In the previously mentioned example, all states of an STD call a function in their entry-behaviour to update a Data out port with their state name as a parameter.

Exit-behaviour The exit-behaviour is executed as the last behaviour before the state is deactivated. Therefore, we need to call it whenever the state is left. A state is left with the activation of an external transition. Multiple external transitions for different change events can exist. The exit-behaviour applies to all of them and is called always immediately before the transition is activated. In contrast to the entry-behaviour where we have a single common entry point for all transitions, we must model the exit-behaviour at multiple places.
We model a transition to another state using the AML *goto* command. To accomplish the desired behaviour, we call the exit-behaviour at all places where we leave the state with a *goto* command. If the transition has effects, we call the exit-behaviour before them. As a best practice, we outsource the entire exit-behaviour into an AML *behaviour* construct. Similar to entry-behaviour, exit-behaviour are deterministic. The processing of change events and the AML structures *choice* and *repeat* are not allowed. States may not have an exit-behaviour specified. In these cases, we simply omit them in the AML model.
Special rules apply if a handover between processes takes place. We will discuss these in the Sections *Orthogonal regions* and *Composite state*.

Internal transitions Like external transitions, internal transitions are triggered with an event but do not affect the active state. The invariant condition of the currently active state is maintained during the entire internal transition. The current state will neither be left nor newly entered. As a result, entry- and exit-behaviour are note executed.
Whenever a change event of an internal transition occurs, the transition is activated. The transition may have a guard assigned which is subsequently evaluated (see Section *Guards*). If the guard evaluates to *true*, the effect of the internal transition is executed. If no guard evaluates to *true*, no effect is executed, and the event is ignored (no behaviour is called). Although there is no rule for an internal transition that requires it to have an effect specified, internal transitions without an effect are redundant.

Pseudostates

Pseudostates are introduced to add semantics to states. According to the Modelling Standard, the difference between a pseudostate and a regular state is that a region can never stay in a pseudostate [30]. However, we will slightly deviate from this definition in our model: In the underlying STS, a state representing a pseudostate is allowed to stay activated in certain cases if our model consists of multiple interacting processes. For example, a process remains in its initial pseudostate whenever the corresponding region is not active. Additionally, we introduce a new *idle* pseudostate that is activated whenever a nested region is active. We discuss the details of these pseudostates in Section *Composite state*.
In the underlying methodology of the Modelling Standard, the following SysML pseudostates are defined:

- Initial pseudostate
- Final state
- Choice pseudostate
- Fork pseudostate
- Join pseudostate

In the considered specifications of this research the last two types are not used. We will define a translation for each used type of pseudostate in the following.

Initial pseudostate Each region has exactly one initial pseudostate. This state defines the initial entry point of the corresponding region and is within the entire STD uniquely identified by the keyword *Initial* and a consecutive number. Whenever a region is activated, it starts from the initial pseudostate. The outgoing transition from an initial pseudostate into another state (any type of state) can have an effect assigned which is executed when this transition is made. It is, however, not allowed to have events or guards assigned to this transition because then it could happen that the initial state can not be left. This is invalid since the active region can never remain in the initial pseudostate after activation. In fact, the outgoing transition is activated implicitly on activation of the initial pseudostate.
We translate the initial state into a normal state in AML. It is obvious that entry- and exit-behaviour are not specified. This reduces our state in AML to the *state* construct followed by the transition effect and a subsequent *goto* command to the next state. In the considered EULYNX specifications, the effect is defined as an ASAL operation that is called during the transition. We translate this ASAL operation into an AML function and call it likewise during this transition. In our case, it is always an *init*-function to initialize values of the Data (Trigger) out ports and block property values.
As mentioned before, with multiple interacting processes a process remains in its initial pseudostate whenever the corresponding region is not active. In this case the transition from the initial state with its effect is only activated when the change event is received which triggers the handover into the corresponding region. Further details about interacting processes are described in Section *Composite state*.

Final state A transition to a final state is activated when the execution of a region has completed. These states are uniquely identified in the entire STD by the keyword *Final* and a consecutive number. Whenever a transition to a final state has been activated, no further transition to another state within the same region can be activated afterwards. In contrary to initial pseudostates, the transitions to a final state may have events and guards assigned. Only if the corresponding event occurs and the guard is satisfied, the transition to the final state is activated and the region completes the execution. An effect can be specified on the transition.
The LCps are sub-components of a reactive system. We can therefore regard each LCp as a reactive system in its own right. This allows us to apply the definition of a reactive system to each LCp.

This definition includes [42]:

- The system is in continuous interaction with its environment.
- The process by which the reactive system interacts with its environment is usually nonterminating. If a reactive system terminates during its availability time, this is usually considered a failure.

With this in mind, we see that LCps are not allowed to have a final state in the outermost region. Only nested regions in form of sequential states are allowed to terminate their execution. We can replace the semantics of a sequential state terminating with handing over the control to the containing region.
We model final states as normal states in AML. We activate them with an external transition from another state. As with any transition, the transition to the final state may have an effect specified. Final states do not contain entry- or exit-behaviour. On activation of the final state, we hand over the control to another process; to the process of the containing region to be more precise. Then, we return to the initial state of the terminated region. Further details about interacting processes are described in Section *Composite state*.

Choice pseudostate Choice pseudostates are uniquely identified in the entire STD by the keyword *Junction* and a consecutive number. A choice pseudostate is used to construct a compound transition path between states. The compound transition allows more than one alternative path between states to be specified, although only one path can be taken in response to a stimulus [30]. The various outgoing transitions have guards assigned but no stimuli. It is not allowed to receive a stimulus in a choice pseudostate in order to activate an outgoing transition because this would require that the region remains in this pseudostate for a certain amount of time. The activated transition out of the choice pseudostate is the one where the guard is satisfied. The validation of the guards takes place *after* the choice pseudostate was activated. All transitions, the one that leads into the choice pseudostate and the ones leaving the choice pseudostate, may have effects specified. As a result, this allows effects executed on the prior transition to affect the outcome of the choice [30].
Because a region can never remain in a pseudostate, one guard of one transition **must** validate to *true* after activating the choice pseudostate. Otherwise, the LCp would be stuck in this choice pseudostate. Additionally, it is not allowed by the SysML standard to have more than one outgoing transition validating to *true* [32]. Following this definition, we can conclude that the conditions of all guards must be orthogonal to each other. Guards are further described in Section *Guards*.
We model choice states as normal states in AML. We activate them with an external transition from another state. Like the other pseudostates, entry- and exit-behaviour are not allowed. Within the choice pseudostate, we use the AML *choice* construct. Each possible option of the *choice* construct represents one outgoing transition, and we specify each guard as *constraint*. We add a *goto* command to another state at the end of every option since it is not allowed that this state remains active.

Composite state

A composite state or **hierarchical state** is defined as a state that has nested regions. Composite states allow state machines to scale in order to represent arbitrarily complex state-based behaviour [30]. A composite state does either have one single nested region (so called **sequential state**) or multiple orthogonal regions. While the first one is the more frequently used form, we analysed the translation of the latter one separately in Section *Orthogonal regions*. Note that composite states can contain other composite states themselves. In our translation we look at one composite state that contains one or multiple sequential states, but it is possible to apply our approach recursively.
Each region or STD assigned to a state has a set of mutually exclusive disjoint subvertices and a set of transitions [30]. A region is therefore self-contained with its own substates, pseudostates, and transitions. A **substate** is a state within a sequential state that makes the containing region a composite state. A sequential state contains exactly one initial pseudostate and may have a final pseudostate. Only sequential states are allowed to have final pseudostates as discussed in Section *Final state*.
Because of the self-contained characteristics of sequential states, we translate them into different *processes* in AML. They are used to describe parallel behaviour of multiple concurrent processes. While the processes are evaluated together, they are translated into a separate and independent STS. According to *Effective AML* [28], a good indication of parallel behaviour is when certain behaviour is possible in virtually every state of a model. This is given in our SysML specifications by interruptions of sequential states, as discussed hereafter.
If the processes are independent from each other, each process is modelled with its own transition system and all interleavings of the transitions are allowed [28]. With sequential states, our regions are widely

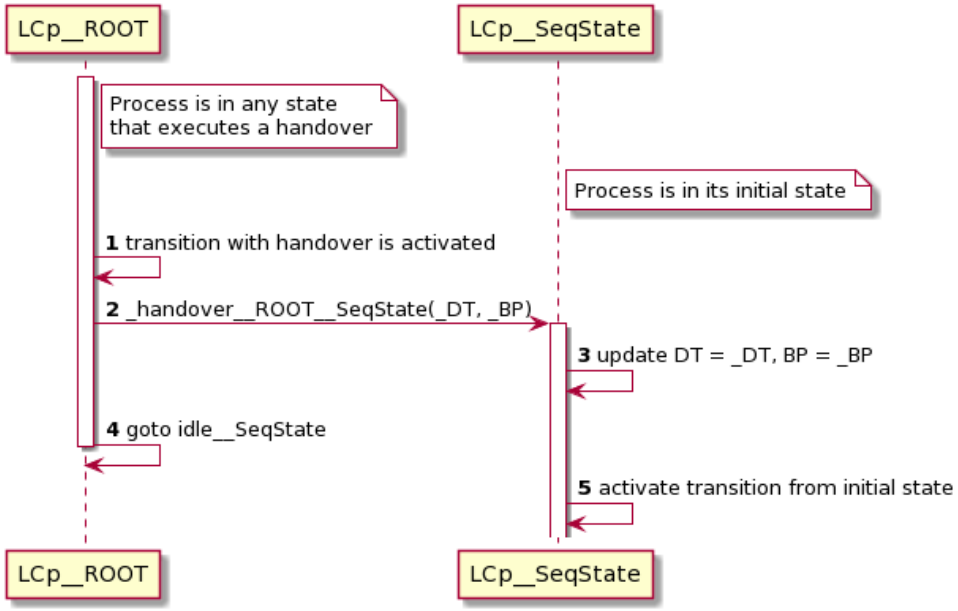
independent except for certain *handover points* where one sequential state is activated or deactivated. These points are the following:

1. Activate a sequential state from a state in the containing region
2. Deactivate a sequential state and activate a state in the containing region
3. Activate a sequential state from another sequential state
4. Interrupt a sequential state from the containing region and activate a state in the containing region
5. Interrupt a sequential state from the containing region and activate another sequential state: This is a combination of the previous process and the first one

AML offers **internal channels** for the communication between parallel STSs. Thus, we define labels on internal channels in all participating processes. Whenever we activate one of these labels, both processes advance on it at the same time. The processes are still fully interleaved, except on internal transitions on which they **synchronize** [28].
When we make the synchronization between processes, we need to "force" the label to be proceeded. In other words, we force both STS to transition on the corresponding labels. In AML, this is called *disruptive communication*. With disrupting internal communication, the internal label takes priority over all other labels that may be possible in the state-vector [28]. Therefore, we implement the synchronization disruptive whenever other labels are possible in a certain state.
As we introduced in Sections *Block property* and *Data in/out port*, an LCp has up to two data structs: one of type `blockProperty__LCp` and one of type `dataPorts__LCp`. While the first one stores all block property values, the latter one caches the values of the Data (Trigger) ports. All data is owned by the LCp and is used throughout the entire semantic interface. The data can change at any time during execution, for example with the receiving of a changed value of a Data in port. If we receive a changed value, we update the cached ones accordingly. Since we divide the semantic interface further into parallel processes with distinct responsibilities, we need to manually ensure the timeliness of data whenever they are used by a process. AML does not support shared variables between parallel processes because this yields synchronization problems. We can, however, use the synchronization points to exchange data. We therefore append each data struct of the LCp as a parameter to the synchronization labels where necessary.
In the following we present a concise overview of our approach to translate sequential states into independent STS. We look at one process representing the composite state, the region that contains one or more sequential states (hereafter simply referred to as *region* or *LCp__ROOT*). Then, we look at one or more processes that represent a sequential state within this region (hereafter referred to as *sequential state* or *LCp__SeqState*). If sequential states are nested, we can use our approach recursively.

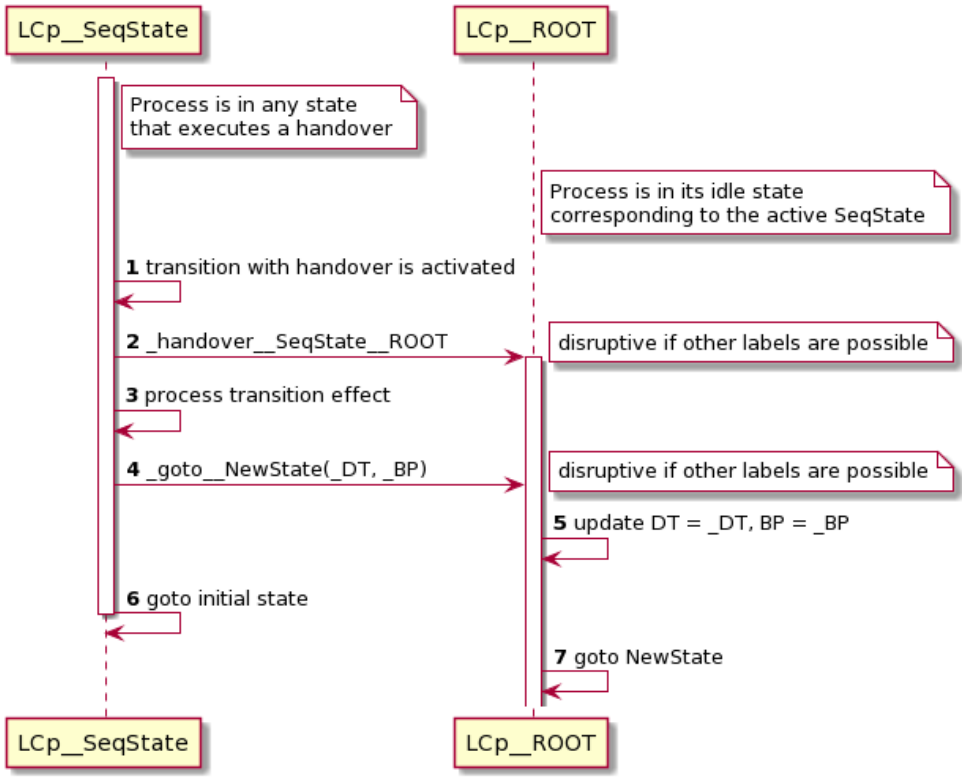
Activate a sequential state from a state in the containing region We define the internal *handover* label for this purpose. We send this label from the active state of the region where the sequential state is activated from. After sending the *handover* label, the region goes to the idle state of the corresponding activated sequential state. Before the synchronization took place, the sequential state was not active and thus in its initial pseudostate. At synchronization, the sequential state updates the data values that are contained in a parameter of the *handover* label. Afterwards, the transition from its initial state is activated immediately. This handover sequence is depicted in Figure 3.3.

Figure 3.3: Composite states: Activate a sequential state from a state in the containing region



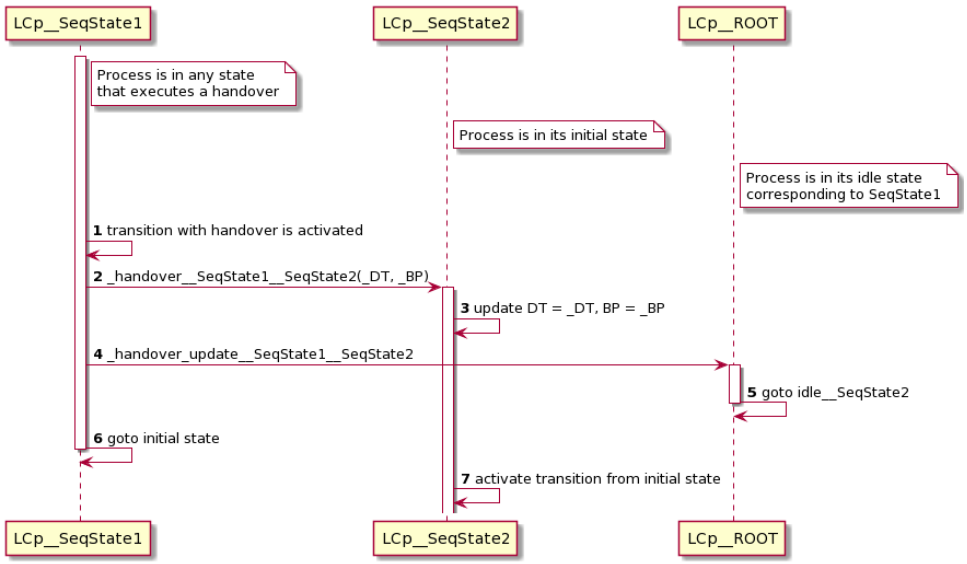
Deactivate a sequential state and activate a state in the containing region For the deactivation of a sequential state and returning into the containing region, we use previously introduced *handover* process with an additional *goto* label. This time, we send the *handover* label in the other direction, from the sequential state to the region. The region is prior synchronization in the idle state corresponding to the active sequential state. The handover label is followed by a *goto* label. After sending this second label, the sequential state returns to its initial pseudostate. Besides updating the data values that are contained in a parameter of the *goto* label, we activate the transition of the region to a specific state with the *goto* label. Which state is determined by the particular *goto* label.
We must send the *handover* label immediately after deactivation of the last state within the sequential state (excluding pseudostates) before we process the transition effect and before we process the exit-behaviour. In contrary, we send the *goto* label as the very last label. In this order it is not possible that the region interrupts the sequential state while making the transition. This handover sequence is depicted in Figure 3.4.

Figure 3.4: Composite states: Deactivate a sequential state and activate a state in the containing region



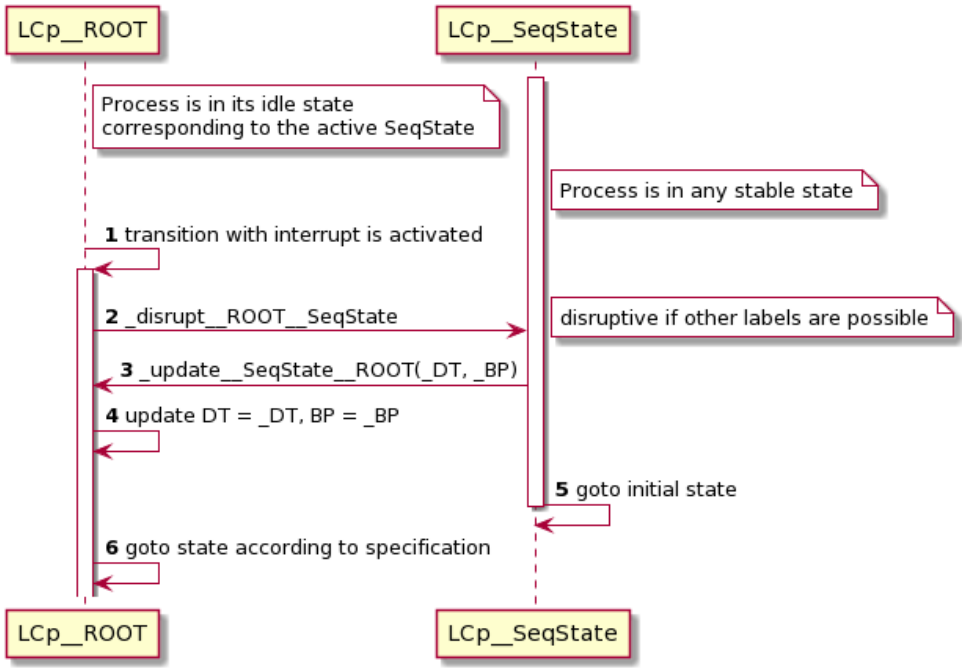
Activate a sequential state from another sequential state We implement this handover similar to the activation of a sequential state from a state in the containing region. Additionally to that procedure, we must update the region where both sequential states are contained in. This is necessary because the region handles the interruptions of the contained sequential states. Therefore, it needs to know which change events to accept to interrupt the currently active sequential state. For this purpose, we introduce the *handover_update* label which we send directly from the previously active status in the first sequential state to the region. The process of the region goes then into the idle state of the newly activated sequential state. A data update of the region is not necessary, this is only done when deactivating all sequential states and returning to the region. This handover between two sequential states is depicted in Figure 3.5.

Figure 3.5: Composite states: Activate a sequential state from another sequential state



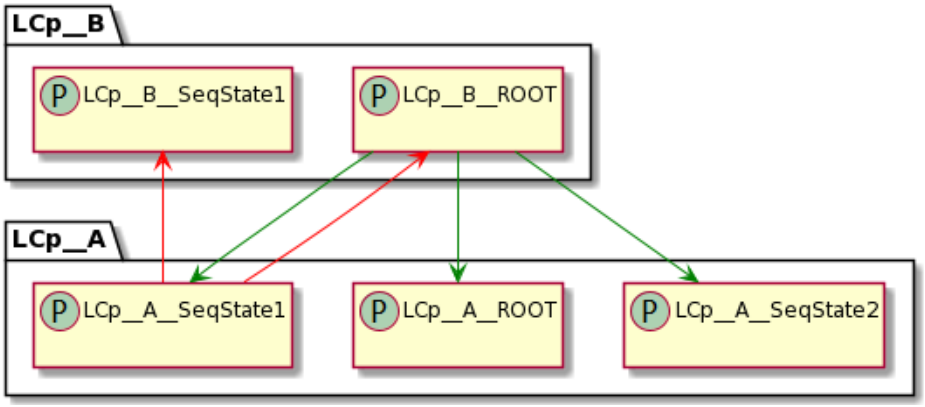
Interrupt a sequential state from the containing region and activate a state in the containing region A region interrupts a sequential state when receiving certain change events. Although this interruption can occur regardless of the active state within the sequential state, it is not possible during transitions within the sequential state nor when the sequential state is in a pseudostate. If the region wants to interrupt the sequential state in these mentioned moments, the interruption is made immediately after the transition is completed and a stable state is activated.
For the interruption, we introduce the *disrupt* label. The region sends this label to the process of the sequential state when it receives a change event that leads to its interruption. The interrupted process replies with an *update* label with the data values contained in a parameter. Afterwards, the sequential state returns to its initial state while the region goes to the state according to the specification. This interrupting sequence is depicted in Figure 3.6.
For convenience, we outsource the interrupt behaviour and the disrupt behaviour into AML functions and AML macros. The macro for the receiving of the disruption must be included in all states within the sequential state where a disruption is valid.

Figure 3.6: Composite states: Interrupt a sequential state from the containing region and activate a state in the containing region



Discussion & alternative approach The introduced translation follows the principle of modelling independent parts with their own behaviour using parallelism [28]. Each sequential state encapsulates its contained substates and transitions in one process. Linking the translation back to the SysML specifications, the semantics of composite states and sequential states are preserved. A region that contains one or more sequential states only knows about the owned sequential states but is agnostic about their internals. From the containing regions perspective, sequential states are mostly treated as simple states. We demonstrated that we can translate sequential states from SysML into independent STS using synchronization between them.
While we showed a valid approach for the translation of a single STD, our approach has limitations when it comes to parallel composition. If we look at a situation where we have two (or more) LCps that consist of multiple processes, their composition becomes complex and soon infeasible. We demonstrate this in Figure 3.7.

Figure 3.7: Parallel composition of LCps with sequential states



The LCp *LCp__A* has three processes. One is the *LCp__A__ROOT* process representing the outermost region. In this region, we have two sequential states *LCp__A__SeqState1* and *LCp__A__SeqState2*. Both are represented with their independent process (and STS). On the other side we have the LCp *LCp__B* with two processes *LCp__B__ROOT* and *LCp__B__SeqState1* following the same structure. For parallel composition, the LCps are connected by internal channels. To be more accurate, the various processes are connected with each other.
Which LCps are connected and what ports are used is defined in the *Logical Architecture* (LA). Since the LA only defines the interconnection of LCp-TypeA or LCp-TypeB with one another, the actually used interfaces by each process may only be a subset of these. In other words, not every process communicates with every other process of the other LCp. This gives us the needed independence of the processes. While Trigger ports seem to follow this principle in the EULYNX specifications, Data ports do not. Data in ports carry per definition valid information at any time. This information is available immediately to the entire LCp. In our case this means that every process within an LCp needs to have valid data from the Data in ports at any time. The various processes share these ports with each other. However, shared data among multiple processes within one LCp is not supported.
Although only one process in each LCp represents the currently activated region or sequential state, the processes forming the other LCp do not know which one is active. Hence, the updated data needs to be sent to every process in the other LCp. In a worst-case scenario, this results in the Cartesian product of all internal labels which becomes infeasible very quickly. Storing the information about the active process in the other LCp yields a similar problem because this information needs to be updated in all adjacent LCps when activating another state. This problem becomes especially obvious if we consider not only two but many more LCps being composed. We therefore suggest another translation approach.

> **Finding 3:** Parallel composition of LCps consisting of sequential states becomes soon infeasible if the sequential states are mapped onto independent STS.

Important to note is that each process represents one region or sequential state within an LCp whereas only one region or sequential state is active at any time. Therefore, we can convert the entire region with all contained sequential states into one process. In terms of SysML STDs this corresponds to the removal of the hierarchy of composite states given by sequential states. As a result, each state of a former sequential state must directly implement the interrupts of the region as transitions. Although not used in considered specifications of this research, entry- and exit-behaviour of the sequential state must be preserved as well. This could be done with the introduction of new pseudostates that represent this behaviour.
Based on our results we recommend the substitution of sequential states in order to remove mentioned challenges with parallel composition.

Orthogonal regions

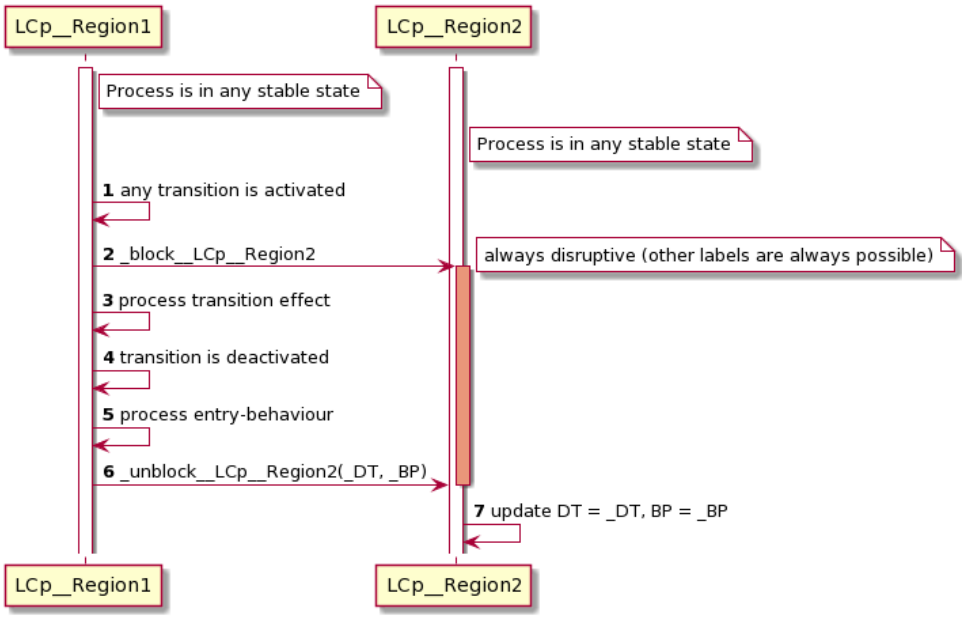
An orthogonal region is a region that consists of multiple (at least two) parallel regions. Orthogonal regions are also known as *concurrent state* or *orthogonal composite state* [30]. In contrast to composite states where only one region or sequential state is active at any time, orthogonal regions have multiple active regions. Whenever the orthogonal region is activated, each nested parallel region is active at the

same time. Thus, orthogonal regions are used in SysML to model concurrent behaviour. As the name suggests, the nested regions are entirely independent from each other and therefore have an initial pseudostate each. This also means that received change events are processed within all regions independently. Transitions that cross the region borders between the orthogonal regions are not valid.
Because of the concurrency between the regions, we translate them into different *processes* in AML. In Section *Composite state* is a description of AML processes and how to synchronize them. The main characteristics of processes is, that each process is modelled with its own transition system and all interleavings of the transitions are allowed [28].
Orthogonal regions are independent from each other. While Trigger ports follow this principle in our specifications, Data in ports and block properties do not. Since both regions are within the same SysML STD representing the behaviour of a single LCp, values of Data in ports and block properties are shared among all regions. For example:

- If one region (process) changes a value of a block property, all other regions (processes) must operate with the new value instantly.
- If a Data in port changes its value, the new value must be propagated to all regions (processes) immediately.

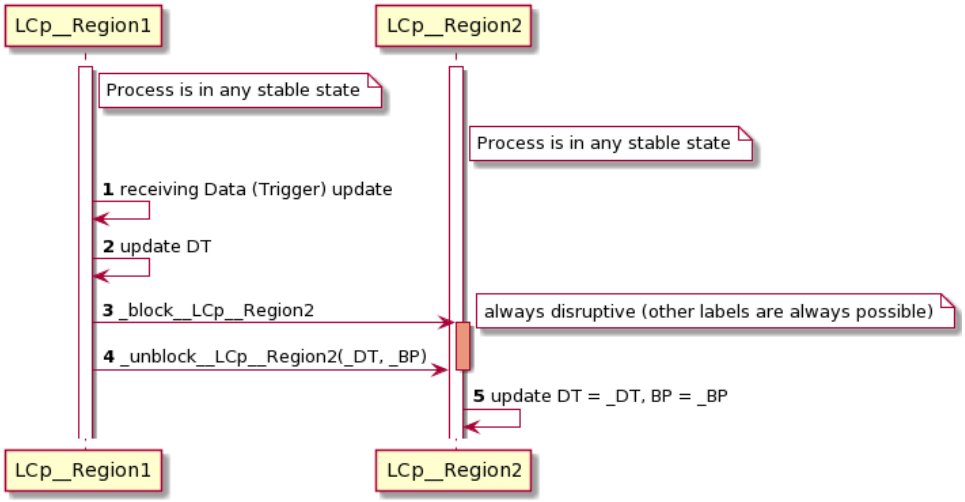
Composite states yielded a similar issue. As a solution, we removed the hierarchy from the STDs and the concurrency from our model. Unfortunately, we can not use the same approach for orthogonal regions because we do need actual concurrent behaviour. Every possible change event for each active region must be possible at all times.
With entirely parallel processes operating on the same data, it can happen that one process alters a value while the others evaluate it. For example, if one process is transitioning from one state to another, a parallel process can influence the transitions outcome by changing data that is evaluated during the transition. These **race conditions** are undesirable. To remove them we follow the run to completion semantics (see Section 3.4.4). Whenever one process is starting a transition, it synchronizes with all other processes. When the other processes receive this synchronization, they are "blocked" and can not receive any other stimuli. Once the process has completed the transition, it synchronizes again with all other processes to "unblock" them.
For this purpose, we introduce the *block* and *unblock* label. Whenever one region activates a transition, we send the *block* label to all other active regions. We send it before we execute any other transition effect. All active regions must accept the *block* label of all other regions in any state (excluding pseudostates because they are part of a transition). After the activated transition is deactivated and all immediate subsequent behaviour such as entry-behaviour is processed, we send the *unblock* label to all other regions. After they have received the *block* label, the only thing they accept is the *unblock* label from the corresponding region. The *unblock* label contains all data of the LCp as a parameter and updates them in all other regions. This blocking-unblocking process with two orthogonal regions *LCp__Region1* and *LCp__Region2* is depicted in Figure 3.8.

Figure 3.8: Orthogonal regions with multiple parallel processes



We delegate the responsibility of receiving changed Data (Trigger) in port values to one of the processes. As without orthogonality, this process is able to receive a stimulus of a Data (Trigger) in port in every stable state. After receiving such a stimulus and updating the stored values accordingly, we update all other processes instantly. To achieve this, we follow aforementioned approach: We first block all other processes and subsequently unblock them, providing the updated data values. This process is depicted in Figure 3.9.

Figure 3.9: Data (Trigger) in port update with orthogonal regions



3.4.4 Transitions

In SysML STDs, transitions are a directed relationship between a source and a destination state [30]. Is the source unequal the destination, we refer to it as *external* transition. External transitions change the active state within an STD. Otherwise, if source and destination are equal, the transition does not affect the active state. These transitions are called *internal* (see Section *Internal transitions*). The following definitions apply for both types of transitions.
A transition consists of the following three parts where we define a translation for each part:

1. Event (Trigger): If the triggering event arises, the guard (if specified) is evaluated. If the transition has a guard specified, the event may be omitted.
2. Guard (Condition): This condition is evaluated whenever the transition is potentially activated. If the guard evaluates to *true*, the transition becomes active. If the transition has an event specified, the guard may be omitted.
3. Effect (Behaviour): The effect is executed during the transition. A transition may have no effect specified.

An inherent concept of transitions in SysML is the **run to completion** semantics. State machines always run to completion, which means that they are not able to consume another event until the state machine has completed processing the current event. Thus, the next event will be consumed only if all effects (behaviour) of the previous event have been completed [30]. This is congruent with the AMP strategy to prioritise the observation of responses over processing stimuli. AMP only processes a new stimulus if a location in the model is quiescent. A location in a model is called quiescent if the model specifies that the SUT may not send any responses at that point [28].

Events

The Modelling Standard defines an event as an occurrence that can be measured with regard to location and time and causes a transition to occur [30]. In this definition, the location can be understood as the LCp. Following different types of events are specified where we define a translation for each type used in considered specifications of this research:

- Change event
- Time event
- Call event (internal broadcast event)

According to the Modelling Standard, events that can not be consumed in the currently active state are discarded. This lines up with the strategy of AMP, which only sends stimuli that are possible in the currently active state. Thus, we do not need to discard events. Because the semantics of internal events are different (as discussed in Section *Composite state*), this is only valid for external events. As a result, we add the receiving of all internal stimuli in every stable state during parallel composition (see Section 3.5).

Change event Change events are continuously evaluated. They are defined as a Boolean expression that was first evaluated to *false* and becomes satisfied subsequently. Every time such an expression is newly evaluated to *true*, a change event is triggered. The Boolean expression of a change event may contain the following arguments [30]:

- Trigger in port: This is the most common type of change event. As discussed in Section *Trigger in port*, such a port evaluates to *true*, whenever the *PulsedIn* data type changes to *true*. We translate this into the receiving of a stimulus without a parameter. The Boolean expression using a Trigger in port as argument may exclusively comprise one Trigger in port [30].
- Data in port: The evaluation is made based on the values contained in a parameter. As discussed in Section *Data in port*, we can not evaluate a continuous flow of data because AML and the underlying STS do not support this. We therefore make the evaluation based on the stored value in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*). After receiving a new value as a stimulus, we update the stored data and re-evaluate the constraints.
- Block property: As described in Section *Block property*, block properties are cached values of Data Trigger in ports. We update the block property value in our struct `BP` of type `blockProperty__LCp` according to the specifications during a transition that was triggered by the corresponding Trigger in port (e.g., during an internal transition or a transition in an orthogonal region). Similar to the Data in port change event, we then re-evaluate the constraints.

Note that in contrast to the Data in port, Data Trigger in ports are not to be used as argument of a Boolean expression of a change event [30].

Time event Time events are used to express timeout behaviour. They measure the passed time since state activation and trigger an event after a finite amount of time. In AML we use constraints according to following description. These constraints are equivalent to the *before* and *after* constraints. Timing constraints only work on external transitions. Timing constraints on internal transitions are currently not supported (see Finding 5). Additionally, the referred time is always the time since the last externally observable label was processed. Thus, if the currently active state was activated with an internal transition, the timing constraint it refers to is not the correct one. This becomes an issue when we combine models during the parallel composition (see Section 3.5).
To model time event dependent transitions, we use a `timeout` variable. The variable is of type `time` and declared within process scope:

```
var 'timeout', :time, :void
```

If a state has outgoing transitions that depend on a time event, we set the `timeout` variable on state activation. Therefore, on all external transitions that lead to the activation of this state, we update the `timeout` variable using the current timestamp and an offset. The offset is the time that is allowed to elapse before the time event is becoming effective.

```
send 'external_label', constraint: 'transition_constraint', update: 'timeout = clock +
    offset', note: 'Timeout: $timeout'
goto 'state_with_time_event'
```

For the time event dependent transition(s), we add the following constraint:

```
constraint 'clock >= timeout'
```

Note that `clock` always refers to the last externally observed label. With this constraint, the transition is only allowed to be activated when enough time has elapsed. All other transitions are not affected by this constraint. If the transition has the semantics of a timeout behaviour, this is not intended. Whenever more time than allowed has passed, all other transitions are not allowed to be activated anymore. This forces the model to advance on the timeout transition. Thus, we add the following constraint to all other transitions via a logic AND link:

```
(clock < timeout)
```

In all transitions that lead to the deactivation of this state, we unset the `timeout` variable, except the transition leads to the activation of another state that is dependent on a time event. In that case, aforementioned rule applies, and we update the `timeout` variable with the current timestamp and an offset.

Call event (internal broadcast event) Call events, or internal broadcast events, are events invoked by other SysML block operations [30]. In other words, one transition triggers another one. For example, this could be used in orthogonal regions to trigger a transition in another active region. In the considered specifications of this research, call events are not used and thus we do not provide a translation pattern.

Guards

A guard is a constraint consisting of a Boolean expression in ASAL. If the transition has additionally an event specified, the guard expression is evaluated whenever this event arises. If the guard expression evaluates to *true*, the transition is activated. If the guard expression evaluates to *false*, the event is consumed without further effect. We translate the Boolean expression from ASAL into a Boolean expression in AML as described in Section 3.4.1. To formulate the conditional cases, we use the AML *choice* construct after receiving the event (which is an *option* within a *choice* or *repeat* construct itself). The first *option* in our *choice* construct has as constraint the guard expression, followed by the transition effect. If the transition is external, the last statement is a *goto* statement. The second *option* has as constraint the negated guard expression with no further statements. These mutually exclusive constraints represent an if-else construct. As a general best practice, we outsource the guard expression to an AML function that returns a Boolean value. The negated guard expression is then equivalent with the negation of the function.

Guards of change events that are only dependant on the values of Data in ports or block property values must be modelled differently. Because we have the change event as a constraint in an *option* of the *choice* or *repeat* construct, it can always evaluate to *true*. Then, we would check the guard expression which is in a nested *choice* construct. If the guard evaluates to *false*, we leave this *option* of the *choice* or *repeat* construct and evaluate all change events once again. If the values contained in the constraint have not changed, the same sequence is executed again. This yields two problems: First, we will not evaluate any subsequent *options* in our *choice* or *repeat* construct (other change events). Second, if we block a concurrent process with internal synchronization after entering the first *option* (e.g., this is the case for orthogonal regions), the system does not converge and causing an infinite blocking-unblocking loop.
To overcome this problem, we lift the guard expression to the level of the change event. In other words, we evaluate the change event (which is dependent on the values of Data in ports or block property values) together with the guard expression and only advance if both evaluate to *true*. This is not necessary for change events produced by Trigger in ports since they only occur punctual.

Guard without event A transition can also consist of only a guard without an event. Where a guard is specified without a preceding event, the guard condition is evaluated immediately after state activation, i.e., after its entry-behaviour has completed [30]. If the state is already active and a guard condition of a transition without event would evaluate to *true*, the guard condition is not evaluated, and no transition activated.
In this case we use a helper flag which we set to *true* as part of the entry-behaviour. We evaluate it to *true* as part of the guard condition and set it to *false* after all *options* (in case an external transition is activated, the state is left, and the flag is not in scope anymore).
Outgoing transitions of choice pseudostates are not allowed to have an event specified because they can not remain active. They always have only a guard assigned to each outgoing transition that defines which transition is activated afterwards. In these cases, we do not add the negated condition with an empty statement block (else clause) but translate exactly the guards as specified. If the specified guards are not mutually exclusive, it could happen that the choice pseudostate can not be left. We regard this as invalid specification.

Multiple guards Multiple transitions with identical events but different guards are allowed to be specified. In these cases, we translate each guard as a separate *option* within our *choice* or *repeat* construct. We only add one extra *option* for the else clause. The constraint of this *option* is equivalent to the negated disjunction of all guard expressions.

Effects

Effects are executed behaviour. We already discussed entry- and exit-behaviour in Section *Simple states*. Additionally, transitions (internal and external) can have effects specified too. For external transitions, the exit-behaviour of the deactivated state is executed before the transition effect is processed. The entry-behaviour of the newly activated state is processed after the transition effect is completed. We model the transition effect just before the *goto* statement. If an exit-behaviour is called, the transition behaviour is modelled after this call (between this call and the *goto* statement). Similar to the entry-behaviour, if an effect is specified multiple times, we outsource it to a function. However, effects are mostly unique and therefore outsourcing them is not beneficial. Many transitions do not have an effect specified. In these cases, we can simply omit it. Note that internal transitions always have an effect because otherwise the transition itself would be meaningless. In the underlying methodology of the Modelling Standard, the following elements of behaviour may be represented as effect [30]:

- **Event-driven responses without additional parameters:** These responses are sent by changing the values of the corresponding Trigger out ports with data type *PulsedOut* from *false* to *true*. As discussed in Section *Trigger out port*, this equates in our model to sending a response without parameter.
- **Event-driven responses with additional parameters:** As discussed in Section *Data Trigger out port*, in our AML model we use responses containing a parameter with the value of the Data Trigger out port. We update the new value in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*) and subsequently send the response containing a parameter with the value of the Data Trigger out port.

- **Responses in form of continuous flows:** As discussed in Section *Data out port*, we can not send out data as a continuous flow. Instead, we use an approach similar to the previous behaviour. We update the value of this Data out port in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*) and subsequently send the response containing a parameter with the value of the Data out port.
- **Call behaviour:** A call behaviour is invoked on demand, executed, and terminated after execution and is supposed to define event-driven data transformations [30]. These effects are operations written in ASAL. We translate them into AML functions as described in Section 3.4.1.
- **Time advance behaviour:** The Modelling Standard defines the time advance behaviour as invoked during system activation and executed continuously. It is supposed to define continuous data transformations [30]. In the considered specifications of this research, time advance behaviour is not specified. We do, however, discuss timing behaviour as events (see Section *Events*) and in parallel composition (see Section 3.5).

3.5 Parallel composition

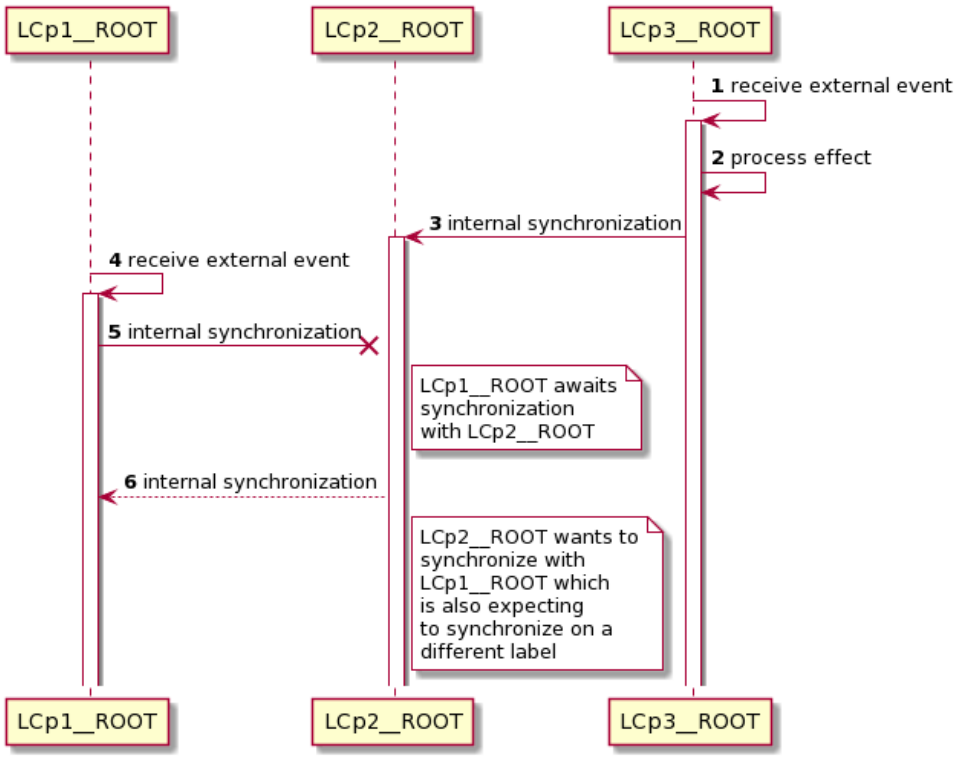
Our translation pattern is based on the independent translations of the behaviour specifications (STDs) of all LCps. Each LCp is represented by one process (except orthogonal regions). This principle follows the general decomposition approach of the EULYNX specifications of each subsystem. For example, the point subsystem consists of 6 LCps on the field-element side and 3 more on the electronic interlocking (EIL) side. The 6 LCps on the field-element side are divided into 4 *generic* LCps and 2 *subsystem specific* LCps. While the generic ones can be found in every subsystem - such as light signalling (LS), level crossing (LX), and radio block centre (RBC) - the latter ones are only to be found in the point (P) subsystem. The communication between the EIL and the field-element is defined in the *EfeS SCI-XX Protocol Stack*, which contains the PDI[6] layer *EfeS SCI-XX PDI* (*SCI-P PDI* for the subsystem point) and the SCP[7] layer.
The observable behaviour of the entire subsystem is only defined by the entirety of all LCps from that specific subsystem. Due to the decomposition, the specifications include internal details, and we regard them as *implementation models* (see Section 6.1). In contrast, AML is designed to model the observable behaviour of a system without considering the internal implementation details (see Section 2.2.1). To match these differences, we need to compose the independent AML models of all LCps likewise to test the stimulus-response behaviour of the complete subsystem. The composition represents a system integration of all LCps to obtain the behaviour model of the entire subsystem. We call it the **composed behaviour**.
The arrangement of the LCps and their interconnections is defined in the logical architecture (LA) by means of LCp-TypeB (see Section 3.1.2). We perform the parallel composition of the AML processes according to the LA.
For the composition, we define one AML process for each LCp. An exception to this are orthogonal regions which contain at least two independent processes within one LCp. Processes representing adjacent LCps share channels with the same labels defined: on one side as stimulus and on the other as response. When composing parallel processes, we change these channels from *external* to *internal*. In our case, internal labels between LCps must always be processed and treated as external labels. We therefore model them as disruptive (see Section *Composite state* for a description of the synchronization between processes).
When a process sends an internal stimulus and the other process is in a state where this stimulus can not be received (because no transition with this stimulus is specified), the first process waits for the synchronization. According to the Modelling Standard, such events that cannot be consumed are discarded [30]. This contradicts with the internal synchronization semantics. To achieve the desired behaviour, we model every internal stimulus in every possible state. If the stimulus does not have an effect, we model the receiving without behaviour. We did this in our model punctual where needed.
The responses on an internal channel must have corresponding stimuli in the process that represents the adjacent LCp. In the EULYNX specifications, the names of connected ports might vary between the LCps. To achieve readability and facilitate the link back to the specifications, we concatenate both names whereby the name of the response comes first followed by two underscores and the name of the stimulus.

[6]Process Data Interface, application layer of SCI (Standard Communication Interface)
[7]Safe Communication Protocol, communication protocol of SCI (Standard Communication Interface)

3.5.1 Functional flow semantics

Channels represent an abstract form of communication, i.e., they reflect the direct communication between LCps or between LCps and interfaces [30]. The internal synchronization between processes embodies the same semantics of direct communication between LCps. The composed behaviour consists of multiple LCp behaviour *synchronizing*. If one LCp receives an event, its behaviour may affect the behaviour of adjacent LCps as well.
For example, lets assume the specified behaviour of a certain LCp reacts to an external event. This event can be either a change event, triggered by a Trigger in or Data in port from the environment, or a time event, triggered by a timeout within the LCp. Based on the specified behaviour, this LCp sends then a stimulus to an adjacent LCp using internal synchronization between the processes. After synchronizing, it continues with the subsequent behaviour as specified. In the adjacent LCp, the received stimulus may have an effect within this LCp which in turn may include stimulating another adjacent LCp, and so on. This raises the question of the functional flow in concurrent processes. While the initial LCp has already processed all effects of an event, the composed behaviour might not be completed yet. Following the run to completion semantics (see Section 3.4.4), an LCp can not consume another event before completing the previous one. However, the run to completion semantics is limited to a single STD. As a result, an LCp is allowed to receive another event while the composed behaviour is not completed in an adjacent LCp.
Applied to our composed model, this means that a process can already receive a stimulus while another process is sending responses. This lines up with the strategy in AMP, because the described approach to always prioritize responses over stimuli (see Section 3.4.4) is limited to the scope of a single process. In our composed model, this can lead to deadlocks when processing multiple stimuli concurrently. An example sequence is depicted in Figure 3.10.

Figure 3.10: Processing stimulus before quiescence in composed model is observed



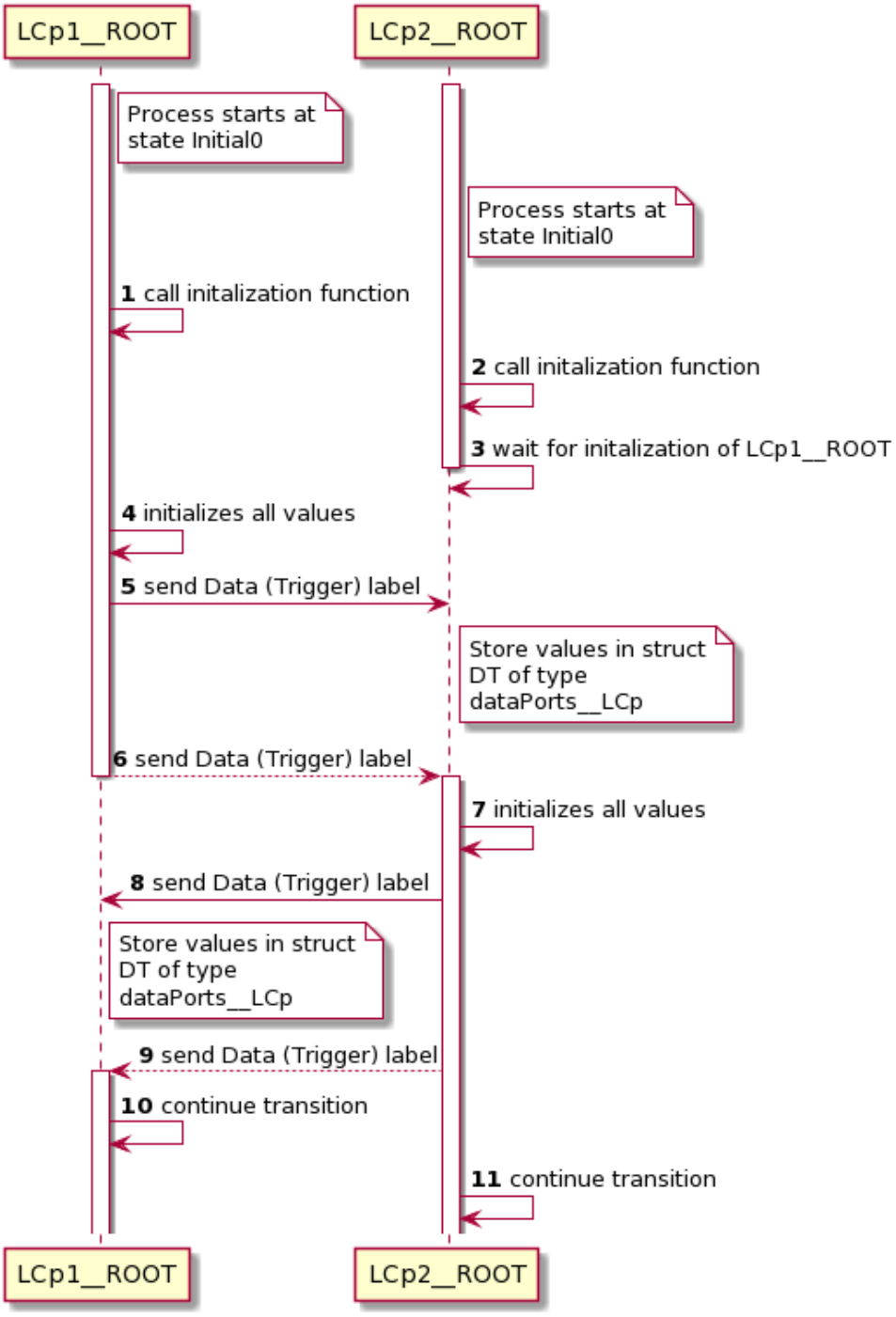
A solution for this problem are *queues* for events. After an event is received in an arbitrary LCp, the composed behaviour is processed. Any subsequent external event will not be processed immediately but queued. Only when all effects of the first event are completed, the next event from the queue is consumed. We do, however, pursue a different approach. Instead of implementing a cumbersome queuing management, we change the strategy of AMP. In the adapted strategy, all responses are consumed first, regardless of the process they are sent from. Only when all responses are consumed, a new stimulus may be sent. This guarantees that the composed behaviour is processed entirely in all processes and therefore all internal synchronizations has taken place. We describe our change in the *Experiment implementation* chapter in Section 4.5.

Finding 4: The functional flow of a composed system is not specified in the considered EULYNX SysML specifications nor in the Modelling Standard. To extend the *run to completion* semantics to the composed behaviour, we adjust the AMP strategy to observe quiescence in the composed model before processing another stimulus.

3.5.2 Initialization of a composed model

After the parallel composition, a deadlock occurs during the initialization operation. The initialization happens concurrently in all processes. Out port values are initialized and propagated to the adjacent LCps (represented by processes). This results in processes trying to synchronize with each other producing a deadlock.
To avoid this, we define an initialization order. After one process has completed the initialization, it propagates its out-port values to the adjacent LCps. After this synchronization took place, another process proceeds with its initialization while the first one waits for its completion, and so on. After all processes have completed their initialization, they synchronize again and proceed with the transition from the initial pseudostate. This sequence with two processes is depicted in Figure 3.11.

Figure 3.11: Initialization order after parallel composition



3.5.3 Timing constraints

In AML, timing constraints are only supported on labels of external channels. If the channel of a label with a timing constraint becomes internal, it is not possible to model timing constraint anymore.

> **Finding 5:** Timing constraints on former externally observable labels are no longer supported if the associated channel becomes internal.

Chapter 4

Experiment implementation

In this chapter, we describe how we implemented our experiments. First, we look at the point subsystem as the basis of our experiments. Then, we discuss the System Under Test (SUT) including the added interface to make it programmatically accessible. Next, we describe how we implemented the plug-in adapter that connects AMP to the SUT including our architecture. Last, we discuss the adjustments in the model and in the AMP strategy.

4.1 Subsystem Point

On the field-element side, the point subsystem consists of 6 LCps while 4 of those form the *generic* part which is present in every field-element. In this work we focus on the point specific LCps. One motivation to do this is that these LCps implement the most important concepts of our translation approach and are therefore suited for a meaningful case study. Another motivation is that one LCp of the generic part (*F_SCI_SCP_Sec_SR*) is not yet specified and not part of the considered EULYNX specifications.
The point specific LCps are namely:

- **F_SCI_P_SR:** Behaviour related to PDI[1] connection status.
- **F_P3_SR:** Operational behaviour of the point. Further divided into the parts *detection* and *drive*. While the first part is describing the behaviour for the detection of the point position, the latter one defines the moving behaviour.

The IBD of the logical architecture (LA) of the point subsystem can be found in Figure *IBD of logical architecture point subsystem [38]*. The architecture of our composed model follows the depicted architecture.
First, we translate the STDs of each LCp separately. In our research, we have used the baseline 3.0 (0.A). Next, we compose them into one model. To do that, we change the channel between these LCps to an internal channel to make synchronization between the processes possible. The labels between the components are internal while all others are externally observable.

4.2 System Under Test (SUT)

As System Under Test (SUT) we use a simulation of the EULYNX point subsystem. This simulator was created by SIGNON[2] using the PTC Windchill Modeler[3] (successor of PTC Integrity Modeler). The PTC model version used is 15.6.14 for the point GUI and 34 for the integrated generic LCps. A partial logical architecture of this point simulator is depicted in Figure *Partial logical architecture of point simulator [43]*. Originally, the simulation does not have an API or command line interface (CLI) that would allow for programmatically use. Tests with this simulation can be done in a manual fashion using its GUI but an interface to use the simulation without GUI is clearly beneficial. For this purpose, the source code of the simulation was made available to the previously mentioned FormaSig project. It is written in Visual Basic and contains automatically generated code from the used PTC Windchill Modeler. For the *FormaSig Deliverable D2.4* [12], Djurre van der Wal and Mark Bouwman modified the

[1] Process Data Interface, application layer of SCI (Standard Communication Interface)
[2] https://signon-group.com
[3] https://www.ptc.com/en/products/windchill/modeler

simulation for their use. While a detailed description of the modifications and the architecture can be found in the mentioned deliverable, we only want to highlight the important changes here.
Mainly, they modified the class *SubS_P_SRForm* (hereinafter *Form*) and the subclasses of *GUIElem*. By configuration, the GUI is not displayed anymore but the GUI elements are still instantiated. The simulation is done in *simulation steps* where each of these steps represent 0.1 seconds. The *Form* observes changes of output GUI elements which occur when the simulation is advanced in time step by step [12]. With an added hook to all GUI output elements, we detect value changes and intercept them. In this manner we can collect all value changes of the GUI output elements and interpret them as a response of the simulation.
The stimulation of the simulation works in a similar manner. Whenever we want to send a stimulus, we change the value of the corresponding GUI input element. The appropriate input element is stored in a lookup table and identified by the label and the channel. Additionally to these changes of the simulation, Djurre van der Wal and Mark Bouwman added an adapter to connect to their model. They also added a new *Main* class as entry point of the simulation which manages the *Form* instance and more.
We use their work as a starting point for implementing a Command Line Interface (CLI) in the new class *CLIAdapter.vb*. The newly added CLI adapter localizes the GUI elements, delivers stimuli from the command line to the input fields and captures eventual responses from output fields which are passed on to the CLI. It also contains a mapping between the labels and the corresponding in- and output fields of the GUI in form of lookup tables. We also extended the *Main* class. It still serves as the entry point of the simulation but additionally handles the initialization and capturing of all responses (via CLI adapter). It prints the responses in a predefined format so we can parse them subsequently in our adapter that manages the communication between our SUT and AMP.
We created an adapter to use our SUT from AMP (see Section 4.3). Our adapter is based on the plug-in adapter from Axini as introduced in Section 2.2.2. Plug-in adapters were only tested for the use in a Linux environment. Our SUT however uses the .NET Framework 3.5 which is Windows only. First, we attempted to migrate the project to .NET (previously named .NET Core) that is supported by Linux. This has proved infeasible because the project relies of numerous assemblies depending on .NET Framework 3.5. Our attempt resulted in build errors. Additionally, a platform compatibility analysis of the project yielded several unsupported APIs. We therefore argue that even without the dependencies on these assemblies a migration would have most likely led to undesirable effects during runtime. Second, we attempted to run the simulation directly on Linux using Wine[4]. Although the initial start-up worked, the program crashed reproduceable after a few seconds. We concluded that the SUT can only be run under Windows and we need to bridge the platform gap using the plug-in adapter.

4.2.1 Input / output format

On start-up, the simulation initializes the test environment and resets it afterwards. The process of these steps is printed out on the command line and completed with the key words `SIMULATION READY`. When we parse this line, we know that the simulation is ready for accepting stimuli. As described in Section 4.2.2, some responses are visible immediately after start-up. Since they are part of the specification as well (e.g., initialization operations), we decided to capture them and include them in our model accordingly. Therefore, we can observe responses immediately after `SIMULATION READY`.

The **format for stimuli** is
`receive! <channel> <label> !Value_<datatype>(<value>)`

- Supported data types are `Bool`, `Int`, and `String`.
- Trigger stimuli are theoretically of type *PulsedIn*. Since the *PulsedIn* data type is a refined Boolean type (see Section *Trigger in/out port*), we substitute it with the data type `Bool`. Thus, Trigger stimuli are always of type `Bool` with the value *true*.
- To conduct the mapping to the input element of the GUI, we are using a lookup table which contains the corresponding input element identified by the combination of channel and label. If a stimulus is sent that is not contained in this lookup table, we throw an exception.

[4]https://www.winehq.org

The **format of responses** is
`!send! <channel> <label> <datatype> <value>`

- Supported data types are `Bool`, `Int`, and `String`.
- Following the explanation of Trigger stimuli, Trigger responses are always of type `Bool` with the value *true*.
- The CLI adapter looks up the changed output element of the GUI in a lookup table. We store additional information for each relevant output element in this table. Based on the element's identification (which is a name derived from the GUI of the simulation), we retrieve information about the channel, label name and data type. If an output element is not contained in the lookup table, we print `??` for the channel and data type and substitute the label with the output elements name. We do not evaluate these outputs eventually.

After a stimulus is sent to the simulation, we advance the simulation in time and capture all responses. When we detect that the stimulation has stabilized (which is assumed to occur within a predefined finite amount of simulation steps), we print the key word `!quiescence`. Whenever we parse this line in the adapter, we can assume that all responses from the last sent stimulus are captured and the simulation is ready for the next stimulus.
The stimulation is terminated with entering `exit`.

4.2.2 Initialization and Bootstrapping

Before any stimulus is sent, the *Form* and the CLI adapter perform a reset. Afterwards, we perform a basic check if we can localize all defined input elements in the simulation. When this initialization is completed, the simulation already yields responses. These responses emerge from various LCps and their initialization operations. We therefore advance the simulation immediately after initializing and collect all responses.
Because we only focus on the point subsystem specific parts, our model is not entirely in-sync with the simulation at this point. The point specific parts assume that the underlying field element subsystem of the generic part has priorly booted. To bring up the field element subsystem we need to send stimuli to the LCp *F_EST_EfeS*. This LCp defines the essential states of the EULYNX field element subsystem [37]. It is an abstract definition where we need to look at the refined LCp *F_EST_EfeS_SR* for our use. Important to note is that while we do not include the generic LCps in our research, we do evaluate the responses of the point specific LCps during this start-up phase of the generic part. These responses are collected and part of our model.
To bring up the subsystem, we send following Trigger in stimuli in the order listed:

1. **T1_Power_On_Detected:** The port *T1_Power_On_Detected* indicates that the voltage has reached the required range for operation [37].
2. **T4_Booted:** The port *T4_Booted* indicates that the booting process was completed successfully [37].

To bootstrap the subsystem, we implemented a dedicated bootstrap function. We call this function immediately after collecting the responses from the initialization.
As described in Section 4.4.1, we furthermore configure our SUT according to the infrastructure manager (IM). We set this configuration also during the bootstrap process, as the first stimulus to be more precise. This way, the following behaviour of subsequent stimuli is already according to the IM configuration.
In the complete bootstrap function, we send the following stimuli in the given order:

```
receive!  env D37_Con_008400 !Value_Bool(true)
receive!  F_EST_EfeS_SR T1_Power_On_Detected !Value_Bool(true)
receive!  F_EST_EfeS_SR T4_Booted !Value_Bool(true)
```

After each stimulus we advance the simulation and collect all responses. We print `!quiescence` only after printing all responses from the initialization and the bootstrapping process.

4.2.3 Data and Data Trigger in/out ports

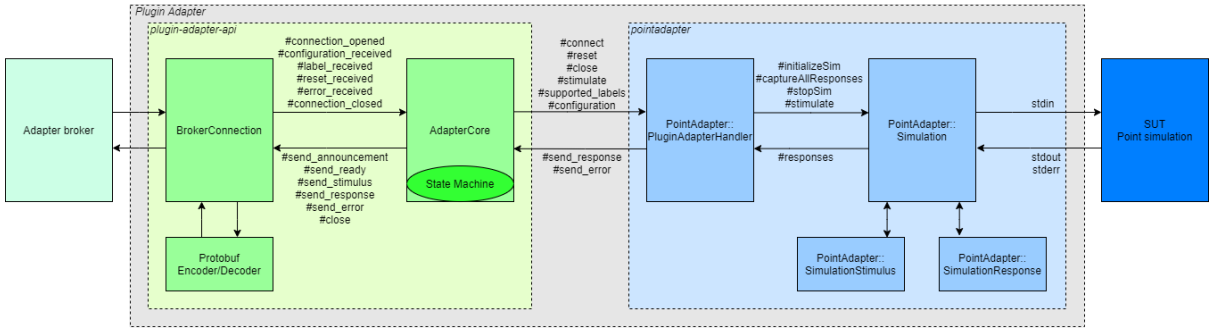
The introduced approach to collect the responses from the SUT has its limitations. As mentioned, the SUT still relies on its GUI that instantiates all input and output fields. For the responses, we collect all changes of the output field values of the GUI. This works reliable for Trigger out ports because they always change their value from *false* to *true*. When it comes to Data and Data Trigger out ports, this does not always work. If the value in the output field of the GUI does not change but we expect a

response of a Data or Data Trigger out port, we do not get a response from our SUT. For example, when entering the first state, a certain value is assigned to a Data (Trigger) out port. We expect this as a response of our SUT. However, if this value was already set during the initialization operation, we do not get a response because the value of the corresponding output field in the GUI does not change. In these situations, we added an additional constraint to our model that only expects a response if the value of the Data or Data Trigger out port actually changes (see changes c4, c5, and c13 in Section *Results qualitative experiment*). We can detect this because we have cached the previous values in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*).
As discussed in Section *Data in/out port*, Data in/out ports have the semantics of a continuous flow of data. We abstract this behaviour and treat them as stimulus / responses with a parameter. Likewise, our SUT does not handle these ports as a continuous flow of data. It only expects data changes as stimuli and provides data changes as responses. Therefore, we do not need to convert these semantics in our adapter.

4.3 Plug-in adapter

As described in Section 2.2.2, we implemented a plug-in adapter that connects AMP to the SUT. Although the plug-in adapter can be implemented in many programming languages, it must support the *plug-in adapter protocol* [28]. Since the *plug-in adapter protocol* relies on Google's Protocol Buffers which is not supported for the .NET Framework 3.5, we could not directly implement the adapter in our simulation. We therefore used the plug-in adapter provided by Axini and ported it to a Windows environment. The plug-in adapter is implemented in Ruby which is supposed to be platform independent. However, some gems[5] do have platform dependencies and need to meet additional prerequisite on Windows (e.g., libgit2/rugged). Subsequently, we extended the plug-in adapter for our use. We only want to briefly spotlight certain aspects hereinafter. The architecture is depicted in Figure 4.1.

Figure 4.1: Plug-in adapter architecture



The simulation is represented in the adapter as type `Simulation`. It is used by the plug-in adapter handler and provides all simulation related operations such as initialization, stimulation, and the capturing of responses. The executable of the simulation is spawned and its *stdin*, *stdout*, and *stderr* captured. We communicate through these pipes with the aforementioned CLI of the simulation (see Section 4.2).
With the stimulate method, the stimulus produced by AMP is converted into type `SimulationStimulus`. This type provides methods for *stimulus mocking* and value handling. Subsequently, the stimulus is written onto the CLI of the simulation using the pipe *stdin*. After sending a stimulus, the plug-in adapter handler captures all responses. We read all lines from the CLI of the simulation using the pipe *stdout* until we reach `!quiescence`. All read lines are parsed into type `SimulationResponse`. This type provides methods for *response mocking*, channel filtering, and value handling. Only responses from channels within the scope of our model are delivered to the plug-in adapter handler.
Because the SUT has GUI input field elements in the background where we entered values during the execution, we dispose the entire simulation and spawn a new process between each independent test cases.

[5]Ruby libraries, see https://rubygems.org

4.3.1 Stimulus mocking

For the selected LCp of our research, the simulation does not provide an input field for all external stimuli. As a result, we are not able to provide certain stimuli to the SUT. As a work around, we *mock* these stimuli to other stimuli where we have a corresponding input field. We hypothesize that these stimuli have the same effect in the system parts we are interested in. We have identified the following stimulus mocking:

Original LCp	*Original label*	*Mocked LCp*	*Mocked label*
F_SCI_P_SR	DT1_Move_Point_Target	S_SCI_P_SR	DT10_Move_Point
F_SCI_P_SR	T1_Cd_Move_Point	S_SCI_P_SR	T10_Move_Point
F_SCI_P_SR	T18_Start_Status_Report	F_SCI_EfeS_Sec_SR	T5_SCP_Connection_Established

Table 4.1: Stimulus mocking

The first two stimuli are part of the LCp *S_SCI_P_SR*. Consulting the STD of this LCp it becomes clear that once reached its state *PDI_CONNECTION_ESTABLISHED*, the mocked stimuli are just forwarded to the LCp *F_SCI_P_SR*. Therefore, we can use the proposed mocking showed above.
The mocking of the last stimulus is more complicated. According to the logical architecture (LA) of the point subsystem, the stimulus *T18_Start_Status_Report* of the LCp *F_SCI_P_SR* corresponds to the response *T6_Start_Status_Report* of the LCp *F_SCI_EfeS_Sec_SR*. Consulting the STD of this LCp we see that this response is sent during the establishing process of the PDI connection. After our initialization and bootstrap sequence, the PDI connection state is *CLOSED READY*. With the stimulus *T5_SCP_Connection_Established* we establish the PDI connection in this LCp and end up in the PDI connection state *ESTABLISHED*. During this establishment, our response *T6_Start_Status_Report* is sent. When we receive the corresponding stimulus *T18_Start_Status_Report* in the LCp *F_SCI_P_SR*, we also proceed to the PDI connection established state. Therefore, we argue that we can mock this stimulus as shown above.

4.3.2 Response mocking

Similar to the stimulus mocking, we need to mock certain responses. These are responses, where we are missing an output field of the GUI for the external response. We hold the same hypothesis that these mocked responses have the same semantics in the system parts we are interested in. We have identified the following response mocking:

Original LCp	*Original label*	*Mocked LCp*	*Mocked label*
F_SCI_P_SR	T3_Msg_Timeout	S_SCI_P_SR	T30_Timeout

Table 4.2: Response mocking

This response is part of the LCp *S_SCI_P_SR*. Consulting the STD of this LCp it becomes clear that once reached its state *PDI_CONNECTION_ESTABLISHED*, the mocked response is just forwarded when receiving the corresponding stimulus *T3_Msg_Timeout* from LCp *F_SCI_P_SR*. Therefore, we can use the proposed mocking shown above.

4.4 Model adjustments

We translated the EULYNX SysML specifications as described in Section 3.4. We used the Axini Modeling Platform in version 3.0 to do that. For our experiments, we changed certain aspects of our model. The motivation to do that is twofold. The first reason is to provide more realistic testing scenarios. For example, while some Data in ports might change their values during execution time, others do not. It is not realistic if these values change during execution and we have observed undesired effects in our simulation when doing so, based on the time the simulation evaluates these ports. The second reason

is that we want to provide the tester with configuration options to adjust the behaviour of the model accordingly.
We implemented a configuration file `default_config` for our model. In this file, various configuration flags are available. These influence the model and the outcome of the test cases significantly. We will discuss the implemented changes and configuration options in the following.

4.4.1 Infrastructure Manager configuration

The point subsystem can be configured for a specific infrastructure manager (IM). Based on this configuration, the behaviour is different. For example, this configuration determines if the point is trailable[6], multiple PMs[7] are supported, and redrive[8] is enabled. The Data in ports *D3x* of the LCp *F_P3_SR* provide the configuration values for which infrastructure manager the point subsystem is configured [38]. Each configuration value is designed as an independent port. We therefore modelled them as different stimuli as described in Section *Data in port*. We receive these stimuli whenever values change. We then cache the received values for a later evaluation to overcome the issue with missing Data in ports with continuous data flow. AMP sends those stimuli at random times with randomized values within the range of the defined constraint. As a result, none or multiple IM configurations can be set at the same time and they change constantly during execution.
This does not reflect the usage of the real system. Although not explicitly specified, we assume that this configuration does not change during execution time. We furthermore assume that exactly one configuration is enabled at all times which is not be changed during system execution.
This means we do not want to receive a stimulus with a different configuration during execution time. Thus, we replaced the stimuli for these Data in ports with configuration flags in our configuration file. We set the values in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*) according to the configured values during the initialization operation of the LCp *F_P3_SR*. This allows us to set the value of exactly the subsystem configuration that should be tested. We implemented the following configuration flags which can be set to either *true* or *false*.

Configuration flag	*Infrastructure manager*
D30_Con_007000	Network Rail
D32_Con_007600	Bane NOR
D33_Con_007900	SZ
D34_Con_008000	DB Netz
D35_Con_008200	CFL
D36_Con_008300	RFI
D37_Con_008400	Pro Rail
D38_Con_008500	SBB

Table 4.3: Infrastructure manager configuration flags

Exactly one point subsystem configuration flag should be set to *true*. With this configuration flag, the model is configured for the specific point subsystem implementation. Correspondingly, the SUT must be configured in the same way. Therefore, according to the set flag, we configure the SUT during the bootstrapping process (see Section 4.2.2). In our experiment, we used the ProRail implementation.
The change is marked in our model with the identifier *c10*.

[6] When a train runs or rolls over a point that is switched to the opposite direction at that time. Some implementations are not *trailable* from a systems perspective, they only manage the endpoint detection.
[7] Point Machines
[8] Switching the point back while it moves into the other direction.

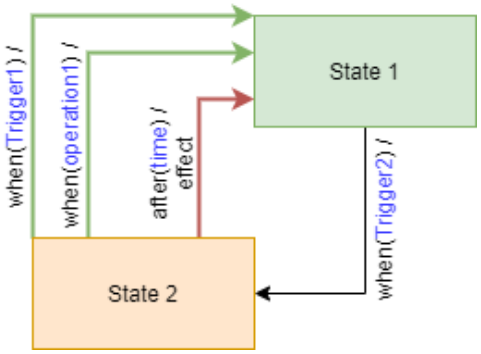
4.4.2 Timing constraints

According to the explanation of the step-based simulation in Section 4.2 it becomes obvious that the simulation is not in real-time. In fact, it differs by a large margin from the actual time. This is due to the fact that responses are observed while advancing the simulation in time. If no response is observed after 10 simulation steps, we assume quiescence (see code snipped below). As a result, after many stimuli without response, the simulation is advanced in time further than the actual time has progressed. On the other hand, if no stimulus is sent at all, the simulation time stands still. Somewhat simplified, AMP executes the model in real-time. As a result, the AMP-time and the simulation time are not in-sync.

```
1  For i As Integer = 1 To 10
2    If CLIAdapter.DequeueOutput(Output) Then
3      Return True 'Return output if a response was observed
4    End If
5    SimMaster.DoStep() 'Do next simulator step (.1 seconds)
6  Next
7  Return CLIAdapter.DequeueOutput(Output)
```

In the EULYNX specifications, timeout transitions are often used. An example is depicted in Figure 4.2. After activating *State 2*, the green transitions can be activated by the specified events. Whenever the amount of *time* has passed, the red transition is activated directly based on a time event.
As explained in Section *Time event*, we set a `timeout` variable to `timeout = timestamp + offset` when entering *State 2*. We then model timing constraints to all outgoing transitions so that only the green ones are allowed to be activated before `timeout` and the red one only after. This forces the model to advance on the timeout transition once the specified offset has elapsed.
This only works if the model and the SUT are both advancing the time in the same manner (e.g., using real-time). But since the SUT generally advances time faster than AMP, the SUT activates the timeout transition too early. The test case fails due to the constraint on that transition in our model. AMP can not advance on this transition yet.
To tackle this issue, we introduce a timeout configuration and change the timeout transitions. These changes are marked in our model with the identifier *c18*.

Figure 4.2: Timeout transitions



Timeout configuration

In the considered specifications of this research, we have exactly two timeout transitions. Both timeout transitions are activated after the time of the Data in port *D4_Con_tmax_Point_Operation* of the LCp *F_P3_SR*. We therefore modelled this Data in port as stimulus as described in Section *Data in port*. We receive this stimulus whenever the value changes. We then cache the received value for a later evaluation to overcome the issue with missing Data in ports with continuous data flow. AMP sends this stimulus at random times with randomized values within the range of the defined constraint. As a result, the timeout value changes at various times during execution.
The port *D4_Con_tmax_Point_Operation* refines the time value for *Con_tmax_Point_Operation* and following standardized time value are configured [38]:

- 12 seconds for IM 008000, 007600, and 007000
- 10 seconds for IM 008200
- 7 seconds for IM 007000, additional value selectable

Although not specified, we argue that the *Con_tmax_Point_Operation* value is set according to the IM and does not change during execution time. This means we do not want to receive a stimulus with a different timeout-time during execution. Thus, we replaced the stimulus for this Data in port with a configuration flag in our configuration file.
We set the value in our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*) according to the configured value during the initialization operation of the LCp *F_P3_SR*. We implemented the following configuration flag which can be set to any positive integer.

Configuration flag	*Value*
default_tmax_point_operation	Timeout-time in seconds

Table 4.4: *Con_tmax_Point_Operation* configuration flag

Additionally, we set the timeout of all model parts to this value plus an additional 15 second margin. 15 seconds has proven to be a good timeout margin (especially if the configuration is set to 0 seconds) which allows for all responses to arrive in a timely manner, particularly during the start-up of the SUT. We also need to configure our SUT accordingly. The GUI of the SUT does have a default value of 12000 milliseconds. We need to change this value according to the configuration. To do that, we add the receiving of the stimulus *D4_Con_tmax_Point_Operation* containing the configured value after the model has reached firstly a stable state. This is the case when it reaches the state *NO_END_POSITION*. We only do that when entering the state for the first time after the start-up has completed. With an additional flag we can skip this step in all consequent activations of this state.

Timeout transitions

Although the timeout-time can be set to a specific value, the aforementioned issue with the time behaviour of the SUT remains. This means, that our constraints on the transitions are mistimed which leads to numerous failed test cases. To overcome this issue, we follow a different approach. Whenever in a state with outgoing timeout transitions, we allow the timeout behaviour to arise optionally at any time. We always accept the timeout behaviour or the behaviour of the other transitions. As a downside, we can not test correct time behaviour of our SUT.
To implement this change, we remove all time related constraints on all transitions. We then wrap the timeout transition in an *optionally* construct of AML. As a result, AMP accepts the behaviour of this timeout transition but does not require it. Whenever a state with a timeout transition is active, AMP has now the option to receive a response from the timeout transition or from another transition. AMP therefore awaits any possible response from the SUT. This means that during our test case execution, AMP waits the maximal allowed time of a response (specified timeout time of our model).
This slows down our test case execution significantly. As our SUT does not advance in time when no new stimulus is received, waiting for the timeout response is useless. To improve test performance, we add timeout constraints to the optional transition. To be more specific, we add `timeout: 5.0` to all potential first responses of the *optionally* construct. 5 seconds has proven to be a good value so that a potential timeout of the SUT is still within this limit.

Value conversion

If the values of Data (Trigger) in ports are handled differently by our model than by the SUT, we need to convert the received values accordingly. Boolean values are not affected since the semantics is equivalent. Furthermore, String values should also not be affected because we preserve all specified String values in our model. Integer values however are subject to conversion. We have identified the following needed conversion:

- Stimulus *D4_Con_tmax_Point_Operation* of LCp *F_P3_SR*

While our model uses seconds, the SUT uses milliseconds. We convert the value directly in the constraint of the label, while assigning the converted value to our struct `DT` of type `dataPorts__LCp` (see Section *Data in/out port*) for all evaluations. This has the advantage that the parameter remains unchanged, and no conversions are needed in the adapter.

4.5 AMP strategy adjustments

As discussed in Section 3.4.4, AMP only processes a new stimulus if a location in the model is quiescent [28]. This lines up with the SysML *run to completion* semantics: State machines always run to completion, which means that they are not able to consume another event until the state machine has completed the processing of the current event.
However, in AMP this is only valid for each individual STS, represented by an independent process. If concurrent processes are executed, a new stimulus might be sent before all responses from another process are consumed. This violates the run to completion semantics.
Therefore, we adjust the strategy of the *sts-engine* in AMP. We change the implementation of AMP accordingly. For our experiments, AMP awaits all responses first before sending a new stimulus - regardless the process of the responses and stimulus.

Chapter 5

Results

In this chapter, we present the results of our Model-Based Testing experiments with the point subsystem. Additionally, we discuss the model configurations used for the qualitative experiment and list the observed non-conforming behaviour in the quantitative experiment. We round the chapter off by stating the differences compared to the approach with mCRL2.

5.1 Model-Based Testing of subsystem Point

Here, we look at the MBT results of the point subsystem. We will use our formal AML model that we retrieved by translating the EULYNX SysML specifications with the introduced translation patterns in Section 3.4. We then test our model in our qualitative experiment against our point simulation as described in Section 4. Next, we look at the failed test cases. We analyse the reason why they fail and categorize them. We reflect the reason back to our translation pattern to see if we need to implement changes in the translation. With an improved model we then start our quantitative experiment where we empirically validate our translation and test our SUT.

5.1.1 Goal

Our first goal is to validate our translation patterns by running our model against our SUT in our qualitative experiment. If we hypothesize that our translation and our SUT are without errors, all our test cases succeed. Since we can not make this assumption of correctness, we need to analyse each failed test case. We are interested in the reason why the SUT is not conform with our model. A failed test case yields either an error in our model or in the SUT.
We are following an incremental approach in our test case analysis. After determining the reason for the failure of a test case, we categorize it and propose a corrective action based on that. The particular corrective action depends on the error location. If the model is faulty, we reflect back to our translation patterns and implement the needed changes. If the SUT is faulty, we conclude with a found bug or a limitation of our simulation.
If we have found a bug or limitation in the SUT, all our subsequent test runs will fail at the exact same transition. This prevents further test steps as the test case will stop when failing. To be able to test successive transitions, we add a configuration flag to the model where possible. Is this flag set, the model accepts the faulty behaviour of the SUT. As a result, the test case does not fail. We call these known bugs of our SUT *flagged bugs*, because we acknowledge the presence of the faulty behaviour with a flag that can be set in the configuration.
With this iterative process we do both, improving our translation patterns and finding non-conformances in our SUT.
The second goal is to achieve a satisfying test coverage by running our improved model against the SUT in our quantitative experiment. We run numerous test cases with an increased amount of test steps each. We validate our model empirically by hypothesise that all test cases succeed. If they fail, it must be due to a non-conforming behaviour of our SUT.

5.1.2 Qualitative experiment

As stated above, in our qualitative experiment we execute our test cases and analyse the results. We give special attention to the failed test cases and propose a corrective action for them. We then proceed to implement changes in the model or add a configuration flag. This aims to circumvent the reason why this specific test case failed. We will list our results from the qualitative experiment below followed by an explanation of the added configuration flags.

Results qualitative experiment

We will list here the results of our qualitative experiment consisting of the analysis of the failed test cases and proposed corrective actions. The *change identifiers (cID)* are also included in the model to make the changes easily traceable.

cID	*c1 / c7 / c8 / c9 / c15*
Category	Response order
Reason	The responses from the SUT are sent in a different order than specified: • *c1*: Operation *cOp1_Init* of the LCp *F_P3_SR* • *c7*: The transition effect in the LCp *F_SCI_P_SR* from the state *REPORT_STATUS* to *STATUS_REPORTED* • *c8*: Entry-behaviour in the LCp *F_P3_SR* of the states *ALL_LEFT* and *ALL_RIGHT* • *c9*: The transition effect in the LCp *F_SCI_P_SR* of the internal transition in the state *PDI_CONNECTION_ESTABLISHED* with the change event of Trigger in port *T20_Point_Position* • *c15*: Entry-behaviour in the LCp *F_P3_SR* of the states *MOVING_LEFT* and *MOVING_RIGHT*
Corrective action	We changed the order of the responses in the corresponding state or function to fit the response order of the SUT.

cID	*c2 / c3*
Category	Deadlock
Reason	Deadlock during initialization of LCps *F_P3_SR* and *F_SCI_P_SR*. The two adjacent LCps each initialize their Data (Trigger) out ports during their initialization operation. Some Data (Trigger) out ports are Data (Trigger) in ports for the adjacent LCp. After the initialization of the out ports, the LCp propagates the values to the adjacent LCp. This LCp uses the received values to initialize its corresponding in ports. This is done using internal synchronization. As discussed in Section 3.5, this results in a deadlock between the two (or more) LCps because each process is awaiting the synchronization with the other.
Corrective action	We defined an initialization order (see Figure 3.11) although the initialization operations are specified as entirely concurrent actions. Where needed, we created an additional label for the needed synchronization. • *DT20_Point_Position* of LCp *F_P3_SR* to *DT20_Point_Position* of *F_SCI_P_SR* • *DT10_Move_Target* of *F_SCI_P_SR* to *DT1_Move_Target* of *F_P3_SR*

cID	*c4 / c5 / c13*
Category	Limitation SUT
Reason	Only data changes are detected as response of the SUT (see also Section 4.2.3). If a value remains the same, the corresponding output field does not change its value and therefore we do not detect a response.

Corrective action We check if the new value is equivalent to our cached one. If that is the case, we do not expect a response from the SUT. If the value changes, a response is required in any case.

- *c4*: *D25_Redrive* in *F_P3_SR*
- *c5* (3x): *D5_Drive_State* in *F_P3_SR*
- *c13* (2x): *D6_Detection_State* in *F_P3_SR*

---

cID *c6*

Category Specification error

Reason Wrong data type for Data port *D6_Detection_State*. In the LCp *F_P3_SR*, the Data out port *D6_Detection_State* is of SySim value type String. During the initialization operation, the Boolean value *false* is assigned which is not valid.

Corrective action The SUT assigns the String "False". We changed the model accordingly.

---

cID *c11*

Category Limitation SUT

Reason Not initialized default values in the GUI of the SUT.
As explained, the SUT still relies on a GUI in the background. This GUI has drop-down input fields for some values that represent Data (Trigger) ports. According to the specification, these values are not initialized. In the GUI however, a value is already selected on start up. Furthermore, the drop-down fields do not allow selecting an empty value.

Corrective action To synchronize our model with the simulation, we must initialize these Data (Trigger) ports according to the GUI. We do this in the initialization function of each process. The values are:

- *F_P3_SR*:
  - `D21_PM1_Position = 'NO_END_POSITION'`
  - `D22_PM2_Position = 'NO_END_POSITION'`
  - `D13_PM2_Activation = 'INACTIVE'`
- *F_SCI_P_SR*:
  - `DT1_Move_Point_Target = 'LEFT'`

Note, that the default value 120000 of `D4_Con_tmax_Point_Operation` in the LCp *F_P3_SR* is not set because it is part of change *c18* (see Section 4.4.2).

---

cID *c14*

Category Specification error

Reason Wrong value for Trigger out port *T5_Info_End_Position_Arrived*. In the orthogonal region *DETECTION* of the LCp *F_P3_SR*, the value *false* is assigned to the Trigger out port in the exit-behaviour of the states *MOVING_LEFT*, *NO_END_POSITION*, and *MOVING_RIGHT*. As discussed in Section *Trigger out port*, it is only valid to assign *true* to a Trigger out port.

Corrective action We ignore this effect. As it is the only specified effect in the exit-behaviour, we can omit the entire exit-behaviour. The SUT behaves accordingly.

---

cID *c17*

Category SUT error

Reason	According to the Modelling Standard [30], exit-behaviours are always the first effect executed when an external transition is activated. Afterwards, the transition effect is executed and finally the entry-behaviour of the newly activated state. Our SUT however executes almost always the entry-behaviour of the newly activated state before the exit-behaviour of the previously active state. The only exception is the transition in the LCp *F_P3_SR* from *MOVING_RIGHT* to *MOVING_LEFT* where the behaviours are executed in the correct order.
Corrective action	Instead of executing the exit-behaviour, we set a helper variable to mark the deferred execution of the exit-behaviour. After activation of the new state and after processing its entry-behaviour, we check if a helper variable marks a deferred exit-behaviour. If this is the case, we execute the corresponding exit-behaviour.

Model configuration

To overcome aforementioned problems, we implement configuration flags into our model. With these flags, certain behaviour of the model can be either enabled or disabled. Each flag can be set to *true* or *false*. The configuration of the model allows us to circumvent known issues of the SUT and explore more states of the STS. In the following, we provide an overview over the implemented configuration flags.

Configuration flag	*cID*	*Description*
set_SUT_default_values	*c11*	With this flag, we initialize the values according to the GUI of the SUT (except `D4_Con_tmax_Point_Operation` in the LCp *F_P3_SR*). If this flag is not set, we only initialize the values according to the specifications. In that case, some values remain uninitialized and are not according to the SUT which leads to the failure of test cases.
response_only_on_data_change	*c4 / c5 / c13*	If this flag is set, we do not expect a response from the SUT if the value of Data (Trigger) in ports do not change. If it is not set, we always expect a response according to the specifications. Since our SUT does not provide that, this leads to failed test cases.
execute_exit_after_entry	*c17*	With this flag, the order of the exit- and entry-behaviour during a transition is swapped. If this flag is set, the entry-behaviour of the newly activated state is executed before the exit-behaviour of the deactivated state. This affects all transitions except the one in the LCp *F_P3_SR* from *MOVING_RIGHT* to *MOVING_LEFT* (the SUT behaves correctly on this transition).

Table 5.2: Model configuration flags

The additional configuration flags *D3x_Con_00xx00* and *default_tmax_point_operation* are discussed in Section 4.4.

5.1.3 Quantitative experiment

From our qualitative experiment, we derived an improved configurable model that allows us to avoid *flagged bugs*. We now take this model to run our quantitative experiment. We set all configuration flags introduced in *Model configuration* to *true*. The goal of this experiment is to achieve a high transition coverage. We argue that with a high transition coverage we gain empirical evidence to support or hypothesis for the proposed translation pattern. We need to analyse failed test cases to determine if we found an issue in the SUT or if our hypothesis is refuted.

Test strategy

AMP supports different test strategies. All strategies choose an action based on the transitions enabled in the current state vectors [28]. The action can be either sending a new stimulus, observing responses, or stopping the test case. Different test strategies result in different test cases and different coverage. Following strategies are supported [28]:

- **Random:** Chooses a random transition from all enabled transitions. This strategy is fast but achieves high coverage only slowly over many test cases. Some parts of the model might remain unexplored.
- **LocalTransitionCoverage:** Chooses a random transition that is not covered yet. When all transitions are covered, it chooses at random. The transition coverage of the currently active state is therefore increased. This strategy is also fast and achieves coverage more quickly than the *Random* strategy. Likewise, some parts of the model might remain unexplored.
- **GlobalTransitionCoverage:** Chooses a random transition that is not covered yet. However, when all transitions in the current state are covered, it uses a *breadth first search* algorithm to find a path to uncovered transitions. This strategy attempt to increase the transition coverage over the entire model and is able to cover hard to reach parts of the model. As a downside, it takes more time to calculate the optimal action.

As stated above, our goal is to achieve a high transition coverage. Comparing these strategies, it becomes evident that *GlobalTransitionCoverage* is the most suitable test strategy for our purpose. This lines up with the *Effective AML* guide [28] that states that for complex models, a *smarter* strategy like *GlobalTransitionCoverage* is often required to achieve decent coverage at all.

Results quantitative experiment

To retrieve the results of our qualitative experiment, we use the *Test set configuration* as shown in Figure C.1. We have increased the *Max Number Of Steps* to 100 and the *Max Number Of Test Cases* to 10. With these values we reach a high transition coverage that does not further increase within the last test cases. We therefore argue that with higher values the test would need substantially more time without covering more transitions.

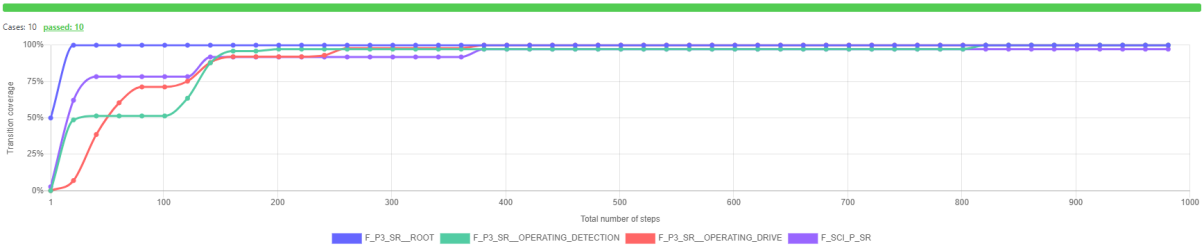
Transition coverage To retrieve the actual transition coverage, we removed arguably unreachable transitions depending on adjacent LCps which are out of scope of this experiment (see Section 4.1). The results are shown in Figure 5.1. We reached the following transition coverage:

Process	*Transition coverage*
F_P3_SR__ROOT	100%
F_P3_SR__OPERATING_DETECTION	100%
F_P3_SR__OPERATING_DRIVE	100%
F_SCI_P_SR	97.3%

Table 5.3: Transition coverage of each process

The reason why the process *F_SCI_P_SR* does not have a complete transition coverage is that the receiving of Data Trigger in stimuli *DT10_Move_Target* is possible in every state (outsourced to an AML macro) but in one state during start-up it is not received.

Figure 5.1: Transition coverage of quantitative experiment



id	started	steps	channel	label	direction	verdict	tags
1	19:14:13.096	100	SCI_P	?DT1_Move_Point_Target	→	passed	
2	19:18:55.460	100	P3	?D21_PM1_Position	→	passed	
3	19:22:48.016	100	SCI_P	?T1_Cd_Move_Point	→	passed	
4	19:23:11.216	100	P3	!D11_Move_Right	←	passed	
5	19:25:48.526	100	SCI_P	?DT1_Move_Point_Target	→	passed	
6	19:33:10.989	100	P3	?D22_PM2_Position	→	passed	
7	19:35:35.327	100	SCI_P	?DT1_Move_Point_Target	→	passed	
8	19:38:52.479	100	P3	?D22_PM2_Position	→	passed	
9	19:41:34.913	100	SCI_P	?DT1_Move_Point_Target	→	passed	
10	19:42:52.153	100	SCI_P	?T1_Cd_Move_Point	→	passed	

Non-conformance of SUT For our conformance testing of the SUT we used different values for the configuration flag *default_tmax_point_operation*. This allows us to test the timeout transition from the states *MOVING_LEFT* and *MOVING_RIGHT* to *STOPPED* in particular or to exclude it.

- **0 seconds:** Whenever the simulation activates a moving-state, it produces a timeout with the next received stimulus. This configuration guarantees us to cover the timeout transition.
- **12 seconds:** This corresponds to a realistic configuration value (see Section *Timeout configuration*). When in a moving-state, the simulation sometimes produces timeouts. In other test cases, a stimulus for end position is sent before a timeout is produced. This allows to cover both, the timeout transition, and the transitions for end position.
- **50 seconds:** It has shown that 50 seconds is an adequate value so that a stimulus for end position is sent before the simulation produces a timeout. With this configuration, the end position transitions are explicitly covered while not covering the timeout transitions.

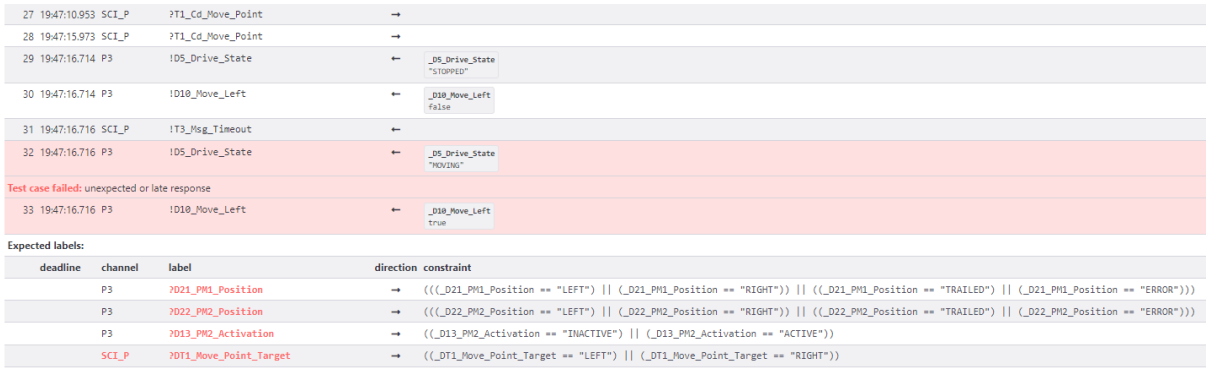
We have discovered following non-conforming behaviour of our SUT.

Moving without moving command When in one of the moving-states, the simulation receives multiple stimuli and advances in time. As the last stimulus, it receives *T1_Cd_Move_Point* at the LCp *F_SCI_P_SR*, which arrives in the LCp *F_P3_SR* as *T1_Move*. Afterwards, we observe the timeout behaviour. The timeout behaviour is characterized by the sequence of the responses

1. D5_Drive_State with `_D5_Drive_State='STOPPED'`
2. D10_Move_Left with `_D10_Move_Left=false`
3. T3_Msg_Timeout

The orthogonal region *DRIVE* of the LCp *F_P3_SR* transitions to the state *STOPPED*. Without a new stimulus *T1_Move*, it is not possible to go back to a moving-state (excluding redrive behaviour). However, we can observe this behaviour at our SUT. It seems that it reacts to the last received stimulus *T1_Cd_Move_Point* although we observed the timeout behaviour afterwards.

Figure 5.2: Non-conforming behaviour: Moving without moving command

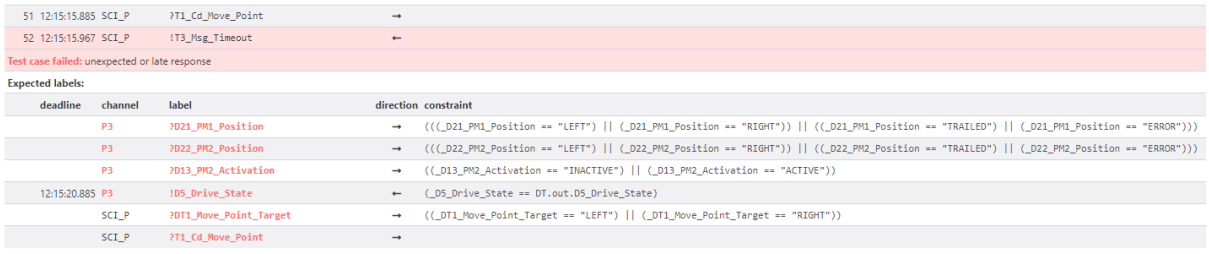


Timeout during transition While in a moving-state, we receive the stimulus *DT1_Move_Point_Target* at the LCp *F_SCI_P_SR*, which arrives in the LCp *F_P3_SR* as *DT1_Move_Target*. As value of the parameter, it contains the opposite direction which the point is moving towards. Afterwards, we receive the stimulus *T1_Cd_Move_Point* at the LCp *F_SCI_P_SR*, which arrives in the LCp *F_P3_SR* as *T1_Move*. We then expect that the transition to the opposite moving-state is activated. During the transition, however, our SUT produces a timeout. This contradicts the *run to completion* semantics.

Figure 5.3: Non-conforming behaviour: Timeout during transition

Unmotivated response *T3_Msg_Timeout* While in a moving-state, we observed that we receive the response *T3_Msg_Timeout* after our SUT has received the stimulus *T1_Cd_Move_Point* at the LCp *F_SCI_P_SR*, which arrived in the LCp *F_P3_SR* as *T1_Move*. We do not expect any response at this point (except the optionally timeout behaviour, which starts with the response *D5_Drive_State* and not with *T3_Msg_Timeout*). This response is not part of the timeout behaviour and the response *D5_Drive_State* of the timeout behaviour is missing. Therefore, it is an unmotivated response that should not be sent according to the EULYNX specifications.

Figure 5.4: Non-conforming behaviour: Unmotivated response *T3_Msg_Timeout*



5.2 Differences to approach with mCRL2

Here, we compare our approach using an AML model to the introduced FormaSig project from Djurre van der Wal and Mark Bouwman who are using mCRL2 as formal specification language and JTorX for Model-Based Testing. To retrieve our results, we looked at the various deliverables [10–12] and conducted interviews with Djurre van der Wal.
After introducing the fundamentals of mCRL2 in Section 2.4, it becomes obvious that its denotational semantics follows a different approach compared to the operational semantics of AML. This results in elementary different models. We list the main differences of the modelling approaches and the gained results in the following. In the future it would be beneficial to research a methodology that allows for a formal comparison.

Symbolism in LTS The generated LPS from the mCRL2 model is a symbolic representation of the state space or Labelled Transition System (LTS) [35]. This LTS can be constructed from the LPS with the use of a state space generator. The symbolic semantics are translated into concrete labels and states. This results in a vast number of states. AMP on the other hand uses STS which is a symbolic LTS. This not only yields substantial benefits in processing expenses but reduces the state space drastically. As of now, mCRL2 only has some experimental symbolic tools.

Test case generation The mCRL2 model is explored for a certain amount of time and the explored states are then used for generating test cases. This is only a partial representation of the entire model. There is no control over what part of the state space is created. The explored states are then used for offline MBT. In our approach, we are using online (or *on-the-fly*) testing where the STS is partly generated on-the-fly and while testing the SUT (see Section 2.1.1). It is possible to influence which part of the model is explored using different test strategies.

Functional flow In the mCRL2 model, a process can always receive stimuli. If the stimulus has an effect in the current state of the model, it is added to an event queue. This process event queue is then consumed when the machine is ready (run-to-completion of previous transitions). For performance reasons, the used queue length is 1. With a larger queue length, the state space becomes unmanageable. We followed a different approach and did not implement an event queue. We changed the AMP strategy to observe quiescence in the composed behaviour model before sending another stimulus. Semantically, this is equivalent with an event queue of length 1.
Additionally, if a stimulus in the mCRL2 model does not have an effect in its current state, the stimulus is not queued and therefore discarded. AMP on the other hand only sends stimuli that are modelled in an activated state. Thus, we do not receive stimuli that we have to discard. Which stimulus is sent in particular depends on the configurable test strategy.

Timing constraints In the mCRL2 model, EULYNX timeout constraints are modelled using a non-deterministic event. Timing constraints are not used. They argue that a lot more time behaviour included in the EULYNX specification is needed before modelling timing behaviour makes sense. In our model, we can model timing behaviour of externally observable transitions but timing constraints for internal transitions are also not possible. Due to the introduced issues of our SUT regarding timing, we modelled the timing constraints similarly as non-deterministic choice.

Initialization LCps In the mCRL2 model, all ports are initialized without triggering events in each LCp during the initialization operation. Thus, the initialization operation is treated differently in this regard than other (translated) ASAL operations.
We follow a different approach in the AML model. We initialize the Data in ports with the Data out ports of the adjacent LCp. Therefore, we use internal synchronization during the initialization operations to distribute the initialized values. This is of importance for the parallel composition.
They have, however, detected some outputs of the LCp *F_P3_SR* in an order that is not permitted by their mCRL2 model. They also state that this is not actually a non-conformance. This confirms the observations we made regarding the response order.

As a closing remark, we want to highlight an interesting aim of the state space exploration facilities and model checking capabilities of the mCRL2 toolset. This toolset is intended to be used in combination with other specification languages [33]. In fact, the mCRL2 toolkit also accepts a subset of executable UML as input [33] although the authors state problems with maintenance. They mention that it is difficult to maintain these conversions from other specification languages into mCRL2 since these languages and syntax evolve.

Chapter 6

Discussion

In this chapter, we first discuss the semantic differences of implementation- and specification models. Next, we have a discussion about the applicability of MBT with AMP in our project. After, we discuss the concrete results of our experiments regarding our point simulation (SUT), AML and our modelling approach, and the EULYNX SysML specifications. Finally, we look at the identified threats to validity of this research.

6.1 Implementation models vs. Specification models

At a general level, a requirement is a statement that describes what a system must do or a quality a system must have. A requirement specification is a set of all requirements that are to be imposed on the design and verification of the system [44]. Thus, we need to define what the system is and where the requirements are elicited from.
The abstract relation between the system and the environment is clear: The system is a solution that was introduced to have effects on the real world in order to solve a given problem [45]. The problem to be solved is rooted in a complex organizational, technical, or physical world [46]. The parts of the world that will affect the system and will be affected by it is its *Application Domain* [45].
Following this definition, we can distinguish between [45]:

- *What* the system does: This is to be sought in the application domain (this is the problem).
- *How* the system does it: This is to be sought in the system itself (this is the solution).

Requirements have their meaning in the problem world whereas software design is concerned with the systems phenomena [46].
However, the EULYNX requirements specifications used for our project do not conform to this definition. Additionally, comparing the EULYNX specifications with the *Principles of good software specification* mentioned by Robert Balzer and Neil Goldman in their early work [47], we see that the first principle of *Separation of Functionality from Implementation* is violated. They state that in specifications the result to be obtained has to be entirely expressed in a *what* (rather than *how*) form.
In the considered EULYNX specifications, internal details of subsystems and design decisions of the solution are specified although the focus should only be on the problem domain. Thus, the specifications are not agnostic of the systems design but model an implementation of one specific solution.
As a reason for that, we look at the applied Model-Based Systems Engineering (MBSE) approach. MBSE is defined as the formalized application of modelling to support system requirements, design, analysis, verification, and validation activities [48]. With MBSE, the output of the systems engineering activities is a coherent model of the system that consists of elements representing requirements, design, test cases, design rationale, and their interrelationships [48]. This model is a representation of one or more concepts that may be realized in the physical world [48].
Design aspects are a central component of MBSE and therefore in the resulting SysML models, making the SysML models *implementation specifications* that implement one specific solution of the application domain. The simulation that we are using in this project as System Under Test (or Implementation Under Test) makes this obvious. It is generated directly from the EULYNX SysML specifications. This makes it apparent that the specifications contain implementation and design details.
AML models on the other hand are *specification models* that accept a set of valid implementations since they are only concerned about the problem domain. For building the AML models we therefore need

a specification that allows for various implementations as long as the requirements of the interface are satisfied. Thus, we conclude that to facilitate effective MBT with AML, a specification on the subsystem interface level that hides all design decisions and implementation details is needed.

6.2 Applicability of MBT with AMP

In their systematic mapping study of *MBT for software safety*, Gurbuz and Tekinerdogan assess relevance to MBT and conclude that MBT can help to test safety-critical systems [49]. The identified recurring motivations were the reduction of the cost of testing and the increased test coverage [49]. In contrast, a disadvantage of MBT is the initial modelling overhead [20]. Despite the fact that approximately 15% of all bugs are found in the modelling phase, it needs a high initial effort. Powerful tools help to mitigate this effort and thus help to further reduce the cost of testing. Gurbuz and Tekinerdogan mention that an important aspect of MBT is the adoption of tool support [49]. The tool support was found in most approaches for both, the model, and the test case generation [49]. Therefore, we see that tool support is an essential part of the applicability of MBT to a domain in general.
Since MBT relies on a formal model which in turn is based on mathematical principles, the support of tools has a large potential to bridge the gap between the mathematical domain and the experts of the application domain. Looking at AMP with its proprietary formal specification language AML, we have been able to create formal models with a gradual learning curve and without a background in mathematics. AMP is designed to be used directly by experts of the application domain, for example by testing experts, architects, and designers at the institution of the SUT. Thus, ease of creating models is one of the main goals of AMP to make the platform usable by the experts of the application domain. In our case, this means that AMP facilitates experts at either infrastructure managers or suppliers of field elements to create formal models themselves. We see here the main benefits of using AMP for MBT compared to other tools like mCRL2 with JTorx.
While the translation of behaviour specifications of single LCps into AML models is achieved with manageable effort, we have demonstrated in our research that the parallel composition is a costly task. The composition is needed to obtain the behaviour model of the entire subsystem which consists of numerous LCps and their interconnections. The composed model is therefore also concerned with internal details of the subsystem. However, these should not be in the scope of the MBT approach where we treat the SUT as black-box without looking at *how* the specified behaviour is implemented. We regard the SysML specifications therefore as *implementation models* (see Section 6.1). In conflict with that, for infrastructure managers and suppliers of field elements we see a benefit in specifications that are restricted to the observable behaviour of the entire subsystem. Suppliers could use this to certify their products for a compliant behaviour on the interface and infrastructure managers could verify the compliance.
On a concluding note, we see a high benefit for infrastructure managers and suppliers of field elements to apply MBT with a tool like AMP that facilitates domain experts directly to create formal models. A specification of the observable behaviour of the entire subsystem would eliminate the need for parallel composition. It would abstract from internal details that are not relevant at the interface between the interlocking system and its subsystems.

6.3 Point simulation (SUT)

Here, we will discuss our results regarding our point simulation which we used as our SUT. We refer back to the results of our qualitative experiment and reason about the found issues. We do not categorize all issues as non-conforming behaviour; some are related to the internal design of the SUT and our experiment implementation. Moreover, we elaborate on the observed non-conformances in our quantitative experiment.

6.3.1 Limitations of our simulation

As described in Section 4.2, the simulation was originally not programmatically accessible. This limited the usability of the simulation as our SUT. We changed the simulation and added a CLI to overcome this limitation. However, the simulation still relies on a GUI in the background. As a result, this yields certain behaviour that leads to failed test cases in our qualitative experiment.
First, we have observed that the **response order** of the SUT is different from the specified one. However, this is allowed according to the Modelling Standard in *Eu.ModSt.1021*. If responses triggered by one and

the same Trigger in port, they have to be sent in random order [30] with the used notation. We therefore do not consider that as a non-conformance of our SUT. The order our SUT sends the responses depends on the internal design of the simulation. It propagates the values of the output fields sequentially in a defined order, starting with the lowest port number. To adapt our model accordingly, we changed the order accordingly in the changes *c1*, *c7*, *c8*, *c9*, and *c15*. Consequently, our model is not agnostic anymore to the internal behaviour of the SUT. Modelling all permutations of response orders would result in a significant larger model that becomes soon infeasible to maintain as multiple responses are specified for almost all effects.
The qualitative experiment showed that the **initialization in the specification and the GUI of the SUT** are not equal. In the SUT, Data (Trigger) in ports are initialized with the default value selected in the underlying GUI. These ports remain uninitialized in the EULYNX specifications. This leads to non-conformance between the model and the SUT as certain operations validate to different outcomes. We regard this not as an error in our SUT but as a limitation of the simulation. To tackle this, we implemented the missing initialization with change *c11* and configuration flag *set_SUT_default_values*.
Additionally, we observed that the **data range of the input fields** in the SUT is not always conform to the specifications. The Data in ports *D21_PM1_Position* and *D22_PM2_Position* can have the value *ERROR* assigned. For example, this is used in the operation *cOp6_Error*. The GUI on the other hand only allows for the selection of the values *LEFT*, *RIGHT*, *NO_END_POSITION*, and *TRAILED*. In this case, we diverted from the approach to use the values of the GUI. We used the values from the specifications which we also assign to the input fields of our SUT (we assign the values directly and circumvent the drop-down selection, therefore we can assign any value). We have seen that the SUT behaves as expected when using the value *ERROR* for these ports.
Due to our experiment setup, we can **only detect value changes** of Data (Trigger) out ports in our SUT. In the EULYNX specification however, a response is also specified when the value does not change. From a systems perspective, the propagated value on a port is valid, even if it remains unchanged. In an embedded system where the values of these ports are read as a continuous stream of modulated data, the assignment in the EULYNX specifications has the semantics of ensuring that this value is at this time propagated on a specific port. Our interpretation of a received response only originates from our mapping onto an STS. Thus, we do not consider that as an error in our SUT. To accept these missing stimuli, we omit them in our model with changes *c4*, *c5*, and *c13* and the configuration flag *response_only_on_data_change*.

6.3.2 Non-conforming behaviour of our simulation

Similar to the response order, we have observed that the **order of the entry- and exit-behaviour** is executed wrongly on certain transitions. Unlike with the arbitrary order of responses within a single effect, the exit-behaviour must be executed as the first effect during an external transition. It is then followed by the transition effect and the entry-behaviour of the newly activated state. Due to the internal design of the simulator, it internally processes all behaviour in the right order but propagates the values to the output fields only once the entire transition is completed. This is referred to as the *SySim model execution loop*[1]. As a result, we detect value changes of the entry-behaviour before value changes of the exit-behaviour. We do not have insight into the internals of the simulation assemblies, and we intend to keep it as a black-box, as MBT is black-box testing. We argue here that this is a limitation of our SUT, but the observable effects are not valid. Therefore, we regard this as an error in the SUT. With the configuration flag *execute_exit_after_entry*, our model is changed in order to accept this erroneous behaviour of our SUT (change *c17*).

6.3.3 Non-conforming behaviour in quantitative experiment

As presented in Section 5.1.3, we have observed several non-conforming behaviours of our simulation. Since the simulation itself is a generated artefact from the SysML specifications where our translation is based on, we want to elaborate on these results. It could be expected that the simulation conforms entirely to our improved model. This is, however, not the case. Reasons for that can be found in different locations.

[1] https://support.ptc.com/help/modeler/r9.2/en/Integrity_Modeler/sysim/SySim_model_execution_loop.html

First, we have to look at how the simulation is derived. This automatic translation from the specification into an executable simulation is a feature of the underlying modelling tool. We do not have insight into the generation process and might have made deviant functional assumptions, based on the specifications (e.g., not according to the SySim model execution loop). Additionally, the generation itself is not proven to be correct.
Second, how the simulation handles timeout behaviour is not purely deducible from the specifications. While we elevated the *run to completion* semantics to the combined behaviour model, the simulation seems to process these events differently and therefore interrupting transitions. We relate the two non-conforming timeout behaviour to this inequality.
Lastly, the precise version of the specification from which the simulation was generated from could not be determined by SIGNON. Although we have managed to evaluate the baseline version of the specification, it could be that the actual simulation is based on a working draft.

6.4 AML and modelling approach

Based on our results, we will here look at our modelling approach with AML, discuss limitations, and propose improvements.

6.4.1 Timing constraints

Many STDs contained in the EULYNX specifications make use of **timeout transitions**. If we model them according to our approach described in Section *Time event*, the timeout transition is not re-evaluated. Our approach concludes to model the timeout transitions as following:

```
1   repeat {
2     o {
3       constraint 'Condition1 && clock < timeout'
4       # effect 1
5     }
6     o {
7       constraint 'Condition2 && clock < timeout'
8       # effect 2
9     }
10    o {
11      constraint 'clock >= timeout'
12      # effect timeout transition
13    }
14  }
```

We set the `timeout` variable on state activation with the current `clock` and an added offset. When entering this state, the constraint of the timeout transition validates to *false*. The clock is then updated every time a response is observed, or a *tick*[2] is generated. The constraints, however, are not re-evaluated after tau[3] transitions. Constraints that can become satisfied after a certain amount of time are currently not supported. For external responses, `before` and `after` are used to model constraints on the clock but they are not supported for tau transitions. To do that, it would be necessary to keep tau transitions with a timing constraint in the state vector without advancing on them directly. Following our model adjustments to tackle the challenges with the step-based simulation (see Section 4.4.2), we did not encounter this issue in our experiment.
Following this discussion, we have a similar problem after the parallel composition. During this process, former external channels become internal. Therefore, timing constraints are not supported anymore on these transitions (see Finding 5).

[2]AMP periodically evaluates the next possible action in the test case. Proceeding to the next evaluation interval is also known as *tick*. Ticks are generated instead of a response when none is available. There is no set rate or interval for them.
[3]Tau transitions model an internal computation step, in formal theory known as an unobservable label τ. The tau transition does not appear as a test step in the test execution. [28]

6.4.2 Response sets

As discussed in Section 6.3.1, the responses of an effect are allowed to be observed in **random order**. Modelling all permutations of all responses within an effect becomes quickly infeasible. A cumbersome approach with a list to keep track of already observed responses could be a possible workaround. For our use case, a dedicated language construct of AML would be useful. We therefore propose to implement a purposeful construct to tackle random responses. For example, a *response set* could be defined where every response has to be received exactly once:

```
1   responseset {
2     o {
3       receive 'Response1'
4       # effect 1
5     }
6     o {
7       receive 'Response2'
8       # effect 2
9     }
10    o {
11      receive 'Response3'
12      # effect 3
13    }
14  }
```

6.5 EULYNX SysML specifications

In this section we discuss the used EULYNX SysML specifications, highlight issues, and propose improvements.

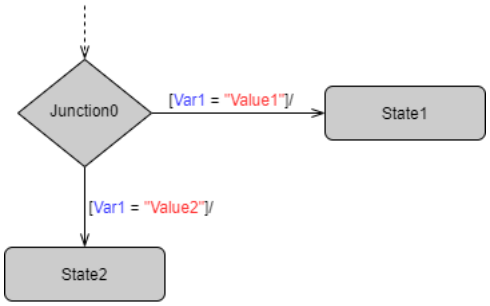
6.5.1 Information type of Data ports

As already mentioned in Section *Data in/out port*, Data ports can either carry **constant information** or **alterable information**. While the first type does not change its value during execution time, the second one might. Unfortunately, the EULYNX specifications fail to mention which information is carried by a specific port. It can only be guessed by the given context.
Since we translate all Data ports to stimulus and response labels, AMP sends stimuli representing a Data in port at any time during execution. AMP chooses a value for the stimuli according to the current testing strategy. As a result, values are changing for Data ports that are supposed to be constant. This might not only lead to undesired behaviour in the SUT but also generates unrealistic test cases. We therefore recommend distinguishing between these types of Data ports by either creating a new *configuration* port type or to specify a distinction in the port signature. For testing with randomized values, we furthermore advice to provide a range that should be covered in the test cases (e.g., timeouts of multiple hours are valid but assumingly not realistic).

6.5.2 Value ranges

Following previous point, the specification of value ranges of Data (Trigger) ports is missing in general. On one side this is important to ensure that AMP covers the entire value range when stimulating the SUT. On the other side this issue gains more significance when looking at guards of choice pseudostates as described in Section *Choice pseudostate*. On all outgoing transitions of choice pseudostates, exactly one guard must validate to *true*, thus they must be orthogonal to each other. The Modelling Standard recommends in *Eu.ModSt.1057* that a catch-all guard technique may be used to ensure the validity of a choice pseudostate [30]. In the EULYNX specifications however, there are guards of choice pseudostates specified like the following example in Figure 6.1.

Figure 6.1: Non orthogonal guards of a choice pseudostate



If `Var1` can only have the values `Value1` or `Value2`, this is valid. However, without a definition of the value range, these are only assumptions. We therefore recommend a specification of the value range of all Data (Trigger) ports. Additionally, we recommend to always have a defensive path as this is one of the principles of defensive programming in the software design techniques for safety-critical systems [50].

6.5.3 Orthogonal regions

Orthogonal regions are executed concurrently. They are located within the same LCp and therefore share all ports, operations, and block properties. This results in certain problems regarding synchronization. In the orthogonal regions of the considered specifications of this research, data separation was done in a way that both regions only use a mutual exclusive subset of Data out ports and block properties. They do, however, share Trigger out ports in both regions. Furthermore, they use the same operations which are in turn dependent on the same Data in ports. This challenges the definition in the Modelling Standard that each region has a set of mutually exclusive disjoint subvertices and a set of transitions [30], where it is arguable if transitions with the same operation as event are mutually exclusive. As a result, a value change of a Data in port may trigger a transition in both regions simultaneously. These activated transitions can process effects including sending responses which in turn may trigger another transition in an adjacent LCp. The responses are sent in a non-deterministic order triggering transitions in adjacent LCps in a non-deterministic manner.
Furthermore, it must be ensured that both regions operate with the same values, especially if the values change after the first region has read the in port and triggered a transition. Additionally, it poses the question if one region is allowed to change values while the other region is processing a transition.

> **Finding 6:** The SysML *run to completion* semantics contradicts the concurrent characteristics of orthogonal regions.

In our opinion, orthogonal regions accommodate various risks and can lead to non-deterministic behaviour and race conditions. In the Modelling Standard, specifications regarding synchronization and data management across concurrent regions are missing. We therefore recommend to either avoid using orthogonal regions or to further specify how these challenges are addressed.

6.5.4 Discussion with SIGNON

Together with the 13 involved infrastructure managers, the EULYNX Consortium develops specifications for all EULYNX subsystems and their interfaces. Basis for these specifications are Use Cases and high-level user requirements including national specifications which are described in natural language for the most part. Based Use Case scenarios, the consortium derives functional and non-functional system requirements to be published in the corresponding requirements specifications. After an elicitation phase with the infrastructure managers, the consortium forwards them to the company SIGNON[4], enhanced with technical constraints to build a model-based specification of requirements. Therefore, SIGNON creates SysML models using the *PTC Windchill Modeler*[5], closely working together with the infrastructure managers in dedicated work clusters. The SysML models capture the Use Cases, their scenarios, and the derived system requirements. To complete the work cycle, the created models are tested and validated together with the infrastructure managers against the use case scenarios.

[4]https://signon-group.com
[5]https://www.ptc.com/en/products/windchill/modeler

We have scheduled a discussion with the involved persons at SIGNON to learn about their processes, tool chain, and modelling approaches. Since these specifications lay the foundation for our project, it provided us with useful insights. Additionally, we set off a dialog to share our results and contribute towards future improvements in the specifications.
We learned that the tool selection (PTC Windchill Modeler, IBM Rational DOORS, and additional extensions) and formalism (SysML) is mostly influenced by the stakeholder needs of the EULYNX Consortium, which in turn are partly historically grown. For example, the consortium requires executable models. As an additional requirement, the toolchain must provide a simulation for the specifications. The simulations are used for the aforementioned testing of the models and validation of the requirement implementation. The generated simulation is the same as we used in our project for validation purposes. At the current stage, SIGNON is facing similar challenges regarding the simulation. Implementing a programmatically usable interface requires time consuming efforts. Thus, the testing and validation itself is carried out in a manual fashion utilizing the Use Case scenarios as a reference. Test cases are derived from the Use Case scenarios.
Additionally, we discussed the challenges of system decomposition and functional analysis. We learned that further improvements regarding the functional flows and meaning of model elements are likely to be added in the near future to increase readability and understanding. For example, this will be done using colour coding that clarifies the meanings of the model elements used in the specifications.

6.6 Threats to validity

6.6.1 Correctness of toolchain

We make the fundamental assumption that our used toolchain is working properly. If this assumption is not correct, it could invalidate parts of our results.
In terms of AMP, this means that we assume the documented behaviour of AMP and that none of the found issues are due to an implementation fault in AMP. Various companies are using AMP and testing processes are in place in the CI/CD pipeline of the development life cycle. Additionally, AMP was validated for Safety Integrity Level 1 (SIL1) where test coverage is one validation criterion. We therefore can assume a certain maturity level of the software.
Furthermore, we assume that the plug-in adapter is implemented according to the explanation in Section 4.3 without any bugs that affect the outcome of this research. We have carried out some basic tests but did not implement and exhaustive test suite. We argue that fundamental problems in the adapter would be evident (e.g., communication problem between AMP and SUT) or become evident during the analysis of the failed test cases.
Moreover, the CLI of the SUT is realized as described in Section 4.2. We are aware that this introduces certain characteristics due to the limitations of our SUT. Since the CLI is part of our SUT, the introduced characteristics can be assigned to the behaviour of the SUT and interpreted as non-conforming behaviour. We address this problematic in Section 6.3 and assume that our implementation works correctly as described.

6.6.2 Isolation of LCps

In our project, we have isolated two LCps and separated the subsystem specific LCps from the generic ones. In the complete system, the subsystem specific LCps are embedded in the generic environment of the LCp. This made it necessary to bootstrap our SUT. We intend to bring the generic system parts to an initialized state so that it is in-sync with our modelled behaviour. We defined this bootstrap process based on analysing the STDs of the generic parts and based on observations of our simulation.
Its correctness, however, is not proven. If done inadequately, the generic parts of our SUT are in wrong states and may propagate different values. Thus, the simulation might behave unexpected and is therefore not suitable for validating our translation approach.
Due to the scope on two LCps, our parallel composition process was of a lesser complexity compared to the entire system. The upscaling of our approach to the extend of the complete system requires additional validation and could show that a different approach is suited better in terms of reducing synchronization overhead.

6.6.3 Stimuli and response mocking

Due to the isolation of two LCps, we had to mock certain stimuli and responses. We did this based on the analysis of the corresponding STDs and we reasoned about the correctness. However, we have no insight on how the simulation actually processes these stimuli and responses. It might affect the state of additionally involved LCps and therefore influence the resulting behaviour of the entire system.
For example, we expect that a certain stimulus has a predicted effect on the state of an LCp and produces a response. This response is the stimulus for the adjacent LCp we are originally interested in. Additionally, our stimulus has additional effects on other LCps that are not in the scope of our project. If there are additional effects where we are not able to observe a response (e.g., another LCp changes its state or the value of a block property), we might assume the wrong states or values. In other words, our mocked stimulus would have a side-effect on another part of the system that is not in the scope of our project but influences subsequent behaviour.

6.6.4 Additional validation

We have run the quantitative experiment numerous times and identified the mentioned *Non-conformance of SUT*. Based on the achieved transition coverage, we have trust that we have found most of the non-conformance and adequately validated our translation. However, we do not claim that we have detected all situations where our test cases fail. Especially regarding to future tests with a focus on timing behaviour, we can not exclude the possibility of failed test cases that challenge our translation approach.

Chapter 7

Related work

Djurre van der Wal and Mark Bouwman of the FormaSig project have implemented an automatic translation from SysML models into a formal mCRL2 model [10–12]. For the automatic translation, they created the framework *jEULYNX* in which the EULYNX SysML diagrams are described. The transformation into a formal mCRL2 model is then carried out automatically.
Since SysML has no formal semantics, it is not possible to validate the translation by a notion of equivalence between the models. For the validation, they compare the translation to the informal specifications of SysML to convince themselves that the semantics are preserved as intended. They compare the translated model to the SysML standard, the EULYNX Modelling Standard, and the simulators that are used for validation purposes within EULYNX. They are using the same simulation as we do in our project.
Together with the provided toolset of mCRL2, they then use the formal model for model-checking. Additionally, they generate test cases with JTorX. They run these obtained test cases against the same simulation of the point subsystem for MBT.
This project is currently in progress.

Shubhangi Salunkhe researched the translation of semi-formal SysML models into Event-B models [14]. Event-B is a modelling method for formalising and developing systems whose components can be modelled as discrete transition systems [51]. The goal of this work was to propose a methodology for a transformation of SysML specifications into formal Event-B models including a toolchain for an automated translation. The transformation was then verified automatically. The project tackled the issue that the previously used manual approach of SysML model transformation was too time consuming. Furthermore, no standardised process for proving the compliance between the generated formal model and the original SysML model existed. In the former modelling approach, the SysML models were first manually translated into UML-B models and subsequently into Event-B models. This work proposed a methodology to directly translate the SysML models into the Event-B model.
They have used the PTC Integrity Modeler (a predecessor of the aforementioned PTC Windchill Modeler) to retrieve the SysML STDs in an *XML Metadata Interchange* (XMI) format which was then further processed using Java XSL Transformer, eMoflon:IBeX[1] and RODIN. eMoflon:IBeX performs model-to-model transformation based on a defined set of transformation rules which were provided by the author of this work. As a result, the formal Event-B model is retrieved.

Linh Ngoc Bui analysed in her work the benefits for ProRail of the EULYNX-style requirements modelling [52]. The main focus of the project was to analyse the EULYNX modelling approach in the context of ProRail to provide suggestions to ProRail in terms of modelling. The general approach of this project was to design a methodology to analyse the current situation of EULYNX and ProRail and subsequently to implement this design using SysML. Since many subsystems are involved in EULYNX, the project was limited to the point and TDS[2] subsystems.

[1] https://github.com/eMoflon/emoflon-ibex
[2] Train Detection System

Chapter 8

Conclusion

In our research we have demonstrated a valid approach to translate SysML models into a Symbolic Transition System (STS). First, we have shown that we are able to translate the specified behaviour of the semi-formal EULYNX SysML STDs into a formal AML model. Second, we presented an experimental setup to enable Model-Based Testing (MBT) using our formal model.
In the current EULYNX specifications, subsystems are further decomposed into logical components (LCps) which partition the external visible stimulus-response behaviour. The decomposition aims for mastering the complexity and facilitates reusability. The atomic LCp specifications encapsulate only a subset of the entire externally observable stimulus-response behaviour. These LCps and their interconnections are arranged in the logical architecture (LA). We performed a *parallel composition* of our formal AML models according to the LA where we have shown an approach to tackle this challenge and proven its feasibility. However, this composition involves modelling internal behaviour and logic between independent LCps since only the atomic LCps remain as black boxes in the EULYNX specifications.
This approach contradicts the Model-Based Testing principles and semantics of the AML *specification models* which focuses exclusively on the externally observable behaviour. The AML models address *what* the system according to the specification should do while omitting *how* this is achieved. The latter one is only relevant for an implementation. Thus, the SysML specifications are considered as *implementation models*. Specifications that represent the entire subsystems as a black-box are missing.
Relating back to our use case, an infrastructure manager (IM) is not interested in the internal logic of a subsystem. For example, the differentiation between generic and subsystem specific sub-components is not relevant from a systems perspective. Relevant for IMs is only the visible behaviour at its interface, which is located between the electronic interlocking system (EIL) and the subsystem as a whole. While the current decomposition may serve its purpose for mastering the complexity during the systems design stage, it creates challenges for the applicability of MBT since we can not abstain from internal details.
We conclude that for facilitating MBT for subsystems, an interface specification of the stimulus-response behaviour of the entire subsystem is vital. Only then the formal model is agnostic to the internal logic. As of now, such a specification is not existent. If it becomes relevant to test LCps independently, their entire syntactic interface must be externally observable to apply MBT.

While creating our translation into our formal AML models, we encountered underspecification regarding the functional flow and some minor mistakes in the semi-formal SysML models. This raises the question if SysML is the most suitable language for modelling this system. Especially when it comes to safety critical systems, ambiguity must be avoided. If not, different implementations from competing suppliers might demonstrate unintended behaviour.
The goal of the EULYNX Consortium is to create high quality specifications that are also understandable for people without a strong mathematical background [30]. This is their main motivation to use SysML as specification language in the MBSE approach. However, they agree that a semi-formal approach raises a number of criticism and that formal methods are one of the correct ways to specify and verify system requirements, but the required mathematical background hinders the appliance of formal methods.
Albeit we see the benefits embodied in the pictographic SysML models, especially in an early systems design phase, we challenge the decision to simply elide a formal approach. We agree that certain formal specification languages such as mCRL2 are limited in their usability for domain experts without a background in formal methodologies. Other formal specification languages like AML however provide a sufficient abstraction from the complex mathematical foundations to become applicable for domain experts.

We conclude that in a more mature stage of the design phase a formal methodology must be considered. A formal model based on mathematical principles combined with a powerful toolset provides various possibilities regarding model-checking and MBT.

8.1 Future work

In our work we have researched the general feasibility of translating EULYNX SysML specifications into AML for MBT. Given that the EULYNX specifications are a work in progress and that there is no actual implementation available yet, there are interesting questions for future research. Additionally, our system scope can be broadened to a model of the entire system for a further validation of our translation.

8.1.1 On system scope

In our research we have focused on the subsystem point where we have limited ourselves to the subsystem specific LCps. A complete translation including the remaining LCps is outside of the scope of this project and left for future research. Likewise, a translation of other subsystem specifications remains to be researched. There is, however, no implementation or simulation of other subsystems available yet that could be used as SUT to validate the translations.

8.1.2 On automatic translation

In the related work mentioned in Chapter 7, Djurre van der Wal and Mark Bouwman of the FormaSig project created the framework *jEULYNX*. They are using a subset of the programming language Java to describe the EULYNX SysML diagrams. Then, they use *jEULYNX* to retrieve mCRL2 models automatically. Currently, *jEULYNX* is a work in progress as well and has some limitations. Nevertheless, a generator for AML models could be added in the future. This would facilitate the *jEULYNX* framework to automatically translate into formal AML models, based on our translation patterns. Once specifications are added to *jEULYNX*, it is easy and fast to retrieve the formal model. This is especially interesting when translating additional subsystems.

8.1.3 On EULYNX SysML specifications

The EULYNX specifications are a work in progress and newer versions are released periodically. It remains unclear how our proposed translation can be applied to future versions of the specifications. In fact, in the baseline 3.1 (2.A) of the generic interface, they applied a new construct in the LCp *F_SMI_EfeS_SR* with a behaviour that can be instantiated for each existing *Item* [53]. For now, the specifications leave many questions unanswered, such as how this complies with the fixed number of ports depicted in the IBD of the same LCp. As future research, our translation pattern could be validated with newer versions of the specifications and extended, if necessary.

8.1.4 On different formalisms

We are using AML to create our formal model, but related works mentioned in Chapter 7 use different formalisms such as mCRL2 and Event-B. While we argue that AML has many advantages in terms of usability, we are lacking a methodological approach for a formal comparison between various formal modelling languages. While many research projects have been carried out concerning software metrics and software measurement methodologies, there is a research gap when it comes to measurement methodologies for formal models. Such a methodology could help to argue which modelling approach is favourable, for example in terms of size, complexity, maintainability, and usability.

8.1.5 On validation

As explained in Section 4.2, we used the point simulation as our SUT to validate our translation. We had to change the simulation in order to make it programmatically accessible. This simulation comes with several limitations. For example, the simulation is step-based and not in real time. Another validation should be done in the future using a real implementation, whenever one becomes available. Analysing the retrieved results might yield further findings, especially regarding timing behaviour.

Acknowledgements

The graduation project is always a challenge, especially during the peak of a pandemic. Nevertheless, I consider myself privileged that I was given the chance to work on this project. I would like to take this opportunity to thank all the people that made this possible.

First, I like to express my gratitude to the team of Axini. They always took time out of their busy schedules to answer questions and assist with troubleshooting. I would like to especially mention Machiel, my daily supervisor, with whom I had inspiring discussions and who has provided me with constructive feedback. He made this project possible in the first place. Additionally, I would like to thank Ana, my academic supervisor, for giving guidance in the scientific world.

Next, I would like to thank Daan and Djurre who were always available for meetings with delightful discussions and explanations. I am specially thankful for the collaboration with the FormaSig project that was made possible by them.

Then, I would like to express my thanks to Xavier, who worked on his graduation project at Axini at the same time, for the enriching talks and comradeship throughout this project. It made working during these home office characterized times much more pleasant and entertaining.

Furthermore, a big thanks to all my study colleagues that have always supported me with helpful tips, entertaining conversations, and - most importantly - with great evenings during otherwise rather secluded times. I experienced great team spirit and selfless dedication to support each other. Special thanks go to *team7*, which is now celebrating its anniversary, with Bart-Jan, Elvin, and Bálint; and the *SP-Group* with Jeffrey, Paul, Sandro, and Georgia.

Lastly, but most importantly, I express my deep gratitude to Mirna, who always supported me without any exception throughout my entire studies and always managed to bring back my motivation whenever I have lost it. You bring out the best in me.

Bibliography

[1] G. Barbu, "Iness - integrated european signalling system," Accessed: 2021-08-09.

[2] *National rules*, https://www.era.europa.eu/activities/national-rules_en, Accessed: 2021-08-09.

[3] R. Safety Unit and S. M. Sector, "Final report of the task force on national safety rules," European Union - Agency for Railways, Tech. Rep., version 1.0, Dec. 13, 2012.

[4] J. Tretmans, "Model based testing with labelled transition systems," eng, in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38, ISBN: 3540789162.

[5] *Cei - en 50126: Railway applications - the specification and demonstration of reliability, availability, maintainability and safety (rams)*, https://standards.globalspec.com/std/1507670/cei-en-50126, Accessed: 2021-08-02.

[6] *Cenelec - en 50128: Railway applications - communication, signalling and processing systems - software for railway control and protection systems*, https://standards.globalspec.com/std/14317747/EN%2050128, Accessed: 2021-07-29.

[7] *Bsi - bs en 50129: Railway applications – communication, signalling and processing systems – safety related electronic systems for signalling*, https://standards.globalspec.com/std/13310429/bs-en-50129, Accessed: 2021-07-29.

[8] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems Advanced Lectures*, eng, 1st ed. 2005., ser. Programming and Software Engineering ; 3472. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ISBN: 3-540-32037-7.

[9] A. Andrews, M. Abdelgawad, and A. Gario, "Towards world model-based test generation in autonomous systems," *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 1–12, 2015.

[10] D. van der Wal and M. Bouwman, "Formasig deliverable d2.1: Translating eulynx to mcrl2: Inventory & strategy," eng, Ph.D. dissertation, Technische Universiteit Eindhoven, University of Twente, 2020, pp. 1–24.

[11] ——, "Formasig deliverable d2.2 translating eulynx to mcrl2: Prototype automatic translator," eng, Ph.D. dissertation, Technische Universiteit Eindhoven, University of Twente, 2020, pp. 1–28.

[12] ——, "Formasig deliverable d2.4: Conformance test suite for eulynx component 2," eng, Ph.D. dissertation, Technische Universiteit Eindhoven, University of Twente, 2021, pp. 1–10.

[13] A. Belinfante, "Jtorx: Exploring model-based testing," eng, IPA Dissertation series no. 2014-09, Ph.D. dissertation, University of Twente, Netherlands, Sep. 2014, ISBN: 978-90-365-3707-0. DOI: 10.3990/1.9789036537070.

[14] S. Salunkhe, "Automatic transformation of sysml model to event-b model," eng, M.S. thesis, Universität Paderborn, 2020, pp. 1–119.

[15] R. W. Whitty, "Software testing techniques, by boris beizer, van nostrand reinhold, second edition, 1990 and testing computer software, by c. kaner, j. falik and h. q. nguyen, van nostrand reinhold, second edition, 1993 (book review)," *Softw. Test. Verification Reliab.*, vol. 2, pp. 215–216, 1992.

[16] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012. DOI: https://doi.org/10.1002/stvr.456. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.456. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.456.

[17] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, *Model-Based Testing of Reactive Systems: Advanced Lectures*, eng, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, vol. 3472, ISBN: 3540262784.

[18] M. Timmer, H. Brinksma, M. I. A. Stoelinga, M. Broy, C. Leuxner, and C. Hoare, "Model-based testing," eng, *Software and Systems Safety: Specification and Verification*, pp. 1–32, 2011, ISSN: 1874-6268.

[19] J. Peleska and W.-l. Huang, "Industrial-strength model-based testing of safety-critical systems," eng, in *FM 2016: Formal Methods*, ser. Lecture Notes in Computer Science, vol. 9995, Springer International Publishing, 2016, pp. 3–22, ISBN: 9783319489889.

[20] T. Ruys and P. Verkade, *Model based testing with axini modeling suite*, Oct. 2020.

[21] T. Ruys, *Introduction to model based testing and the axini modeling language*, Axini B.V., Oct. 2020, pp. 1–23.

[22] T. Chow, "Testing software design modeled by finite-state machines," eng, *IEEE transactions on software engineering*, vol. SE-4, no. 3, pp. 178–187, 1978, ISSN: 0098-5589.

[23] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, "Test case generation by means of uml sequence diagrams and labeled transition systems," eng, in *2007 IEEE International Conference on Systems, Man and Cybernetics*, IEEE, 2007, pp. 1292–1297, ISBN: 142440990X.

[24] C. B. Jones, *Systematic Software Development Using VDM (2nd Ed.)* USA: Prentice-Hall, Inc., 1990, ISBN: 0138807337.

[25] J. Spivey, "Understanding z—a specification language and its formal semantics," eng, *Science of Computer Programming*, vol. 11, no. 2, pp. 180–184, 1988, ISSN: 0167-6423.

[26] G. J. Holzmann, "Design and validation of protocols: A tutorial," *Computer Networks and ISDN Systems*, vol. 25, no. 9, pp. 981–1017, 1993, Protocol Specifications, Testing and Verification, ISSN: 0169-7552. DOI: https://doi.org/10.1016/0169-7552(93)90095-L. [Online]. Available: https://www.sciencedirect.com/science/article/pii/016975529390095L.

[27] L. Frantzen, J. Tretmans, and T. A. Willemse, "Test generation based on symbolic specifications," eng, in *Lecture Notes in Computer Science*, vol. 3395, 2005, pp. 1–15.

[28] A. B.V., *Effective aml*, Axini B.V., Jun. 2021, pp. 1–86.

[29] *Principles of railway interlocking*, https://www.railwaysignalling.eu/railway-interlocking-principles-railwaysignalling, Accessed: 2021-07-29.

[30] E. Initiative, *Modelling standard: Eu.doc.30*, 3.0 (0.A), Eulynx, pp. 1–190.

[31] K. Pohl, H. Hönninger, R. Achatz, and B. Manfred, Eds., *Model-Based Engineering of Embedded Systems The SPES 2020 Methodology*, eng, 1st ed. 2012. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN: 1-283-90881-6.

[32] O. M. Group, *Omg systems modeling language (omg sysml)*, 1.3, OMG, pp. 1–272.

[33] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse, "An overview of the mcrl2 toolset and its recent advances," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Piterman and S. A. Smolka, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 199–213, ISBN: 978-3-642-36742-7.

[34] J. F. Groote, J. J. A. Keiren, B. Luttik, E. P. de Vink, and T. A. C. Willemse, "Modelling and analysing software in mcrl2," eng, in *Formal Aspects of Component Software*, ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 25–48, ISBN: 9783030409135.

[35] *Introduction to mcrl2*, https://www.mcrl2.org/web/user_manual/introduction.html, Accessed: 2021-06-23.

[36] J. F. Groote, A. Ponse, and Y. Usenko, "Linearization in parallel pcrl," eng, *The journal of logic and algebraic programming*, vol. 48, no. 2, pp. 39–70, 2001, ISSN: 1567-8326.

[37] E. Initiative, *Generic interface and subsystem requirements: Eu.doc.20*, 3.0 (0.A), Eulynx, pp. 1–51.

[38] ——, *Requirements specification for subsystem point: Eu.doc.36*, 3.0 (0.A), Eulynx, pp. 1–41.

[39] *Asal structured action language (sysim)*, https://support.ptc.com/help/modeler/r9.0/en/index.html#page/Integrity_Modeler/sysim/SySim_Atego_structured_action_language.html, Accessed: 2021-06-08.

[40] ——, *Interpretation rules for model-based requirement: Eu.doc.29*, 1.3 (0.A), Eulynx, pp. 1–41.

[41] O. M. Group, *Omg unified modeling language (omg uml)*, 2.5, OMG, pp. 1–794.

[42] R. Wieringa and M. Jackson, *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*, ser. ITPro collection. Elsevier Science, 2003, ISBN: 9781558607552.

[43] P. Wolber, *Sup_p_sr logical architecture*, SIGNON, pp. 1–1.

[44] I. Hooks, *What is the difference between a requirement and a specification?* `https://reqexperts.com/resources/requirements-articles/articles-what-is-the-difference/`, Accessed: 2021-09-10.

[45] M. A. Jackson, *Software requirements & specifications : a lexicon of practice, principles and prejudices*, eng, ser. ACM Press Books. Addison-Wesley, 1995, ISBN: 0201877120.

[46] A. V. Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*, eng, 1st ed. Wiley, 2009, ISBN: 8126545895.

[47] R. Balzer and N. Goldman, "Principles of good software specification and their implications for specification languages," in *Proceedings of the May 4-7, 1981, National Computer Conference*, ser. AFIPS '81, Chicago, Illinois: Association for Computing Machinery, 1981, pp. 393–400, ISBN: 9781450379212. DOI: `10.1145/1500412.1500468`. [Online]. Available: `https://doi.org/10.1145/1500412.1500468`.

[48] R. Steiner, S. Friedenthal, and A. Moore, "Integrating sysml into a systems development environment," eng, in *Practical Guide to SysML - The Systems Modeling Language*, 3rd Edition, Elsevier, 2015, pp. 2–2, ISBN: 0128002026.

[49] H. G. Gurbuz and B. Tekinerdogan, "Model-based testing for software safety: A systematic mapping study," *Software Quality Journal*, vol. 26, no. 4, pp. 1–46, Dec. 2018. DOI: `10.1007/s11219-017-9386-2`. [Online]. Available: `https://doi.org/10.1007/s11219-017-9386-2`.

[50] *Safety-critical software design techniques*, `https://www.leedeo.es/l/safety-critical-software-design-techniques/`, Accessed: 2021-07-23.

[51] T. S. Hoang, *An introduction to the event-b modelling method*, eng, 2013.

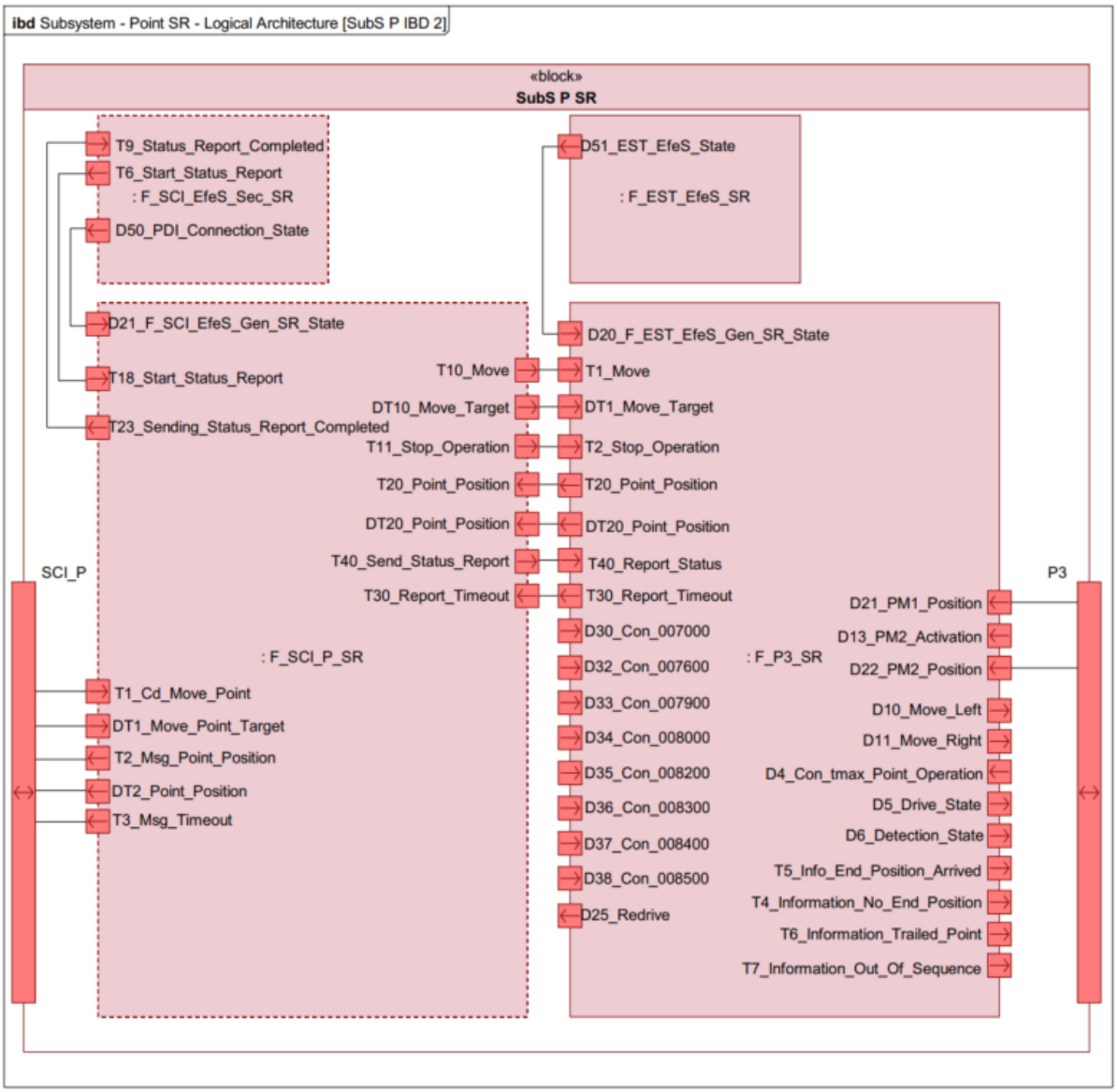
[52] L. N. Bui, "An analysis of the benefits of eulynx-style requirements modeling for prorail," eng, Ph.D. dissertation, Technische Universiteit Eindhoven, 2017, pp. 1–123.

[53] E. Initiative, *Generic interface and subsystem requirements: Eu.doc.20*, 3.1 (2.A), Eulynx, pp. 1–50.

Appendix A

IBD of logical architecture point subsystem

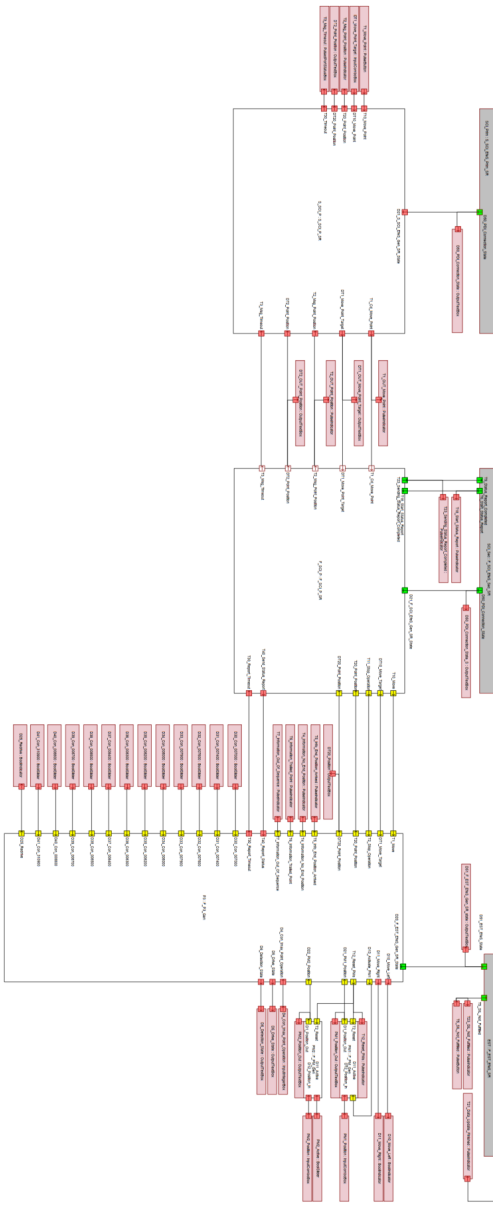
Figure A.1: IBD of logical architecture point subsystem [38]



Appendix B

Partial logical architecture of point simulator

Figure B.1: Partial logical architecture of point simulator [43]



Appendix C

Test set configuration for quantitative experiment

Figure C.1: Test set configuration for quantitative experiment