Scenario-Based Design and Simulation of Mechatronic Systems

Bachelor Thesis
Software Engineering Group
Prof. Dr. W. Schäfer
Institut für Informatik
Fakultät für Elektrotechnik,
Informatik und Mathematik
Universität Paderborn

vorgelegt von
Jens Frieben
am
27.03.2009

Betreuer: Joel Greenyer

Gutachter: Prof. Dr. Wilhelm Schäfer
Prof. Dr. Heike Wehrheim

Jens Frieben
Matrikelnummer: 6397644
Kampstrasse 7
59505 Bad Sassendorf
Contents

1 Introduction 1

2 Foundations 5
  2.1 Design process of mechatronic systems 5
    2.1.1 Development process 7
    2.1.2 Mechatronic Modelling Language 9
  2.2 Sequence Diagrams 10
    2.2.1 UML Sequence Diagram 11
    2.2.2 Live Sequence Charts LSC 12
    2.2.3 Modal Sequence Diagram MSD 15
  2.3 S2A and AspectJ 17
  2.4 Other tools and technologies 20
    2.4.1 Activestructure-Editor 20
    2.4.2 EMF & GMF 20
    2.4.3 UML2 project 22
    2.4.4 OCL 22

3 Concepts 23
  3.1 Modelling 23
    3.1.1 Scenario: Railcab switches track 26
    3.1.2 Scenario: Detect Railcab - create convoy 32
    3.1.3 Scenario: Enter switch 33
    3.1.4 Scenario: Disband convoy 34
  3.2 Tool vision 35
  3.3 Scenario-Playout 36
    3.3.1 Trace semantics of MSDs as a Petri net 37
    3.3.2 Initialization 38
    3.3.3 Hot/Cold Violations 39
    3.3.4 Hot/Cold CombinedFragments 40
    3.3.5 Asynchronous messages 40
    3.3.6 Message selection and Playout-Strategies 41
    3.3.7 Playout 42
  3.4 Object Binding 43
    3.4.1 Symbolic lifelines 43
    3.4.2 Conventional Binding 44

III
## Contents

3.4.3 Extended Binding ........................................ 44  
3.4.4 Binding problems .......................................... 45  
3.4.5 Assignment ................................................. 47  
3.5 Hierarchical refinement ...................................... 48  

4 Implementation .................................................. 51  
4.1 Modal Petri nets ............................................... 51  
4.1.1 Petri net elements ........................................... 52  
4.1.2 Conversion ................................................... 53  
4.2 Simulator components ......................................... 56  

5 Conclusions and Future Work .................................. 59  

A Scenario diagrams ............................................. 63  

List of Figures ..................................................... 69  

Bibliography .......................................................... 71
1 Introduction

As part of the Collaborative Research Center 614 (CRC 614) 'Self-Optimizing Concepts and Structures in Mechanical Engineering' \(^1\), new methods and languages are developed for the design of mechatronic systems. A characteristic of these systems is that multiple engineering domains are involved in their development. These are generally mechanical-, controlling-, electrical-, and software engineering. Advanced mechatronic systems consists of software-intensive, interacting components, which makes the development of these systems even more challenging.

Based on existing development guidelines like VDI 2206 [22], the CRC’s new development concepts introduce methods and languages to cope with these complex systems. At the beginning of the design process, all domains work together on the conceptual design. The resulting design document of this combined efforts is called the principle solution, and is created in four phases. One of these phases is the 'Conceptual design on system level'. Here, the scenarios that are identified in a previous phase, are used to specify function hierarchy, create the systems structure, and to model its behavior. If the principle solution is approved, the domains begin their own, domain specific development independently.

The techniques currently used in the CRC are not yet sufficient, since they cannot be formally analyzed. The formal analysis of system behavior is important in order to detect errors early in the development stages. If errors remain undiscovered, the costs to correct them later in the development are significantly higher. If they are not even identified and solved in later development phases, this also affects the safety and reliability of the mechatronic system.

Therefore new ways to formally specify and analyze the behavior are required. One analysis method, which is also the focus of this thesis, is the simulation of the mechatronic systems behavior. Such an analysis allows the developer to find inconsistencies and errors right in the conceptual design phase.

This thesis introduces concepts and methods to specify and simulate behavior of mechatronic systems. I present a way to model the behavior by using Modal Sequence Diagrams (MSD) and a play-out engine I developed to simulate the sys-

\(^1\) Collaborative Research Center 614 – www.sfb614.de
1 Introduction

tem afterwards. The decision to use sequence diagrams is based on two reasons. First, the design is driven by application scenarios and second are requirements often given in the form of sequences of interactions between environment and system components. A play-out is the monitoring and execution of previously modeled behavior diagrams. Modal Sequence Diagrams are used for this modelling, because they allow the distinction between possible behavior and necessary behavior. MSDs are a form of Live Sequence Charts that was developed in order to formalize the semantics of assert and negate in UML [12]. They allow the developer to model certain interactions between system and environment components that may or must occur.

My solution is similar to the play-out engine developed by Harel and Maoz [13, 11, 10]. They use the Modal Sequence Diagrams to specify the behavior of Java-based applications. These MSDs are then compiled into an application specific logic which is used to control the simulation. I denote this compiler and play-out the S2A approach.

Like Harel and Maoz, I propose MSDs to model the behavior in the conceptual design of mechatronic systems. Based on an example I show how the scenario-based behavior specification for a mechatronic system can be realized.

Since the Railcab case study is often used in the CRC, this was the first choice for an example. The Railcab system is a novel transportation system that is currently developed at the University of Paderborn. In the future, they shall autonomously transport passengers and goods. Furthermore, they autonomously make certain decisions during their operation, for example to form convoys with other Railcabs or to optimize the comfort level for passengers. The Railcab-system is a challenging example with many networked systems and components/subcomponents on different hierarchies, making it an ideal basis for an example.

During the course of elaborating the concepts for playing-out the MSDs, some problems were identified that could not be resolved in the scope of this thesis. Concepts for the desired use case for modeling and simulating in hierarchical component structures are investigated, but could not be implemented in time. Also the assignment of values and references is left open for implementation. This restricts the current available simulation procedure drastically, since no changes to the model can be made. Discovered problems with play-out when dealing with systems with dynamic structure could be solved and integrated into the current implementation.

The second chapter Foundations introduces the development process for mechatronic systems in more detail. A short overview to the Railcab and its modules and components is given. Following this are the descriptions of interaction dia-
grams that are used in this thesis. I introduce Live Sequence Charts (LSC) and give a short description of a LSC Play-out - a procedure to simulate modeled behavior. An extended version of the LSC, the Modal Sequence Diagrams, that are used in the S2A play-out [10] and my approach. Therefore I explain them in more detail before I proceed to the S2A approach, on which my simulation engine is based.

A main topic of this thesis is how the behavior of mechatronic systems can be modeled in a scenario-based way. Therefore, I will start the third chapter Concepts with the example and its scenarios. The behavior of scenario each is modeled with MSDs. With these behavior diagrams at hand, the vision of a tool for modelling and simulation is sketched. Driven by this vision I introduce the concepts for the play-out I developed. Problems that I identified during the modelling phase are the focus of the last sections of the concepts chapter. To complete this thesis, the Implementation chapter show some details about the representation of MSDs as Petri nets and a conclusion give a resume and an outlook on revealed or still open questions.
1 Introduction
2 Foundations

This chapter gives an overview of the terminology and some basic technologies that are used within this thesis. First I explain what mechatronic systems are, the problems in design of these systems, and the new development approach. Afterwards the basics of sequence diagrams and the two derivative specifications 'Live Sequence Charts' and 'Modal Sequence Charts' are explained. Based on these prerequisites, I introduce the S2A compiler which makes use of these diagrams. AspectJ, an aspect oriented extension to Java, is a vital part of the S2A play-out. Therefore I give a short overview of this topic. To conclude the foundations chapter the Partialmodel-Editor and its dependencies Eclipse Modelling Framework (EMF), the UML2 project, Object Constraint Language (OCL) are explained in detail.

2.1 Design process of mechatronic systems

A characteristic of mechatronic systems is that several engineering domains are involved in their development. These domains are usually mechanical engineering, electrical engineering, control engineering, and software engineering. So every system that combines mechanical parts with electrical elements and information processing is a mechatronic system. Today these systems can be found everywhere in the form modern washing machines, cars, and airplanes.

Mechatronic systems usually consist of multiple reusable modules called *Mechatronic Function Module* (MFM). These MFMs are designed, created and tested independently from the other components. If a MFM acts autonomously - it reacts and acts self-determinedly - it is called an AMS, in short *Autonomous Mechatronic System*. If two or more AMSs are connected via any kind of communication, this network is called DMS, an acronym for *Distributed Mechatronic System*. For further information on these definitions, see [6, 23].

Figure 2.1 shows an overview of these systems in a real world example. The Railcab\(^1\) [15], that is also the basis for most examples in this thesis, incorporates all three types MFM, AMS and DMS. The Railcab system is a novel transportation system that is currently developed at the University of Paderborn. Part of this

\(^1\)Neue Bahntechnik Paderborn http://www-nbp.uni-paderborn.de/
system are the so called *Railcabs* - autonomous vehicles that are able to make decisions on their own. In this hierarchy the MFM modules like Drive-and-Braking, Energy-supply, and Track-guidance build the foundation of an AMS. These parts give the Railcab the ability of self-optimization with inherent intelligence. It can independently and flexible change operating conditions. One advantage of the Railcab system is the ability to create a convoy. For example are the Railcabs able to detect each other and exchange information about direction, current speed and other important data needed to form a convoy. Based in these information, the Railcabs decide on their own whether to create a convoy or not. This interaction and communication is exactly the definition of DMS. Some more details about the Railcabs possible decisions, their behavior, and internal components will be given in the concepts chapter 3.

![Figure 2.1: Hierarchy of MFM, AMS, and DMS](image)

Accompanying the immense possibilities are also the problems in the design of these systems. The complexity in the development of these system can be much higher than in a single domain. Interaction and self-optimizing mechatronic systems result in many new problems that have to be solved. These can range from shared resources and possible conflicts when accessing them, to basic communication issues. A very important source of errors and design issues lies in the separated design processes of the involved domains. An explanation of these
problems and how to cope them is given in the following section.

### 2.1.1 Development process

The design of self-optimizing mechatronic systems introduces some new challenges for the development process. Many aspects have to be specified including not only structure, but also goals for self-optimization, behavior of this intelligent systems, situations/scenarios the system must handle and many more. Therefore guidelines like ’VDI 2206’ [22] are not sufficient any more.

For the special purpose of designing mechatronic systems, a new development process is being developed [8, 9, 7]. It is an advancement to the normal mechanical development process and incorporating the other three domains software-, electrical-, and control engineering. Figure 2.2 shows the ’VDI Guideline 2206’, a directive for the design methodology for mechatronic systems. On the left side of this ’V’ is the system design. This is the phase where all domains work together on the single, cross-domain design document, the principle solution.

The conception stage is partitioned into four phases (see illustration 2.3) - ’Task Analysis’, ’Conceptual design on system level’, ’Conceptual design on module level’, and ’Concept Integration’ - each with their own outcome design documents.

![Figure 2.2: VDI-Guideline 2206 'Design methodology for mechatronic systems'.](image)
In the Task Analysis phase, the desired functions, requirements, and tasks of the future system are investigated. In addition to this, scenarios are collected which are used in the second phase. The Task Analysis phase is also used to specify the environment of the mechatronic system. The second phase called 'Conceptual design on system level' is of most interest for this thesis. As shown in figure 2.3 this phase consists of three steps. In the first step the initial function hierarchy is specified. It is a hierarchical breakdown of the application functions, covering conventional and self-optimizing functionality. Step two uses the scenarios identified in phase one and represents the application-scenario driven development part of this design process.

In each scenario, the necessary functions are determined. Previously defined functions can be reused and new ones are added to or update the function hierarchy. This information is combined with additional information like for example structure, internal behavior or shape into elements. The reused elements are called Solution elements and new created elements are denoted as System elements. The resulting structure of system and solution elements can be modeled with the Mechatronic Modelling Language and results in the active structure. Also specified in each scenario is the systems behavior. This includes activities, states and transitions of the system as well as the communication and cooperation with other systems or subsystems. The analysis of the behavior gives a first idea of the internal optimization processes, so that finally the 'System of Objectives' can be defined. Last step of the 'Conceptual design on system level' phase is the integration of all scenarios, resulting in the system level principle solution.

If the phases three and four have been completed, the final principle solution document is available. As soon as the final version of the principle solution is
approved and the conception phase has ended, the domains start their own, more
detailed design independently from the other domains. Is their development com-
plete, the different designs are merged again during the systemintegration phase,
creating the complete specification for the final product.

2.1.2 Mechatronic Modelling Language

The principle solution consists of a set of models, which allow different views on
the system under development. In the context of the CRC 614, they are often
referred to as 'the partial models', however, in the following I will refer to this
language as the Mechatronic Modelling Language (MML). This modelling lan-
guage is currently developed by the CRC and captures the relationships between
these models. Figure 2.4 shows an overview of the eight models: Requirements,
Environment, System of Objectives, (application) Scenarios, Functions, active
structure, Shape, and Behavior.

![Diagram of partial models](image-url)

Figure 2.4: Principle solution overview showing the seven connected partial mod-
els
• **Requirements**: This is the representation of the requirements, like values for height, width, maximum speed and so on.

• **Environment**: This model describes the environment of the system and its integration into the environment. Influences, possible interference factors and interactions with the system are identified here.

• **System of Objectives**: It is the representation of objectives and their relations. These could be for example objectives like driving comfort, travel time, deterioration, reliability and their relation to each other.

• **Scenarios**: Application scenarios describe small parts of the functionality of the system that is being developed. They specify how the system should behave in a particular state and situation.

• **Functions**: It is a hierarchical breakdown of application functions and used to define the functionality of the system. They can be conventional or for self-optimization purposes.

• **Active Structure**: Here are the selected system elements, their properties and relationships illustrated. The aim is to describe the basic structure of the self-optimizing system, including all anticipated system configurations.

• **Shape**: This model includes data/values about shape, location, alignment, sizes of surfaces and many more. This model is often accompanied by CAD models.

• **Behavior**: This group includes different types of behavior. Basically, the system states with the associated process flow and the state transitions with the underlying adaption processes is modeled here. The adaption or adjustment processes describe the actual implementation of the self-optimization. Behavior represents a group, due to the fact that there exist many types of behavior. These could be for example be the behavior for a logic circuit, dynamic behavior of multi-body systems, electromagnetic compatibility or interaction behavior.

For this thesis are the three partial models *Active Structure, Behavior* and *Scenarios* of importance. The *Active Structure* model defines all objects that are needed for modeling the behavior. *Behavior* in combination with the *Scenarios* specify how the system will react in certain situations.

### 2.2 Sequence Diagrams

This subsection gives a short overview of the message sequence diagrams that are used in this thesis. Section 2.2.1 introduces the basic concepts, elements
and constructs of message sequence diagrams. Based on this, the Life Sequence Charts (LSC) are explained. The last kind of sequence diagram described in this foundations chapter is the Modal Sequence Diagram (MSD), which is an extension to LSCs.

Please note that in some diagrams the arrows representing the messages have also a dot at the beginning. This dot denotes the send event of a message.

### 2.2.1 UML Sequence Diagram

A sequence diagram is an interaction diagram in UML, that shows how objects communicate with each other. This is specified with an ordered sequence of messages send between them. The current version of the OMG\textsuperscript{2} specification is UML 2.1 \[19\].

Sequence diagrams are used to model the communication between objects. The possible flow of control throughout the processes are described in two dimensions: horizontal are the involved active objects and vertical the ordering of time. Lifelines represent these involved active objects, and come in two kinds. Actual lifelines map a specific object to a lifeline and symbolic lifelines are placeholder for objects of the given type.

Messages are send from one lifeline to another and are represented by horizontal arrows. The only invariant is that sending of a message must occur before the reception of that message. The sending and receiving of a message are called event. In the sequence diagrams, we specify the occurrences of an event on a lifeline. An event is a point in time and has no duration. A trace is a sequence of events ordered in time and describes the history of message-exchange corresponding to a system run. Messages that have no duration between sending and receiving event are called synchronous. The opposite type, asynchronous messages, are not instantaneously transmitted.

A gate a special kind of an event that is used to model the passing of information between a sequence diagram and its context. They are fixed on the diagram borders and can have incoming or outgoing messages. Also included in the UML 2.0 specification are constructs like alternatives, loops, parallels, asserts, and more. The description of these elements is postponed to the Modal Sequence Diagram section.

Figure 2.5 shows a simple UML sequence diagram with three lifelines representing two Railcabs and a TrackControl. Message getRCSOnTrack is the first message that is being send from rc1 to the tc TrackControl, followed by the sending of notifyRailcab to rc2. After this message is received, the second Railcab sends the registerAtRailcab to rc1. Activation boxes, or method-call boxes, are the opaque rectangles drawn on top of lifelines to represent that processes

\textsuperscript{2}Object Management Group – www.omg.org
are being performed in response to the message. The asynchronous message `notifyRailcab`, indicated by the half-arrow and the bend, meaning is is not instantaneously transmitted. Finally, the X at the end of lifeline `tc:TrackControl` denotes the end of the instance lifetime.

### 2.2.2 Live Sequence Charts LSC

Damm and Harel proposed [5] in 1998 an extended version of the UML sequence diagram, called Live Sequence Diagrams (LSC). An important advantage over UML is that LSCs allow the distinction between possible behavior, necessary behavior, and a new concept to specify forbidden scenarios.

One deficiency of UML sequence diagrams is their inability to enforce progress, meaning if a sequence diagram has reached a specific message, this message must be sent/processed. Otherwise this would violate the modeled behavior.

To overcome this drawback, *temperatures* are introduced. A message has a temperature that can either be *hot* or *cold* (indicated with a blue color and dashed lines for cold, red and solid lines for hot). A hot message means the progress is enforced, whereas a cold temperature means the message may be lost.

In order to make statements about the state of the system, *boolean conditions* referring to attributes or data items of the involved entities are used. Conditions also come in two variants hot (mandatory) and cold (possible). A hot condition must be satisfied, or else an error occurs. Cold conditions do not generate an error when they are not evaluated true, but force an exit from the enclosing LSC. A condition is evaluated if all previous events been reached/visited. Hence a condition that covers multiple lifelines is kind of a synchronization point in the
diagram, because all previous events must be reached before the condition can be checked. The following table gives a short summary of these definitions.

<table>
<thead>
<tr>
<th>Element</th>
<th>hot</th>
<th>cold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>message must be processed</td>
<td>message can be lost</td>
</tr>
<tr>
<td>Condition</td>
<td>condition must be satisfied</td>
<td>non-satisfied condition exits LSC</td>
</tr>
</tbody>
</table>

Events, like SendEvent or ReceiveEvent, are attached to locations which are points on an instance axis. A cut is a mapping of every instance to one of its possible locations in the LSC. They represent the borderline between already visited locations and those which still have to be considered. The elements directly below the cut are those, which are currently enabled. The definition for a hot/cold cut is: An LSC cut is hot if one of its locations is hot and is cold otherwise.

LSC diagrams come in two views. *Existential* LSC specify scenarios that may happen if a condition comes true. The can be used to specify systems that should be monitored, but do not affect the execution.

An *universal* LSC forces the system to satisfy the scenario if the given condition is evaluated true. These universal LSCs can be activated if a specific prefix or history of messages has occurred, which is specified in the pre-chart. However, the message sequence of the pre-chart is not required to hold in the system, but must be observed before activating the actual LSC, called *main-chart*.

Figure 2.6 illustrates a basic LSC. Messages getRCsOnTrack and notifyRailcab are in the prechart, denoted by the dashed hexagon. If the sending of these two messages is observed, the mainchart is activated. First, the registerAtRailcab is send from rc2 to rc1. Followed by this is a condition, checking the value rc1.isConvoyAvailable. If this condition is fulfilled, requestConvoy is sent.

LSCs do also allow the instantiation of multiple diagrams. If two or more LSCs have the same pre-chart, they all will be created if the pre-chart occurs. Each diagram can be instantiated several times too, based on the triggering pre-charts. This actually allows an infinite recursive instantiation if the pre-chart of a diagram is also modeled in the mainchart. In addition to this, LSCs can be synchronized by observing the same message. This circumstance is shown in figure 2.7. Two instantiated LSCs observing different messages, but have getNextTrackControl in common. A trace that satisfies both LSCs would be registerRailcab, register-Complete, sendTrackInfo, updateTrackData, and getNextTrackControl. At this point, both charts would be synchronized.

If messages do not occur in the order they are specified, a violation is thrown. There are two kinds of violations: hot and cold - depending on the temperature of
the current cut. A cold violation leads to a simple discarding of the current LSC. In contrast, a hot violation indicates an error and inconsistency. This applies also for conditions.

**LSC Play-out**

The introduced elements and semantics of LSC allow to simulate and/or check a behavior in form of a given message trace. First, several diagrams are created, each modelling behavior for a different part of the whole system. This set of LSCs is then used to check if a specific order of messages is valid or not. An order that is not valid causes a hot violation and the termination of the simulation. The process is called play-out, because the previously collected behavior specifications are played out immediately when their prechart matches the trace.

The following examples describe this process. I used only abstract names because of space restrictions. Assume there exist multiple diagrams and three messages will be executed in the order msgA, msgB, msgC. In step 1 (2.8) msgA is a minimal event for two LSCs, so both are instantiated. The objects a of type A and b of type B are bound to the lifelines representing A and B. In step 2 (2.9) both diagrams advance their cuts after msgB is send. No violations are thrown due to the still correct behavior definitions.

Step 3 (2.10) raises a cold violation. Message msgC is sent from a to b. For
2.2 Sequence Diagrams

Figure 2.7: Synchronization across Live Sequence Charts using the message getNextTrackControl.

scenario1_inst this works out fine, but scenario2_inst expected the message msgA. The resulting cold violation discard the scenario2_inst and the simulation continuous. In this example message msgC is also a minimal event for scenario3, so here again a new LSCs is being instantiated.

Additional information and a good overview about LSCs and the play-out can be found in the book by Harel and Marelly [14] and [4, 5, 11, 18].

2.2.3 Modal Sequence Diagram MSD

This type of sequence diagrams has been introduced by David Harel and Shahar Maoz in [13]. MSDs extend LSCs and allow the specification of traces that are possible, mandatory and forbidden. MSDs also remove the need for precharts/maincharts and allow a formal interpretation for UML sequence diagrams. Although MSD is very close to UML 2.0 sequence diagrams as they are described in the standard documents, they define it more precisely [2, 12].

To distinguish lifelines representing the environment from the ones for system internals, a new boolean attribute isEnvironment has been introduced. Additionally, a distinction between messages that should be executed or just be observed was needed. For this, a second attribute execution mode has been introduced. The two possible values are executed and monitored.

The assert is the condition element of the MSDs. Because MSD extends the
Figure 2.8: Step 1: Message msgA forces two ActivePNConfiguration instantiations.

Figure 2.9: Step 2: Message B is executed in both scenarios.

UML 2.0 specification, this naming for conditions has been carried over. Figure 2.11 shows a simple assert, just checking if the attribute trackIsFree is true. To represent alternative traces in a diagram, the alternative element can be used. It consists of one or more alternative parts representing the different cases. Each part can have a guard/condition that has to be satisfied to enter the according part. In figure 2.12 is an alternative used with one condition and an else part, which is entered if no other condition could be satisfied.

In Modal Sequence Diagrams the prechart construct from LSCs is dropped. Instead, a more general approach is used by David Harel and Shahar Maoz, where cold fragments inside universal interactions serve prechart-like purposes: a cold fragment does not have to be satisfied in all runs. But if it is satisfied, it enables the satisfaction of its subsequent hot fragment; and this is true in all runs.

Also a new way to initialize the diagrams is the use of a minimal event. Each MSD contains exactly one event at the beginning that triggers the instantiation similar to the LSC pre-charts. After this instantiation messages are observed to identify possible hot/cold violations.
2.3 S2A and AspectJ

Important to note is that LSCs and MSDs can be represented as automata. The idea is to use possible cuts of the LSC as states of the automaton and a transition for each legal occurrence specification. Two important end states are the hot and cold end states. If the automaton stops these states, a violation has occurred. More information on the conversion process can be found in [12, 18].

2.3 S2A and AspectJ

S2A\(^3\) is a compiler that translates Modal Sequence Diagrams into AspectJ code. This AspectJ code in combination with a play-out controller is used to simulate the behavior specified in these MSDs. S2A is designed and programmed by Asaf Kleinbort and Shahar Maoz. The compilation scheme is presented in the papers/articles [18, 10, 2].

\(^3\)S2A compiler http://www.wisdom.weizmann.ac.il/~maoz/s2a
The idea is to specify parts of the program behavior with the MSDs and then use the existing Java code together with the Java aspects to get a fully functioning program. The generated aspects are used to observe and control the method invocation of the existing code. This requires besides the Java code, a tool to specify the MSDs, the S2A compiler to generate the aspects, and finally an AspectJ compiler to translate the aspects to Java bytecode.

To run a program with the aspects controlling the behavior, first the necessary Java objects (like GUIs) must be created. After that, Asaf Kleinbort and Shahar Maoz use these objects as lifelines to create the MSDs. In these diagrams the actual object methods are the messages send between the lifelines and conditions refer to object attributes. Finally, if all interactions been modeled, the S2A compiler takes the MSDs and generates the aspects and other needed classes to run the program.

In section 2.2.2 the transformation from LSCs to automatons is described. These automatons are now represented by ScenarioAspects, listening for the relevant messages with Pointcuts and reacting to those with execution of the Advices. These are used to advance the states of the automaton and thereby being able to check if the current message was allowed to execute or violates any hot/cold conditions. This check is performed by the play-out logic that is also generated by the S2A compiler.

To trigger the instantiation of ScenarioAspects the minimal event is used. Every time during the runtime of the application a minimal event is send, a new instantiation, called activeMSD, is created. ActiveMSD store the current state of the automaton and to which objects the lifelines a bound.

To distinguish messages that should be executed by the controller from the ones
that should only be monitored, the previously mentioned attribute *execution* is used. If a message is set to *monitored*, the S2A controller just waits until this message is eventually executed by the given Java code. In case of an *executed* message, the play-out logic has the right to execute the message on its own. This process is repeated until no more executable messages are left. The play-out then waits for new induced messages from environment. The evaluation of conditions followed by choosing and sending a message is called *step*. All play-out initiated steps that are in between two environment messages, are combined to a so called *super-step* [11].

A vital part of this simulation is the binding of objects to lifelines. The S2A playout-controller distinguishes between two types of lifelines, *environment* and *system*, and therefore the MSD attribute *isEnvironment* is needed. An environment lifeline is assigned by the engineer, modelling the diagram. It signals the playout-controller that an object bound to this lifeline will be given by an observed message. The binding of system lifelines is more complex. If a message is marked as *monitored*, the controller waits for the program to choose the right callee object, which can then be bound. But in case the message marked as executed, the controller is in charge to find the appropriate object to bind to the target lifeline. For this search, the special MSDenv class must be implemented which returns the object for a given lifeline. This class is not generated by the S2A compiler.

**AspectJ**

AspectJ is an aspect-oriented extension for the Java programming language. In this very short introduction to AspectJ, I will focus only on the important parts needed for the S2A compiler, the *Pointcut* and *Advice*.

The *Pointcut* describes a point in the execution of a program, or preciser a specific method call during the execution. For example the snippet 2.13 shows the pointcut *logEndOfTrack*. It listens for the execution of a method with the signature *Railcab.endOfTrack(..)*, meaning no return type (*), callee must be of class *Railcab* and the method name is *endOfTrack*, regardless which parameters are given (...). If such a pointcut is defined, one or more *Advices* can be attached to it. The advice shown in 2.14 is attached to the pointcut *logEndOfTrack* and just writes a message to the console. This particularly kind of advice is called

```java
pointcut logEndOfTrack (): execution (* Railcab.endOfTrack(..));
```

Figure 2.13: AspectJ pointcut code snippet.
after() returning, which obviously starts directly after the method defined in the pointcut is executed. Other possible kinds of Advices are before or around. It is to mention, that in the advice body any Java code can be executed. The

```java
after () returning: logEndOfTrack () {
    System.out.println("logEndOfTrack has been executed");
}
```

Figure 2.14: AspectJ advice code snippet

pointcuts and Advices are stored in the equivalent to .Java files, the .aspects which must be compiled by an AspectJ compiler. The resulting bytecode can be executed by a standard Java interpreter.

2.4 Other tools and technologies

2.4.1 Activestructure-Editor

The Activestructure-Editor is currently being developed at the University of Paderborn. With this editor the active structure can be modeled. The long-term goal is it to create multiple editors for the Mechatronic Modelling Language as well. An advantage would be that each editor could use the same metamodel and thereby enable a cross-referencing between the editors. For example could the structure be modeled with the Activestructure-editor and later be used in the beha viormodel-editor to specify the behavior with sequence diagrams. Screenshot 2.15 shows the current development version of the Activestructure-Editor. The editor views left (freeform diagram) and right (tree-editor) show the same model. A Systemelement Railcab containing a RadioSystem and an EnergyModule connected with a TrackControl via an InformationFlow. This InformationFlow is relayed by the InformationFlowPorts.

The Activestructure-Editor is not a standalone application but consists of several plugins for eclipse. Eclipse is an open source platform and has a plugin model that allows an easy extension to its basic features. Important plugins like EMF, GMF, and UML2 are described in more detail.

2.4.2 EMF & GMF

EMF is a set of eclipse plugins that allow the creation of metamodels and from this the generation of model, edit and editor code. EMF provides a user interface to first create a metamodel and based on this the ability to generate the

---

4http://www.eclipse.org
2.4 Other tools and technologies

Figure 2.15: A screenshot showing the current development status of the Activestructure-Editor.

according model classes and editors. The modeled objects are stored in the XML Metadata Interchange (XMI) [20] which allow interchangeability with other tools. Upon this basis, the Graphical Editor Framework\(^5\) (GMF) is able to generate the appropriate GUIs for the editor. With both frameworks, application development can be significantly accelerated.

EMF and its tools is used for many tasks in this thesis. A very good example for the possibilities of this framework: The Petri net model that is explained in section 4.1 was taken from http://www.pnml.org/, where an ecore model made available. This model is used as a basis from which the modalPetrinet.ecore inherits. From this extended model, the Java classes were generated, without writing a single line of code. If changes on the model are necessary, a re-generation combines selfwritten code and generated code.

\(^5\)Graphical Editor Framework website http://www.eclipse.org/gmf
2 Foundations

2.4.3 UML2 project

The UML2 project\(^6\) aims are to provide a fully realization of the UML 2 standard and the exemplary tools for developing models based on those metamodels. With the help of ecore models and EMF + GMF their editors for viewing and editing UML models already support class, component, state machine, and activity diagrams. Of course, the ecore metamodel can be extended and therefore reused in new plugin-projects or other editors. The models created by the UML2 project editors are stored in XMI \([20]\), which is already a common and well used format.

2.4.4 OCL

The Object Constraint Language is used to describe rules that apply to any metamodels and models. The rules are formalized in a precise text language and can be used to set constraints or create object query expressions. OCL can be used to specify invariants of objects and pre- and post conditions of operations, making UML (class) diagrams more precise (using vocabulary of UML class diagrams). They add the ability to access attributes and navigate through UML class diagrams and, even more important for this thesis, through instantiated EMF models. OCL supports collections, special operations, filters, quantifiers, iterations, and many more constructs. For this work, only simple conditions and the navigation through the class model are used.

The code snippet 2.16 checks two conditions: first if railcab1 and the passingRailcab have the same trackcontrol object (isEqual). The second condition is a simple check if the attribute speed of the railcab1 is smaller than 100.

\[
\text{railcab1.currentTrackControl} = \text{passingRailcab.currentTrackControl} \\
\text{AND railcab1.speed} < 100
\]

Figure 2.16: OCL code snippet for simple condition checks.

\(^6\)http://www.eclipse.org/uml2
3 Concepts

This chapter introduces concepts how the system behavior can be modeled in a scenario-based way. I also explain in which way this approach is connected to the existing modeling techniques. Therefore a basic example which includes multiple scenarios is given. These scenarios are then modeled using Modal sequence diagrams. Based on this, a Tool-Vision explains the desired process and functionality of this modeling approach. It is also explained how a seamless integration in the existing concepts can be achieved. One of the desired features is a simulation of the previously specified behavior, which enables the analysis of the system during the conceptual design.

Section Modelling starts this chapter and shows the modeling of multiple scenarios for a mechatronic system with Modal Sequence Diagrams. Following this is the Tool-Vision (3.2) explaining the desired modeling techniques and tool functionality for this scenario-based development approach. Afterwards Scenario-Playout (3.3) details the prerequisites for a behavior simulation. Section 3.4 ObjectBinding investigates some problems that occurs when binding symbolic lifelines in a dynamic object system. To conclude this chapter, I propose ideas and possible solutions how Object refinement (3.5), the detailing of the objects innerworkings, can be achieved.

3.1 Modelling

It is desired to model (self-optimizing) mechatronic systems based on the design process currently development by the CRC 614. During the 'Conceptual design on system level' phase, in each scenario new Functions, Structure and Behavior are created and existing ones reused. They are closely related to each other, therefore a common metamodel and modelling language should allow to cross-reference between the different partial models. For example would this allow the use of components defined in the active structure as lifelines in the behavior specification.

In this section, I assume that the 'Task Analysis' phase is completed and the relevant scenarios are identified. The basis structure of the example is given and specified in an active structure diagram. It is modeled with the Mechatronic Modelling Language and is accompanied by the function hierarchy as described
in section 2.1.2. Based on these scenarios, I will specify the behavior with the use of Modal Sequence Diagrams. The MSDs use the components defined in the active structure as lifelines. Hence, the communication between the components can be easily modeled by just drawing the messages.

The Railcab-system consists of many individual systems and subsystems, but in this example only a few will be used. These systems are primarily Railcabs, tracks, switches, and TrackControls. The Railcab is the core of this system. The autonomous vehicles are able to solve problems on their own. This means that they are able to optimize themselves depending on the situation. These decisions could for example be, how to optimize the driving mode for the next track section or should a convoy be created with the railcab in front.

Railcabs need TrackControls along the tracks. These TrackControls are important for the Railcabs linear drive to work, because it needs precise information of its current position. Each TrackControl is responsible for one track and can perform additional tasks. Some of the tasks include managing the Railcabs that are currently on their track, queueing cabs that want to enter, or just store information about the state of the track. With these information the railcabs are able to autonomously decide which driving mode (fast, energy saving, comfort and so on) to use. To make this example simpler, I assume that each TrackControl can only manage one track and knows only its previous and next TrackControl. SwitchControl is kind of a TrackControl. It is a system for managing switches, which has a slightly more complex Railcab management and more previous/next associations. The example in which all scenarios take part is relatively small.

Figure 3.1: Track plan for this example

Figure 3.1 shows a track plan in a more abstract form. There are two Railcabs
on the route which consists of a ring of tracks and a shortcut that is branched by two SwitchTracks. The individual track sections are separated from each other by the lines with blue dots.

For this thesis it was not possible to use the Mechatronic Modelling Language to model the structure, so an UML based model is used instead. Due to the fact that the current MML version is also based on UML, an UML representation of the active structure elements is not a problem. The metamodel, that I use to create the examples basis structure, is called Railcab. An overview of this metamodel is given in figure 3.2 and shows the used classes, attributes and associations.

The modeled basic structure consists of the 2x Railcabs, 2x Switches, 2x Switch-

![Figure 3.2: Railcab metamodel created for this example](image)

Track, 13x Track, 13x TrackControl and 2x SwitchControl. This corresponds to the schematic trackplan given in figure 3.1.

Scenario overview

For the Railcab project already numerous application-scenarios have been identified and defined on index cards. The basic content of these cards consists of Name, short description of the scenario, creation/modification date, rough description of the environment, objectives/goals and their priorisation, as well as a sketch of the scenario. In this thesis, only a simple description of the scenarios is given. Additional information like objective priorities are not of concern for the modelling.

For this example, four basic scenarios have been identified:

...
1. **Railcab switches track**: An important and regularly used scenario, specifying a Railcab switching its current track. This scenario is also used for an object refinement, introducing the internal subcomponents of the Railcab.

2. **Detect Railcab - create convoy**: A Railcab that detects the proximity of another one - communication takes place to negotiate a possible convoy mode.

3. **Enter switch**: This scenario is similar to 'Railcab switches track', but with an improved Railcab management.

4. **Disband convoy**: The counterpart to the detect Railcab scenario. A convoy should be disbanded.

Despite that the scenarios are all independently modeled, they might overlap in some degree with each other. These intersections represent the cross-scenario specification, and possible inconsistencies can be revealed during the simulation. Figure 3.3 also shows the 'Detect track end' circle which is not a scenario, but fits perfectly to show that it can be used by both scenarios. Each of them is now described in detail and the according behavior modeled with Modal Sequence Diagrams.

The following scenarios are modeled based on the MSD and LSC specification.

### 3.1.1 Scenario: Railcab switches track

Since the first scenario shows many of the most important characteristics and semantics, it is described in more detail than the other scenarios. The end of a track has been reached and the Railcab must initiate steps to switch to the next
track section. On a high abstraction level, the individual steps are shown that must be executed to achieve this track change. Later on, the internal components of the Railcab are specified.

In order for a Railcab to leave its current track and switch to the next, it is necessary to register at the TrackControl that is responsible for the upcoming track-section. Because the Railcab does not know the new TrackControl, it has to query its current one for this additional information. Finally a registration follows, allowing the Railcab to request the entering of the next track. The scenario ends with the un-registration at the Railcabs current TrackControl.

But there also exists the possibility that the route is blocked by an overthrown tree (alternative 'Track blocked'). In this case the new TrackControl sends a requestDenied message, forcing the Railcab to stop the switching process immediately. A second alternative for this example is a high utilization of the next track (alternative 'Track full'). The TrackControl recognizes this problem and sends a signal to the Railcab that entering the track is currently not possible. Additionally, an estimated time for a free slot is sent to the Railcab, which could adjust its speed to save energy. After a short period of time, the requestEnter signal is again sent to the TrackControl.

Figure 3.4 shows an object model, similar to a collaboration diagram, for the current situation. The upper half represents a schematic of this scene. Included are two Tracks with their TrackControls and the Railcab that wants to switch. The vertical dashed line at the end of the first track indicate a trigger for this scenario - the railcab will reach the track end. In the lower half are the involved classes. Between the TrackControls and Tracks is the association next which is later used to navigate through the model. The Railcab uses onTrack to find the currently used track and currentTrackControl to get the TrackControl it
is registered at. The dashed lines represent the state after successfully switching the tracks.

The modal sequence diagram 3.5 is modeled as follows. The first message after initialization is `getNextTrackControl`. This message and almost all following ones are asynchronous messages, because communication between TrackControl and Railcab is carried out wireless. The Railcab has to query its current TrackControl to addresses the one responsible for the next track. Then it has to register (`registerRailcab`) itself at the new Trackcontrol and send a `requestEnter`. The next message is marked as cold, because this is the point in the scenario where the two alternatives can continue the process. I like to compare this with a predetermined breaking point. The next step is to allow the Railcab to enter the track (`allowEnter` and `enterNewTrack` messages). To complete the scenario, the Railcab unregisters itself from its old TrackControl (`unregisterTrackControl`).

Figure 3.5: 'Railcab switches track': Assumed MSD representing the behavior of the 'main' or successful trace.

The result of a successful `main diagram` execution is shown in figure 3.6. It is
similar to the one at the beginning of this scenario, but here the associations have been moved. The diagram shows now the railcab on the next track.

![Diagram showing railcab on the next track](image.png)

**Figure 3.6: 'Railcab switches track':** The datamodel after a successful execution of the main diagram.

Very important is the fact that the mentioned predetermined-breaking-point message is marked with the temperature cold. This is the crux of this scenario and allows multiple endings without creating errors or semantic inconsistencies. But before I go into more detail, the other two alternatives MSDs must be presented.

**Alternative 1** shown in figure 3.7, named 'Track blocked', starts with the execution of message `requestEnter`, denoting it the minimal event for this diagram. The request to enter for the railcab is denied (`requestDenied`), forcing it to stop immediately.

**Alternative 2** also denies the request to enter but sends an `sendETA` (expected time of arrival/expected time of availability). The railcab alters its speed accordingly (`adjustSpeed`), sets a timer, and sends the `requestEnter` message again. This scenario is named 'Track full'.

During a play-out, the message `requestEnter` triggers an instantiation of both alternative MSDs. Depending on the answer of the new TrackControl (allowEnter, requestDenied, sendETA) only one of the diagrams will remain active. The other two must be discarded due to a cold violation.

Assume that the next track is already at its maximum railcab capacity. When the track end is reached, the main scenario is instantiated and all messages will be executed without any problems up to the `requestEnter` 'predetermined-breaking-point'. Here, the simulation creates two more diagram instantiations, namely alternative 'Track blocked' and 'Track full'. Due to the fact that the new TrackControl can not handle any more Railcabs, it sends the `sendETA` message. This causes ColdViolations in the main and alternative 'Track blocked' diagrams and
3 Concepts

Figure 3.7: 'Railcab switches track': The two, small alternative endings to the main diagram.

leads to their termination. Only 'Track full' remains and the simulation proceeds normally. Figure 3.8 is a visualization of this process. By using this modelling technique, an engineer who creates such a behavior diagram is now able to specify alternative endings. Of course it would also be possible to use an alternative, but this approach allows a separate modelling of these diagrams. Beside this, other scenarios may use the alternative endings as well. The set of endings can be extended or reduced by simply adding or removing diagrams - and all without the need of modifying existing ones.

Figure 3.8: 'Railcab switches track': Visualization of the different endings for the case of an 'track full' end. Cold violations discard the remaining two diagrams.
3.1 Modelling

Refinement: Detect track end

An engineer who has already modeled the first scenario at a high abstraction level might want to bring more detail into the behavior. It is desired to model an object refinement independently from the scenario, which allows the re-use of the refined behavior in other scenarios. This refinement is done by defining the inner workings of the Railcab and their behavior.

This refinement can be interpreted like a zoom into the diagram and the creation of a new MSD for the included subcomponents. Messages send and received by the Railcab are indicated by incoming and outgoing *gates*. Gates allow messages to that start or end at the border of the diagram, and can be an easy way to model the passing of information between a sequence diagram and its context. In section 3.5 'Hierarchical refinement', two ways to model such a refinement are explained.

For this scenario the Railcabs internal subcomponents 'Distance Measurement Unit' (short DMU) in combination with a 'Track-End-Coordinator' (TEC) and a RadioSystem are used. The DMU is a sensor that scans the rail bumpers for a marking that indicates the end of the track. If such a bumper is discovered, the Track-End-Coordinator is informed. This component is responsible for getting information about the next track, requesting the next TrackControl and many more. The third new subcomponent is a RadioSystem to communicate with other TrackControls or Railcabs. The according Active Structure is modeled in figure 3.9 and includes also a modified TrackControl.

The following two MSDs show only the needed lifelines & messages to save some space and to focus on the important parts. The complete diagrams are given in the appendix. The first diagram 3.10 is a modified version of the 'Railcab switches track' scenario. I added a dashed, gray rectangle to indicate the lifeline/object I want to refine. The second diagram 3.11 represents the refinement of the Railcab.
Figure 3.10: 'Detect track end': MSD from scenario 'Railcab switches track' that is used to refine the Railcab, indicated by the scenario name above it.

The original incoming and outgoing messages are relayed by the gates at the diagram borders.

### 3.1.2 Scenario: Detect Railcab - create convoy

This scenario is about the recognition of other nearby Railcabs and the question whether to form a convoy or not.

The Railcabs can detect each other via a short-range radio. This radio system is also used to exchange information between them.

The behavior modeled in figure 3.12 shows such an information exchange. The first Railcab approaches and recognizes the radio signals of a second Railcab in its proximity. It sends a signal to this Railcab and introduces itself. Triggered by this, the detected Railcab sends its current information on speed, target destination, modes of operation, etc. to its finder. In addition, the callee Railcab asks for the same information in return. Based on these information, the caller Railcab decides whether it should form a convoy or not.

The according Modal Sequence Diagram reflects this communication in detail. The diagram is initialized with the message `detectRailcabProximity`. When the message is received, the Railcabs exchange their information by sending the `sendCruiseInformation` message. In this example the assert condition
3.1 Modelling

Figure 3.11: 'Detect track end': Behavior of the internal components of the Railcab modeled in this refinement diagram.

\[ \neg \text{rc.convoyMode} \land \text{rc1.direction} = \text{rc.direction} \]

checks only two simple conditions. First, if Railcab 1 is already in a convoy, and second, if both Railcabs are in the same driving mode.

3.1.3 Scenario: Enter switch

This scenario is quite similar to the first 'Railcab switches track' one. Here the Railcab enters not a new track, but a switch. Due to the splitting and joining of tracks, the SwitchControl has to carefully instruct the entering Railcabs. Two diagrams are created to model the behavior for this scenario. The first one is the recognition of the track end, where also the requestEnterSwitch send to the SwitchControl. The second diagram represents the re-request if previous request have been denied. Both diagrams are given in the appendix.

Diagram A.1 shows the first part of the modeled behavior. The Railcab is informed about the track end with the message \text{endOfTrack}. Similar to the 'Railcab switches track' scenario the current and next TrackControl/SwitchControl are identified. After this a requestEnterSwitch is send. If the switch is busy, it sends sendETA, an approximated time, to the railcab. The Railcabs reaction to this message is modeled in the second diagram A.2 of this scenario. In case the SwitchControl allows entrance to the next track, a similar trace to one given in the 'Railcab switches track' main diagram is executed.
Figure 3.12: 'Detect Railcab - create convoy': Simple message trace completed through an alternative fragment.

To keep the scenarios and diagrams simple, the MSD specifying the Railcabs waiting to enter the switch, consist only of three elements. The first element is the sendETA message send by the SwitchControl. Then a timer is started. After the time has passed, a new requestEnter message is send. The timer can of course be replaced by more complicated actions, as for example slowing down the Railcab to save energy.

3.1.4 Scenario: Disband convoy

The initial situation for this scenario is an already formed convoy with two or more Railcabs. One of the Railcabs has to leave the convoy. A switch is the only possibility for a Railcab to leave the current track. A station is not directly attached to the main track, but separated from it by a switch.

If a Railcab wants to leave the convoy, an adequate distance to the predecessor and successor Railcab must be established. In case the Railcab is the leader of the convoy, only the predecessor has to be informed. If the Railcab is the last
shuttle in the convoy, its successor has to be informed. In the last case both Railcabs. The resulting MSD for this scenario is modeled in diagram A.3. The diagram is triggered by listening to the allowEnter message, which is sent from a SwitchControl to the Railcab. If the Railcab wants to leave the track, the condition in the next assert must be evaluated true. Next, one of the following alternative-parts is entered. It is checked whether the Railcab is leader of the convoy, at the end or somewhere in between. Each alternative-part informs, depending on this, its predecessor and successor Railcabs. After the disbandNotifications has been sent, the Railcab prepares to leave the track. It reduces its speed to create the necessary distance.

3.2 Tool vision

The goal is to analyze the created models as early as possible. One way to analyze a design is to simulate it, which can be realized as follows. First the Functions, the Active Structure, and the Behavior are modeled for each scenario independently. The MML metamodel (2.1.2) allows us to capture the relationship between the structure and behavior. This provides the ability to cross-reference elements. In section Modelling I presented this behavior modelling process for some scenarios.

A scenario-playout (see section 2.2.2) is used to simulate the behavior that is modeled independently for each scenario with these diagrams. The play-out should allow an engineer to simulate a modeled system by

- using previously defined structure elements to create an example environment for simulating the static and dynamic parts.
- start, observe, and influence the behavior simulation by inducing environment events

A possible use case can look like the following. An engineer working on one or more scenarios uses the defined structures to generate the needed classes to model an example system. After creating the test system the developer can execute a simulation by inducing external events into the system. This permits him to take the role of the environment. Inducing external events can of course be automated by defining test cases in form of event sequences. Based on the previously defined Modal Sequence Diagrams, the play-out engine starts the simulation. The components are dynamically bound to the lifelines, allowing a flexible change of the instance structure. During this simulation the engineer can observe the communication between the components. A message trace and changes directly at the instantiated example objects can be used to analyze the behavior. Discovered errors and violations can indicate contradictions in the design. It is also possible to interact with the play-out, by forcing the play-outs strategy to prefer some message.
The task of this thesis was to investigate how to model and simulate the Modal Sequence Diagrams in the context of the tool vision sketched above. I first investigated two other existing play-out approaches. The S2A compiler, which I introduced in section 2.3, was not suited for our purposes, because it needs fully functioning (implemented) code to execute. In addition to this a second compiler is necessary to compile the AspectJ code into bytecode. To support a dynamic object system, a special controller has to be manually created to bound objects to lifelines according to the current context. This controller has to be modified for each change in the object model. S2A is specifically designed for Modal Sequence Diagrams. If a change is necessary, like adding additional elements or constructs, the play-out engine can not be modified easily.

The second available simulation tool is the Play-Engine from Harel and Marelly[14]. An integration of this standalone tool into the existing eclipse plugins is very difficult. The objects system of this approach would complicate the realization of dynamic structures and component systems.

Because of this I decided to develop a new eclipse based simulation plugin.

The concepts and tool features which I introduce in the following are the basis for the vision given above. I developed concepts to play-out the created diagrams and thereby being able to simulate the systems behavior. For this, the already presented Modal Sequence Diagrams need to be modified. I also provide an elegant way to cope the dynamic object system and the resulting binding problems. To be able to integrate my simulation approach into the existing tool suite, all simulation related concepts are based on the eclipse platform. This enables the use of plugins like EMF to generate model and editor code easily. My idea is to be able to simulate the modeled behavior right after specification, by using a system/environment to test with.

The toolsuite will allow an engineer to model the different aspects of each scenario and afterwards analyze the system by simulating it.

3.3 Scenario-Playout

In the following I explain the concepts of the play-out I developed. It introduces the interpretation of MSDs as modified Petri nets and their new elements. After this, each section covers a different part of the simulation, which concepts are described independently. This way important topics, like initialization and violations, are introduced step by step. To complete the play-out, the previously presented concepts are combined in the the basic play-out simulation routine.
3.3 Scenario-Playout

3.3.1 Trace semantics of MSDs as a Petri net

The first and most noticeable difference to S2A is that I don’t use automaton to represent the Modal Sequence Diagrams. The conversion process maps a possible trace described by an MSD to an extended Petri net. This transformed Petri net has some advantage over its LSC source, because model checking can be performed more easily [3, 1]. Additional, but minor advantages are that parallel events lead to complicated states in an automaton, but remain small and recognizable in a Petri net. Events that have more than one dependency to be enabled, can be expressed better. For example it is possible that a receive event has a predecessor on its lifeline and a sending event on a different one. Both preceding events must be enabled, before the receive event can be fired. This kind of behavior can be modeled very well with Petri nets.

Here I use Petri nets and extend them by additional attributes like temperature which is also known from LSCs. I denote this kind of Petri net as modalPetrinet. It is based on existing approaches converting sequence diagrams to Petri nets like [17, 16, 1]. The basic idea is to translate each event in a MSD to a transition in a Petri net. All the MSD elements that are currently needed for the simulation can be converted in this way.

Each lifeline of a sequence diagram is covered by MessageOccurrenceSpecifications. These MessageOccurrenceSpecifications represent events which can be SendEvent, ReceiveEvent or CombinedFragmentEvent that I explain in the following. SendEvent and ReceiveEvent are the ends of a message. CombinedFragmentEvents are different, because they can actually cover multiple lifelines. The two types of CombinedFragement used in this thesis are AlternativeCombinedFragement and AssertCombinedFragement, which are both described in the foundations chapter. More information about these elements can be found in OMGs UML specification [19]. An important fact of sequence diagrams is, that all the elements on a lifeline are ordered. With this information it is possible to insert arcs and places to recreate the lifeline ordering using the according transitions. The relationship of send and receive events can also be modeled this way, regardless of the lifeline they are covering.

Figure 3.13 and 3.14 show such a conversion. On the left side is a simple sequence diagram including three lifelines, four messages and an assert CombinedFragment. The numbers next to the events and transitions are added to better identify the related elements in both diagrams. The Petri net visualizes the fact that assert events can also be used to synchronize the lifelines. At the top and bottom of the Petri net are so called BoundaryEvents. They help to identify the start and end of a diagram.
There are additional properties that can not be found in normal Petri nets. Each transition has two more attributes taken from their sequence diagram event counterparts. First one is temperature and the possible values hot/cold and second one is execution with executed/monitored. Both attributes have the same semantics used in the S2A approach and their use will be explained in sections 3.3.3 and 3.3.6. MessageEvents include also the messagename plus messagekind attribute, and a reference to the lifeline their event covered in the original MSD diagram. CombinedFragments can cover multiple lifelines, so this attribute is a reference list. Alternative events are in need to store their alternative parts with their conditions.

In this thesis I do not use the term lifeline in the context of Petri nets any more. Lifelines represent objects that participate in a sequence diagram, so I denote their modal Petri net counterparts as participants.

3.3.2 Initialization

If communication between objects should be monitored by the play-out, the S2A approach creates an activeMSD that is responsible for this object pair. The activeMSD represents a Modal Sequence Diagram that has been initiated for this particular object context. This instantiation is similar for the modal Petri nets.
Each Petri net has also a triggering minimal event. If the play-out recognizes this event, the according modal Petri net is determined. Instead of creating a new instance of the diagram, a new configuration is initialized. A configuration is used to store the current cut and to keep keep track of the binding. The cut is represented by set of places that currently hold at least one token. In a MSD, objects are bound to a lifelines. The modal Petri nets on the other hand, manages a list of participants and their bound objects. To check which object is bound to an event or vice versa, the play-out just needs to get the participant and lookup the object.

The configurations are called `ActivePNConfiguration` and represent the counterparts to the S2A’s activeMSD.

### 3.3.3 Hot/Cold Violations

Checking for violations in modal Petri nets is different to the check S2A uses. In the S2A approach the controller uses a state automaton to represent the MSD as described in section 2.2.3 ‘Modal Sequence Diagrams’. In addition to the normal end state are the Hot-Violation and Cold-Violation states. If the automaton stops in these special states a violation has occurred.

For Petri nets exists an adequate way to find violations with the use of cuts. To recall, the definition of a hot/cold cut is: “A cut is hot if one of its locations is hot and is cold otherwise [18]. The cut is advanced by firing a transition and thereby enabling the following, depending transitions. A cut in a MSD is represented by a set of places containing tokens in a modal Petri net. Figure 3.15 illustrates this representation. The MSD has a cut through the message `setNextTC`, after event three but before event four.

With the additional transition attribute temperature, I can now check for a cold or hot violation by inspecting the places of a cut, respectively their next transitions. The following steps take only effect if the diagram is already instantiated and the participants are bound. Similar to the S2A approach, a message sent between two objects is checked if it is used anywhere else in the diagram. This is the equivalent to the set $M$, containing all messages of the whole diagram. If the message is not found in the set $M$, the check ends without a violation. Otherwise one of the following cases is executed:

- **Message available**: all tokens to fire the transitions for the message are available. The message is executed regardless of the cuts temperature. This advances the cut, consuming tokens and moving them to the transitions outgoing places. No violation is thrown.

- **Message not available**, the cut is cold: one or more tokens are missing and
the event can not be fired. Because the current cut is cold a ColdViolation is thrown.

- Message not available, the cut is hot: one or more tokens are missing and the event can not be fired. Because the current cut is hot a HotViolation is thrown.

### 3.3.4 Hot/Cold Combined Fragments

A minor topic but non the less needed for modelling scenarios are the temperatures for combined fragments. Not just messages can have the attribute hot and cold, but also asserts and alternatives. If an assert element is cold and the evaluation of the condition returns false, a cold violation is thrown - in case of a hot temperature a hot violation. The same applies to alternatives. If all conditions of the alternative parts evaluate to false and no else exists, the whole alternative fragment is skipped.

An alternative represented by a petrinet is illustrated in figure 3.16. The alternative can only be executed if all predecessor places have tokens. This is the case, when the message \texttt{getNextTrackControl} is fired. The following green and yellow transitions represent the two alternative parts. These parts come, like the alternative, in pairs to enclose their containing transitions.

### 3.3.5 Asynchronous messages

To be able to represent asynchronous messages in a Petri net is an important issue. Due to the strict differentiation between send and receive transitions, merely the
3.3 Scenario-Playout

The execution of a message is slightly different. The rules for executing both kinds of messages are:

- **Synchronous messages** can only be send if both (send and receive) transitions can be fired.

- **Asynchronous messages** only need tokens in every place that will be consumed by the sender, respectively receive transition.

It is also valid to model the send and receive event of an asynchronous message with different temperatures. A good example for this is a wireless communication. A message must be sent (hot) but is not necessarily received at the other end (cold). This opens new ways to model behavior and can be used to create a diagram dedicated for wireless communication. Everytime such a communication is initiated, this diagram is triggered and a it must be decided, if the message should be altered, dropped, or (perfectly) transmitted.

3.3.6 Message selection and Playout-Strategies

In the introduction to the modal Petri nets, the additional attribute execution has been mentioned. A message with the attribute set to *monitored* will not be chosen by the simulation engine for execution. Therefore is a *executed* message added to a pool of possible next messages. The S2A controller sorts messages in...
This leads to the **PlayoutStrategies** and the **PlayoutStrategyManager**. A Playout-Strategy is the component of the simulation that chooses the next message that should be executed. This selection can be based on rules, a predefined order or just a random pick. The task of the PlayoutStrategyManager is to choose the strategy that fits best for the current situation.

### 3.3.7 Playout

Now, that all simulation related concepts are introduced, they can be combined in the scenario-playout. I now explain the scenario play-out based on the following pseudo code.

```
TRIGGEREVENT(sender, receiver, messagename)
1     instantiate new ActivePNConfigurations
2         fire MessageEvent
3     while MessageEvent available
4         do execute non-MessageEvents
5         choose new MessageEvent
6         if next MessageEvent is minimal
7         then instantiate new ActivePNConfigurations
8         fire MessageEvent
```

The simulation starts with the invocation of `triggerEvent`. This method has three parameters: the sender object, the receiver object and the MessageEvent. First, all ActivePNConfiguration have to be instantiated followed by firing the message. Afterwards the non-MessageEvents are executed, like condition evaluations and detection of a diagram boundary. After that, a new message is chosen and executed. If this message is a minimal event for a diagram, the according ActivePNConfigurations will be instantiated before the execution. This process is repeated until no more messages are available that can be played-out. The details to these operations are given now.

`instantiate new ActivePNConfigurations` is responsible for instantiation new ActivePNConfigurations. If the send message is a minimal event in a MSD, the according modal Petri net is determined. For this Petri net a new ActivePNConfiguration is created and the initial places added to the current cut. Because a message can only be send if sender and receiver objects are given, these both objects can be used for the binding in the new ActivePNConfiguration. The newly created ActivePNConfigurations is added to a list containing all other currently
3.4 Object Binding

This section is about how the objects are bound to the lifeline in a dynamic object model. Binding is used to map a lifeline to a specific object, respectively a participant to an Active Structure solution or system element. In a dynamic object system, binding is more complicated, because it is not possible to refer to a specific (static) object. The fixed sub-components of a Railcab represent a static object system. A simulation on a higher abstraction level with several, moving Railcabs and TrackControls is an example for a dynamic object model. In an dynamic environment, each lifeline is must be bound to an object depending on its current context. If a lifeline can not be bound it is not possible to use it to send or receive messages.

Next, I introduce two types of binding that I named 'Conventional Binding' and 'Extended Binding'.

3.4.1 Symbolic lifelines

An important characteristic of this approach is the use of only symbolic lifelines. Despite the fact that all lifelines are named, these names only represent identifiers like variable names.

As previously mentioned are both, static and dynamic objects systems used. To be able to cope with both types, all lifelines are set to symbolic. This allows the
developer to design the diagram like a pattern, matching the current situation. Now lifelines can, even if differently named, reference the same object. It would not be possible to use the extended binding or the other OCL condition checks without this representation.

In case a non-symbolic lifeline is needed, this can be easily achieved by using an appropriate OCL query to bind the desired lifeline always to the same target object.

### 3.4.2 Conventional Binding

Conventional Binding is used at the creation of an ActivePNConfiguration. The minimal message that triggered the instantiation is send between already bound lifelines. This information can be used to bind the first lifelines in the new ActivePNConfiguration. Conventional Binding occurs only if minimal messages are send, all other lifelines are bound to objects with the Extended Binding. Messages that are induced by the environment always contain sender and receiver objects.

### 3.4.3 Extended Binding

At least one lifeline is bound during the creation of an ActivePNConfiguration. The problem is how to map the remaining lifelines to their expected components/elements if there exist no additional information how the lifelines are related to each other. An example for this is a diagram with three lifelines. At initialization, the first two lifelines are bound, because sender and receiver objects were given by the minimal message. The third lifeline, which is somehow related to the other two lifelines, must also be bound. To find the appropriate object for a lifeline I developed a new concept named extended binding. It is a combination of modified assert events and the Object Constraint Language (OCL).

Each ActivePNConfiguration manages its binding in the form of Lifeline-Object pairs. A lifeline has the attributes `representsClass` to reference the required type, and `name` which is the lifelines identifier. With these information it is possible to create the *OCL Environment*, representing a set of name-to-object mappings. For each already bound Lifeline-Object pair, a variable in the OCL environment is created using the lifelines name and the bound object. Now OCL expressions can be used to navigate through the instances and query attributes and references. These queries are used to return an object and to bound it to its according lifeline. To use this kind of binding in MSDs easily during modelling, an assert event with a modified condition is used. The query expression to find the value or reference
3.4 Object Binding

is contained in the condition of the assert.

The following example illustrates this process. Figure 3.17 shows three lifelines \( rc1 \) of type \( \text{Railcab} \), \( tc1 \) and \( tc2 \) of type \( \text{TrackControl} \). The minimal event bound \( rc1 \) and \( tc1 \) with the conventional binding. Before the simulation is able to send the message \( \text{msgB} \) to the lifeline \( tc2 \) it is necessary to bind it to an object. The assert event with the condition \( tc2 := tc1.\text{nextTrackControl} \) is used for this purpose. The query uses the \( rc1 \) variable to navigate to the first \( \text{TrackControl} \), and from there to the target \( \text{TrackControl} \).

![Diagram](image)

Figure 3.17: Example of an assert fragment with OCL binding and the according part of the metamodel.

The S2A approach uses pointcuts to identify the sending and receiving objects. To enable a flexible binding for a dynamic object system and to send messages to objects that are still not bound, S2A queries the \( \text{MSDenv} \) class. This class returns the next object to bind. However it must be implemented for each application and be modified for every new object type that is added. This is of course not a desired way to simulate the modeled system behavior.

3.4.4 Binding problems

An important question remains - when to bind the lifeline to the object. A binding is necessary before a message can be sent to the target object. But this is not early enough, as the next example illustrates. In figure 3.18 two ActivePNConfigurations for \textit{scenario1} and \textit{scenario2} are shown. On the left side has \textit{scenario1} recently been instantiated and \( rc1, tc1 \) bound. The last lifeline of type \( \text{TrackControl} \) has still no assigned object, which is indicated by the questionmark as the variable name.
If the binding of `?:TrackControl` follows after messages `msgA` and `msgB`, a potential hot violation is missed. This can happen if in scenario2 the TrackControl `tc2` sends a message to `rc1`, and the objects are the same in both diagrams. If the last remaining variable `?` is then bound to `tc2`, a hot violation should have occurred at `msgB`, but has not. To avoid these late binding problems, all lifelines must be bound during creation of the diagram instantiation. A binding as late as possible is not allowed any more. With these restrictions it is now regardless where in the diagram the bindings are defined as long as they are all processed during initialization. Whether to use the binding assert fragments before the first use, or right at the top of the diagram remains a question of visual appealing - as long as all lifelines get bound.

Another idea how this binding can be achieved is to use an external list, attached to the diagram called 'Object Navigation Constraints' (see illustration 3.19). This way no additional elements in the diagram are needed. The ONC-list is represented by a directed acyclic graph with the top elements bound by the conventional binding. The order of this binding is indicated by the directed graph. In the figure, `track1:Track` and `rc1:Railcab` represent these elements. They are also connected with an arc with rectangles at both ends to indicate this conventional binding. The other lifelines are bound by using existing associations or OCL queries like the assert bindings. It is not necessary to add variable names here any more. This could improve the readability in more complex scenarios. For this thesis, I continue to use the modified asserts in order to show the binding in the diagrams.

One last issue that needs to be mentioned is the **NULL-binding**. Scenario 'Disband convoy' (3.1.4) indicates, or reveals such a problem. If the Railcab that wants to

![Diagram](image-url)
leave the convoy also represents the first or last shuttle, then either predecessor or successor are not existing. This implicates that the lifelines can not be bound (or bound to a null-object), but could be used during the execution. Therefore a similar semantic to the hot/cold violations should be added, allowing to set the temperature attribute to lifelines, too. For this thesis I continue to handle null-bindings like hot violations. However, is this a topic that needs to be investigated.

3.4.5 Assignment

The current concept and implementation has a small drawback for simulating complex scenarios. The advantage to use code generated directly from the metamodel creates also a flaw in the design. Since methods are generated without method body no assignments or other operations are performed. This means that sending and receiving of messages can be indicated, but no changes to the underlying objects are made. This is the reason to design an assignment assert fragment - to assign new values and references.

All needed techniques and prerequisites are already introduced, so all that is left is the definition of such an element. With the exceptional features EMF provides, combined with OCL it is possible to use assignment statements as shown in 3.20.
3 Concepts

To prevent a misunderstanding - it is of course not allowed to change lifeline variables, only object values and references.

3.5 Hierarchical refinement

Object refinement is used to specify the details of components and system elements. During simulation it allows the engineer to get a more detailed view of a specific objects behavior. To reflect the design process in the 'Conceptual design on module level' phase, an incremental refinement of components and diagrams should be possible. For a Railcab, this design process starts on the level of the distributed mechatronic system (DMS II). At this level it is for example specified, how transport requests are generated or how the shuttle transports people and goods from one location to the other. At the next level (DMS I), it is modeled how the Railcab has to operate in certain situations, e.g. when forming a convoy, switching a track, detecting an obstacle, and so on. Based on this level, sub-components like Sensors, Communication modules, Energy-supply and their interactions are derived. Scenario 'Railcab switches track' (3.1.1) gives an idea how such a refinement could look like. In the following I denote a diagram that is used to refine an object or lifeline as refinement diagram. The diagram that contains the object or lifeline that is refined is denoted in this thesis parent diagram.

There is a differentiation between two types of refinement. The first way is to create a new diagram that represents a whole lifeline of the parent diagram. This is illustrated in figure 3.21 which shows a simplified version of the 'Railcab switches track' scenario. The Railcab lifeline in the upper DMS I level is refined by a scenario modeled in the lower, module/components level. Gates at the diagram border are used to model the passing of information between the sub-components and its context. This way, the original message endOfTrack send from the Track to the Railcab is carried over to the internal components. The refinement diagram is connected to its parent diagram lifeline in a 1:1 relation, so there exists no other refinement for it. This modelling technique to specify behavior of sub-components is introduced by Atir, Harel,Kleinborg, and Maoz in [2]. The S2A compiler makes use of the 1:1 relationship to combine all refinement diagrams into their parent diagrams and then to compile it to Sce-
3.5 Hierarchical refinement

Figure 3.21: The MSD Railcab_decomp_1 is the refinement diagram for the upper rc1:Railcab lifeline.

...narioAspects. This kind of refinement I call lifeline refinement, due to its relation to a specific lifeline.

The second way to model the behavior of subcomponents is not to refine a lifeline. Instead, multiple refinement diagrams can be created for one object, resulting in a 1:n relationship between object and diagrams. This allows the developer to model different parts of the behavior like he would usually do. Alternative endings and cross-diagram synchronization with messages can be achieved with this kind of refinement. Most important is the fact that the refinement diagrams can be used in other parent diagrams as well. This technique I call object refinement. Figure 3.22 shows a high level diagram with Track, TrackControl, and Railcab lifelines. For the TrackControl and Railcab also multiple refinement diagrams are created (e.g. Railcab_decomp_1). If a message is send to either TrackControl or Railcab, these MSD are initialized like normal diagrams with the minimal event. On the contrary to lifeline refinement may the resulting simulation of the Railcabs internal components lead to other outgoing messages than modeled in...
Figure 3.22: High level diagram and two sets of refinement diagrams for Track-Control and Railcab.

the parent diagram. Therefore it is necessary to play-out all refinement diagrams first, and then return to execute the main message trace. A strict differentiation between these the hierarchy level is needed. This can for example be realized by a dedicated play-out for each level.

Both refinement techniques are still not fully investigated. Open questions remain how to handle cold and hot violations in such hierarchies, or how to separate the play-out. Some of these topics are addressed in [2].
4 Implementation

In this chapter I present two main topics. The conversion from UML sequence diagram to a Petri net structure and the simulation plugin. Section 4.1 'Modal Petri nets' explains why Petri nets are used for the simulation, how the resulting data structure looks like, and the advantages and disadvantages of such a representation. After this, the basic steps how the UML elements like Messages, Alternative Combined Fragments, and Assert Combined Fragments are transformed into their equivalent Petri net analog is explained.

In section 4.2 the simulation-plugin is introduced. To conclude the implementation chapter, all major components be listed with a short description and the functionality of the observer, the core of the simulation, is introduced.

4.1 Modal Petri nets

In this work the representation by Petri nets preferred, not like the MSD automaton used by the S2A approach 2.2.3. This has the advantage that parallel events lead to complicated states in an automaton, remain easily recognizable in a Petri net. Additionally, events that have more than one dependency to fire, can be expressed better. For example is it possible that a receive event has a predecessor on its lifeline and a sending event on a different one. Both preceding events must be enabled, before the receive event can be fired. This kind of behavior can be modeled very well with Petri nets. There already exist different approaches to convert a sequence sequence diagram to a Petri net, like [17, 16, 1]. The implementation of the conversion process described in section 4.1.2 is based on these approaches.

Choosing a Petri net as a basis for the simulation datastructure, induce the decision to create a new metamodel or to reuse an existing one. Due to the fact that only minor extensions to a basic Petri net were necessary for a simulation, an existing metamodel was chosen. A free implementation of Petri Net Markup Language (PNML) form of an ecore file was taken from the pnml.org reference site\(^1\). This metamodel is the basis of my new model.

The eclipse EMF plugins provide graphical editors to create new metamodels and easy ways to extend existing ones. With these plugins the new SimulationPN metamodel could be created easily and efficiently. One advantage of using EMF

\(^1\)www.pnml.org
4 Implementation

and ecore models is the ability to generate code (interfaces and implementations) from it. This helped to speed up the implementation process and to prevent errors. As a bonus, a graphical presentation of the diagrams can now be generated as simple as the creation of the metamodel, with tools like the GMF\textsuperscript{2} plugin.

The following two subsection will now detail the model elements and the conversion from UML sequence diagrams to SimulationPN Petri nets.

4.1.1 Petri net elements

The simulationPN metamodel extend the pnml.org metamodel. Therefore it is possible to re-use most of the elements like e.g. the places and arcs. Here I only explain the new elements that were added to achieve the new requirements.

- **TransitionEvent**: With only a few exceptions all new elements inherit from the original TransitionNode of the pnml.ecore model. Since all of our key elements from the sequence diagram are events, it was best to use one common point for the inheritance hierarchy. The EventTransition inherits from the TransitionNode, splitting the hierarchy-tree in messages and combined fragments. At the lower end of the MessageEvent hierarchy we distinguish between MessageSendEvent and MessageReceiveEvent. Combined Fragments are always represented as a pair consisting of a Start-and EndEvent. An example for this structure would be an AssertEventPair with the members AssertStartEvent and AssertEndEvent. Alternatives include a member pair for each alternative part, too.

- **SimulationPN**: This element corresponds to the sequence diagram and inherits directly from pnml.org Petri net. It provides additional methods to provide faster access to specific objects like boundary events and contains additional attributes.

- **ActivePNConfiguration**: The ActivePNConfiguration is used as an instantiation of a SimulationPN and is the equivalent to the activeMSD used in the S2A approach (2.3). This class stores the current cut as a reference list and most important, the binding from EObject to Participant. Also additional methods for violation checks, event availability, and other methods are implemented here, too.

- **Participant**: This class just represents a lifeline from the original UML sequence diagram. A Participant object is part of a SimulationPN and referenced in multiple ActivePNConfigurations and events that cover this Participant.

\textsuperscript{2}http://www.eclipse.org/gmf/
4.1.2 Conversion

For the conversion from UML sequence diagrams to modalPetrinet, an approach similar to the one described in ‘Transformation from live sequence charts to colored Petri nets’ [17] is used. In a rough overview I will give an understanding of the transformation process and after that a small example.

Again, only Lifelines, MessageEvents and CombinedFragments are of interest, so I focus in this description only on these elements. The basic idea is to handle each event, whether CombinedFragment or MessageEvent, as a transition. The events covering a lifeline are ordered. This order is represented by creating places and arcs in an appropriate way to set the transitions in the same dependency relation as the events before. Each sequence diagram in an UML file is separately converted, generating one equivalent Petri net. This Petri net starts and ends with DiagramBoundaryEvents which have the functionality to let the simulation algorithm know when the end of the sequence diagram is reached and that the current instantiation can be discarded.

In Petri nets, the ordering of events on the same Lifeline is represented by predecessor and successor relationships. With the use of transitions it is now also possible to represent messages by connecting the corresponding sender-receiver pairs. Figure 4.1 shows on the left side two lifelines and a message send from A to B. On the right side is the equivalent Petri net. In addition to the ordering induced by the lifeline is the SendEvent-ReceiveEvent dependency added. The white transitions at the top and bottom represent the BoundaryEvents for this diagram. For the CombinedFragments things are a bit different. These special elements can cover more than one lifeline and so they generally have more predecessors/successors. Like the normal TransitionEvents all predecessors must have at least one token to enable the CombinedFragment-Transition. In figure 4.2 such a CombinedFragment can be seen. For this example it is not filled with content.
and the BoundaryEvents are missing, because only the predecessor and successor relations are of interest here.

Figure 4.2: Illustration how a CombinedFragment is integrated into a Petri net.

As previously described CombinedFragments are always represented by event pairs, bounding their contained fragments. This makes the distinction whether a transition is in an CombinedFragment more easier. Therefore an AssertEventPair consists of an Condition, an AssertEndEvent and AssertStartEvent. The same applies to Alternatives, whereas they can contain one or multiple Alternative-parts that are also represented by AlternativeEventPairs. This structure is illustrated in figure 4.3.

Figure 4.3: Structure of CombinedFragments showing the start-end-pairs.

The next figure 4.4 shows a small, but complete sequence diagram and the equivalent Petri net. The First message msgA is send from A to B, followed directly by
an AlternativeFragment. This Alternative consists of Condition1 with a message send from A to B and vice versa. The else part contains a self-invocation on A. In the end, the final message msgE is send.

The Petri net on the right side shows how the DiagramBoundaryEvents at the top and bottom surround the elements contained in the diagram. After the two transitions 1 and 2, the AlternativeEventPair labelled with number 3 is created. The two AlternativeConditionPairs (condition1 and else) are not connected to the AlternativeEventPair with places and arcs like other events, because these events can not be considered separately. Just like the DiagramBoundaryEvents, each ConditionEventPair can contain transitions or other CombinedFragments.

Figure 4.4: Small, complete example of a conversion from sequence diagram to equivalent Petri net.
4.2 Simulator components

The simulation is distributed on three eclipse plugins (dependencies are not listed here).

- **de.upb.swt.msdsimulation.petrinet.modal**: Contains the model for the SimulationPN and all generated interfaces and modified implementations.

- **de.upb.swt.scenarios.simulationextension**: This is the heart of the simulation and currently contains the observer for the play-out, Menu-Extensions, the sequence diagram to petrinet converter, violations and the play-out strategies.

- **org.pnml.core**: The original petrinet model in form of an.ecore file taken from pnml.org. All other files are generated with EMF and have not been modified.

The core component is the PetrinetSimulationObserver which is instantiated after the conversion process of the UML file to one or more simulationPNs. Now I explain the functions of the BindingManager, PoolManager, PlayoutStrategy(s), and how the observer operates.

**BindingManager**: This class is responsible for all ActivePNConfigurations. Everytime the Observer detects a minimal event and a new ActivePNConfiguration is created, this class stores them in an easy way to access them. With references similar to a hashmap, it is possible to get fast access to all ActivePNConfigurations that are currently bound to a specific EObject. A schematic of this class is shown in figure 4.5.

**PoolManager**: All converted SimulationPNs are stored in this class along with additional information like participant count. The mapping of minimal event to their occurring SimulationPNs is realized similar to the mapping in the BindingManager. A schematic diagram for the PoolManager can be seen in 4.6.

**PlayoutStrategy**: The interface PlayoutStrategy is used to chose the next available message/event. Currently only one strategy is implemented, the RandomBestPlayoutStrategy. This strategy sorts the available messages into violation categories (NoViolation, ColdViolation, HotViolation) and chooses one message randomly from the preferred NoViolation category. If that category contains no messages, the category ColdViolation is used and after that from the HotViolation category. There also exists a PlayoutStrategyManager which is a collection for all created strategies.
Observer: The Observer combines all the previously mentioned classes from the modalPetrinet metamodel plus the Binding-, Pool- and PlayoutStrategyManager. It is the core of the simulation and responsible for the execution of tasks like evaluating conditions. The activity diagram 4.7 gives an overview of this process. The primary and only public method of the observer is triggerEvent which is invoked from the simulation menu. The parameters are messagename, sender and receiver. With these information the observer queries the PoolManager for all SimulationPN that are linked to this minimal event. Then for each of these SimulationPN an ActivePNConfiguration is created and registered at the BindingManager. The sender and receiver EObjects are immediately bound to the appropriate Participants. If the binding is complete, the next step is to fire the message.

After this initialization phase, the simulation enters a loop until no more messages are available. First, all available non-message events are executed like AlternativeEventPairs and AssertEventPairs. Then a PlayoutStrategy is requested from the PlayoutStrategyManager which returns a message that has to be executed next. If no more messages are available, the loop ends at this point. Is the new message for some SimulationPN a minimal event, all the associated ActivePNConfigurations are created before the message is fired. Now the loop starts over again with the execution of all non-message events.
Figure 4.7: Activity diagram showing the different operations of the simulation.
5 Conclusions and Future Work

The goal of this bachelor's thesis was to investigate how the discrete behavior of a mechatronic system can be modeled with a scenario-based approach in the early development stages. In addition to this a simulation of the specified behavior should support the analysis of this system.

The development of mechatronic systems, especially self-optimizing systems, is a challenging task. Currently new methods and languages for the development of these systems are being researched by the CRC 614. In self-optimizing mechatronic systems not only the mechanical and electrical engineering is involved, but also the domains software- and control engineering [8].

For these systems, many aspects have to be specified including structure, shape, functions, goals for self-optimization, and behavior. The techniques currently used for behavior specification in the principle solution are not yet sufficient, since they cannot be formally analyzed. The formal analysis of system behavior is important in order to detect errors early in the development stages.

To model the interaction of the environment and system components independently for application scenarios, I used Modal Sequence Diagrams. Based on an example, I showed how the scenario-based behavior specification can be realized. For this example the Railcab, a novel transportation system, was chosen.

Several problems could be identified during the modelling of the scenarios, that required additional concepts for their modeling and simulation. The dynamic object system makes the binding of objects to lifelines more difficult. This is because it is not possible to refer to a specific (static) object. To cope with this problem, I used modified assert elements and Object Navigation Constraints. This allows the binding of lifelines depending on the current context. As a side effect, it is now possible to access object attributes and references by using OCL queries.

Due to the hierarchy induced by components and subcomponents, the behavior specification becomes a challenging task. Issues how to to model the internals of a component without creating inconsistencies with the higher level specifications still have to be solved. Therefore I made use of existing concepts [2] and added ideas that allow the refining of lifelines. Now it is possible to model refinement diagrams which preserve the relation to their high level diagrams. This solution allows the simulation of high level components and the observation of their internal behavior. The editors used to create other MML models are already
developed as eclipse plugins. It also proved difficult to adapt existing simulation engines. Therefore I implemented a plugin that integrates with the existing eclipse modelling tools.

Not all of the above mentioned topics could be fully implemented in time. Some missing LSC/MSD elements like loops and parallels have to be integrated into my implementation. Also the assigning of values and references is incomplete. This restricts the current simulation procedure drastically, since no changes to the model can be made. I showed in section 3.5 Object Refinement how to use refinement diagrams in a simulation. To use this concept, the according changes to the simulation routine must be made.

During the course of elaborating the concepts for playing-out the MSDs, some problems I identified that could not be resolved in the scope of this thesis. The following points are open to be investigated in future work.

To increase the modelling possibilities for engineers, additional constructs or elements could be added. MSD and LSCs allow the use of parallels, loops, and many more. So it has to be investigated if these constructs influence the refinement of diagrams and components. In addition to this is it also necessary to find a corresponding modal Petri net representation.

Is a formal specification language used to model the behavior, the play-out of system messages can be extended. There is a lot of non-determinism in the play-out of MSDs. So, the simulation algorithm has to make a lot of choices. It may happen that the simulation leads to a hot violation that could have been avoided by making another choice in the past. A smart play-out [11] improves the simulation by creating a decision tree. It looks a limited number of steps ahead to avoid choices that lead to invalid path on this tree. To be sure never to run into hot violations, the complete synthesis of the specification is needed [21].

Section 3.5 Hierarchical refinement introduced two methods, object refinement and lifeline refinement. The lifeline refinement replaces a lifeline in a high level diagram by a new created refinement diagram. In this refinement diagram is the behavior of the sub-components modeled. Object refinement is a relation between an object and multiple refinement diagrams, each describing the internal behavior. For both techniques some questions still remain. It should be investigated how to handle cold and hot violations in such hierarchies [2], or how to separate the play-out and the differentiation between simulation levels. It is yet unclear if these refinement techniques can be integrated into my solution and which fits best into the development process.

There are also some additions that may improve my implementation and support an engineer modelling scenarios. Visual representation of the messages send
between components and highlighting of diagram cuts could be integrated, like realized in the play-engine. The current possibilities to influence the simulation are restricted to environment messages like mentioned in [10]. Here, it could be useful to force the play-out to prefer some messages.
5 Conclusions and Future Work
A Scenario diagrams

Scenario: Enter switch

Figure A.1: 'Enter switch': First diagram for this scenario modelling a Railcab entering a switch and requesting entrance.
Figure A.2: 'Enter switch': The re-request send from the Railcab to the Switch-Control.
Scenario: Disband convoy

Figure A.3: 'Disband convoy': MSD showing the different cases that have to be checked for a separation from the convoy.
# List of Figures

1.1 Two Railcabs autonomously formed a convoy to optimize their energy management. .................................................. 3

2.1 Hierarchy of MFM, AMS, and DMS .................................................. 6
2.2 VDI-Guideline 2206 'Design methodology for mechatronic systems'. 7
2.3 The four phases of the conception stage and their resulting documents. ................................................................. 8
2.4 Principle solution overview showing the seven connected partial models .............................................................. 9
2.5 Basic UML sequence diagram ...................................................... 12
2.6 Live Sequence Diagram ............................................................... 14
2.7 Synchronization across Live Sequence Charts using the message getNextTrackControl. ........................................ 15
2.8 Step 1: Message msgA forces two ActivePNCongfiguration instantiations. ............................................................. 16
2.9 Step 2: Message B is executed in both scenarios. ......................... 16
2.10 Step 3: Cold violation caused by msgC discards scenario2 and creates scenario3. .................................................... 17
2.11 After message requestEnter is received, the assert sc.trackIsFree is evaluated. ......................................................... 17
2.12 Alternative used to model two different traces, depending on the isTrackFree boolean. ................................................. 18
2.13 AspectJ pointcut code snippet .................................................... 19
2.14 AspectJ advice code snippet ....................................................... 20
2.15 A screenshot showing the current development status of the Activestructure-Editor. .................................................... 21
2.16 OCL code snippet for simple condition checks. ......................... 22
3.1 Track plan for this example ............................................................ 24
3.2 Railcab metamodel created for this example ................................. 25
3.3 Some of the identified scenarios and how they might overlap. ...... 26
3.4 'Railcab switches track': The datamodel before the scenario is executed. ............................................................... 27
3.5 'Railcab switches track': Assumed MSD representing the behavior of the 'main' or successful trace. .......................... 28
List of Figures

3.6 'Railcab switches track': The datamodel after a successful execution of the main diagram. ........................................ 29
3.7 'Railcab switches track': The two, small alternative endings to the main diagram. .................................................. 30
3.8 'Railcab switches track': Visualization of the different endings for the case of an 'track full' end. Cold violations discard the remaining two diagrams. .................................................. 30
3.9 'Detect track end': New, refined Active Structure of the Railcab showing two subcomponents and the connected TrackControl. .................. 31
3.10 'Detect track end': MSD from scenario 'Railcab switches track' that is used to refine the Railcab, indicated by the scenario name above it. ................................................................. 32
3.11 'Detect track end': Behavior of the internal components of the Railcab modeled in this refinement diagram. ......................... 33
3.12 'Detect Railcab - create convoy': Simple message trace completed through an alternative fragment. .......................... 34
3.13 Simple sequence diagram with assert CombinedFragment .................. 38
3.14 Result of the transformation process from MSD to Petri net ................ 38
3.15 Cut in a MSD and the equivalent Petri net cut, represented by the places (containing tokens). ........................................ 40
3.16 On the left side a MSD with an alternative, on the right side the equivalent representation as a modal Petri net. ................. 41
3.17 Example of an assert fragment with OCL binding and the according part of the metamodel. ........................................ 45
3.18 A potential hot violation is missed due to the late binding of tc2. ...... 46
3.19 Possible external binding solution, the Object Navigation Constraints, for scenario 'Railcab switches track'.......................... 47
3.20 Assigning a new speed value of 400 to the railcab1. .................... 48
3.21 The MSD Railcab_decomp_1 is the refinement diagram for the upper rc1:Railcab lifeline. .................................................. 49
3.22 High level diagram and two sets of refinement diagrams for TrackControl and Railcab. .................................................. 50
4.1 Conversion of one message into the Petri net equivalent. ................. 53
4.2 Illustration how a CombinedFragment is integrated into a Petri net. 54
4.3 Structure of CombinedFragments showing the start-end-pairs. ........ 54
4.4 Small, complete example of a conversion from sequence diagram to equivalent Petri net. ............................................... 55
4.5 BindingManager schematic ............................................. 57
4.6 PoolManager schematic ............................................. 57
4.7 Activity diagram showing the different operations of the simulation. 58
List of Figures

A.1 'Enter switch': First diagram for this scenario modelling a Railcab entering a switch and requesting entrance. .......................... 63
A.2 'Enter switch': The re-request send from the Railcab to the Switch-Control. ............................................................... 64
A.3 'Disband convoy': MSD showing the different cases that have to be checked for a separation from the convoy. ................. 65
List of Figures
Bibliography


Bibliography


[16] Dipl. Inf, Olaf Kluge, Dr. Julia Padberg, Prof Dr, and Hartmut Ehrig. Modeling train control systems: From message sequence charts to petri nets, 2000.


Erklärung der Urheberschaft

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Paderborn, den March 25, 2009

..........................  Jens Frieben