Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Analyzing Scenario-based Specifications

Master's Thesis

v1.0.1 Modified Edition

Author
Benedek Horváth

Advisor
Zoltán Micskei, PhD
András Vörös

January 31, 2018

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Horváth Benedek*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2018. január 31.

Horváth Benedek
hallgató

Kivonat

Az elmúlt évek során az informatikai és azon belül is a biztonságkritikus rendszerek komplexitása rohamosan növekedett. Az összetett rendszerek tervezésében a bonyolultságukból adódóan meghatározó paradigmává vált a modell alapú megközelítés. Ennek során magasszintű modellekből kiindulva származtatjuk a rendszer konfigurációs beállításait, dokumentációt és akár forráskódját is. Így a magasszintű terveken különböző ellenőrzéseket lehet végrehajtani, miközben a folyamatos konzisztenciát megtartva automatikusan generálhatóak a platform-specifikus megvalósítások.

A dolgozatomban a rendszerkomponensek kommunikációjának modellezésére gyakran használt forgatókönyv-alapú specifikációs nyelv formalizmusokat vizsgálom. Ezek előnye, hogy még a konkrét komponensek megvalósítása nélkül, csupán a komponens szintű tervekből kiindulva vizsgálhatóak az egyes kommunikációs szekvenciák. Ezáltal korai fázisban detektálhatóvá válnak az inkonzisztens viselkedések, melyek a specifikációban rejlő hiányosságokra, ellentmondásokra vezethetőek vissza.

A dolgozatomban áttekintem a gyakran használt forgatókönyv-alapú specifikációs nyelveket, kiértékelem a hozzájuk tartozó nyílt forráskódú tervezőszoftvereket is. A szoftvereket két szempontból hasonlítom össze: az általuk támogatott specifikációs nyelv szabványossága, illetve az eszköz által megvalósított analízis és tervezési funkciók tekintetében.

Elosztott, beágyazott, biztonságkritikus rendszereket állapotgép formalizmussal is lehet tervezni. A Gamma állapotgép kompozíciós keretrendszer [14, 15] egy eszköz, amely támogatja a komponens alapú, hierarchikus, reaktív rendszerek tervezését és analízisét. Ennek ellenére az eszközben jelenleg nem lehetséges egy adott komponens adott portjára vonatkozó kommunikációt modellezni és megtervezni.

Ezért dolgozatomban kiegészítem a Gamma keretrendszert egy modell alapú forgatókönyv nyelvvel, a Gamma szkenárió nyelvvel (GSL). A formalizmus a Live Sequence Chartokon alapul. A nyelv absztrakt és konkrét szintaxisát nyílt forráskódú modellezési technológiákkal készítettem el. Ezenkívül definiáltam a GSL formális működési szemantikáját is azáltal, hogy az egyes szkenáriókat véges automatákra képzem le.

Ezután a dolgozatomban bemutatok egy eljárást forgatókönyvek egymással való kompatibilitásának ellenőrzésére. Az eljárás olyan eseménysorozatok keresésén alapul, amelyek ellentmondásos döntésre juttatnak két forgatókönyvet. Azaz az eseménysorozatot az egyes forgatókönyveken szimulálva az egyik elfogadja (érvényesnek jelöli meg), míg a másik elutasítja (érvénytelennek jelöli meg) azt. A validációs eljárás eredményét visszavetítem a forgatókönyv szerkesztőbe azon célból, hogy a tervezőmérnök értesüljön az ellentmondásos definíciókról.

Végül bemutatom a GSL-nek a Gamma keretrendszerrel való integrálhatóságát egy modellvasút esettanulmányon keresztül. Ezenkívül méréseket is végeztem a szkenárió kompatibilitási eljárás futásidejének vizsgálatára.

Abstract

The complexity of IT and safety-critical systems has increased rapidly in recent years. Due to the complexity of the systems in system engineering, the model-based paradigm has become the decisive approach. The goal of the approach is to automatically derive configuration, documentation and implementation from high-level models. In this way different analysis methods can be applied on the high-level models. Finally, consistency can be checked and the platform-specific implementation can be generated automatically.

In the thesis, I analyze the often used scenario-based specification languages for modeling communication of system components. The advantage of these languages is that the communication sequences of different components can be analyzed without actual implementation details, using only the component-level design models. In this way inconsistent behaviors and design flaws can be identified in the early design stages. These flaws are usually due to ambiguous requirements and specification.

Besides the overview of the frequently used scenario-based specification languages, I evaluate the existing tool support for them by comparing the tools from the level of standardization of the modeling language, that is used by the tool, and the analysis and design functionalities the tool offers.

Distributed embedded safety-critical systems can be developed by statechart formalism. Gamma Statechart Composition Framework [14, 15] is a tool which supports the model-driven design and analysis of hierarchical, component-based reactive systems. However, it is not possible to specify the communication through a given port of a component.

Hence I enrich the Gamma Framework with a model-based scenario language: Gamma Scenario Language (GSL). The formalism of GSL is based on Live Sequence Charts. I designed the abstract and concrete syntax of GSL using open-source modeling technologies. I also defined the formal operational semantics of the language by transforming scenarios into finite automata.

I propose and implement a scenario compatibility validation workflow which finds ambiguous traces in scenario definitions and then back-annotates these traces to the scenario editor to notify the engineer about the ambiguities.

Finally, in order to evaluate the applicability and the integrity of GSL to the Gamma Framework, I apply the language on a model railway case study. Then I perform several measurements, in order to get a preliminary overview for the runtime performance of the scenario compatibility validation workflow.

Chapter 1

Introduction

The complexity of software and safety-critical systems has increased rapidly in recent years. In software engineering it led to the increasing number of lines of source code. For example, on an Airbus A380 airplane several million lines of code are responsible for the safety of the passengers. In addition to that, in order to be fault tolerant and handle extra functional requirements, these systems have to be distributed which adds an extra complexity to the system design.

Motivation Therefore in the latest years the model-based approach became an important paradigm in the field of distributed and safety-critical systems. The goal of this approach is to provide methods to develop models and to automatically generate implementation from them, thus eliminating the cost of bugs caused by manual implementation.

Such development methodology is the Y-model [6]. On each level of the Y design method, the design plans and the details of the respective component are contained within the models. In this way the platform-specific implementation, source codes and configuration files, can be automatically derived using code-generators. Besides, from each V&V model different test cases can be generated automatically, which can verify the correctness of the code-generators.

Testing and analyzing the correct behavior of such systems with the conventional approaches usually requires large amount of human effort. However, the model-driven approach can also provide means to apply formal methods that allow the discovery of both design and behavioral errors in an early stage of development.

Another advantage of model-driven development, besides the code generation and the formal reasoning about correctness, is one can design models about every aspect of the system, including the communication between the components.

Specification languages The modeling of communication originates from the telecommunication industry, where the first scenario based specification language, the *Message Sequence Chart* (MSC), was standardized by the International Telecommunications Union (ITU) in 1993 [35]. This language enabled the graphical representation of message exchanges and actions done by entities (instances) participating in the scenario.

Over the years, several other formalisms were derived from MSC. For example, Object Management Group (OMG) [46] standardized *UML Sequence Diagrams*. It extends the MSC with several new elements, thus enabling modeling more complex scenarios. Another formalism, called *Live Sequence Charts (LSC)* was first published in 2001 [8]. LSC was motivated by the fact that MSC and UML Sequence Diagrams cannot easily express either mandatory or optional behavior. Hence LSC introduced *message-level* and *chart-level* modalities. A fourth scenario based language is *Modal Sequence Diagram* (MSD). It

is the combination of LSC with UML Sequence Diagram, because by applying stereotypes it adds modalities to the elements of the UML Sequence Diagrams [24]. Thus the methods which were developed for those formalisms can be also applied to MSD.

There are several open-source modeling tools which support designing scenario-based specifications using the aforementioned languages. I compared several such tools and found that usability, supported functionality and their popularity is changing from one tool to another. Also the maturity of design analysis and validation capabilities these tools have varies on a wide range. The results of the survey is detailed in the further chapters of the Master Thesis.

Context Distributed embedded safety-critical systems can be developed by statechart formalism. *Gamma Statechart Composition Framework* (*Gamma Framework* in short) [14, 15] is a tool which supports the model-driven design and analysis of hierarchical, component-based reactive systems. Automatic transformation of individual statecharts, as well as their composition to formal models, has been developed to support the formal analysis of the design models. Additionally, the framework supports the back-annotation of the analysis results to the compositional language enabling the users to analyze the behavior of their models in a familiar domain.

Although with the *Gamma Framework* the engineer can compose composite models from statecharts and formally prove their correctness, it still lacks a functionality to model and specify communication scenarios between components. More specifically, it is not possible to specify the communication through a given port of a given component.

Language development Hence the aim of my Master Thesis is to enrich the *Gamma Framework* with a model-based scenario language. This scenario language, called *Gamma Scenario Language* (GSL), has on hand practical syntax with modern tooling support to ease its usage within the framework. On the other hand, it has formal semantics to support formal validation of such scenarios. It validates the compatibility of scenario definitions by finding ambiguous traces. Ambiguous traces are such traces that are accepted by one scenario definition but rejected by another one. For implementation I used open-source modeling technologies, such as Eclipse Modeling Framework [52], Xtext [12] and Xtend [4].

Finally, I evaluate the work from two perspectives. First, by introducing the application of GSL on a model railway case study for designing and validating scenarios on a port of a component. Second, by presenting the results for the preliminary runtime performance of the scenario compatibility validation workflow implementation.

The rest of the work is structured as follows:

- Chapter 2 presents the concepts of model-driven engineering, the aforementioned scenario-based specification languages, the *Gamma Framework* and the theoretical concepts behind the formal semantics of GSL scenarios.
- Chapter 3 defines some aspects for analyzing scenario-based specifications and it also compares existing modeling tools for scenario-based specification languages.
- Chapter 4 introduces the decisions that were considered during the design of GSL.
- Chapter 5 introduces GSL in details, including its abstract syntax, concrete syntax and operational semantics, moreover it also details the validation of GSL scenarios.
- Chapter 6 evaluates the work through a case study and preliminary runtime performance measurements.
- Chapter 7 concludes the work and provides ideas for future work.

Chapter 2

Background

System engineering provides solutions for multidisciplinary problems. It includes the business and technical processes, that are necessary to achieve the solutions and to eliminate risks which affect the success of the project. The technical processes include both system specification, design, engineering and the testing, validation and verification of the built system [13, p. 15].

2.1 Model-Driven Systems Engineering

Definition 1 (Model). Model is the simplified representation of the real world. It keeps information that is relevant from the design perspective, and abstracts or simplifies other, irrelevant aspects and perspectives of reality [13]. ▪

Definition 2 (Model-Driven Systems Engineering). Model-Driven Systems Engineering is a formalized application of modeling requirements, design, analysis, verification and validation processes. The primary artifact is the model, that is the chief information container medium [30]. ▪

In model-driven engineering the emphasis is put not on writing detailed electrical documentation but designing a coherent system model and automatically deriving platform-dependent source code and configuration from it. Besides, component- and system-level design models are created, that both contain the necessary information of the respective aspect of the system with the related constraints as well.

The greatest advantage of the model-driven approach is the automatic transformation of the platform-independent models to platform-dependent source code and configuration files using code generators. In this way the consistency of model and the generated code can be satisfied and the bugs from the manual implementation can be eliminated.

Besides source code, documentation can be automatically derived too. In this way it will be maintained along with models, so they will be always consistent and up-to-date.

Model-driven engineering is primarily applied in safety-critical systems, e.g. air traffic control and railway control systems, where the fault tolerant operation is essential. Thus these software and hardware components of such systems should meet strict standards, e.g. DO-178C [48], DO-278A [49], DO-331 [50], EN-50128 [36]. The EN-50128 standard contains the model-driven and formal verification rules that should be applied in the engineering of avionics systems.

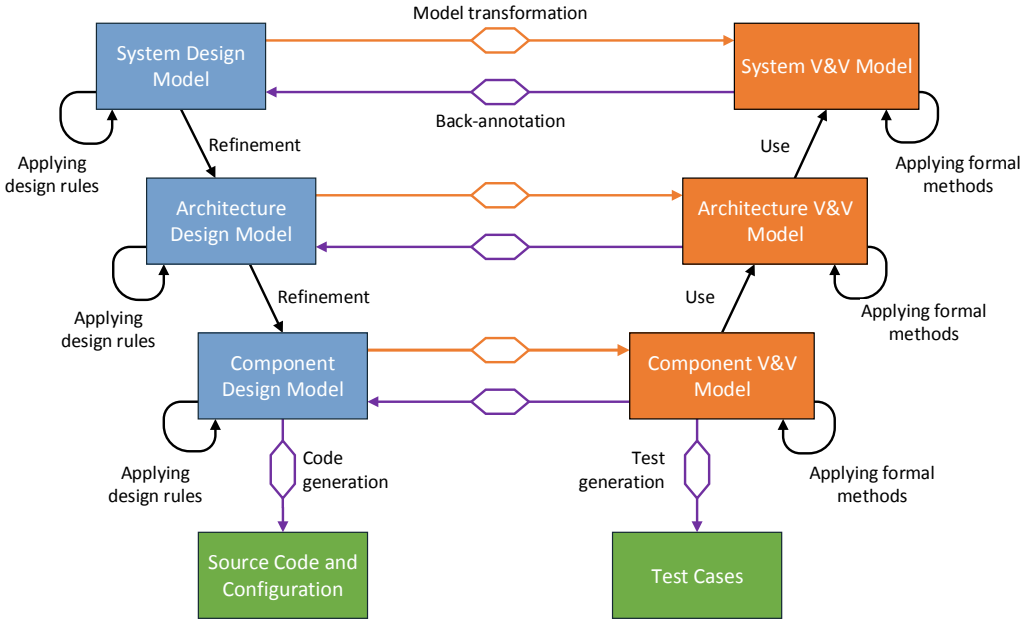


Figure 2.1: Y-Model of Model-Driven Systems Engineering [43]

2.2 Y-Model in Model-Driven Systems Engineering

The model-driven approach is usually applied in the engineering of safety-critical systems where the validation and verification of the system is essential in all design phases. The system, along with its components, should be compliant with the requirements and should fulfill additional formal and informal safety rules. In order to help engineers design such systems, the formerly used V Systems Engineering Process model [51, p. 40] has been extended into a Y-model [6] depicted in Figure 2.1.

The Y-model is similar to the former V-model, because in order to fulfill the complex requirements against the system, a hierarchical design process is elaborated. First, the high-level requirements are collected, refined and analyzed. Then, the System Design Model is elaborated from the requirement analysis artifacts. It contains the construction of different architectures that build-up the entire system which conforms to the requirements.

On the third level the heterogenious interacting components are designed. The internal structure of these architectures can be different, the details are contained in each Architecture Design Model.

By decomposing each layer to different components, the complexity of the initial problem, the complex requirements which were set against the system, can be reduced and the validation and verification of the system can be managed, if the Component Design Models are small and simple enough.

In the Y-model, correspondingly to the V-model, on each design level a validation and verification model can be derived through model transformation rules. Formal methods can be applied on the validation and verification (V&V) models. Besides, verification results from the lower levels can be reused on the higher levels, e.g. Architecture V&V Model can reuse the verification results of its Components V&V Models.

The validation and verification information can be driven back to the design models through back-annotation. In this way engineers can tell how the design models should be modified in order to correct design flaws and errors.

On each level of the Y design method, the design plans and the details of the respective component are contained within the models. In this way the platform-specific implementation, source code and configuration files, can be automatically derived using code generators. Besides, from the each V&V model different test cases can be generated automatically, which can verify the correctness of the code generators.

Extending the V-model with code generators and formal verification can eliminate bugs which would be added by manual source code implementation. In this way the design quality of the system and its components can be improved and the implementation and maintenance costs can be greatly reduced.

2.3 Model-Driven Software Engineering

The model-driven approach can be applied not only for hardware systems, but for software systems as well, and in this way for the software engineering field too. In order to elaborate Model-Driven Software Engineering, standard modeling languages are required. They have precise semantics that is essential for attaching formal meaning to the elements of the language. A standard modeling language is defined by four attributes:

- *Abstract syntax*: describes the elements of the language and their relations. Abstract syntax describes the language in a platform-independent way.
- *Concrete syntax*: concrete representation, denotation of the elements of the language.
- *Formal semantics*: semantics of the elements of the language.
- *Well-formedness constraints*: constraints of the language that cannot be described by the abstract syntax.

The different modeling languages defined by the *Object Management Group* have become the de-facto standards in software engineering. These languages include the *Unified Modeling Language* (UML [46]), the *Systems Modeling Language* (SysML [45]) that can be used for Model-Driven Systems Engineering too.

The first, 1.0, version of UML was published in 1997. The most recent version of the language is 2.5, that was published in 2015. The abstract syntax of the language is its metamodel. The concrete syntax is defined by the different UML diagrams. The well-formedness constraints are described in the *Object Constraint Language* (OCL [44]). The formal semantics of UML was not defined. Instead, a derivation of UML exists that is fUML [47] and it contains the formal semantics of UML activity diagrams.

2.3.1 Eclipse Modeling Framework

Eclipse is a free, open-source integrated software development environment that comes with an extensible plug-in framework. One of the most widely used plug-in set is the Eclipse Modeling Framework (EMF) [52], that has become one of the de-facto modeling toolsets in Model-Driven Software and Systems Engineering.

Ecore is the metamodel of EMF. The most important concepts of Ecore, which resembles to the UML metamodel since it uses similar concepts, are depicted in Figure 2.2. EMF comes with a metamodel editor, which enables the engineer to design the domain-specific metamodel, similarly to designing class diagrams.

Besides, an editor can be generated for creating models that conform the formerly designed metamodel. The editor enables developing models in a tree-like structure. In this way designing domain-specific metamodels and creating instance models can be done

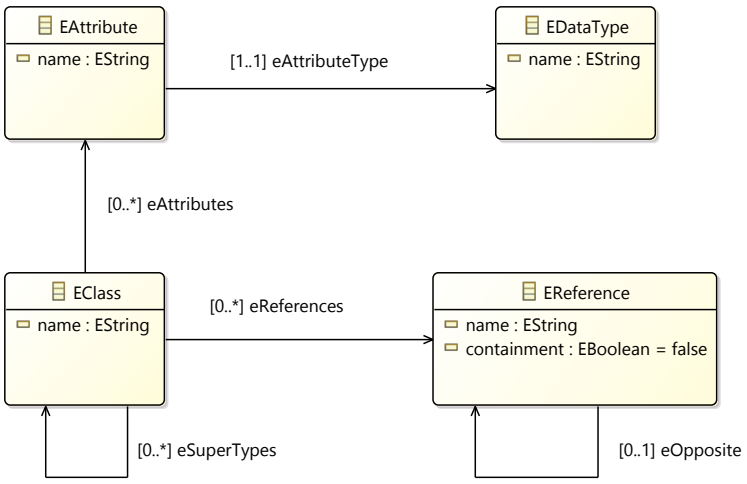


Figure 2.2: Core concepts of the EMF metamodel

in the same framework. It enables rapid design and platform independent development, because both the model and metamodel are serialized in an XMI structure.

Platform-specific codes can be generated from the designed metamodel and the instance model by using the built-in code generation facility. This facility is extensible with new generators depending on the target platform where the final model or the system should be deployed.

2.4 Scenario-based Specification Languages

The model-driven engineering process introduces scenario-based specification languages to describe the collaboration of system components and inter-object communication. It helps engineers to get a quick overview of how each part relates to the others, what messages they exchange and which are the critical message sequences. Scenario-based specification languages are designed to overcome these difficulties and help system and software developers model the inter-object communication scenarios.

Scenario-based languages provide means to describe interactions between different participants with individual lifelines to achieve a goal. Although a scenario can prescribe mandatory, forbidden, or optional behavior, there are other possible behaviors of the system which are not prescribed by either scenario.

Scenario-based Specification Languages can be used in the Model-Driven Systems Engineering methodology, if these languages allow us to create different models which describe scenarios. In this way the communication between the components or entities of the system can be described at a high-level, if we only know what messages shall be transferred between them. Besides, if we know what methods these entities have, then we can describe lower-level behavior too, e.g. method calls, attribute constraints.

There are different scenario languages which have a common origin, the purpose of describing high-level communication, but the details of each formalism are elaborated differently. They are going to be introduced in the following sections.

Harel proposes a methodology [19, 20], that resembles Model-Driven Systems or Software Engineering, which allows computer engineers to synthesize running code from high-level use-case diagrams and intuitively designed scenarios. The consistency of the specification can be verified with formal methods, and the synthesis can be done via automata and statecharts. Final implementation code can be generated from statecharts by platform-specific code generators.

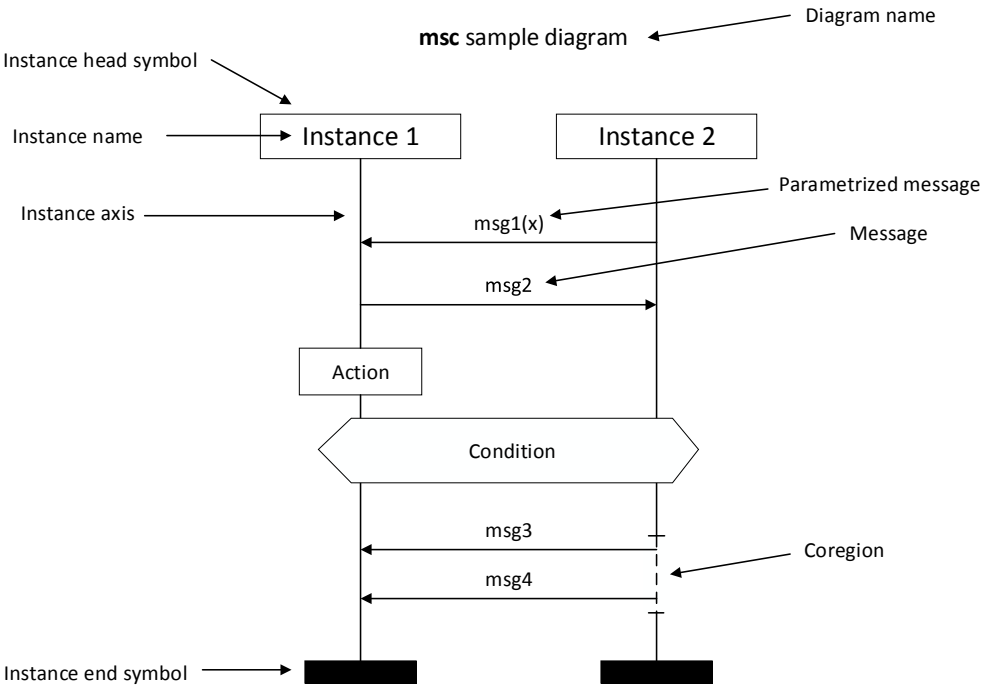


Figure 2.3: Basic concepts of Message Sequence Charts

Although, as an evidence of feasibility the articles focus only on the Live Sequence Chart formalism, there are some aspects which are disregarded in papers: e.g. applying fault tolerant and safety methods on complex reactive systems, handling the complexity during formal verification due to the state-space explosion problem.

2.4.1 Message Sequence Chart

Message Sequence Chart [29] was standardized by the International Telecommunications Union (ITU) in 1993 [35]. The formalism was originally called *Extended Sequence Chart*, an auxiliary notation for the *Specification and Description Language* [34] that is a standard for the specification of telecommunication systems' structure and behavior. The MSC formalism is introduced below as it was by Klose [38, pp. 46–47]:

A possible graphical notation for MSC is similar to the ones for UML Sequence Diagrams (as introduced in Section 2.4.3). The participating entities have vertical lifelines (*instance axis*), the communication (*messages*) between them is represented by horizontal lines with messages noted on the top of them, as depicted in Figure 2.3.

Every communication is asynchronous and (similar to UML Sequence Diagrams) consists of two events: a send event and a receive event. The two restrictions for message events are the following: the send event should be before the receive event, and only one message may be sent or received by each entity at one point of time.

Besides communication, there can be *actions* done by the entities themselves and *conditions* which may involve more entities. Conditions may include entities' local variables and value restrictions for message parameters.

The events (messages, actions, conditions) of an instance axis are ordered from top to bottom, unless they are contained in a *coregion*. All events within a coregion are completely unordered, i.e. they may occur in any order, but not simultaneously.

2.4.2 Live Sequence Chart

Live Sequence Chart was first published in 2001 [8] and there are many tools which support it. For example the Play-Engine [27] and its advanced version the PlayGo framework, which provides a Play-In / Play-Out approach [28] through that LSCs can be both specified and simulated.

The design of the LSC formalism was motivated by the fact that sequence charts, like Message Sequence Chart and UML Sequence Diagrams, cannot make difference between mandatory and optional behavior.

LSC resembles MSC in a lot of perspectives, but the biggest difference between them is that LSC attaches modalities to scenarios and many researchers have defined its semantics over the years [8, 21, 22, 23, 38]. Modality can be defined on two levels: the first level is the whole scenario (chart), the second level is the level of messages.

Chart-level modality Chart is called *universal* if it consists of two parts: *prechart* which describes the preconditions of the scenario. If the interactions in the prechart are successfully executed, then the chart body, also called as *mainchart* has to be executed successfully too. If the mainchart is not successfully executed, then the chart is violated, unless the violation occurred because of a cold message (see *message-level modality* paragraph for more details). Universal charts describe expected behavior over all possible system traces.

An *existential* chart consists of only the *mainchart* part which describes scenarios without preconditions, e.g. some exceptional behaviors which may occur by accident. However, existential charts should be satisfied by at least one trace.

Message-level modality On the message level, a message can be either *cold* (optional) or *hot* (mandatory). A *cold* message is optional, if it does not arrive (due to the fact that it is lost or an other message arrives instead of that [28]), then the scenario is exited and the trace remains inconclusive. On the other hand, a *hot* message prescribes a mandatory behavior, so the specified message must arrive, otherwise the whole chart is violated and the trace is invalid.

A condition is a boolean expression which consists of logical and arithmetical operators, constants and attributes. In a chart a condition can be cold, meaning that it *may* be true (otherwise control moves out of the current block or chart), or hot, meaning that it *must* be true (otherwise the system aborts due to specification violation).

According to Damm et al. [8] locations carry a sequential annotation representing progress within a lifeline. The annotation starts from 0 and incremented by 1 every time the lifeline sends or receives a message, or a condition is progressed there. Locations in a chart can be either *cold* or *hot* locations. *Cold* locations mean the run *does not need to* move beyond the location, on the other hand *hot* locations mean the run *must* move beyond the location.

The basic concepts of Live Sequence Charts with a simultaneous region are depicted in Figure 2.4. The chart is a universal chart that has a prechart and a mainchart. The prechart is activated, if a message $m1$ arrives from process A to B. After this message arrives, the condition $x < 2$ must be true, otherwise the whole chart is violated.

After the condition became true, the mainchart is activated. First, process B sends a message $m2$ to process C. Then, process B must send a message $m3$ to A, otherwise the whole chart is violated. Finally, process A checks condition $y >= 4$. If the condition is false then the mainchart is exited, otherwise the whole chart is successfully executed.

The following formal definitions of LSCs, simregion, partial-order semantics and cut were translated from Horányi's paper [31, pp. 12–14] and Larsen et al.'s paper [39].

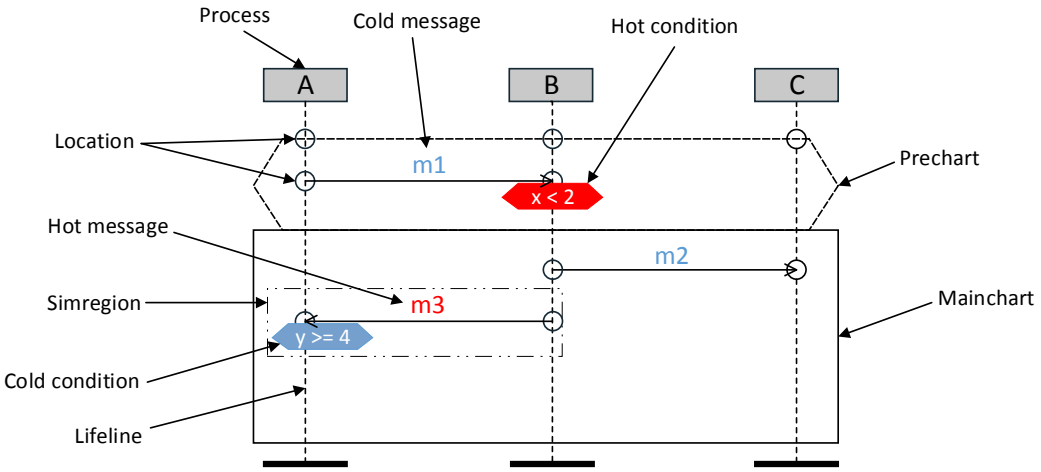


Figure 2.4: Basic concepts of Live Sequence Charts

Definition 3 (Live Sequence Chart). An $\mathcal{L}$ Live Sequence Chart describes the interaction among a finite set of processes denoted by I. On the lifeline of each $i \in I$ process, there are finite sets of positions ($pos(i) = \{0, 1, \ldots p\_max_i\}$), that denote the possible locations of messages, interactions and conditions. The locations of $\mathcal{L}$ are defined as a set $L = \{\langle i, p\rangle | i \in I \wedge p \in pos(i)\}$.

A universal chart can be separated into two parts. The first one is the *prechart*, which prescribes the preconditions of the scenario. If the interactions in the *prechart* are successfully executed, then the chart body, also called as *mainchart* has to be executed successfully too, otherwise the chart is violated. An existential chart usually consists of only the *mainchart*.

A universal LSC is depicted with solid border lines, an existential LSC has dashed borders.

Let Σ be the set of all possible messages. A message $m = (\langle i, p\rangle, \sigma, \langle i', p'\rangle) \in L \times \Sigma \times L$ is sent at the p position of process i with the label σ to the process i' that is at position p'.

From the delivery point of view, messages can be either synchronous or asynchronous. Synchronous messages are denoted by filled arrowhead, asynchronous messages are denoted by open arrowhead. From modality point of view, messages can be either *hot* (mandatory) or *cold* (optional). Hot messages *must be* delivered successfully, while cold messages *can be* either lost or delivered. ▪

Definition 4 (Simultaneous region (simregion)). The simregion s is a set, which contains a message ($m \in M$) and a condition ($c \in C$), $s \subseteq (M \cup C)$:

- It is not empty: $\exists e \in (M \cup C) : e \in s$
- Unique: $\forall m, n \in M : (m \in s \wedge n \in s) \Rightarrow m = n$
- Non-overlapping: $\forall s, s' : \forall e \in (M \cup C) : (e \in s \wedge e \in s') \Rightarrow s = s'$

Let $S \subseteq 2^{(M \cup C)}$. Simregions allow to group several elements, which should be observed at the same time [38]. ▪

Definition 5 (Partial-order semantics of LSCs). The locations of an LSC are partially ordered according to the following rules:

- in the chart if l_1 is above l_2 , then $l_1 \leq l_2$,
- all locations in the same simregion have the same order:
  $\forall s \in S, \forall l, l' \in L : (\lambda(l) = s) \wedge (\lambda(l') = s) \Rightarrow (l \leq l') \wedge (l' \leq l)$

where λ function assigns the locations to simregions. The partial-order relation $\preceq \subseteq L \times L$ can be defined as the transitive relation of $\leq$. ▪

Definition 6 (Cut in an LSC). The cut of an LSC is a bottom-closed set of locations which go through on each lifeline of all processes. The bottom-closedness means if a location is in a cut, then every other locations are in the cut too which are before the given location owing to the partial-ordering relation: $\forall c \subseteq L, \forall l, l' \in L : (l \in c \wedge l' \preceq l) \Rightarrow l' \in c$.

In an LSC there is always a minimal and a maximal cut. The minimal cut is empty, so it contains no location. On the other hand, the maximal cut contains all locations. If the LSC has a non-empty prechart, then there is a cut in the beginning of the mainchart, which contains all locations within the prechart. ▪

Definition 7 (Run in an LSC). According to Harel et al. [21] the sequence of cuts constitutes a run, if all locations are enumerated in the same order as message sending should happen, so:

1. The first element of the sequence is: $(\langle i_1, 0\rangle, \langle i_2, 0\rangle, \ldots, \langle i_n, 0\rangle)$, where $i \in I$ is a process and $|I| = n$.
2. In the sequence that location is incremented which originates the next message delivery: $(\langle i_1, l_1\rangle, \langle i_2, l_2\rangle, \ldots, \langle i_j, l_j + 1\rangle, \ldots, \langle i_n, l_n\rangle)$ where l_1 denotes the latest location in the lifeline of i_1, according to the previous sequence element; and the id of the sender of the next message is i_j so its location is incremented by one.
3. The location of the latest message's recipient is incremented in the recent cut: $(\langle i_1, l_1\rangle, \langle i_2, l_2\rangle, \ldots, \langle i_k, l_k + 1\rangle, \ldots, \langle i_n, l_n\rangle)$ where l_1 denotes the latest location in the lifeline of i_1, according to the previous sequence element; and the receiver of the last message was i_k so its location is incremented by one. ▪

The sequence of cuts is constructed according to the enumeration above, repeating phases 2 and 3 until all locations and message deliveries were exceeded.

Definition 8 (Trace of a run). According to Harel et al. [21] the trace of a run is the sequence of processes and messages in the order they were delivered: $\forall m \in M : (i, m)$ where $i \in I$ the sender of the message m.

To illustrate cuts, run and trace in an LSC, a sample LSC is depicted in Figure 2.5. Cuts in the LSC are for example:

- minimal cut: $\emptyset$
- maximal cut of the prechart:

$$\{(\langle A, 0\rangle, \langle B, 0\rangle, \langle C, 0\rangle), (\langle A, 1\rangle, \langle B, 0\rangle, \langle C, 0\rangle), (\langle A, 1\rangle, \langle B, 1\rangle, \langle C, 0\rangle)\}$$

- maximal cut of the LSC, including the prechart and mainchart:

$$\{(\langle A, 0\rangle, \langle B, 0\rangle, \langle C, 0\rangle), (\langle A, 1\rangle, \langle B, 0\rangle, \langle C, 0\rangle), (\langle A, 1\rangle, \langle B, 1\rangle, \langle C, 0\rangle),$$
$$(\langle A, 1\rangle, \langle B, 2\rangle, \langle C, 0\rangle), (\langle A, 1\rangle, \langle B, 2\rangle, \langle C, 1\rangle), (\langle A, 1\rangle, \langle B, 3\rangle, \langle C, 1\rangle),$$
$$(\langle A, 2\rangle, \langle B, 3\rangle, \langle C, 1\rangle)\}$$

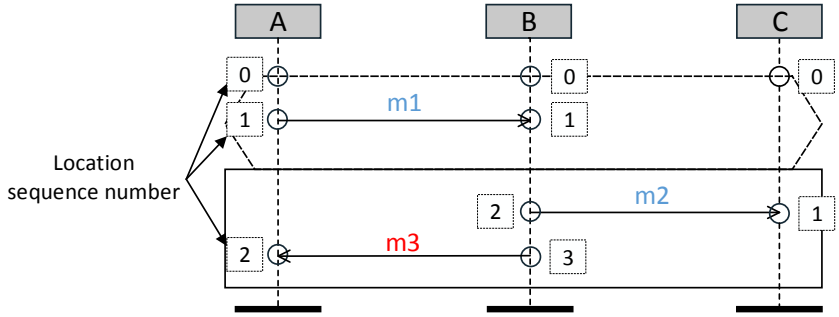


Figure 2.5: Sample Live Sequence Chart with locations

The sequence of cuts in the maximal cut of the LSC above constitutes a run, according to Definition 7. The trace of that run is:

$$(A, m1), (B, m2), (B, m3)$$

Simulating or executing an LSC can end in three results:

- The LSC is successfully executed, if all hot messages were delivered and all hot conditions were true and all hot locations were exceeded.
- The LSC is exited, when either a cold condition or the prechart is violated. It is not an error, it simply means the scenario cannot be fitted to the recent execution.
- The LSC is aborted, when a hot condition was violated or a hot location was not reached. It means there was a serious deterrence from the specified behavior.

Live Sequence Chart formalism was successfully applied in telecommunication and hardware domains.

Bunker et al. applied the LSC approach for high-level modeling of telecommunication applications to help detecting feature interaction at early development stages [5]. They introduced the results of applying the methodology to the specification, animation and formal verification of a telecommunication service.

Combes et al. applied the formalism for hardware requirements specification where it was used as a high-level visual notation for writing specification [7]. An automatic link was developed that inputs an LSC specification and outputs temporal properties suitable for model checking.

2.4.3 UML Sequence Diagram

Sequence Diagram is a notation in the *Unified Modeling Language* for modeling scenario-based interactions which focuses on message interchange between a number of lifelines. Similarly to Live Sequence Charts, Sequence Diagrams borrow some elements from Message Sequence Charts [35], but it extends the latter language with new elements too.

Interactions chapter of UML specification defines the elements that constitute an interaction, whose representation can be a *Sequence Diagram* [46]. Although there are other notations with different purposes for modeling *Interactions*, e.g. *Communication Diagrams*, *Interaction Overview Diagrams*, *Timing Diagrams*, I will use only *Sequence Diagrams* in my thesis, because they are most widely used for modeling interactions.

First, I am going to highlight the concrete syntax of the most important parts of the language, then give some references to semantic definitions of UML *Interactions* and *Sequence Diagrams* in the following sections.

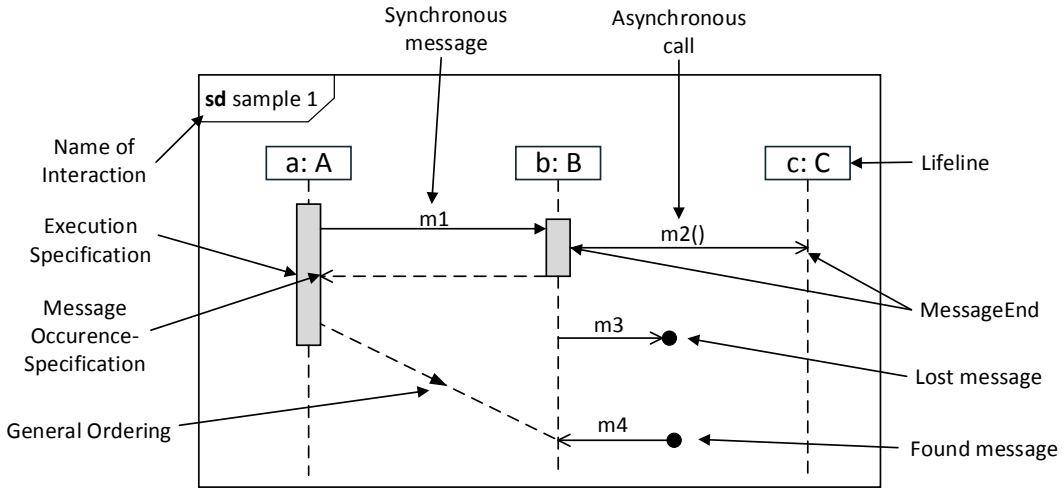


Figure 2.6: Concepts of UML Sequence Diagram

Concrete Syntax The concrete syntax of UML *Sequence Diagrams* is depicted in Figure 2.6. Each *Sequence Diagram* should have a unique name. Entities who participate in the interaction and communicate via *Messages* that are represented on *Lifelines*.

A *Message* can be either *synchronous*, depicted with filled arrowhead, or *asynchronous*, depicted with empty arrowhead. Message can mean either a *Call* or a *Signal* or a *Reply*. These properties are defined by a message's *MessageSort* attribute. The *MessageKind* property of the message specifies, whether it is a *lost* message, whose receiver is not know, or a *found* message, whose sender is unknown. If both the sender and the receiver are known, then it is called a *complete* message.

Messages have two *MessageEnds*. Each MessageEnd is either a *Gate* or a *MessageOccurenceSpecification*. Gate links two parts of the message, e.g. a message crosses the boundaries of a *CombinedFragment*, depicted in Figure 2.7. MessageOccurenceSpecification represents a send event or a receive event associated with a message between two lifelines [46].

ExecutionSpecification is represented as a gray rectangle on the lifeline and it specifies the execution of a unit of behavior in a Lifeline [42]. The start event and the end event of the ExecutionSpecification are marked with *ExecutionOccurenceSpecifications*.

GeneralOrdering is applied for ordering otherwise unrelated messages, e.g. a found message.

Interactions can be composed together by referring from one to another via *InteractionUse* elements. InteractionUse can have input parameters and return values, which can be used in the referrer Interaction.

Interaction can contain *StateInvariants*, which are runtime constraints on participants of the Interaction. They can refer to the state of an instance of the given Lifeline, or they can be expressions constraining attributes and variables. Both variants are depicted in Figure 2.7. *StateInvariants* are assumed to be true at runtime, otherwise the execution trace is considered invalid.

More complex *Interactions* can be constructed with *CombinedFragments*. They consist of one or more *InteractionOperands*, which can be guarded with *InteractionConstraints*, depending on the type of the CombinedFragment. Each CombinedFragment contains an *InteractionOperatorKind* attribute that defines what kind of fragment it is. The definitions are given according to the UML specification [46, pp. 581–582].

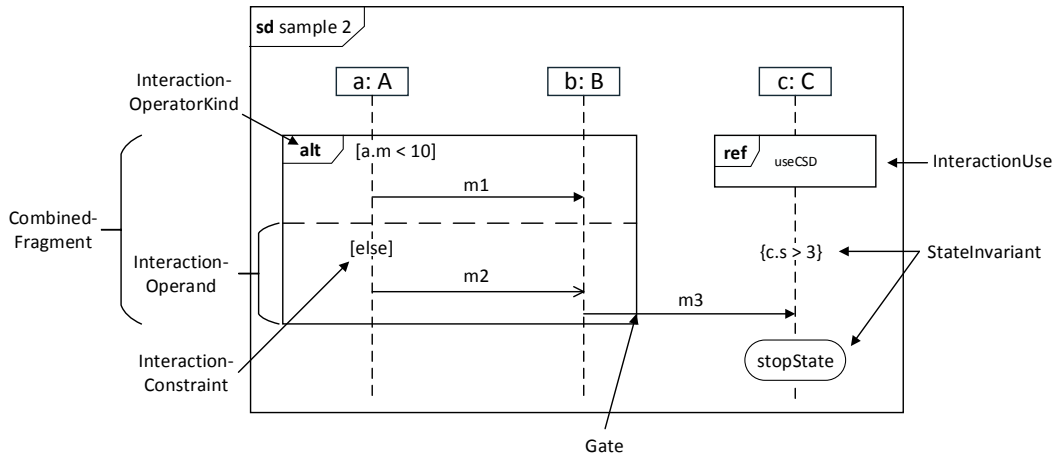


Figure 2.7: Sequence Diagram with CombinedFragment [42]

- *seq*: the CombinedFragment represents a weak sequencing between the behaviors of the operands.
- *strict*: the CombinedFragment represents a strong sequencing between the behaviors of the operands.
- *alt*: the CombinedFragment have different operands, which are guarded with InteractionConstraints. The first operand whose constraint is true, will be executed.
- *opt*: the execution of the CombinedFragment is optional.
- *par*: the operands of the CombinedFragment should be executed in parallel.
- *loop*: the CombinedFragment should be executed until the loop constraint is true.
- *critical*: the CombinedFragment represents a critical region, whose traces cannot be interleaved by other OccurenceSpecifications on the same Lifeline as this region.
- *neg*: the trace represented by the CombinedFragment is negative, it must not occur, otherwise the result is an invalid trace.
- *assert*: the possible sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace.

Semantics Although the semantics of model elements of UML Sequence Diagrams are defined in text in the UML standard [46], it contains some flaws due to the complexity and the increased expressive power of the language. The greatest problem is that the semantics are not formally defined which allows the construction of ambiguous Sequence Diagrams.

To increase the expressive power of the language, so that liveness and safety properties can be formulated, *assert* and *negate* (neg) were introduced as types for CombinedFragments. However, the semantics of assert and negate prevents their applicability, because of their ambiguous meanings. According to Harel et al. [24] applying *assert* and *negate* can result in an ambiguous construction. It is due to the fact that according to the UML standard, assert is "the sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace." [46, p. 582]

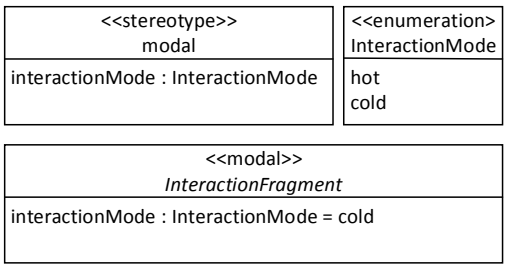


Figure 2.8: Concepts of Modal Sequence Diagram [24]

and "neg designates that the CombinedFragment represents traces that are defined to be invalid" [46, p. 582]. It seems contradictory and ambiguous if assert or negate contains the other. To address the problem, they proposed a new formalism, called *Modal Sequence Diagram* that is introduced in Section 2.4.4.

What is not trivial either, how to calculate partial ordering of events in an Interaction with complex CombinedFragments [42, p. 502].

To address the problem, several semantics definitions have been proposed by various authors during the years. In a survey Micskei et al. [42] collected and categorized different formal semantics and the semantic choices made in papers dealing with UML 2 Sequence Diagrams. They present the different options for the collected choices and the relations between them in a structured format. So one can get a broader overview how the semantics of Sequence Diagrams can be defined based of the purpose of the diagram.

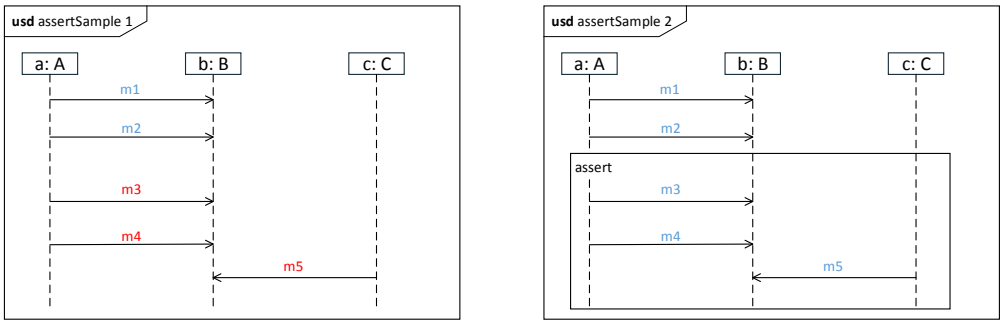
2.4.4 Modal Sequence Diagram

Harel et al. proposed a new formalism [24] to address the problem caused by *assert* and *negate* CombinedFragments in UML Sequence Diagrams. The new formalism is called *Modal Sequence Diagram* (MSD) that is on one hand the extension of the former formalism with modalities, on the other hand the combination of Live Sequence Chart with UML Sequence Diagram. Thus the methods which were worked out for those formalisms can also be applied to MSD. The formalism and its operators are introduced in this section according to the definition by Harel et al. [24].

To allow the specification of modalities over Interactions, Harel et al. defined a stereotype *modal* with an attribute *interactionMode*, which, in the case of LSC, can be either hot (*universal*) or cold (*existential*). The new stereotype is introduced, depicted in Figure 2.8 based on [24, p. 240], as an extension of the abstract class *InteractionFragment*, and hence also of its subclasses: *Interaction*, *InteractionUse*, *ExecutionSpecification*, *OccurrenceSpecification*, *CombinedFragment*, *InteractionOperand* and *StateInvariant*. Technically, the modal stereotype is applied to messages too. The *interactionMode* of a message is derived from the modes of the message's send and receive *MessageOccurrenceSpecifications*. In general, a message is hot if at least one of its ends is hot, and is cold otherwise.

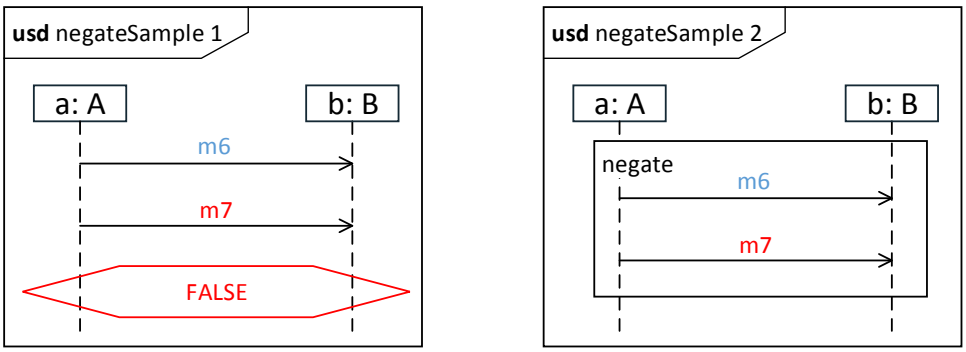
The semantics of MSD is roughly the same as LSC's [8, 21, 23]: a system-model satisfies an MSD specification if (1) every one of its runs satisfies each universal diagram in the specification, and (2) every existential diagram is satisfied by at least one possible system run. The semantics of an MSD interaction fragment depends both on the partial order induced by its occurrence specifications and on its mode.

In order to support sequential composition of MSDs, Harel et al. leave out the *prechart* construct from LSCs [24]. Instead, a general approach is taken, where cold fragments inside universal interactions mean prechart-like purposes: a cold fragment does not have to be



(a) MSD with implicit assert

(b) MSD with explicit assert



(c) MSD with implicit negate

(d) MSD with explicit negate

Figure 2.9: Assert and negate in MSD

satisfied in all runs but if it is satisfied then the subsequent hot fragment should also be satisfied.

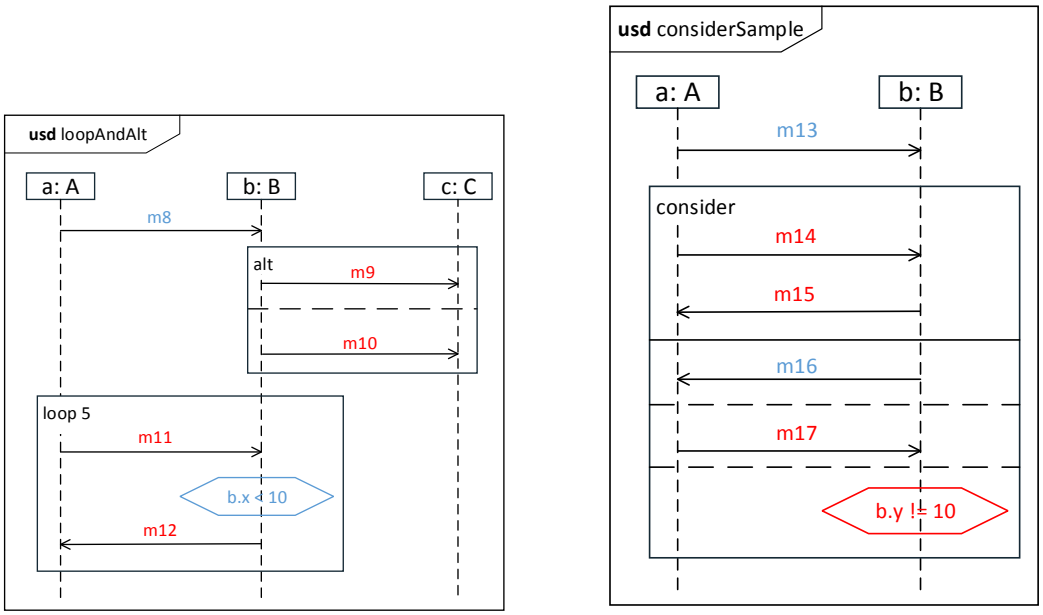
Because *StateInvariant* is an *InteractionFragment* that has a modality property, the StateInvariant also inherits such property. Several cases can be differentiated, depending on the StateInvariant (condition)'s modality as follows:

- A universal (hot) condition must evaluate to TRUE, otherwise the chart is violated.
- An existential (cold) condition can be evaluated to FALSE, then the chart fragment is exited and the enclosing chart is continued. If the cold condition violation happens in the topmost chart fragment, then the chart is exited and no error happens.

Assert is interpreted in MSD as assigning a *hot* mode to all the OccurrenceSpecifications inside the interaction fragment operand, as depicted in Figure 2.9b.

Negate is interpreted in MSD as adding a hot constant FALSE after the last *OccurenceSpecification*, as the maximal location element in the fragment, as depicted in Figure 2.9d. Thus the *InteractionFragment* specifies a forbidden scenario, because if the fragment is satisfied and the hot FALSE constant is reached, then it is a contradiction and the chart is violated.

Alt and *loop* in MSD, depicted in Figure 2.10a, have the same meaning as in Sequence Diagrams. For the former one the operands of the CombinedFragment are executed, if the condition belonging to that operand evaluates to TRUE. If there are multiple operands which are enabled, then a nondeterministic choice is made to execute either operand. For the latter one the operand is executed repeatedly until the condition is TRUE, and there is no trace or condition within the loop that violates it.



(a) Alt and loop in MSD

(b) Consider fragment in MSD

Figure 2.10: Alt, loop, consider fragments in MSD

Consider, and respectively its dual fragment *ignore*, allows to design messages which must be considered, respectively ignored, during the execution of the interaction operand. Adding more messages and conditions in the *considered* set results in more restrictive specifications. A run not only needs to perform the occurrences in a matching order specified by the interaction operand, but it also should not exhibit any of the additional occurrences in the prohibited set.

In comparison with *consider*, *ignore* results in a more permissive specification, because in the run besides the interactions of the operand, any other interactions from those, which were specified in the *ignore* set, may be carried out.

For example, the *consider* fragment depicted in Figure 2.10b specifies that sending $m16$ from b to a is a cold violation; sending $m17$ from a to b is a hot violation; $b.y = 10$ is a hot violation too, if they occur during the execution of the interactions $m14$, $m15$ in the interaction operand of *consider*.

Should a cold violation occur, then only the *consider*, respectively *ignore*, fragment is exited. However if a hot violation occurs then the chart is violated too.

According to Harel et al. [24] the use of *InteractionFragments* as operands for the *consider CombinedFragment*, together with the general modal semantics provided for MSD, significantly increase the expressive power of the language and allow more compact and intuitive specification of complex behavior.

2.5 Gamma Framework

Gamma Statechart Composition Framework[1] [14, 15] is an Eclipse Modeling Framework-based tool which supports the model-driven design and analysis of hierarchical, component-based reactive systems. The main functionality of the framework is supporting the hierarchical composition of component-based models.

[1] `https://inf.mit.bme.hu/en/gamma`, last access: 29/11/2017

Based on an intermediate statechart language, a new language is defined to facilitate the composition of statechart models with precise semantics. To support the modeling process, validation rules have been defined for the intermediate statechart language to find design flaws as soon as possible. Furthermore, the automatic transformation of individual statecharts as well as their composition to formal models has been developed to support the formal analysis of the design models. Additionally, the framework supports the back-annotation of the analysis results to the compositional language enabling the users to analyze the behavior of their models in a familiar domain.

Finally, the framework includes a code generator that produces the implementation of the composed system, assuming the implementation of the statechart models are given and following the semantics of the compositional language.

At the time of writing the thesis, Gamma supports YAKINDU Statechart Tools[2] as an input modeling language for the state-based components and UPPAAL [3] as a verification framework for formal analysis.

The relevant parts of the Gamma metamodel are depicted in Figure 2.11a. With the framework the user can create *ComponentDefinitions* which can either be a *Statechart-Definition* (that is basically a statechart in the Gamma's internal statechart language) or a *CompositeDefinition*.

A *CompositeDefinition* consists of multiple *ComponentInstances* that refer to a *ComponentDeclaration*, which will refer to a *ComponentDefinition* or a *StatechartSpecification*. A *ComponentDeclaration* can define several *Ports* which can be bound via *PortBindings* to the border of the external (encapsulating) component. In this way component of components (composite components) can be built.

Each *Port* is bound to an *Interface*, which has *EventDeclarations* (events) as depicted in Figure 2.11b. Each *EventDeclaration* has a direction which defines if the respective event is an *incoming* (IN), *outgoing* (OUT) or a *bidirectional* (INOUT) one. *Incoming* events are received, *outgoing* events are sent, *bidirectional* events are received and sent by the component which implements the *Interface*. Besides, each *Port* realizes the *Interface*, it is bound to, either in *provided* or in *required* mode:

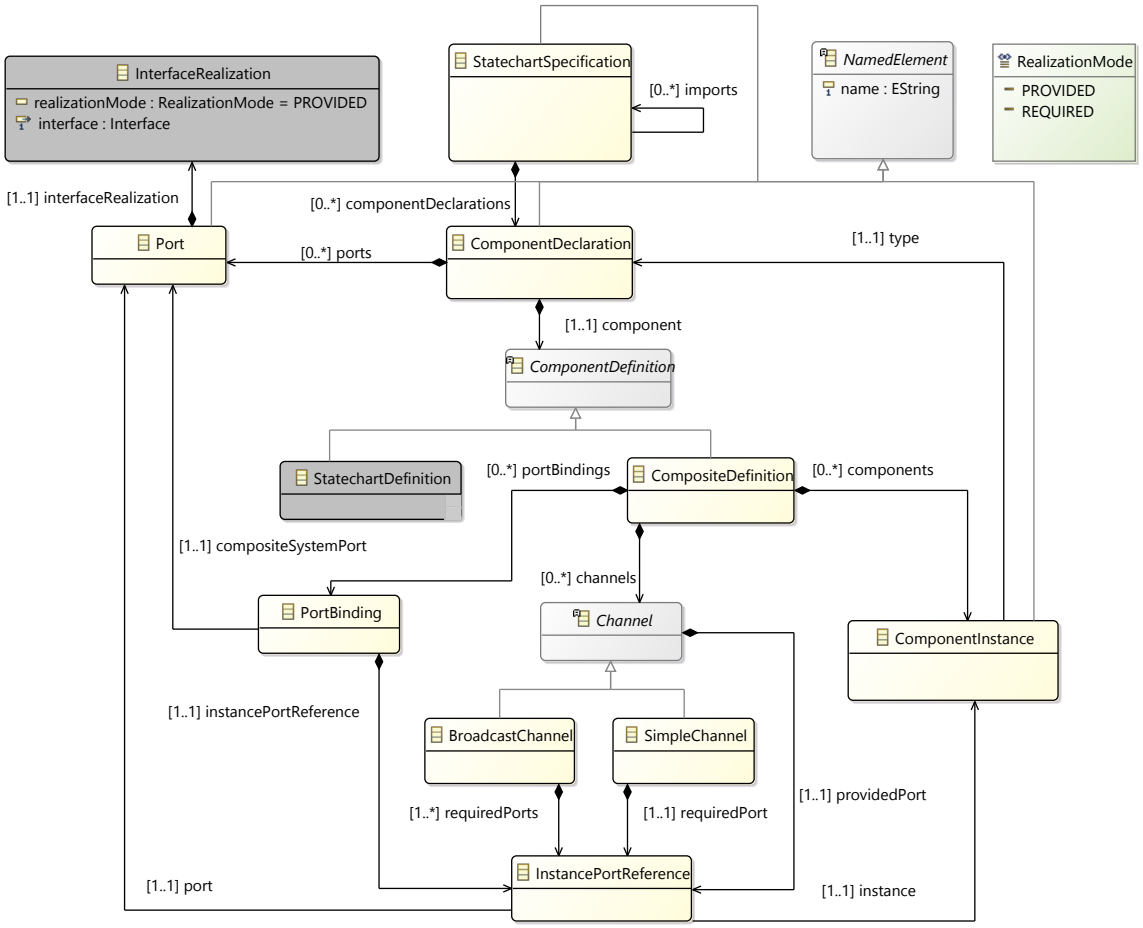
- *provided* mode means, the events of the realized interface are transferred and received according to their specified directions.
- *required* mode means, the directions of the events are reversed, so that *outgoing* events can be received, *incoming* events can be sent through the port.

A *CompositeDefinition* defines multiple *Channels* between the *ComponentInstances*. Each *Channel* can either be a *SimpleChannel* or a *BroadcastChannel*:

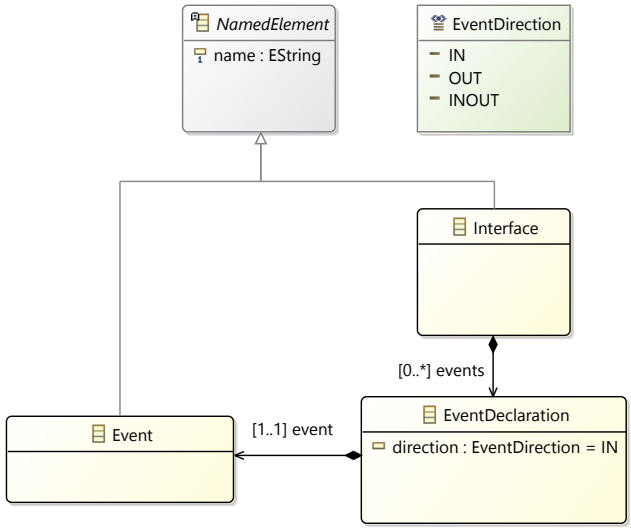
- A *SimpleChannel* connects two *Ports* which realize the same *Interface* in different modes so that they can be connected. This construction guarantees that each event sent by either *Port* can be processed by the other one.
- In a *BroadcastChannel* one *Port provides* an *Interface* and multiple other *Ports* on the *Channel require* that *Interface*. However, a restriction had to be introduced for *Interfaces*. Only such *Interfaces* can be used here, whose events are outgoing, so that those *Ports* which *require* the *Interface* cannot send events.

As an example for *ComponentDeclaration*, see the textual specification in Listing 2.1 or the graphical illustration is depicted in Figure 2.12. This example shows a component which consists of two *SectionDeclarations* (the internal hidden elements) which are connected to each other, however some of the ports are offered to the external world.

[2] `https://www.itemis.com/en/yakindu/state-machine/`, last access: 29/11/2017



(a) Component metamodel



(b) Interfaces metamodel

Figure 2.11: Relevant parts of the Gamma metamodel

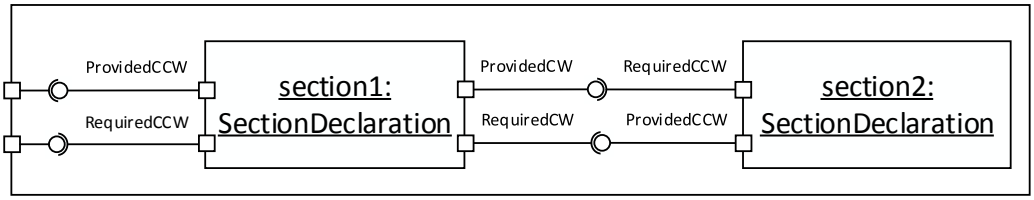


Figure 2.12: Graphical representation of component TwoSections

The TwoSections component declares two ports (ProvidedCCW, RequiredCCW) which realize the same *Protocol* interface but with different realization modes. These port realizations are then bound to the respective ports of the section1 internal component, and they are offered to the outside world, to any component which may use the services of the TwoSections component. Besides, there are two *SimpleChannels* which connect section1 and section2 with each other. Each channel connect those ports which are compatible with each other, in a sense that they implement the same interface but with different realization modes. Thus the section1.ProvidedCW and section2.RequiredCW ports form one *Channel*, and section2.ProvidedCCW and section1.RequiredCW ports form another *Channel*.

```
import Section

specification TwoSection {
  component TwoSections := {

    port ProvidedCCW : provides Protocol
    port RequiredCCW : requires Protocol

    {
      components {
        section1 : SectionDeclaration
        section2 : SectionDeclaration
      }

      bindings {
        ProvidedCCW -> section1.ProvidedCCW
        RequiredCCW -> section1.RequiredCCW
      }

      channels {
        [section1.ProvidedCW] -o)- [section2.RequiredCCW]
        [section2.ProvidedCCW] -o)- [section1.RequiredCW]
      }
    }
  }
}
```

Listing 2.1: Textual specification of a component in the Gamma Framework

2.6 Finite Automata

Finite Automata are such graphs which can recognize patterns in a sequence of input events. There are many different types of Finite Automata from which three of them are used in the Master Thesis: *Nondeterministic Finite Automaton*, *Deterministic Finite Automaton* and *Minimal Deterministic Finite Automaton*. All of them recognize the

same class of languages, called *regular languages*. In the followings these automata will be defined according to Ullman et al. [1].

Definition 9. A *Nondeterministic Finite Automaton (NFA)* is a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where:

- Q is a finite, non-empty set of states,
- Σ is a finite, non-empty set of input symbols, also called as the *input alphabet* or *events* of the automaton. It is assumed that ϵ, which stands for an empty is string, is never a member of Σ,
- $\delta : Q \times \Sigma \to Q$ is the state transition function, that gives for each $q \in Q$ and for each symbol in $\Sigma \cup \epsilon$ a set of next states,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of the accepting states (final state). ▪

Definition 10. A *Deterministic Finite Automaton (DFA)* is a tuple $\langle Q, \Sigma, \delta_d, q_0, F \rangle$ where:

- Q is a finite, non-empty set of states,
- Σ is a finite, non-empty set, representing the event set of the automaton,
- δ_d is a subset of tuples $\langle Q \times \Sigma \times Q \rangle$, and the number of outgoing edges from each state for each event is only one. So $\forall s_i \in Q, \forall e_j \in \Sigma : |\langle s_i, e_j, s_{i+1} \rangle| = 1$, where $i \in [0, n-1]$ and $j \in [0, m-1]$ and $n = |Q|$ and $m = |\Sigma|$. Edge labeled by event ϵ is not allowed;
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states. ▪

Definition 11. A *Minimal Deterministic Finite Automaton (MFA)* is a DFA that accepts the same language as the DFA, but it has minimum number of states. MFA minimizes the computational cost for pattern matching as it contains the least number of states to recognize a regular language. ▪

Definition 12. A *synchronous product* $(A_1 \times A_2)$ of Finite Automata $A_1 = \langle Q_1, \Sigma, \delta_1, q_{a1}, F_1 \rangle$ and $A_2 = \langle Q_2, \Sigma, \delta_2, q_{a2}, F_2 \rangle$, is a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where:

- $Q = Q_1 \times Q_2$ is a finite, non-empty set of states,
- Σ is a finite, non-empty set, representing the event set of the automaton,
- $\delta : Q \times \Sigma \to Q$ is the state transition function:$\langle \langle p, q \rangle, e_i, \langle p', q' \rangle \rangle \in \delta$, iff $\langle p, e_i, p' \rangle \in \delta_1$ and $\langle q, e_i, q' \rangle \in \delta_2$, where $\langle p, q \rangle, \langle p', q' \rangle \in Q$ and $p, p' \in Q_1$ and $q, q' \in Q_2$ and $e_i \in \Sigma$;
- $q_0 \in Q$ is the initial state which is created from q_{a1} and q_{a2},
- $F = F_1 \times F_2$ is the set of accepting states.

Such *Synchronous Product Finite Automaton (SFA)* accepts the language which is accepted by both A_1 and A_2. Also marked as: $L(A_1 \times A_2) = L(A_1) \cap L(A_2)$. ▪

Definition 13. A *trace* in a finite automaton is a sequence of states $s_0, s_1, \ldots, s_n$ (where $\forall i : s_i \in Q$) for the input sequence $e_1, e_2, \ldots, e_n$ (where $\forall j : e_j \in \Sigma \cup \{\epsilon\}$) if $\forall s_i \in Q, \forall e_j \in \Sigma$ exists a transition between state s_i and s_{i+1} with the label of e_j, so $\langle s_i, e_j, s_{i+1} \rangle \in \delta$, where $i \in [0, n-1]$ and $j \in [0, m-1]$ and $n = |Q|$ and $m = |\Sigma \cup \{\epsilon\}|$ and ϵ is the empty event that is never a member of Σ.

A trace is accepting if $s_0 = q_0$ and $s_n \in F$. ▪

Definition 14. A finite automaton *accepts* an input sequence if exists an accepting trace for the input sequence. ▪

Definition 15. The *language defined* (or *accepted*) by a finite automaton M is the set of input sequences which are accepted by the automaton. Also marked as: $L(M)$ ▪

2.6.1 Determinization of a Nondeterministic Finite Automaton

The algorithm introduced by Ullman et al. [1], is described in ALGORITHM 1. The general idea behind the determinization of an NFA N is, each state of the constructed DFA D corresponds to a set of states in N. After reading input $e_1, e_2, \ldots, e_n$ the D is in that state which corresponds to the set of states that the N can reach, from its initial node, following paths labeled by $e_1, e_2, \ldots, e_n$.

The algorithm constructs a transition table *Dtran* for D. Each state of D is a set of states in N. *Dtran* is constructed so that D will simulate "in parallel" all possible moves N can make on a given input trace. However, ϵ-transitions of N have to be eliminated according to in ALGORITHM 2.

The initial state of D is the ϵ-*closure*(s_0), and the accepting states of D are all those sets of states in N that include at least one accepting state of N.

Algorithm 1: The subset construction of a DFA from an NFA

input : an N NFA
output: a D DFA accepting the same language as N

```
1  initially, ϵ-closure(s_0) is the only state in Dstates, and it is unmarked;
2  while there is an unmarked state T in Dstates do
3  |  mark T;
4  |  foreach input symbol a do
5  |  |  U = ϵ-closure(move(T, a));
6  |  |  if U is not in Dstates then
7  |  |  |  add U as an unmarked state to Dstates;
8  |  |  end
9  |  |  Dtran[T, a] = U;
10 |  end
11 end
12 Dstates is Q_d, the finite, non-empty set of states of D
13 Dtran[T, a] is the value of the δ_d transition function for state T and event a in D
14 move(T, a) is the set of states that are reachable from state T for event a in N
```

2.6.2 Minimization of a Deterministic Finite Automaton

The prerequisites and steps of the minimization algorithm are defined according to the papers by Ullman et al. [1] and Csima et al. [37].

In order to minimize a deterministic finite automaton, this automaton must be *complete*, which formally means: $\forall i, \forall j : \langle s_i, e_{j+1}, s_{i+1} \rangle \in \delta_d$, where $s_i \in Q$ and $e_j \in \Sigma$ and $i \in [0, n-1]$ and $j \in [0, m-1]$ and $n = |Q|$ and $m = |\Sigma|$.

Algorithm 2: Computing ϵ-*closure*(T)

input : a set of states T
output: a set of states ϵ-*closure*(T)

```
1  push all states of T onto stack;
2  initialize ε-closure(T) to T;
3  while stack is not empty do
4  |  pop t, the top element, of stack;
5  |  foreach state u with an edge from t to u labeled ε do
6  |  |  if u is not in ε-closure(T) then
7  |  |  |  add u to ε-closure(T);
8  |  |  |  push u onto stack;
9  |  |  end
10 |  end
11 end
```

If DFA D is not *complete*, then it has to be made so by defining the missing transitions in δ_d.

Then every state s has to be removed which is not reachable from initial state q_0 via any input sequence.

The states of D are partitioned into groups according to the following rule: two states are in the same group if no trace exists that would distinguish those states from each other, as defined in Definition 16, until the partitioning cannot be refined further by breaking any group into smaller groups. These steps are detailed in Algorithm 3.

The complexity of the Algorithm 3 is quadratic in the number of states Q of the automaton and linear in the size of the input alphabet Σ. The reason for the computatinal complexity is the following: the size of the table is $O(|Q|^2)$ so less than $|Q|$ rounds are needed and in each round $|\Sigma|$ transitions have to be checked [37].

Then we have the minimum-state DFA (MFA).

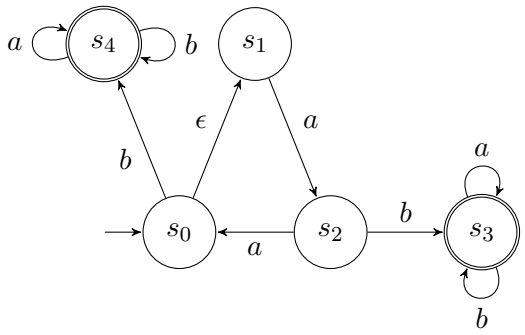
Definition 16. The input sequence x *distinguishes* state $s_i \in Q$ from state $s_j \in Q$ if exactly one of the states reached from s_i and s_j by following the path with label x is an accepting state $\forall i, j : 1 \leq i, j \leq n, i \neq j$ where $n = |Q|$.

State s_i is *distinguishable* from state s_j if there is some input sequence that distinguishes them [1]. ▪

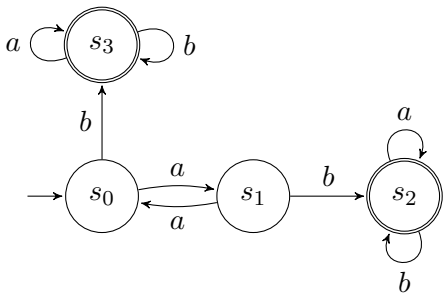
Algorithm 3: Computing MFA from DFA

input : a D DFA
output: an M MFA accepting the same language as D

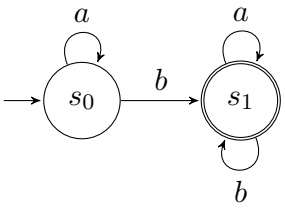
```
1  let T be an empty array whose size is |Q| × |Q| that is indexed by the states;
2  let T[p, q] = 0, if (p ∈ F ∧ q ∉ F) ∨ (p ∉ F ∧ q ∈ F);
3  while change in T occurs do
4  |  if T[p, q] is empty, but there is an a ∈ Σ and for p' = δ(p, a) and q' = δ(q, a),
   |  p', q' ∈ Q, T[p', q'] is not empty, then let T[p, q] = i;
5  end
6  Column-wise reading below the main diagonal of T, those T[p, q] cells belong to the
   same partition which are empty in the end. These partitions will be the states of M.
7  M's δ_m can be constructed from D's δ by looking at one of the states in each
   partition and checking the transition in δ from this state for each e ∈ Σ.
```



(a) Nondeterministic Finite Automaton



(b) Deterministic Finite Automaton



(c) Minimal Deterministic Finite Automaton

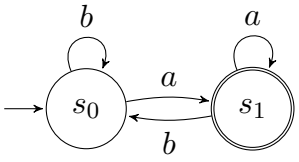
Figure 2.13: Sample Finite Automata for the same language

2.6.3 Examples

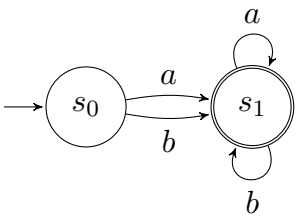
An NFA, DFA, MFA can be represented by a *transition graph*, where the nodes are states and the labeled edges represent the transition function δ. There is an edge labeled by $e \in \Sigma$ from state s_i to state s_{i+1} (where $\forall i : s_i \in Q$ and $1 \leq i \leq n$ and $n = |Q|$) iff s_{i+1} is one of the next states for s_i and input e is received according to δ.

The transition graph for an NFA, DFA, MFA recognizing the language of regular expression **(a)*b(a|b)*** is depicted in Figure 2.13. The inital state of the automata is shown by an arrow with no trigger and no input state.

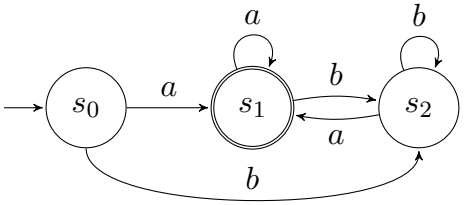
In Figure 2.14 the *Synchronous Product Finite Automaton (SFA)* of two Finite Automata over the same Σ is depicted. Automaton A_1 recognizes the **(b*a*)*a** regular expression, A_2 recognizes the **(a|b)+** regular expression, so $A_1 \times A_2$ recognizes the **(b*a*)*a** regular expression.



(a) Finite Automaton A_1



(b) Finite Automaton A_2



(c) Finite Automaton A_3

Figure 2.14: Sample Synchronous Product Finite Automaton

Chapter 3

Existing Tooling for Scenario-based Specification Languages

In order to support model-driven engineering of systems designed by scenario-based specification languages, modeling tools are required. In this chapter several tools will be studied which support the design of scenario based system descriptions. It should be noted, that for comparison only open-source and for academical and research purposes free softwares were selected.

These tools are going to be compared regarding two aspects: level of standardization of the modeling language and the functionality offered by the tool. These aspects are going to be detailed in Section 3.1. Then in the following sections the existing tool support of formalisms for scenario-based specification languages are compared.

3.1 Tool comparison aspects

The aspects for comparison have two different dimensions: one dimension is the *level of standardization* of the modeling language which is used by the tool, the other aspect is the functionalities the different tools have.

Based on these two dimensions, the different tools can be evaluated as shown in Table 3.1 and as detailed in Section 3.2 – Section 3.5 of this chapter.

3.1.1 Standardization level of the modeling language

From this perspective one end of the scale is the de-facto standardized modeling language, the other end is the non-standardized, custom modeling language.

The advantage of having a standardized language is, its concepts are reviewed by many people, widely known, unambiguous and probably formalized which enables the application of formal methods over the language. The disadvantage is, if something should be changed in the language then it can be a rigorous and time-consuming procedure to carry out due to the number of people being involved in the process.

On the other end of the standardization scale, the advantage of having a custom modeling language is that language designer has higher level of freedom in building the language. Since one can decide on its own concepts, they can be changed quickly. However, the elements of the language might not be widely known, thus it may be ambiguous and the whole tool support has to be implemented also since there is no de-facto implementation.

	UML metamodel	Subset of UML extended with own metamodel	UML metamodel with modalities and execution kinds	Custom metamodel
Tooling	Papyrus	PlayGo	ScenarioTools MSD	ScenarioTools SML
Design	✔	✔	✔	✔
Simulation	✗	✔	✔	✔
Consistency analysis	✗	✔	✔	✔
Structural analysis	✗	✔	✔	✔
Code generation	✗	✔	✗	✗
Verification	✗	✔	✗	✗
Runtime monitor	✗	✗	✗	✗

Table 3.1: Comparing tools for scenario-based specification languages

In between the two ends, there are many stages which involve extending a standard language with new elements whose semantics should be defined by the designer of the language. Moreover, if the language would be used in a field then having a corresponding tool which supports this formalism is advantageous to have.

3.1.2 Functionality of the tool

The tool's functionality aspect mostly focuses on how much it supports the engineer to design consistent models and what kind of functionality the tool provides for validating such models and generating implementation artifacts from them.

The first and most important functionality is the *editor*, where the model can be designed using the elements of the language. It is also advantageous if the editor has *analysis* capabilities.

In *analysis*, I make a difference between *structural* and *consistency* analysis. The former includes the ability to ensure the references between the objects exist and conform to the well-formedness constraints. In comparison with that, *consistency* analysis means the model elements do not contradict to each other and has a consistent view of the (sub)system as a whole. For example, if multiple communication sequences can be designed then these sequences may have to consider each other so that they do not prescribe ambiguous communication situations.

Besides the editor, a *simulator* is also good to be included, if the metamodel of the language includes elements that can be dynamically executed and simulated in design time. Thus the probable errors can be detected and fixed early in the development process, which can result in lower costs.

The advantage of the model-driven methodology is, the platform-specific implementation, deployment and runtime artifacts can be automatically generated from the models, thus eliminating the possibility of having code bugs, assuming the *code generator* and the model are correct (Section 2.1). Thus it is very good if the tool has a code generator which automatically generates the implementation from the models.

Model analysis and validation is usually not enough to ensure the error-freeness of the design. Thus formal *verification* should be applied. Such methods check every possible configuration and state which may occur during the lifetime of the system and warn the system engineer to prepare for these situations. In general case the possible state space might be infinite, thus these methods have difficulties in real world applications.

Although formal verification might not be applicable in every environment, the mitigation of runtime errors and failures is not impossible. Should the tool be able to generate *runtime monitors*, the system can be prepared for these failures and act automatically without human intervention, e.g. in safety-critical embedded systems.

3.1.3 Evaluation

As it can be seen in Table 3.1, even though *Papyrus* implements a standardized modeling language (UML Sequence Diagram) and was claimed as having been used in numerous industrial case studies [9, 10, 11], it still lacks basic analysis and simulation functionalities.

On the other hand, *PlayGo* which extends a subset of UML with custom metamodel (LSC) has richer functionality. Even though there is there is no reference for industrial case studies for this tool, numerous publications and PhD dissertations were inspired by it.

Although *ScenarioTools MSD* and *ScenarioTools SML* use rather different metamodels, they implement the same functionalities. Probably, it is due to the fact that the same team implemented both tools. Comparing the last two tools, they use totally different metamodels. The latter one has a custom metamodel which can express assumptions

about the (physical) environment much easier, than the first one. Moreover, the latter one (SML) was under active development, at the time of writing the thesis, but the former one (MSD) had been obsolete.

The conclusion regarding the tool comparison is, PhD dissertations and research papers have higher motivation for implementing experimental features for modeling tools, and having a custom metamodel gives higher flexibility in modeling and implementation.

In the following sections the aforementioned tools (Section 3.2 *Papyrus*, Section 3.3 *PlayGo*, Section 3.4 *ScenarioTools MSD*, Section 3.5 *ScenarioTools SML*) are going to be presented, with respect to the comparison aspects.

3.2 Papyrus

Papyrus is a modeling environment that is constructed by sets of plug-ins in Eclipse. The great advantage of Papyrus is that, it supports designing models that conform to different modeling standards, e.g. UML, fUML, SysML, MARTE, EAST-ADL, RobotML, UML-RT, ISO/IEC 42010.

Although the extent of the tool support for each standard varies, developers can extend the different editors and create new views or enhance the model editors with automatic code generators that create platform-specific implementation from the platform-independent models.

Papyrus has been successfully applied in several industrial case studies, e.g. in the avionics as a collaboration platform [9], in humanoid robot design [10], and in transition from document-centric to model-centric design [11].

3.2.1 Design

As far as scenario-based modeling is concerned, Papyrus contains an editor that supports designing UML-conformed Sequence Diagrams. The engineer can add different lifelines that represent different entities that communicate with each other via messages. Diagrams can contain all the elements that were introduced in Section 2.4.3.

3.2.2 Simulation

At the time of writing the thesis, Papyrus offers no simulation for Sequence Diagrams. However, an extension called *Moka*[1] provides an execution engine for fUML models which are practically activity diagrams with formal semantics.

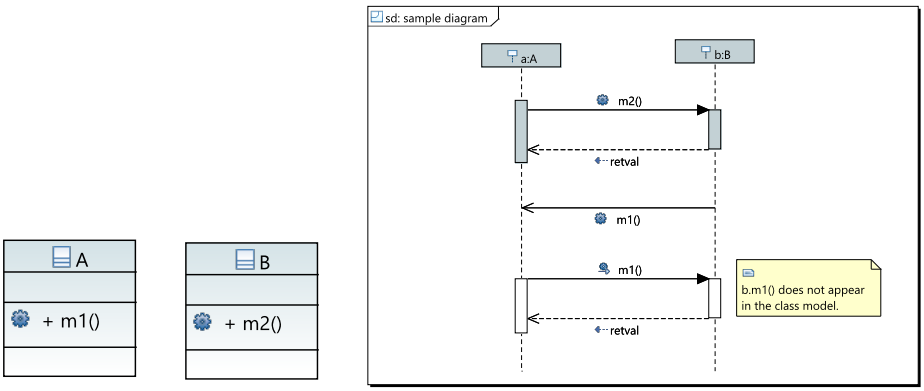
3.2.3 Consistency analysis

At the time of writing the thesis, Papyrus offers no consistency analysis of Sequence Diagrams.

3.2.4 Structural analysis

Although Papyrus offers no built-in structural analysis for Sequence diagrams, it can be extended with custom OCL constraints, that shall be registered in the Papyrus's Validation framework. A *class model* can be created, which contains the corresponding classes that are represented in the sequence diagrams as lifelines. The classes store what operations they can perform, and these operations can be directly referred from the sequence diagrams as synchronous or asynchronous messages.

[1] `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`, last access: 15/12/2017



(a) Sample class model **(b)** Sample Sequence Diagram

Figure 3.1: Sample Papyrus diagrams

Due to the lack of built-in validation, such messages can be represented in the sequence diagrams, which are not in the *class model*. What's more, even those operations can be referred by a message, which belong to the opposite class depicted in Figure 3.1.

3.2.5 Code generation

At the time of writing the thesis, Papyrus offers no code generation from Sequence Diagrams. However stub of Java classes and methods can be generated from the elements of the *class model*.

3.2.6 Verification

At the time of writing the thesis, Papyrus offers no verification for Sequence Diagrams.

3.2.7 Runtime monitor

At the time of writing the thesis, Papyrus does not offer generating runtime monitors for Sequence Diagrams.

3.3 PlayGo

PlayGo[2] [26] is a comprehensive tool for behavioral, scenario-based programming, built around the language of LSC. It is an Eclipse-based derivative of *Play-Engine*[3] [27] that was the first proof-of-concept tooling for Live Sequence Charts.

PlayGo supports designing, simulating LSCs, and generating Java code from them. Besides, consistency analysis is applied during simulation via statechart synthesis and model checkers.

3.3.1 Design

Designing Live Sequence Charts in PlayGo is done through the *play-in* process. It means user can create charts by telling in controlled natural language sentences what the chart should do: what objects shall be on the chart, how they should interact. The built-in natural language processor parses the sentence and constructs the chart automatically. The

[2] `http://wiki.weizmann.ac.il/playgo/index.php/Main_Page`, last access: 29/11/2017
[3] `http://www.wisdom.weizmann.ac.il/~playbook/`, last access: 29/11/2017

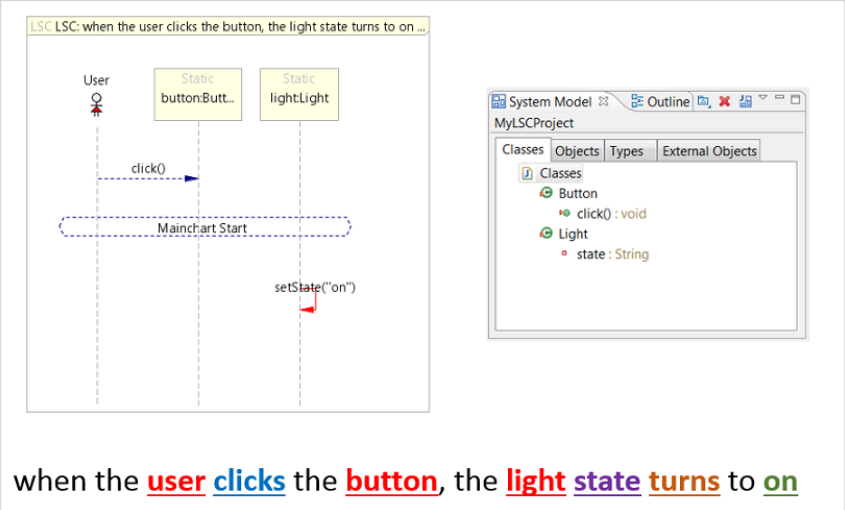


Figure 3.2: Sample LSC via natural language specification

part-of-speech tagger can be helped by manually assigning tags (such as object, subject, message, method) to the words.

Besides, according to the manual[4] of the software, a complete requirements document can be imported and transformed into LSCs in batch. Each sentence will be transformed into a single LSC. Decisions are made by using a statistical model that is trained on previously analyzed requirement documents.

An example Figure 3.2 taken from the manual of the software, depicts an LSC that was specified via the natural language play-in method. Words of the sentence were underlined according to their tags: red means subject (lifeline), blue means operation (message), from purple brown and green words an attribute value change method call is generated. It should be noted that although it is a *universal* chart that is depicted a bit strangely: the *prechart* is indicated with the cold *click()* message and the *Mainchart Start* cold condition.

Although natural language processor could make it easier to design a chart, it has some serious limitations. First, each chart has to be composed by one sentence. Second, the parser is error prone: once it parsed the sentence it is not willing to change the tags for the words, because the part-of-speech tagger is disabled.

3.3.2 Simulation

One big advantage of PlayGo is the simulation of Live Sequence Charts via the so-called *play-out* process. According to Marelly et al. it is a process that is used for testing the behavior of the system via firing user and system actions in any order, and checking the ongoing responses of the system [28]. The process is depicted in Figure 3.3.

PlayGo engine has to monitor precharts of all *universal* charts to choose if there is any whose prechart can be applied to the occurred input sequence. If there is any, then that universal chart's mainchart should be executed as well. If any hot message or hot condition violation is occurred, then the system should abort, because a universal chart's hot messages should be delivered and their hot conditions should evaluate TRUE always.

It should be noted that during play-out *existential* charts are *monitored*, that means the engine simply tracks the events in the chart as they occur.

[4] `http://wiki.weizmann.ac.il/playgo/index.php/How_to_Play-In`, last access: 29/11/2017

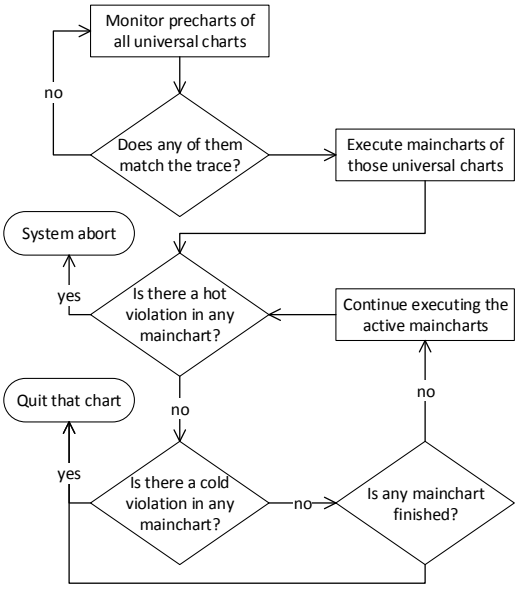


Figure 3.3: Play-out process in PlayGo

A network of LSCs satisfies a system run, iff (1) all *universal* charts were always executed successfully, and (2) all *existential* charts were executed successfully at least once in that run.

As it can be guessed, choosing the next method to be fired in the play-out is a nontrivial task for simulating and testing LSCs, and it has great effect on the simulation result. According to the *PlayGo manual*[5], the tool has four built-in strategies for play-out:

- *naïve*: "The naïve play-out strategy is the simplest one. It arbitrarily chooses a non-violating method from among the current set of methods that are enabled for execution in at least one chart, but which are not violating in any chart."

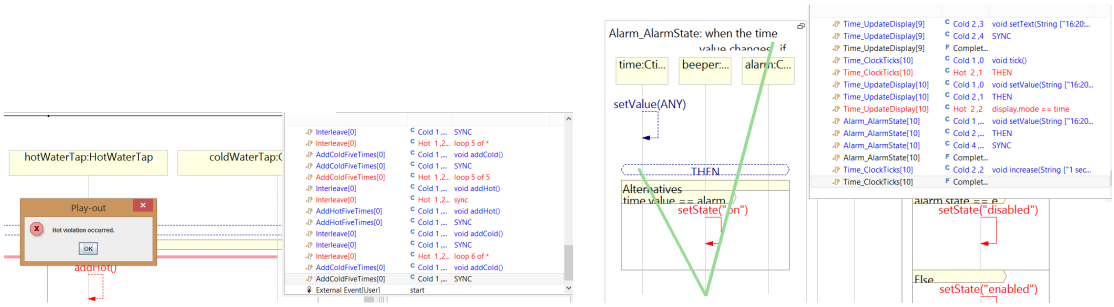
- *random*: "The random play-out strategy is similar to the naïve play-out strategy. However, it chooses the next method to execute randomly, using a 'seed' number. The user can either choose a constant seed, in which case the same method will be selected in repeated runs, or ask PlayGo to use random seed, thus causing different, random method selection in each run."

- *interactive*: "The interactive play-out strategy allows the user to choose the next method to execute. TA dialog is presented to the user that lists all the currently enabled non-violating methods and the user is expected to select one of them. The selected method is returned to the play-out mechanism for execution."

- *smart*: "Smart play-out is a smarter, safer way of choosing the next method to execute. It considers not only the current set of enabled non-violating methods, but also looks ahead and picks up a finite sequence of methods that will lead to a successful (non-violating) superstep, if any such sequence exists. For details see Section 3.3.6."

A hot violation occurred during the play-out of the built-in Water Tap example, depicted in Figure 3.4a. While the successful play-out of a universal chart of the built-in Wrist Watch example is depicted in Figure 3.4b.

[5] `http://wiki.weizmann.ac.il/playgo/index.php/PlayGo_Feature_List`, last access: 29/11/2017



(a) Hot violation via play-out **(b)** Successful play-out of a universal chart

Figure 3.4: Play-out examples in PlayGo

3.3.3 Consistency analysis

Harel et al. proposed a method [21] that checks the consistency of LSCs. First, they formalized the LSC specification. Then they defined that a system satisfies an LSC specification, if for every universal chart and every run, whenever a message arrives the run must satisfy the chart, and if, for every existential chart, there is at least one run in which the message holds and then the chart is satisfied.

They stated that an LSC specification is satisfiable if there is a system that satisfies it. After that they proved that LSC is satisfiable if it is consistent. The consistency analysis is done through a translation from LSC to deterministic finite automaton (DFA) accepting the language of a universal chart. For every universal chart a DFA is constructed, then an automaton accepting exactly the runs that satisfy all the universal charts can be constructed by intersecting these separate automata. This intersection automaton will be used in the algorithm for deciding consistency.

Besides the consistency analysis process, they proposed an algorithm for synthesizing a state-based object system, e.g. state machines or statecharts, from consistent LSCs.

Unfortunately these consistency analysis features were not implemented in PlayGo, instead the *smart* play-out mechanism is used there.

3.3.4 Structural analysis

PlayGo provides *structural analysis* as well. It means the messages between the lifelines should be consistent with the operations of the lifeline's type. Thus only those methods can be called in a message, which are explicitly set in the system object model. The model stores what types in the system exist and what methods they have, as depicted in Figure 3.2.

3.3.5 Code generation

Harel et al. proposed a compilation process that generates AspectJ code from LSC [40]. The idea came from the main similarity between the aspect-oriented programming paradigm and the inter-object, scenario-based approach to specification, in order to construct a new way of executing systems based on the latter. The translation process was originally developed as a UML2-compliant tool, called *S2A* that was later integrated into PlayGo.

3.3.6 Verification

As it was mentioned earlier in Section 3.3.2, a *smart* play-out strategy was implemented in PlayGo and was originally proposed by Harel et al. [22]. It considers not only the current

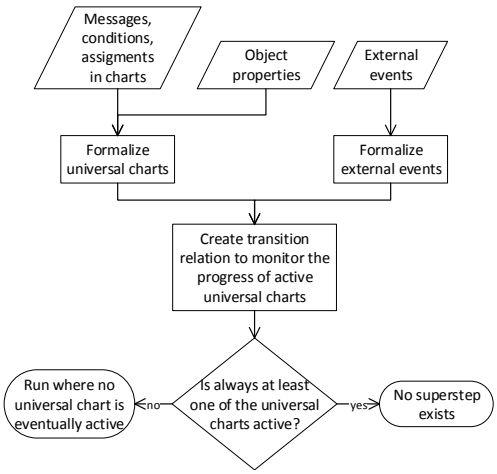


Figure 3.5: Process of smart play-out

set of enabled non-violating methods, but also looks ahead and picks up a finite sequence of methods that will lead to a successful (non-violating) superstep, if any such sequence exists.

Definition 17 (Superstep). A sequence of events carried out by the system as response to the event input by the user. ▪

The approach that was proposed by Harel et al. is depicted in Figure 3.5 and is introduced as follows [22]: they formulated the play-out task – including the charts, messages, precharts, activation of charts, object properties and conditions, assignments, symbolic messages, branching, loops – as a verification problem, and used a counterexample provided by model-checking as the desired superstep. The system on which they perform model-checking is constructed according to the universal charts in the specification. The transition relation is defined so that it allows progress of active universal charts but prevents any violations. The system is initialized to reflect the status of the application just after the last external event occurred, including the current values of object properties, information on the universal charts that were activated as a result of the most recent external events and the progress in all precharts.

The model-checker is then given a property claiming that always at least one of the universal charts is active. In order to falsify the property, the model-checker searches for a run in which eventually none of the universal charts is active; i.e. all active universal charts completed successfully and by the definition of the transition relation no violations occurred. Such a counterexample is exactly the desired superstep. If the model-checker verifies the property then no correct superstep exists.

The smart play-out was originally implemented for the Weizmann Institute model-checker TLV and the CMU SMV model-checker. Later it was adapted in JTLV[6] to be integrated with PlayGo.

3.3.7 Runtime monitor

At the time of writing the thesis, PlayGo offers no runtime monitor service. But the framework could be extended with this feature via the generated AspectJ codes and the model-based representation of the charts.

[6] `http://jtlv.ysaar.net/`, last access: 29/11/2017

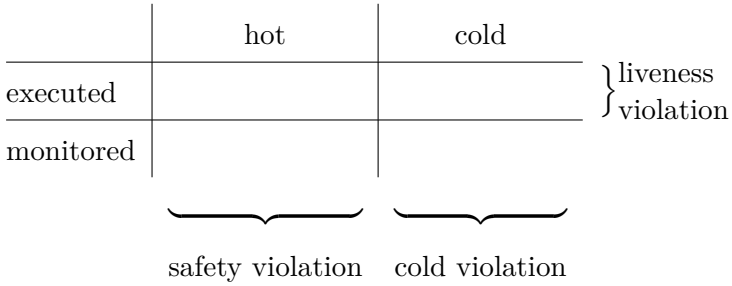


Table 3.2: Message with modalities and execution kinds

3.4 ScenarioTools MSD

ScenarioTools MSD[7] [17] is a framework for design, analysis, and simulation of Modal Sequence Diagrams. Although the framework mostly uses the same semantics for the formalism as introduced in Section 2.4.4, but it was changed in some aspects.

First, besides the formerly used *hot* and *cold* attributes, so-called execution kinds *monitored* and *executed* are introduced. Execution kinds are orthogonal properties for messages, as shown in Table 3.2. *Monitored messages* are those, which may or may not occur. On the other hand, if the message is *executed* it must eventually occur.

Second, the semantics of *hot* and *cold* messages was refined [17]. If the message is *hot*, no message must occur that the scenario specifies to occur earlier or later. With other words: only the referred *hot* message can occur next. If the message is *cold* and a message occurs that is specified to occur earlier or later, the progress of the MSD is aborted. In addition to that, messages which are not specified in the MSD are ignored.

Third, they make a difference between *safety* and *liveness* violations. The former happens if the forthcoming message in the execution of the MSD is either an *executed* or a *monitored hot* message, but instead of that message another one was sent between the two lifelines. If this violation happens for a *cold* message, then it is called a *cold violation*. *Safety* violation must never happen, while *cold* violation may occur and result in terminating the active containing MSD.

Liveness violation happens, if the forthcoming message is an *executed* message that cannot be progressed. On the other hand, an active MSD is not required to progress in a *monitored* cut, that contains a *monitored* message as introduced in Section 2.4.2.

Fourth, besides the system operations and interactions, they model environment behavior so the engineer can make assumptions for that. It was necessary, because the original play-out mechanism proposed by Marelly et al. did not make assumptions about the environment and it was possible that the system had to wait endlessly for an event before making any progress [28].

Thus *assumption MSDs* were proposed by Greenyer et al. [17]. Syntactically, assumption MSDs are the same as other MSDs, but they are marked with stereotype *«assumption MSD»*. Semantically, a sequence of events satisfies an MSD specification if it does not lead to a safety or liveness violation in any requirement MSD or if it leads to a safety or liveness violation in at least one assumption MSD. So the system is only obliged to satisfy the requirement MSDs if the environment satisfies all assumption MSDs.

However, there are some ambiguities in the semantics of the recently introduced *execution kinds*. The definition of *monitored* message requires that, *it may or may not occur*. Besides the definition of *monitored* cut yields that the execution of the referred MSD is not required to progress. Although the *not-required-to-progress* property allows the MSD

[7] `http://scenariotools.org/projects2/msd/msd-specifications/`, last access: 29/11/2017

not to progress even a hot or a cold message occurs. But it can lead to a *safety* or a *liveness* violation in the next execution step, unless the prescribed message arrives.

Another ambiguity comes from the definition of *monitored* message, because it resembles the former definition of a *cold* message: *it can be lost*, which is similar to the *message not being occurred*, introduced in Section 2.4.2. But if it is a *monitored hot* messages, then it means the message *must* (not only *may*) occur otherwise it is a *safety* violation and the system terminates.

3.4.1 Design

ScenarioTools MSD comes with a Papyrus-based Modal Sequence Diagram editor, that extends the formerly introduced Sequence Diagram editor with new MSD-specific properties.

Similarly to Papyrus, it supports creating the system model (*class model*, and *composite diagrams*), consisting of different classes which have different methods. Each lifeline can refer to a class whom it will represent in the interaction. Although structural inconsistencies emerge because of the Papyrus editor.

3.4.2 Simulation

ScenarioTools MSD comes with an Eclipse-based debug feature, that enables the simulation of the Modal Sequence Diagrams.

It extends the original play-out mechanism proposed by Marelly et al. [28]. That has been extended by taking assumption MSDs into consideration as well. Plus the process allows the user to choose the next message that is going to be sent and see its forthcoming effects. So user can choose which message leads to hot or cold *safety* violations or *liveness* violations. The assumption or the requirement MSDs leads to better understanding of the causal relation between the different MSDs and helps the designer to make additional assumptions about the environment.

3.4.3 Consistency analysis

ScenarioTools MSD comes with an extended play-out mechanism that determines and offers the user to choose from the set of forthcoming enabled executable messages. If it leads to a safety or liveness violation in any requirement MSD, then the MSD specification is inconsistent and unsatisfiable by the environment.

Besides, ScenarioTools MSD can synthesize a global finite-state controller based on Büchi automaton for an MSD specification if it is consistent. The construction of the Büchi automaton is detailed in Joel Greenyer's PhD Thesis on *Scenario-based Design of Mechatronic Systems* [16, pp. 27–30].

3.4.4 Structural analysis

ScenarioTools MSD reuses the Papyrus sequence diagram editor for creating Modal Sequence Diagrams. As it was mentioned earlier the editor lacks structural analysis and validation capabilities, because such messages can be represented in the MSDs, which are not represented on the *class model*. What's more, even those operations can be referred by a message, which belong to the opposite lifeline's class.

3.4.5 Code generation

At the time of writing the thesis, ScenarioTools MSD offers no code generation from Modal Sequence Diagrams.

3.4.6 Verification

At the time of writing the thesis, ScenarioTools MSD offers no verification for Modal Sequence Diagrams.

3.4.7 Runtime monitor

At the time of writing the thesis, ScenarioTools MSD offers no runtime monitor for Modal Sequence Diagrams.

3.5 ScenarioTools SML

ScenarioTools SML [18], where SML stands for *Scenario Modeling Language*, is a tool suite for the scenario-based modeling and analysis of reactive systems. The tool is implemented as a framework which consists of custom Eclipse plug-ins for design, analysis and simulation of scenario-based specifications.

The metamodel of Scenario Modeling Language is depicted in Figure 3.6. A *Scenario* that consists of an *Interaction* and has different *role bindings* (dynamic or static), which can have a *kind*:

- assumption: that makes assumptions about the environment.
- requirement: that makes requirements about the system itself.
- specification: that specifies system behavior, similarly to the *universal* charts in Modal Sequence Diagrams.
- existential: that specifies system behavior, similarly to the *existential* charts in Modal Sequence Diagrams.

An *Interaction* consists of different *Interaction Fragments* which can be *Loop*, *Alternative*, *Parallel*, a *Condition*, a *Modal Message* or an *Interaction* itself. These fragments are similar to the Sequence Diagram's *Combined Fragment* concept in some respect.

A *Modal Message* can be *strict*, which means that it must be delivered and no other messages can occur, while waiting or sending this message, as it was for *hot* messages in case of LSC or MSD. Besides, a *Modal Message* can be *requested* that is similar to the *executed* property of MSD messages. It is because, if a *requested* message does not occur eventually, then it is a *liveness violation* for the system.

Besides, an *Interaction* may contain *Constraint Blocks*, e.g. *interrupt*, *forbidden*, *ignore*, *consider* messages. If an *interrupt* message occurs, then the execution of the scenario is interrupted and shall be continued from the last message that was executed. A *forbidden* message means that it should not occur during the execution, otherwise the scenario is aborted. *Ignore* and *consider* messages are those that should be *ignored* and *considered* during the execution of the scenario too.

From different *Scenarios* a *Collaboration* can be composed. A *Collaboration* contains different *Roles* that can either be *static* or *dynamic*. As stated in the *help pages of ScenarioTools SML*[8]: "*static role* can be bound to one specific object in the object system of the runtime environment. It is a fixed role that can not be played by every object. In comparison with that, *dynamic role* can be dynamically bound to any objects that is compatible with the class of the role during runtime. Multiple objects can have this role."

Consequently a *Message* has a sender and a receiver object, which are represented as *Roles* in the metamodel of SML, as depicted in Figure 3.6.

[8] `http://scenariotools.org/scenario-modeling-language/`, last access: 29/11/2017

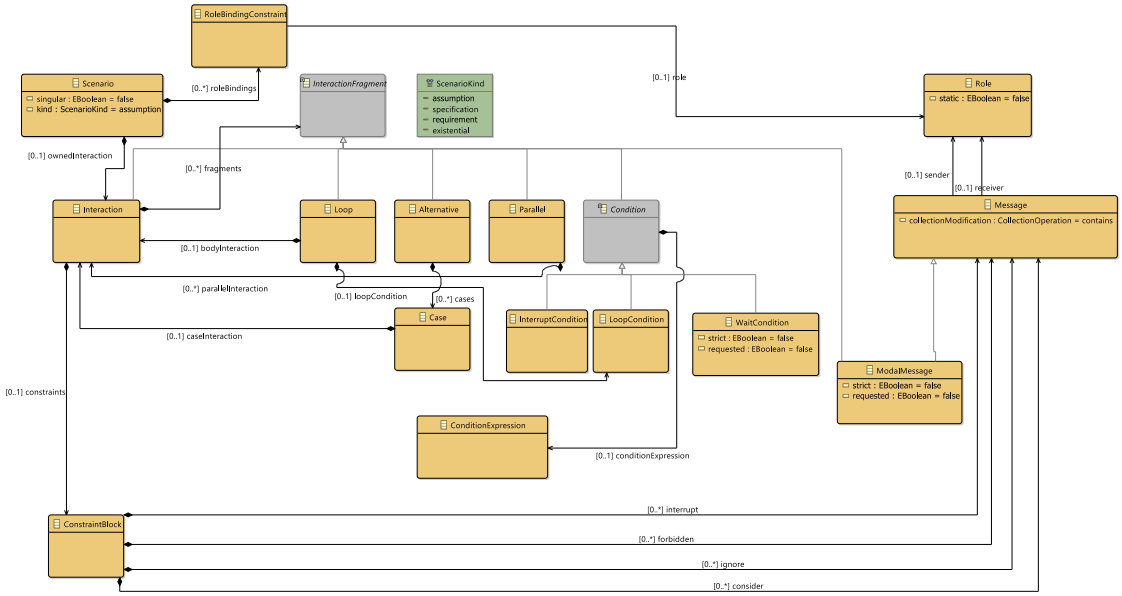


Figure 3.6: Scenario Modeling Language metamodel

3.5.1 Design

The *Scenario Modeling Language* provides a textual editor via the *Xtext Framework* for designing scenarios. The editor is based on a custom Domain Specific Language (DSL) for modeling scenarios. It provides a user-friendly editor with syntax-highlighting, syntactical and semantical analysis functionalities.

An *object system model* can be designed as a class diagram. The object system describes the domain concepts: the elements of the domain concept will communicate with each other and this communication is modeled by scenarios. *Controllable* and *uncontrollable* elements can be chosen from the set of elements in the *object system model*. *Controllable* classes are part of the system, while *uncontrollable* classes are part of the environment for which *assumption* scenarios can be prescribed.

Events can be *spontaneous* or *non-spontaneous*. *Non-spontaneous* events are events that cannot occur randomly and are specified in some scenarios. On the other hand, *spontaneous* events can occur at any time during the execution of the system.

3.5.2 Simulation

Although the *homepage of the ScenarioTools SML*[9] outlines a play-out simulation, depicted in Figure 3.7, that is similar to the one described formerly in Section 3.4.2 for Modal Sequence Diagrams, it could not be made work in the *Scenario Modeling Language* tool, at the time of writing the thesis.

3.5.3 Consistency analysis

Although the *homepage of the ScenarioTools SML* outlines a synthesized controller, that is similar to the one described formerly in Section 3.4.3 for Modal Sequence Diagrams, it is not clear whether it was also implemented for the *Scenario Modeling Language* tool.

[9] `http://scenariotools.org/simulation/`, last access: 29/11/2017

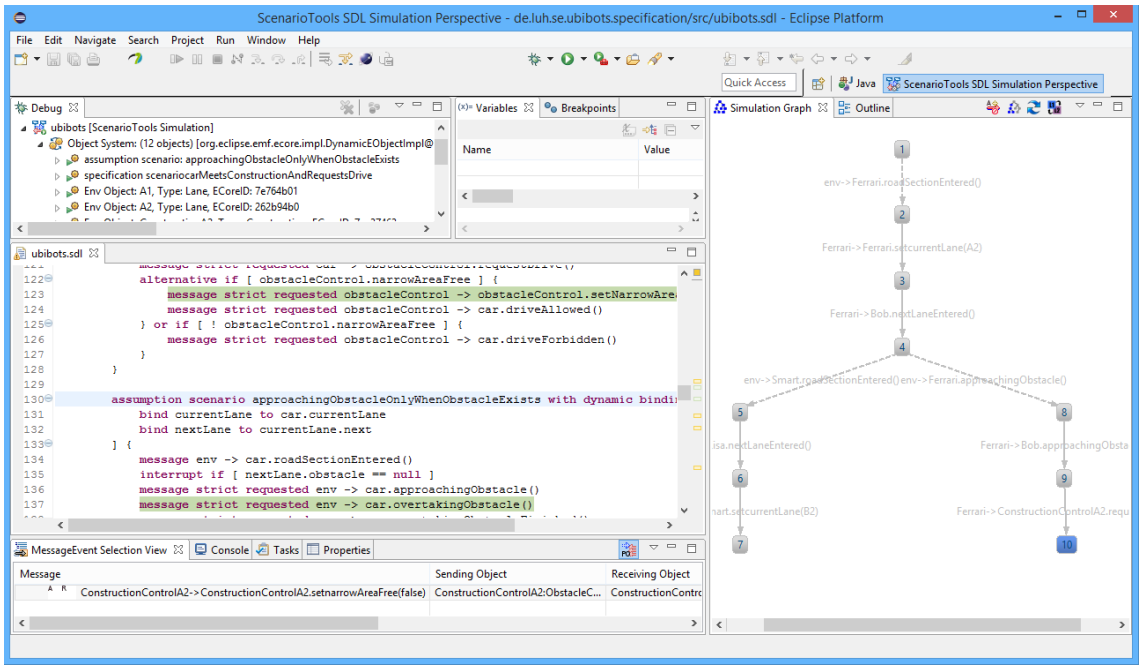


Figure 3.7: Simulation in ScenarioTools SML

3.5.4 Structural analysis

As it was mentioned earlier, the custom DSL-based editor provides syntax-highlighting, syntactical and semantical analysis functionalities. So in the collaboration-descriptor model file, which contains the *scenarios* together with the *roles* of the referred elements from the *object system model*, the referential consistency between the different objects is guaranteed.

It means, only those methods of the objects can be called as messages that were described in the *object system model* earlier. In this way the engineer can be sure that the referred object expects for that message and may be able to handle that.

3.5.5 Code generation

At the time of writing the thesis, ScenarioTools SML offers no code generation from scenarios that conform with the Scenario Modeling Langauge metamodel.

3.5.6 Verification

At the time of writing the thesis, ScenarioTools SML offers no verification for scenarios that conform with the Scenario Modeling Langauge metamodel.

3.5.7 Runtime monitor

At the time of writing the thesis, ScenarioTools SML offers no runtime monitor for scenarios that conform with the Scenario Modeling Langauge metamodel.

Chapter 4

Design of GSL

The aim of this Master Thesis is to design and implement a scenario-based specification language, which supports the validation of scenarios. I wanted to integrate the scenario language as a plug-in into an existing framework, that is:

- *Eclipse Modeling Framework*-based, so that Model-Driven Software and Systems Engineering methodology and the related Eclipse-based modeling technologies can be applied
- *open-source* and *extensible*, so that the framework is extensible with a plug-in that can be attached to the original metamodel through some extension points
- under *active* development at the time of writing the thesis, so that the authors can be reached and there might be a chance for future usage of the scenario language

Gamma Statechart Composition Framework seemed to be a suitable candidate for this, as it was introduced in Section 2.5. The framework is developed at the Fault Tolerant Systems Research Group (*FTSRG*[1]), where I do my Master Thesis.

Disclaimer: the language is called *Gamma Scenario Language* (GSL), because it is going to be included in the next release of the framework. Although the design decisions, which are detailed in the following sections, were discussed with the developers and advisors of the framework, but I developed the GSL.

In order to define a scenario language, it has to be decided *what the purpose of the scenario definitions* (Section 4.1) is, *where those scenarios are used* (Section 4.2) and *what their semantics* (Section 4.3) is. In the following sections these aspects are going to be discussed in details, including some examples of *how a certain trace can be categorized* (Section 4.4) with respect to the scenario.

4.1 Purpose of the scenario definitions

In a state- and component-based system one may define scenarios with many different purposes, e.g. validate the scenario against a given system trace, validate the scenario against the internal behavior of a component, find traces which violate a scenario but does not violate an other one etc. Different tooling is needed to support the different purposes, so deciding on this point is essential in the further elaboration of the work.

In the following subsections several purposes will be discussed, comparing their strengths (advantages) and difficulties (disadvantages) also.

[1] `https://inf.mit.bme.hu/en`, last access: 29/11/2017

Describe the behavior of a component

In most cases scenario definitions are there to define the expected behavior of a component, by regarding the component as a 'black box' and only relying on how it will communicate with other ones. If the component's real (runtime) behavior deviates from this prescription, then it might imply the component is faulty.

It should be noted that, not only the expected behavior can be specified in a scenario definition but also the faulty behavior. But in this case it must be stated, which behavior the scenario describes. It must be decided early if the scenario describes an *expected* or an *erroneous* behavior, otherwise the interpretation can be confusing. See Section 4.3 for more details.

The advantage of this approach is, these scenarios can define contracts which should be adhered to. On the other hand, designing every possible scenario definitions can be a tedious work and it might not be exhaustive with respect to every possible scenario. So deriving them from automatically generated formal models might be easier.

Validate the conformity of the component's internal behavior

Does the communication through the interface conform to the internal behavior of the component? Are the incoming events always handled or is there any of them which gets ignored, are the outgoing events sent at all? Should the component receive a signal through a port, is the component able to send the response signal as it is described by the scenario?

Although some of these questions might be addressed by static analysis too, but it might be interesting to see in what cases the internal behavior deviates from the expected one.

The advantage of this approach is, the component's conformity to the port's interface can be validated.

On the other hand the disadvantages include:

- the internal behavior of each component can be different from component to component, so the reusability of these scenario definitions is questionable.
- more specific to the internal behavior, than to the communication itself. The emphasis is more put on the communication *between the port and the internal behavior*, than on the communication *between the components*.
- validation might be dependent on the representation of the internal behavior. However, this representation can be transformed to another one. For instance, if the internal component is a YAKINDU statechart, then it can be transformed to a timed automaton, by the Gamma framework, and the questions raised above can be answered there.

Generate runtime monitors from the scenario definitions

One may generate runtime monitors from scenario definitions, in order to alert if the system's runtime behavior deviates from the predefined correct behaviour [31, 32]. It is essential in safety-critical embedded domain, because a malfunctioning system may threat human lives and cause harm to them, e.g. in a nuclear power plant, in a railway interlocking system or in the control system of an airplane.

The advantage of the runtime monitors is that the user can get an instant notification if the system deviates from its acceptable behavior and the safety system can react automatically to solve the problem as quick as possible. Supposing that the safety system is functioning well and the notification can be received.

The disadvantage of runtime monitor is that the tools which help identifying deviating behavior, e.g. timed automaton, might not scale very well if the system has many different configurations and states.

Generating traces for model-based testing

In the Model-Based Testing methodology a simplified representation of the System Under Test is created, where each configuration of the system is represented. Different traces (event sequences or change sequences) are generated from this model which can be used as test scenarios to validate the system.

The advantage of this approach is that traces can be generated from a representation which is independent from the test framework. On the other hand, manually designing scenarios from which traces can be generated is a tedious work and it might not be exhaustive with respect to the transition coverage that could be more easily achieved by other formal representations (e.g. timed automaton or Petri nets).

Log validation

Log validation means that given trace can be verified if it conforms to a predefined scenario, in order to classify the trace as *erroneous* or *acceptable*. This purpose is similar to generating traces for model-based testing, but here the direction is inverse: a scenario definition has to be found in a log, which decides if the trace is an *erroneous* or *acceptable* one. The advantages and disadvantages of this purpose is similar to aforementioned approach of generating traces for model-based testing.

Decision

From the possible purposes above the *describe the behavior of a component* was chosen, because it is the most general one, In the future, building on top of my work, one may define scenarios from which *runtime monitors* or *test traces* can be generated.

There is another usage of the scenarios in this approach. Scenarios can be validated in a sense that if there is a trace for which two scenario definitions end up in an ambiguous decision, one of them says it is an *accepted* behavior while the other one says it is *erroneous*, then a feedback can be given to the user. So the contradiction between scenarios can be already found at design time.

4.2 Places for scenario definitions

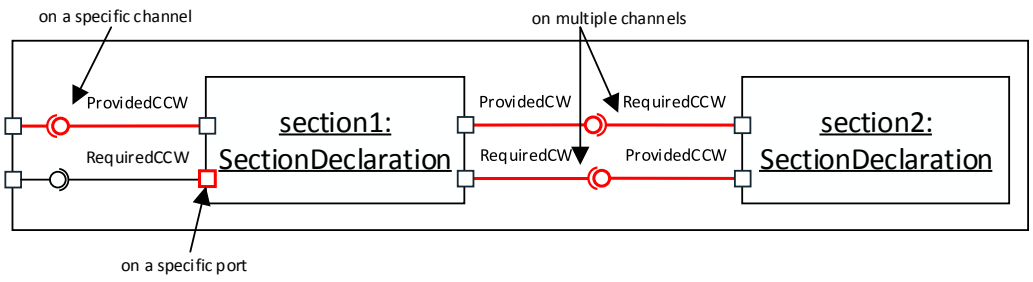


Figure 4.1: Places for scenario definitions

Scenarios can be defined at various points in the system design as it is depicted in Figure 4.1. In the following sections the possible places will be discussed by comparing their advantages and disadvantages.

On a specific port

Defining many scenarios for a particular port that realizes an interface in either *provided* or *required* mode, and validating the scenarios' compatibility with each other as introduced in Section 4.1.
The advantages of this approach are:

- scenario definitions could be reused on other ports also, which realize the same interface in the same mode.
- it can be also reused on ports that are either on the border of the internal component or a composite system.
- existing scenarios can be extended to more specific ones on a particular port and in this case the compatibility of that specialization with the other scenario definitions might be also examined.
- compatibility of scenario definitions, depending on the realization mode, can be also examined. Can two definitions be compatible with each other if the port's realization mode is changed for the other one (from *provided* to *required* or vice versa)?

However, a big disadvantage of this approach is that the scenario definition does not include the other port in any way, because it just refers to one particular port. It might have a drawback when other end (port) of the communication also has to be considered.

On a specific channel

Defining scenarios for a channel, by explicitly marking the two (or more) endpoints. It is similar to the previous subsection, but now the channel is defined.

The advantage of this approach is that scenario definitions might be also reused on other channels if ports realize the same interface in the same mode and the channel type is the same as in the previous scenario.

On the other hand the disadvantages include:

- the channel type might have an implication for the scenario definition, because in *broadcast* channels the receiver ports must *require* the specific interface and they cannot have any *outgoing* event (due to the semantics of Gamma Framework).
- every possible combination of realization modes (*provided*, *required*) should be defined by the user in the scenarios.
- if a specific port of a specific component expects the scenario to be different from the other ones, then it should explicitly say how it expects the other port to "behave". Compared to the previous subsection, where the only thing the user has to define is that particular port in which he is interested in and does not have to explicitly model the other port.

On multiple channels

Defining scenarios for multiple channels at once in a composite component. It is similar to the previous subsection, but on the diagram multiple interfaces and events would be presented.

The advantage of this approach is that the user will get an overview of the whole communication in the composite component.

On the other hand, the disadvantages include:

- defining a new scenario might be cumbersome, if many channels are involved.
- since the communications on different channels might be independent from each other, parallel regions should be introduced early.
- tooling has to be implemented well, especially the editor so that the scenario is easily comprehensible.
- additional complexity for validating the compatibility of the scenarios in the same composite component.
- scenario definitions might not be reusable in other composite component, unless the internal components and their port connections are the same as in an other composite component.

Decision

From the possible places itemized above, scenario definitions *on a particular port* was chosen, because this is the base for every other case and many things can be validated already that would also have to be validated in the other cases. For example, are the scenario definitions compatible with each other, does it conform to the port declaration.

4.3 Interpretation of a basic scenario

In order to design a scenario-based specification language, its expressive power and interpretation have to be decided. In this section the *formalism* of the scenario language is introduced on a high level. Then the *interpretation* of a scenario is discussed, following the aspects raised by Micskei et al. in the *Interpretation of a basic Interaction* section of their survey [42]. Since GSL has some similarities with UML Sequence Diagrams, the same aspects should be discussed also.

Finally in Table 4.1 several example scenarios are shown together with traces in order to decide when will the trace be *valid, invalid or inconclusive* for the given scenario.

Formalism

Regarding the expressive power, in order to include explicit modality that is easier to be expressed by the nature of the language, than in UML Sequence Diagrams, I extended the Live Sequence Charts (LSC, introduced in Section 2.4.2) formalism with some extra elements, detailed in Section 5.1.

Harel et al. compared papers on the expressive power of the LSC formalism in their survey [25]. Some papers were referred in the aforementioned survey, which suggested translations from fragments of the language into temporal logics. A first embedding of a kernel subset of the language (which omits variables and some other elements) into a CTL* was given. It was shown that existential charts can be expressed using CTL, while

universal charts are in the intersection of linear temporal logic and computation tree logic: $LTL \cap CTL$.

In the Gamma Scenario Language (GSL), only *universal charts* are supported, with non-empty *prechart* which contains only *cold* messages (signals). The *mainchart* cannot be empty either, but it can contain *cold* and *hot* messages also. In this way, the scenario's 'activation' can be specified by preconditions and it prescribes a certain behavior.

Representing signals

Signals are identified by a tuple ⟨modality, direction, interface, interface event name⟩, and a unique sequence number, which shows the place of the signal in the sequence of signals. The problem of overlapping signals is circumvented, because (1) the editor only supports defining non-interleaving signal sequences and (2) in the validation phase no new signal can be sent (in any direction) until the one that is being processed is not finished yet.

In a trace, that is a sequence of signals, a particular message may be sent multiple times, in the same direction, between the same participants. However, in the sequence of signals, the signal's sending occurrence is always before its reception occurrence and no other signal can be sent / received in the meantime. It means, there is a strict ordering of signals.

Categorizing traces

Combining the message-level modality with the chart-level modality results in having three different conclusions for the trace (sequence of signals), against which the scenario was running:

- The trace is *valid*, if in the scenario all the hot messages were delivered and all the hot conditions were true and all the hot locations were exceeded.
- The trace is *invalid*, if in the scenario, a hot condition was violated or a hot location was not exceeded. It means there was a serious deterrence from the specified behavior.
- The trace is *inconclusive*, if in the scenario either a cold condition or a cold location was not exceeded. It is not an error, it simply means the scenario cannot be fitted to the recent execution.

Complete or partial traces

Due to the semantics of the precharts of the LSCs, there can be any prefix in the system trace, before the behavior specified by the scenario occurs. If the prechart is a succeeded, then the mainchart must also either succeed or end in a *cold violation*. If so, then any signals can arrive, but the analysis starts the matching process from the beginning of the scenario. If the mainchart ends in a *hot violation*, then it will stay in this location for the whole suffix of the trace.

So the scenario language supports matching partial traces until this subtrace ends in either success or in *cold violation*. As soon as there is a subtrace which ends in a *hot violation*, the whole trace is invalid (see the previous section on *categorizing traces*).

4.4 Example traces against scenarios

In the following table several scenarios are shown, along with some traces. The category (valid, invalid, inconclusive) for the trace is decided, based on the scenario definition.

In the scenario definition it is assumed that there is only one sender, who may send signals. Sending a signal called a is denoted by $!a$, receiving this signal is denoted by $?a$. A period (.) between the signals is just a concatenation character in the trace. $\langle ?a \rangle$ denotes that receiving an a is in a prechart. Cold signals are written in *italics*, hot signals are written in **bold**.

Scenario	Trace	Trace category
$?a.?b.!c$	$?a.?b.!c$	valid
$?a.?b.!c$	$?a.?a.?b.!c$	if at least $?a$ is in prechart → valid if there is no prechart → invalid
$?a.?b.!c$	$?d.!c$	if at least $?a$ is in prechart → inconclusive otherwise → invalid
$\langle ?a \rangle.?b.!c$	$?a.!b$	if $?b$ → inconclusive if $?b$ → invalid
$\langle ?a \rangle.?b.!c$	$?a.?b.!c.!d$	valid, but another scenario may decide it is invalid, due to $!d$

Table 4.1: Example trace categorizations based on scenarios

So any prefix is allowed until the *prechart* matches, after that any *cold* violation results in an *inconclusive* trace, any *hot* violation results in an *invalid* trace, *no violation* results in a *valid* trace for the scenario.

Chapter 5

Language Development

The aim of this Master Thesis is to design and implement a scenario-based specification language, which supports the validation of scenarios. This chapter introduces the implementation of the *Gamma Scenario Language*'s (GSL).

In Section 5.1 the abstract syntax of the LSC-based scenario language is given, then in Section 5.2 the concrete syntax of the language is defined along with the structural validation rules which are necessary to ensure correct models.

In Section 5.3 the operational semantics of GSL is defined by giving a mapping between the elements of the language and a finite automaton. Then this semantics is compared to the one introduced by Maoz et al. [40].

Finally, in Section 5.4–Section 5.6 the compatibility validation of scenarios is explained. First, the concept of *scenario compatibility* is defined, then a workflow is proposed and implemented for validating scenarios from their compatibility point of view.

5.1 Abstract syntax

As it was introduced in Section 4.3, GSL is an extension of the Live Sequence Chart (LSC) formalism. The abstract syntax (metamodel) of the language is depicted in Figure 5.1.

The *ScenarioDefinition* depends on a *StatechartSpecification* (Section 2.5) and a *PortReference*. A *PortReference* refers to a *ComponentInstance* and to one of its *Ports* in the previously stated *StatechartSpecification*. In this way several scenarios can be defined (*ScenarioDefinition*) on the same port of a certain component, as it was decided in Section 4.2.

A *ScenarioDefinition* consists of a *Prechart* and a *Mainchart*. A *Chart* is a non-empty list of *Interactions*. The concept of *InteractionFragment* was introduced for the list of *Interactions* in order to allow their reuse, see *CombinedFragment*.

An *Interaction* can be either a *ModalInteraction*, or an *InteractionDefinition* or a *CombinedFragment*.

A *ModalInteraction* is the representation of the message-level modality in LSC. It encapsulates a *Signal* which can be a mandatory (*hot*) or an optional (*cold*) one.

An *InteractionDefinition* represents an interaction between the *Port*, that was referred by the *PortReference* and some other ports which are not represented at all. The *InteractionDefinition* is abstract, because the concrete interaction depends on its semantics. For example, is it a *Signal*, a *Message*, a synchronous or asynchronous *Function Invocation*, etc.

The only extension of an *InteractionDefinition* is a *Signal*, which refers to *Interface* that is realized by the *Port* and to an *Event* from the ones declared by the interface.

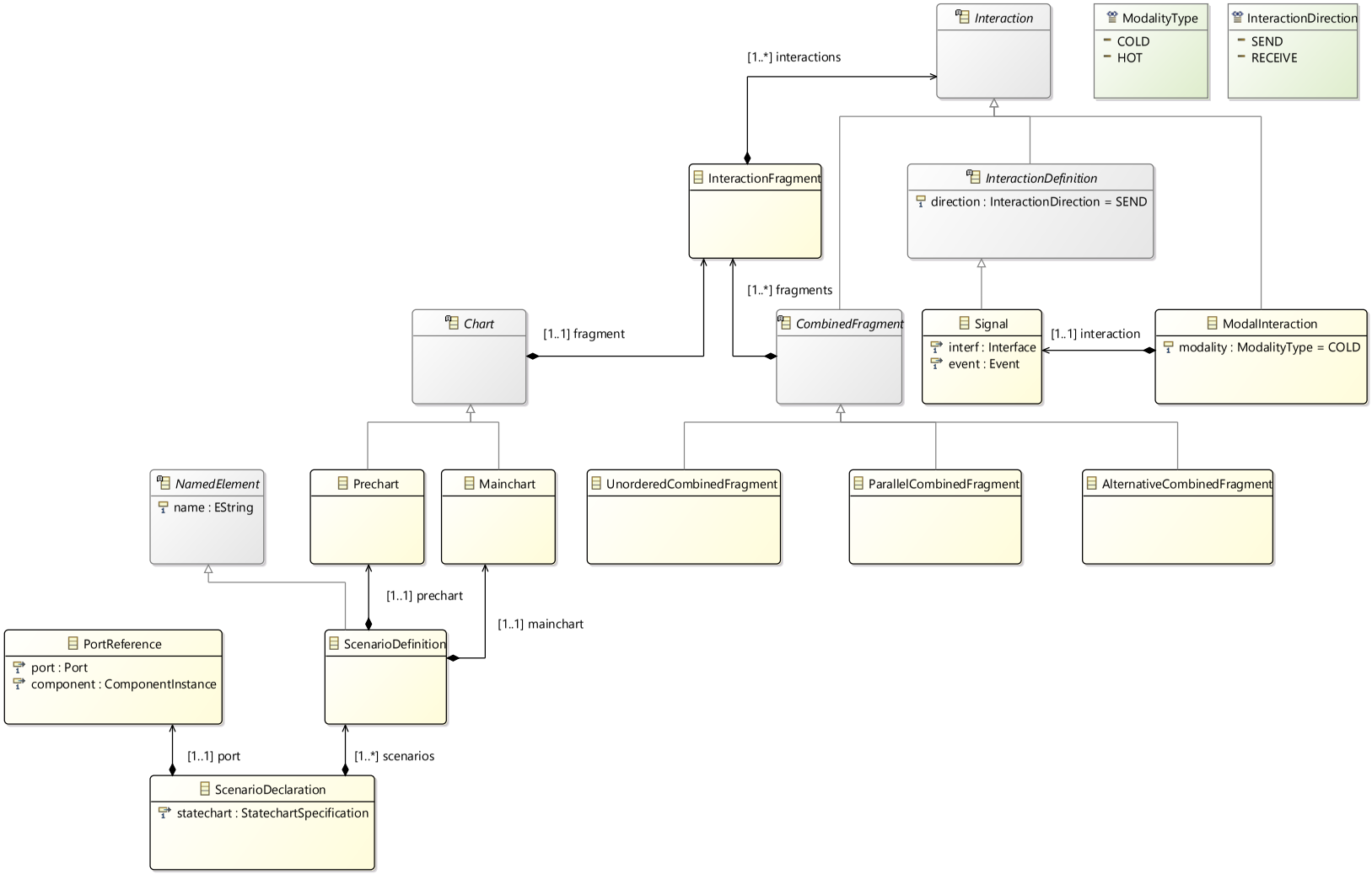


Figure 5.1: Abstract Syntax of GSL

The *CombinedFragment* is an extension of the LSC formalism. The semantics of this concept is similar to the semantics of *CombinedFragments* in the context of UML Sequence Diagrams as depicted in Figure 2.7 in Section 2.4.3. However, there are some differences between the two. In the context of GSL, the *CombinedFragments* do not have a dedicated *InteractionOperatorKind*, but there is a separate concrete class for every implementation (*AlternativeCombinedFragment*, *UnorderedCombinedFragment*, *ParallelCombinedFragment*) whose semantics are covered in the corresponding sections. Moreover, the *InteractionOperands* in this context are non-empty *InteractionFragments* and there is no *InteractionConstraint* implemented yet which could be used as guard condition for each *InteractionFragment*.

An *AlternativeCombinedFragment* contains at least two non-empty *InteractionFragments* from which any of the them can occur in the trace as the next *Interaction*. For its semantics see Section 5.3.

An *UnorderedCombinedFragment* contains at least two non-empty *InteractionFragments* whose contents are concatenated after each other in every possible combinations, but each *InteractionFragment* is handled atomically. It means (1) the order of *Interactions* in the *InteractionFragment* is preserved and (2) between two *Interactions* from the same *InteractionFragment* no other *Interaction* can come from the other *InteractionFragments*. For its semantics see Section 5.3.

A *ParallelCombinedFragment* contains at least two non-empty *InteractionFragments* whose contents are combined in every possible combinations. The only property that is preserved in ordering is the *happens-before* relation (Definition 19 in Section 5.3.2). It means, the order of *Interactions* in the combined *InteractionFragment* has to respect their original ordering within their original *InteractionFragment*, but between two *Interactions* from the same *InteractionFragment* any other *Interaction* can come from the other *InteractionFragments*, assuming their ordering is also preserved. Compared to *UnorderedCombinedFragment*, in this case the atomicity of *InteractionFragments* is not required, only the *happens-before* relation of events within the same *InteractionFragment* has to be preserved. For its semantics see Section 5.3.

5.2 Concrete syntax

In order to support the model-driven engineering and the easy usage of the scenario language, an Xtext[1]-based [12] textual editor was developed. Xtext is a framework for developing Domain-Specific Languages.

From the language designer's perspective, the only thing he has to design is the grammar of the language, the other classes (e.g. editor with syntax highlighting and autocomplete, lexer, tokenizer, parser) will be automatically generated by the framework. On the other hand, these classes can be customized if necessary (e.g. scope provider, model validation).

From the perspective of the language user the text that is typed into the editor will be parsed, based on the grammar rules, and a model will be automatically created which conforms the grammar. Should a parsing or a custom validation error occur, an error notification will appear in the editor and the user can either automatically fix the problem, by predefined auto-fixes, or manually find a solution. It is very convenient both from the perspective of the language designer and the user.

Although the complete grammar of the scenario language is included in the appendices in Listing A.1, a short excerpt is shown in Listing 5.6 how a *ScenarioDeclaration* refers to a *StatechartSpecification* and a *PortReference*; and how a *ScenarioDefinition* is constructed

[1] `http://www.eclipse.org/Xtext/`, last access: 15/12/2017

```
cold receives Protocol.reserve
hot receives Protocol.cannotGo
hot sends Protocol.canGo
cold sends Protocol.release
```

Listing 5.1: Modal Interactions

```
alternative {
  hot sends Protocol.canGo
} or {
  hot receives Protocol.release
  hot sends Protocol.canGo
}
```

Listing 5.2: Alternative Combined Fragment

```
unordered {
  hot sends Protocol.canGo
} and {
  hot receives Protocol.release
  hot sends Protocol.canGo
}
```

Listing 5.3: Unordered Combined Fragment

```
parallel {
  hot sends Protocol.canGo
} and {
  hot receives Protocol.release
  hot sends Protocol.canGo
}
```

Listing 5.4: Parallel Combined Fragment

```
import FourSection
port section1.ProtocolProvidedCW

scenario A {
  [
    cold receives Protocol.reserve
    cold receives Protocol.cannotGo
  ]
  {
    hot sends Protocol.canGo
    hot receives Protocol.release
    hot sends Protocol.canGo
}

scenario B {
  [
    cold sends Protocol.reserve
    cold receives Protocol.canGo
  ]
  {
    alternative {
      hot sends Protocol.cannotGo
    } or {
      hot receives Protocol.canGo
      cold sends Protocol.cannotGo
    }
}
```

Listing 5.5: Multiple Scenario Definitions for a Port

Figure 5.2: Scenario Concrete Syntax Examples

from a *Prechart* and a *Mainchart*. Besides, in Figure 5.2 several examples illustrate the outlook of the syntax of the elements in an editor. For clarity purposes, the *PortReference* is not shown in the short examples, only in Listing 5.5.

Several structural validation rules were implemented for scenarios to ensure the correctness and well-formedness of the created models. In Section 5.2.1 these rules are itemized and categorized by severity.

```
ScenarioDeclaration returns ScenarioModel::ScenarioDeclaration:
  'import' statechart = [StatechartModel::StatechartSpecification]
  port = PortReferenceDefinition
  (scenarios += ScenarioDefinition)+
;

PortReferenceDefinition returns ScenarioModel::PortReference:
  'port' component = [CompositeModel::ComponentInstance] '.' port = [
    StatechartModel::Port]
;

ScenarioDefinition returns ScenarioModel::ScenarioDefinition:
  'scenario' name = ID '{'
    prechart = PrechartDefinition
    mainchart = MainchartDefinition
  '}'
;

PrechartDefinition returns ScenarioModel::Prechart:
  '['
    ^fragment = FragmentDefinition
  ']'
;

MainchartDefinition returns ScenarioModel::Mainchart:
  '{'
    ^fragment = FragmentDefinition
  '}'
;
```

Listing 5.6: Gamma Scenario Language (GSL) Grammar (excerpt)

5.2.1 Scenario validation rules

In order to ensure the correctness and well-formedness of the created models, several structural validation rules were implemented for the scenarios. These are categorized by ascending severity as follows:

Info

- If there is a *ParallelCombinedFragment* in the *InteractionFragment*, then a marker is put there in order to notify the user about the complexity of generating every possible partial orderings of the fragments.

Warning

- The modality of each *ModalInteraction* in a *Prechart* should be cold. *Precharts* filter the traces before the *Mainchart* is evaluated, so using the hot modality does not make sense in this setting.

- There should be at least one *ModalInteraction* with hot (mandatory) modality in the *Mainchart*, otherwise the respective *ScenarioDefinition* does not prescribe any compulsory behavior and every trace which runs against the scenario will eventually be either *valid* or *inconclusive* as it was introduced in Section 4.3.

Error

- *ComponentInstance* referred by the *PortReference* should exist in the respective *StatechartSpecification*, otherwise there is no such component whose *Port* could be referred.
- *Port* referred by the *PortReference* should belong to the respective *ComponentInstance* that was referred by the same *PortReference*, otherwise there is no *Port* for which the *ScenarioDefinitions* could be defined.
- *ScenarioDefinitions* should have unique names, otherwise the result of the compatibility validation cannot be back-annotated correctly into the editor. See Section 5.4.
- Every *Signal*'s direction should conform to the referred *Event*'s direction with respect to the *Port*'s realization mode. For instance, if the *Port* realizes the *Interface* in provided mode and the *Interface* contains the respective *outgoing Event*, then the *Signal* should *send* this event. However, if everything remains the same as before, but the *Signal receives* the respective *Event*, then it is wrong.
- First *Interactions* modality (hot or cold) in the *InteractionFragments* which belong to the same *CombinedFragment* has to be the same, otherwise the violation locations cannot be generated correctly as it is detailed in Section 5.3.
- There cannot be two *ScenarioDefinitions* which are structurally the same even if they have different names.
- For those interaction fragments which have a common prefix that consists of the same *Interactions*, the next *Interaction* after the common prefix must have the same modality, otherwise the violation locations cannot be generated correctly.

Most of the validation rules were implemented in an extension of the auto-generated language validator class of Xtext. These validation rules are executed upon every model change, so they have to be completed quickly, otherwise the editor may be frozen which causes a not-so-good user experience. Thus the *redundancy* and the *common prefix* validation rules are implemented in a separate validator that is only triggered before the scenario transformations, which are detailed in Section 5.4.

5.3 Formal semantics

Harel et al. defined the formal operational semantics of LSCs using formal expressions [22]. Moreover, they synthesized state-based object systems and state machines from LSC specifications [21, 23]. However, for my Master Thesis and for scenario compatibility validation an automaton based approach would be more suitable.

Thus in Section 5.3.1 I shortly describe the approach introduced by Maoz et al. [40] where they transformed multimodal scenario-based specification into *scenario aspects*. Then in Section 5.3.2 I describe my contribution, the scenario analysis approach in Gamma. It is based on the work of Maoz et al., but was simplified and modified to fit our needs.

Finally in Section 5.3.3 I give a few examples for how the examples of the concrete syntax, depicted in Figure 5.2, look like as automata.

5.3.1 Semantics given by automaton – S2A approach

Maoz et al. [40] translate each LSC into a *scenario aspect* which is responsible for monitoring relevant events by simulating an abstract automaton whose states represent cuts (see Section 2.4.2) along the LSC lifeline and whose transitions represent enabled events.

The automaton representation of an LSC is built by doing static analysis over the LSC. The analysis involves simulating a run over the LSC, to capture all its possible cuts. Each cut is represented by a state. Transitions between states correspond to enabled events. An additional transition from each cold cut state to a designated completion state corresponds to all possible (cold) violations at the cut. An additional transition from each hot cut state to a designated error state corresponds to all possible (hot) violations at the cut. The scenario ends in a non-cold-violation completion state if it accepts the given run.

For each LSC in the specification, following the construction of the automaton of the LSC, AspectJ code is derived from the respective automamton.

This approach is very similar to the one proposed by Harel et al. [24], where the construction yields an alternating weak word automaton (AWW) to define the semantics of LSC. In the AWW the partition of the states is induced by the partial-order of events specified in the LSC.

5.3.2 Semantics given by automaton – Gamma approach

In this subsection I define the transformation algorithm which translates each *Scenario Definition* into a separate finite automaton, along with those extra elements (various specializations of a *CombinedFragment*) that were not part of the original LSC specification.

The high-level transformation algorithm is defined in Algorithm 6 in Section A.2, some necessary concepts are also formulated in Definition 21. In the following paragraphs the transformation of each model element will be introduced, together with the states naming conventions and the handling of violations and error states. Finally, my approach and the one proposed by Maoz et al. [40] is compared.

Due to space limitations all algorithms of this subsection is defined in Section A.2 of the Appendix, together with Definition 21.

Naming convention for states The initial state of the automaton is always named as *initial*. Every other state's name has a prefix which depends on if the respective signal is in the prechart or the mainchart (*prechart_* and *mainchart_* respectively) and a sequence number appended to it (which is unique with respect to a *Chart*).
Violation and *accepting* states are exceptions in the naming convention. *Violation state* represents *cold* or *hot* violation which is denoted by its name's postfix (*coldViolation* and *hotViolation* respectively). The only *accepting state* can be in the end of the mainchart, thus it is always called as *mainchart_end*.

Violation states and error state The violation states are always reused in each *Chart* of the same *ScenarioDefinition*. So there is only one instance of *coldViolation* state in the *Prechart* and there is only one *coldViolation* and *hotViolation* state in the *Mainchart*. There is no hot violation state in the Prechart, as it is discussed in Section 5.2.1.

Besides, only the *mainchart_hotViolation* violation state is regarded as an error (trap) state, which means for every *Event* on the *Interface* of the *Port* referred by the *ScenarioDeclaration* and for every *InteractionDirection* a loop transition is created in this state, so it cannot proceed. The reason for this construction is that the trace which runs against the *ScenarioDefinition* can be regarded as *invalid* and any continuation of this trace will be *invalid* also.

However, every other *violation state* is connected to the *initial state* via an ϵ transition. In this way the *universal scenarios* can continue until the corresponding trace is regarded as *invalid*.

Accepting state The only state which accepts the trace is the one that is created in the end of the *ScenarioDefinition* after the last *ModalInteraction* is transformed. Although this state is an accepting state, it is connected to the *initial state* via an ϵ transition.

In order to clarify the previous two paragraphs, see Section 5.4 where it is explained why only *mainchart_hotViolation* state has to be differentiated from the other *violation states* and *accepting state*.

ModalInteraction Every *ModalInteraction mi* is transformed into a transition between two states as in ALGORITHM 9.

Should the *Signal* referred by *mi* happen, then a new state and transition is created from the last state before *mi* and the new state. The trigger of the transition contains the *direction* and the *Signal* of *mi*.

For every *Event* and *InteractionDirection* which are not the ones referred by the *mi*, transitions are generated from the last state before *mi* and the corresponding violation state. The chosen violation state depends on the *modality* of *mi* and the type of the *Chart* which contains it.

Signal Every *Signal s* is transformed as introduced in the *ModalInteraction* paragraph. The difference is that a *Signal* does not have *modality*. So it is never transformed alone, but always as part of a *ModalInteraction*.

InteractionDefinition There is no other *InterationDefinition* than *Signal* yet, so see the corresponding paragraph for the transformation.

AlternativeCombinedFragment Every *fragments* of an *AlternativeCombinedFragment acf* are transformed into separated branches starting from the last state before *acf*. For those *fragments* which have a common prefix, the prefix is transformed only once and the corresponding *fragments* continuations are attached to that state of the common prefix where they start from as in ALGORITHM 12. In this way there is no unnecessary repetition of interactions in the beginning of branches.

The continuations are transformed as a list of *Interactions*. The end states of the continuations are finally connected via ϵ transitions to a new state, so that every possible branch of *acf* is finally connected together into one state so that continuous semantics of the transformed *Interactions* is preserved.

In the following there is a short explanation for connecting the last state of each branch to the same state: looking at the *Interactions* from a very high-level, they should look similar to a transition between two states with a trigger as the *Interaction*. However it is too general, so it has to be refined for each concrete *Interaction*. But irrespectively of their complexity, finally each of them should preserve a continuation of the control flow. So connecting the transformation of the next *Interaction* should be made as easy as possible. Thus having only one former state to connect to is the easiest way possible. So every continuation and every branch which does not lead to a *violation state* should be connected by ϵ transitions to the same state in the end.

Besides the behaviors prescribed by the *fragments*, the necessary transitions are also generated if the first *ModalInteraction* in each *InteractionFragment*, let they be *firstModalInteractions* as defined in Definition 21, does not occur in the runtime trace. So for every

Event and *InteractionDirection* which are not the ones referred by the *firstModalInteractions* in each *fragment*, transitions are generated from the last state before *acf* and the corresponding violation state. It should be considered that the triggers of these transitions cannot contain those *Signals* which are referred by the *firstModalInteractions*. The chosen violation state depends on the *modality* of *firstModalInteractions* and the *Chart* type of *acf*. Thus the *firstModalInteractions* should have the same modality thus exists the corresponding validation rule in Section 5.2.1.

UnorderedCombinedFragment Every *UnorderedCombinedFragment ucf*, as defined in Definition 18, is transformed into a corresponding *AlternativeCombinedFragment acf*, then *acf* is transformed as defined in the corresponding paragraph in ALGORITHM 10.

Definition 18. An *UnorderedCombinedFragment ucf* can be regarded as an *AlternativeCombinedFragment*, whose *fragments* are the permutations of *ucf*'s fragments. In each permutation each original *fragment* of *ucf* is regarded atomic, so the *Interactions* of different *fragments* are not mixed together, only the *fragments* are appended as a whole. ▪

ParallelCombinedFragment Every *ParallelCombinedFragment pcf*, as defined in Definition 19, is transformed into a corresponding *AlternativeCombinedFragment acf*, then *acf* is transformed as defined in the corresponding paragraph in ALGORITHM 11.

Definition 19. A *ParallelCombinedFragment pcf* can be regarded as an *AlternativeCombinedFragment acf*, whose *fragments* are the permutations of interactions in the fragments of *pcf* by taking every *Interaction* together and preserving the partial order between only those *Interactions* which belong to the same *fragment*.

With other words: the partial order of those *Interactions* is preserved which belong to the same *fragment*, but the *InteractionFragments* themselves are not regarded atomic. So between two *Interactions* from the same *fragment* another *Interactions* can come from different *InteractionFragments* considering they also obey to their partial ordering.

It should be noted that it has a higher computational complexity, than transforming an *UnorderedCombinedFragment*, because now another *fragments*' *Interactions* can be injected also into each *InteractionFragment*. Thus there is an info-level marker in the editor for such elements, as shown in the validation rules in Section 5.2.1. ▪

In the following, several differences and the similarities between the approach proposed by Maoz et al. and the one introduced by me are the followings [40]:

- in [40] *cold violation* states are regarded as *accepting states*, but a trap state is created from each of them. In this way they do not reconnect these states to the *initial state* via an ϵ transition. Compared to them, I regard *cold violation* states as normal states, so the trace which ends in them is not accepted.

- in [40] in the graphical representation of the LSC the first interactions with dashed arrows and conditional expressions are regarded as cold modalities which form a *prechart* together. Compared to [40] in my LSC formalism there is an explicit notation for *prechart* and for *cold* interactions. However, *conditional expressions* are not supported yet.

- in [40] those states which are transformed from the LSC's *prechart*-like structure are regarded as *accepting states*. These states are trap states for every interaction that is not the one prescribed by the LSC. Compared to them, I do not treat those states which are transformed from the *prechart* as accepting states. However, should

a *cold violation* occur, then there is a transition to the corresponding *violation state* from which an ϵ transition takes back to the *initial state*. The only similarity of the approaches is that the *initial state* is a trap state for every Signal that is different from the first Signal of the *prechart*. But the *initial state* is not an accepting state.

- in [40] an *accepting state* is regarded as a trap state for every interaction that occurs after the last one. Compared to them, I do not regard *accepting state* as a trap state, but it is connected via an ϵ transition back to the *initial state*. See Section 5.4 why it is done so.

- although I have an explicit naming convention for each state in the automaton, as described in the *State's naming convention* paragraph above, it is not clear if Maoz et al. also have such a convention.

- both approaches regard *hot violation* states as trap states for every interaction following the *hot violation*.

- both approaches have an *accepting state* in the end of the transformation, after the last interaction of the mainchart. What is not clear if Maoz et al. also have only one *accepting state* or multiple of them, in case there is an alternative way to end the scenario for a trace to be accepted.

5.3.3 Examples

An example for the difference between an *UnorderedCombinedFragment* and a *ParallelCombinedFragment* is depicted in Figure 5.3 together with their *ScenarioDefinitions*.

Due to space limitations, the *mainchart_hotViolation* state and the corresponding transitions, as defined in the previous section above, are not depicted.

Moreover, on the figures Σ symbol represents the set of every possible combination of *InteractionDirections* and *Events* of the *Interface* referred by SECTION1.PROTOCOLPROVIDEDCW port (both *ScenarioDefinition* refer to the same port). The $\Sigma \backslash \{receive\ Protocol.reserve\}$ expression is a shorthand for enumerating every transition whose trigger is from the specified set.

In Figure 5.3b it can be seen, that taking the permutations of *Interactions* but preserving their *partial ordering* in the same *InteractionFragment* yields more states, than just simply appending each *fragment* as a whole after the others.

It can be also noted in Figure 5.3b that the branch which starts from *mainchart_s*1 consists of two sub-branches which originally had a common prefix (*send Protocol.cannotGo*) that was transformed only once, so they start from this state.

5.4 Compatibility validation of scenarios

A central aim of the Master Thesis is deciding which *ScenarioDefinitions* are incompatible with each other from the ones specified for a certain *Port*. If there is a trace for which two scenario definitions end up in an ambiguous decision, then there is a contradiction between scenarios that should be found at design time. For example, one of the scenarios say it is an *accepted* behavior while the other one says it is *erroneous*.

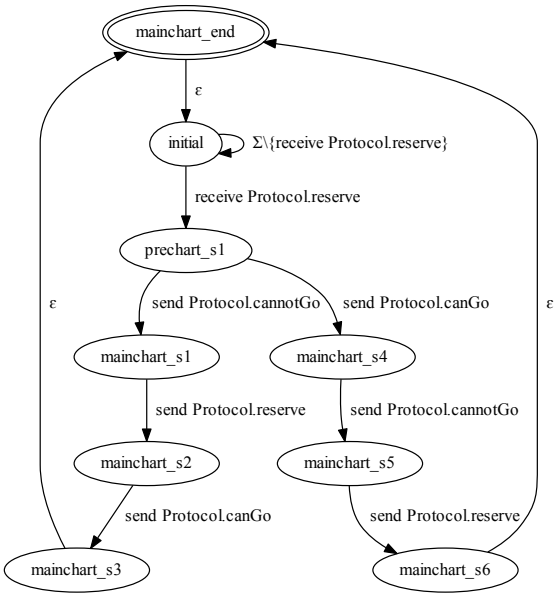
In order to do so, first in Section 5.4.1 I define *Scenario compatibility* which is illustrated in Table 5.1. Then in Section 5.4.2 I introduce the workflow which validates scenarios from their compatibility perspective.

```
scenario A {
  [
    cold receives Protocol.reserve
  ]
  {
    unordered {
      hot sends Protocol.cannotGo
      hot sends Protocol.reserve
    } and {
      hot sends Protocol.canGo
    }
  }
}
```

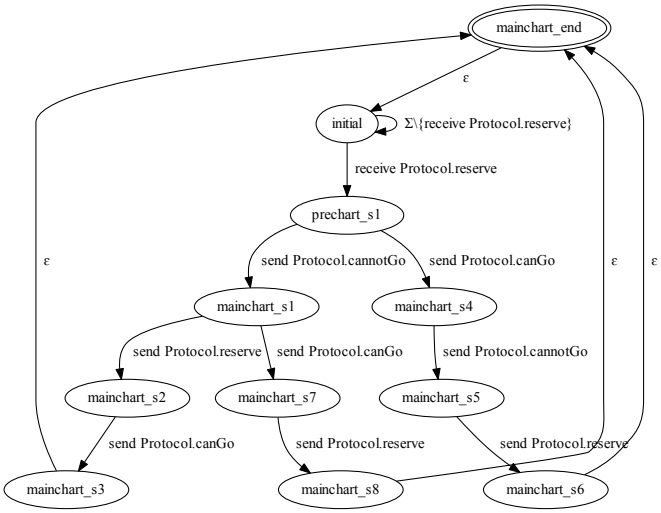
Listing 5.7: Scenario S_1

```
scenario B {
  [
    cold receives Protocol.reserve
  ]
  {
    parallel {
      hot sends Protocol.cannotGo
      hot sends Protocol.reserve
    } and {
      hot sends Protocol.canGo
    }
  }
}
```

Listing 5.8: Scenario S_2



(a) Automaton of S_1



(b) Automaton of S_2

Figure 5.3: Automaton of scenarios without hot violation state

$Scenario_1$ \ $Scenario_2$	accept	hot violation	cold violation
accept	✔	✗	✔
hot violation	✗	✔	✔
cold violation	✔	✔	✔

Table 5.1: Scenario compatibility table

5.4.1 Scenario compatibility definition

Definition 20 (Scenario compatibility). Two *ScenarioDefinitions* are *incompatible* with each other, if there is a trace which drives one automaton to a *hot violation* state and the other to an *accepting* state.

Two *ScenarioDefinitions* are *compatible* with each other if they are not incompatible.▪

The concept was defined according to Definition 20, because a *trace* can be either *valid* (ends in an *accepting state*), or *invalid* (ends in a *hot violation*), or *inconclusive* (ends in a *cold violation*) with respect to the *ScenarioDefinition* and its automaton representation.

So, if there is a *ScenarioDefinition* (scenario) for which the trace is *accepted* (*valid*) but in the meantime there is another scenario for which the respective trace ends in a hot violation (*invalid*), then the scenarios are not compatible with each other, because they end in an contradictory conclusion for the trace. Both scenarios are *universal* for the respective *Port* as defined in Section 4.3.

5.4.2 Compatibility validation workflow

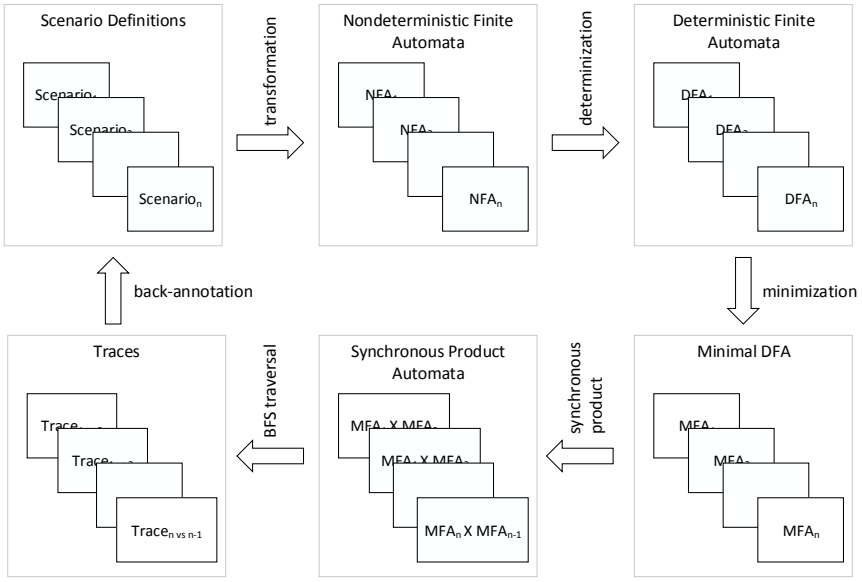


Figure 5.4: Scenario compatibility validation worklfow

In this section I propose a workflow, depicted in Figure 5.4, which decides if there is a trace for which two *ScenarioDefinitions* (scenarios) on the same *Port* are incompatible with each other according to Definition 20.

It transforms every scenario to their nondeterministic finite automaton (NFA) representation by the transformation rules introduced in Section 5.3.2. After that, in order to remove the ϵ transitions and the nondeterminism from the automata, each of them is determinized and minimized so the unnecessary states are removed, thus the size of the automaton is reduced but its accepted language (traces accepted by the automaton) remains the same. This transformation chain (scenario $\rightarrow$ MFA) for each scenario can be done independently from each other. They are done concurrently on separate threads, each scenario $\rightarrow$ MFA transformation is executed on a different thread.

Then every possible pairwise ordered combinations (permutations) of the minimized DFAs (MFAs) are taken, because the scenarios are pairwise validated. In the pairwise combinations each automaton is paired with every other automaton but itself, and the permutation preserves the ordering of the elements. So for k MFAs, the considered number of pairwise combinations are: $\frac{k \cdot (k-1)}{2}$. The ordering of the automata in the pairwise combination is randomly decided.

Then from every pairing a synchronous product finite automaton (SFA) is created.

For every transformation phase between the automata representations (NFA $\rightarrow$ DFA $\rightarrow$ MFA $\rightarrow$ SFA) a mapping is created to store the traceability relation. The role of the traceability relation is to support the interpretation of the analysis results at the high-level models. The mapping is created between each transformation phase, and it can be traversed in both directions.

Let $mfa_1 \times mfa_2 = sfa$ be the SFA, where $mfa_1, mfa_2 \in MFA$. The order of the automata in synchronous product is important. Because for the mfa_1 automaton those states of the SFA are searched, which are created from the *accepting state* of the NFA from which the corresponding mfa_1 was created. For mfa_2 similarly those states of the SFA are searched, which are created from the *mainchart_hotViolation* state of the NFA from which the corresponding mfa_2 was created. Then the intersection of these two sets of states in the SFA is calculated, let's call this set of states $ErroneousStates$. If there is an intersection, then it means there is a trace that mfa_1 accepts but mfa_2 rejects. So those scenarios from which mfa_1 and mfa_2 were created respectively, are incompatible with each other. The process is depicted in Figure 5.5.

This shortest trace is created by traversing the *sfa* from the *initial state* until either state in the $ErroneousStates$ is found by Breadth-First-Search algorithm. The trace is generated by recording the triggers of the transitions during the traversal.

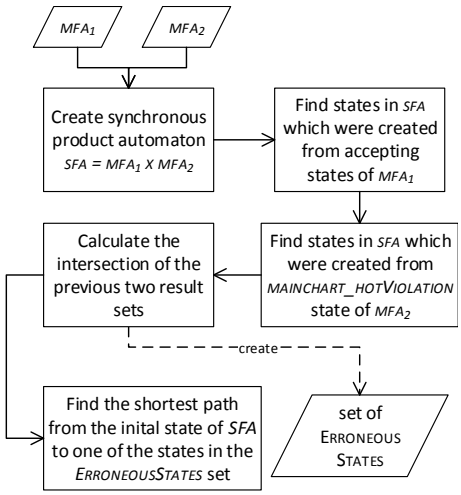


Figure 5.5: Process of finding the error trace in an *sfa*

Plug-in name	Description	Xtend SLOC	Java SLOC
automaton.model	Finite State Automaton metamodel	0	1063
automaton.model.validation	VIATRA validation rules for automata	153	7016
automaton.util	Automata transformation, construction and traversal algorithms	1305	2053
automaton.util.tests	Automaton transformation tests	1326	1580
language	GSL grammar and Xtext tooling	675	3881
language.ide	Xtext tooling for GSL	31	4744
language.ui	Xtext tooling for GSL	119	405
model	GSL (scenario) metamodel	0	2102
model.util	GSL error trace back-annotation	485	814
transformation	Scenario compatibility validation workflow	639	955
ui	Eclipse GUI dialog feedback	60	102
ui.contribution	Eclipse menu contribution	76	85
util	Custom extension methods for collections, scenario equality validator	210	325
		5079	25125

Table 5.2: Plug-in project source lines of code

Finally, this error trace is back-annotated into the scenario editor as a validation report. The back-annotation of the trace is done by mapping this trace to a corresponding scenario in mfa_2 and finding the first *ModalInteraction* for which the scenario goes to a *hot violation*.

The back-annotation algorithm handles *CombinedFragments* which consist of multiple interaction fragments. In case of a *CombinedFragment cf* the back-annotation algorithm is looking for an interaction fragment of *cf* which matches the trace. If it finds a matching fragment and the trace is not over yet, then it continues matching the remaining modal interactions in the error trace after *cf*. If there is no matching fragment or the trace is shorter than the number of modal interactions in the fragments, then the algorithm puts a warning marker to the first fragment of *cf*. The warning marker shows the whole error trace to the user.

5.5 Technical implementation

I implemented the proposed workflow as a set of plug-ins in Eclipse which will be included in the next release of the *Gamma Framework*. I used the Eclipse Modeling Framework (EMF) for creating the metamodel of the scenario language, depicted in Figure 5.2, and the automaton language depicted in Figure 5.6. I used Xtext to implement a grammar which gave a concrete syntax and an editor for the scenario language, as it was introduced in Section 5.2. Besides the grammar I also implemented structural validation rules for the scenario models. Validation rules ensure that the transformations start with a structurally correct model.

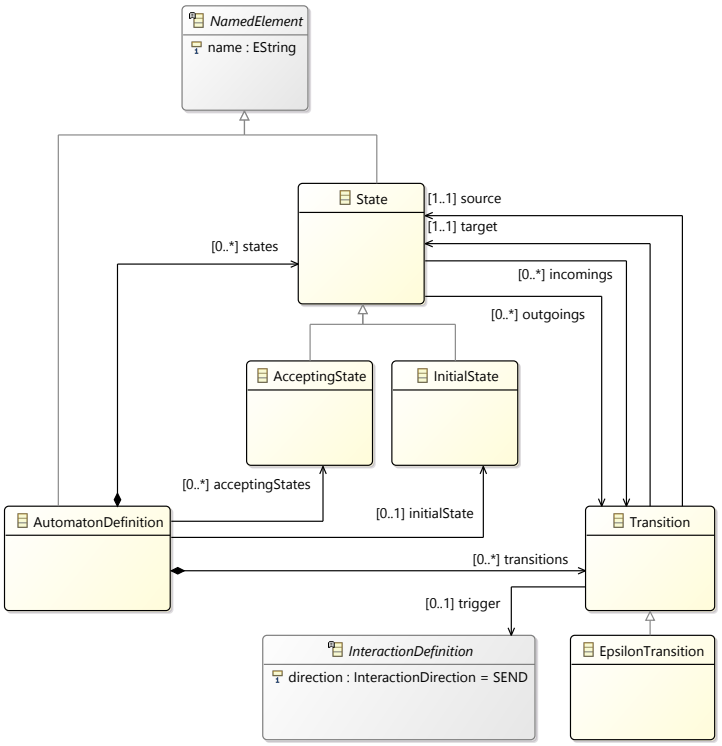


Figure 5.6: Automaton metamodel

As an implementation language I used Xtend[2] [4], that is a more flexible and expressive dialect of Java which compiles into readable Java 5 compatible source code. The advantage of this language is, it utilizes the advantages of Java 8 streaming API with more convenient extension methods. Moreover it reduces the verbosity of the Java language. What I mean by this expression is, the type informations are automatically inferred and in most cases there is no need for explicit type declaration for variable initialization. Thus the code is less verbose and more readable.

The set of plug-ins which realize the workflow are itemized in Table 5.2 together with the total number of source lines of code (SLOC) in those plug-ins. As it can be seen there is much less Xtend code, than generated Java code. It is due to the fact that on one hand Xtend is more compact, on the other hand there are two projects which contain the metamodels of the *scenarios* (`model`) and the *automata* (`automaton.model`). These metamodels are in ecore files which are XML and from these serialized formats EMF automatically generates the Java implementations, hence the relatively many Java SLOC.

As it can be seen in the aforementioned table, the projects that are necessary for the GSL grammar and its editor and validation consist of relatively few lines of Xtend code, but there are much more generated Java code. It is due to the fact that most of the work is done by Xtext. However, I had to implement the grammar (87 lines which are accumulated into the Xtend SLOC for `language` plug-in) and the validation rules.

In order to have the finite state automata in models also, I designed a simple automaton metamodel, as depicted in Figure 5.6. An automaton consists of States and Transitions. There are two special states, the InitialState and the AcceptingState. Each Transition is activated by a trigger, that is an InteractionDefinition in the corresponding ScenarioDefinition model, from which the initial NFA in the transformation workflow is created as depicted in Figure 5.4. An EpsilonTransition is a special transition, because it does not have any trigger.

[2] `http://www.eclipse.org/xtend/`, last access: 15/12/2017

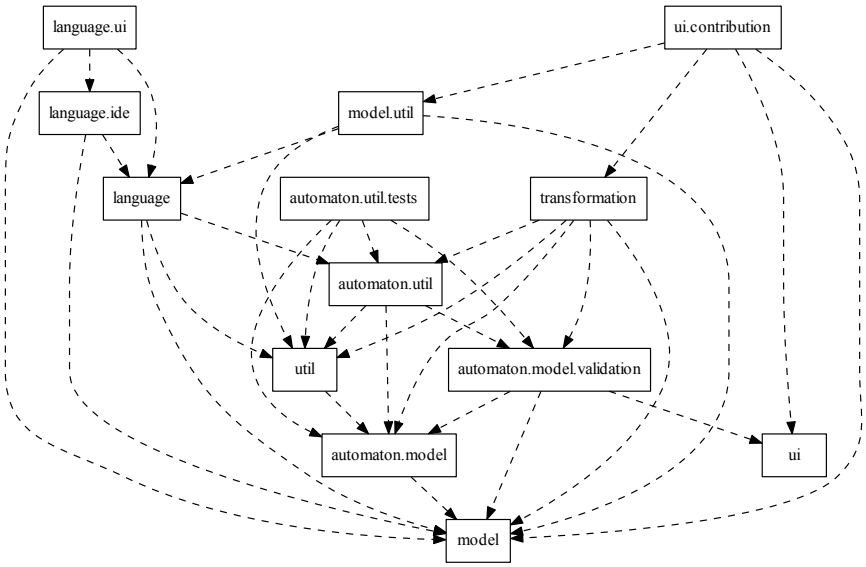


Figure 5.7: Plug-in dependencies diagram

Since every automaton in the workflow conforms to the same automaton metamodel, but they have semantically different rules, I had to implement structural validation rules for them too. These rules are in the `automaton.util` plug-in. Some of these structural validation rules are implemented as graph patterns in VIATRA[3] (formerly known as EMF-IncQuery [53]) framework. Other rules are implemented as imperative Xtend code.

The scenario to automata transformation workflow is implemented according to the algorithms in Section 2.6. Although the whole workflow is implemented as a batch transformation in the `transformation` plug-in, each step of the transformation is implemented in different classes. For example the automata related transformations and traversals, including finding the accepting-rejecting trace in the synchronous product finite automaton, are implemented in the `automaton.util` project.

In order to ensure that the automata transformations are correct, the output automaton of a transformation is validated against these patterns. By collecting the matches of these validation patterns on the respective automaton models, it can be ensured that the transformation is done correctly and the created automata meet their formal specification's rules. Should a transformation error occur, the user will get a corresponding error message which says which validation rule was violated.

For the back-annotation of the error traces, I implemented a feedback mechanism which puts warning markers for those places in the editor, where the deviation from the behavior prescribed by the scenario happens according to the error trace. The back-annotation is implemented in the `model.util` plug-in.

In order to integrate these plug-ins into Eclipse, I had to implement two projects. The first one is the `ui.contribution` that adds a menu contribution to the *Project Explorer* and *Package Explorer* views of Eclipse. The other UI contribution to Eclipse is the UI project, in which the info and error dialogs are shown to inform the user about the progress of the workflow.

Dependencies among theses plug-ins are depicted in Figure 5.7, except the ones for the Gamma Framework and for Xtext and Eclipse plug-ins for clarity purposes.

[3] `http://www.eclipse.org/viatra/`, last access: 29/11/2017

5.6 Testing the implementation

I implemented unit tests (`automaton.util.tests` plug-in) to validate the automata determinization, minimization and synchronous product algorithms. In the unit tests the input automata are built from code, by using the factory methods of the metamodel generated by the Eclipse Modeling Framework. The output automata are validated by invoking JUnit[4] assertions which is a tedious work. This should be changed for Model Based Testing and the input models should be in resource files as objects, instead of constructing them via code.

Other parts of the workflow were tested by running sample ScenarioDefinitions through it and inspecting the back-annotated error trace manually.

[4] `http://junit.org`, last access: 15/12/2017

Chapter 6

Evaluation

As part of the Master Thesis, I intended to evaluate my work from the following aspects:

Q1: How can scenario definitions be used for specifying the communication through a port of a given component?
Q2: How can these scenario specifications be validated regarding their compatibility with each other?
Q3: How much time does the scenario compatibility validation workflow take, depending on the number of interactions in the scenarios?

In order to address **Q1** and **Q2** I applied *Gamma Scenario Language* (GSL) in a case study in Section 6.1. In order to address **Q3**, I performed measurements on execution time with various model sizes. The methodology that was applied for preliminary runtime measurement of the compatibility validation workflow, together with evaluation of the measurements results, are detailed in Section 6.2.

6.1 Case study

In this section, I am going to introduce, how scenarios designed by GSL can complement composite statecharts in a model based demonstrator. In Section 6.1.1 and Section 6.1.2 MoDeS^{3}is going to be introduced together with a simplified component model of composite statecharts, designed in the Gamma Framework.

Then in Section 6.1.3 some sample scenarios are defined and evaluated in order to examine the scenario editor's and the scenario compatibility validation algorithm's capabilities on increasingly more complex scenario definitions.

6.1.1 MoDeS^3

MoDeS^3(Model-based Demonstrator for Smart and Safe Systems[1]), is a complex research and educational demonstrator, which, at the time of writing the thesis, is being developed by students and researchers at the Fault Tolerant Systems Research Group.

The demonstrator presents the combined use of model-driven and safety engineering with IoT technologies for smart and safe cyber-physical systems. MoDeS^3 is a model railway network where the system has to fulfill both functional and safety requirements. Safe behavior is aimed to be ensured by the combined use of design-time and runtime verification and validation techniques [2, 33].

There is a multilayer safety system which supervises the train tack. The top layer of the system processes a camera stream by marker recognition, in order to map locomotives

[1] `https://modes3.inf.mit.bme.hu`, last access: 29/11/2017

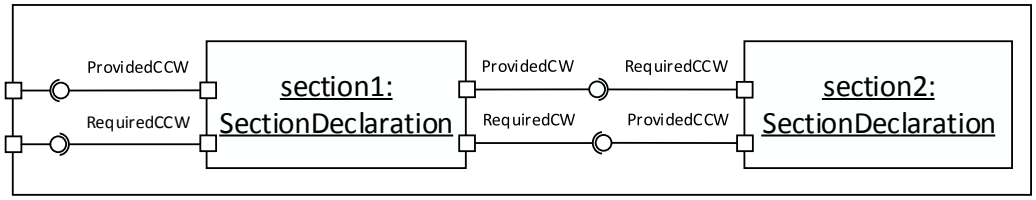


Figure 6.1: Graphical representation of TwoSections component

to trail segments. Then the changes in the segment occupancy are propagated as incremental graph pattern changes which are recognized and the necessary measures are taken in order to prevent train collision [2].

Although this solution recognizes dangerous patterns quickly, but the drawback is that it is a centralized, monolithic safety system which is not tolerant to outages.

Thus a lower-level safety system was developed by using distributed statecharts. The distributed statechart consists of a statechart for *segments*, *turnouts* (switch) which communicate with each other via events. For each physical switch a composite statechart is deployed, that consists of a *turnout* and three or four *segment* statecharts. The statecharts can communicate with each other either via direct method calls, if they belong to the same physical component, or via a communication network if they belong to different physical components [33]. Statecharts are prepared for network failures in a fail-safe way: should a response not arrive in time, a default response is constructed which is always assumed to be a negative response. Thus the trains would stop, even if there no real dangerous situation.

This fault-tolerant distributed composite statechart system was originally designed in xtUML [41] in my previous work [33]. It was later refined in YAKINDU Statechart Tools together with the textual composition specification as a case study for the Gamma Framework (Section 2.5) by Bence Graics.

```
interface Protocol {
  inout event reserve;
  inout event release;
  inout event canGo;
  inout event cannotGo;
}
```

Listing 6.1: Interface event directions

6.1.2 Component model

I designed a simplified component model which consists of two segment statecharts. The component model was introduced by a textual specification (Listing 2.1) in Section 2.5. In order to recap, only the component's graphical representation is depicted in Figure 6.1.

The statechart model is the *SectionDeclaration*, that was designed by Bence Graics. The declaration of the *Protocol* interface, which represents the events of the safety protocol, is in Listing 6.1. Every event of the interface is bidirectional (inout), thus they can be both sent and received, irrespective from the corresponding port's realization mode.

The semantical meaning of the events is the following:

- reserve: represents a *request*, that a train would like to *reserve* the next track element (segment or switch) on its trajectory.
- release: represents a *request*, that the train on the *recipient* track element is allowed to go.

- canGo: represents a *response*, that the train that is on the *recipient* track element *can go* to the *sender* track element
- cannotGo: represents a *response*, that the train that is on the *recipient* track element *cannot go* to the *sender* track element

6.1.3 Scenarios

As a case study, I defined several scenarios for the TwoSections component's section1.ProvidedCCW port which realizes the Protocol interface in provided mode.

The scenario definitions were implemented in more rounds with different purposes. In the first round with *simple scenarios* I would like to illustrate the usage of the editor.

Then with *scenarios derived from statecharts* I would like to illustrate that scenario definitions for a given port can be derived from a statechart, and how can the scenarios' compatibility be validated.

Finally with *complex scenarios* I would like to illustrate the express power of GSL by combining the complicated elements of the language.

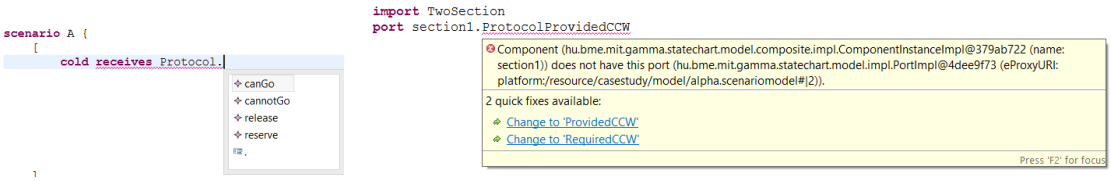
Simple scenarios

```
import TwoSection
port section1.ProvidedCCW

scenario A {
  [
    cold receives Protocol.release
    cold sends Protocol.canGo
  ]
  {
    alternative {
      hot sends Protocol.canGo
    } or {
      hot sends Protocol.reserve
      hot receives Protocol.cannotGo
    }
    cold receives Protocol.release
  }
}

scenario B {
  [
    cold receives Protocol.release
    cold sends Protocol.canGo
  ]
  {
    hot sends Protocol.canGo
    cold receives Protocol.release
  }
}
```

Listing 6.2: Simple scenarios

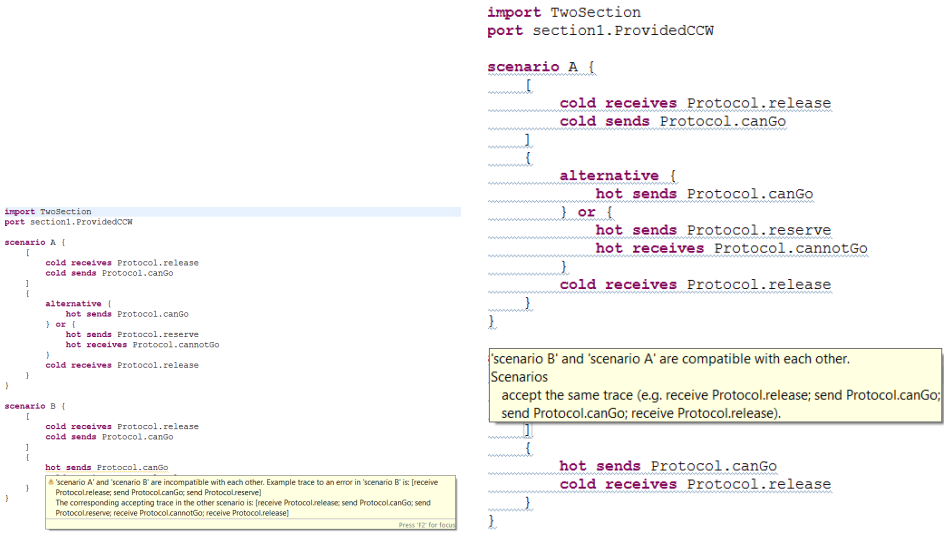
The purpose of the *simple scenarios* is to illustrate how the different *scenario definitions* can be created in an editor for the same port in GSL. The questions which I would like to answer are the followings:

Q1: How easy is the editor to be used?
Q2: What kind of structural analysis does it provide?



(a) Keyword suggestion **(b)** Structural validation with automatic fixes

Figure 6.2: Some features of the GSL editor



(a) Scenario A vs Scenario B validation **(b)** Scenario B vs Scenario A validation

Figure 6.3: Back-annotation of the scenario compatibility validation

Q3: How easily can the results of the scenario compatibility validation be back-annotated into the editor?

Scenario A and B under study are illustrated in Listing 6.2. They are defined for the same port and their prechart is the same, thus finding a common prefix for the error trace is relatively easy. Although their maincharts are different, Scenario B's mainchart is the subgraph of Scenario A's mainchart regarding their finite automata representations. It is due to the fact that the *AlternativeCombinedFragment* in Scenario A contains a fragment whose only interaction is hot sends Protocol.canGo which is the first interaction in Scenario B's mainchart, and their continuation is the same.

Designing the scenario in the editor is relatively easy since the editor provides keyword and object suggestions based on the GSL grammar, that is detailed in Listing A.1. If there are automatic problem resolutions (fixes) then the editor suggests some as depicted in Figure 6.2.

There are two possible permutations in which Scenario A and Scenario B can be validated against each other regarding compatibility. In the first permutation, where Scenario A's accepting states (accepting states in MFA_A, Minimal DFA for Scenario A) and Scenario B's hot violation states (error states in MFA_B) are intersected, the result is a trace for Scenario B which takes this scenario to hot violation.

The error trace is: receive Protocol.release; send Protocol.canGo; send Protocol.reserve that continues as receive Protocol.cannotGo; receive Protocol.release to be accepted by Scenario A. This trace is depicted in Figure 6.3a

and shows that the warning marker is put on the first interaction of Scenario B's main-chart, because that is the point when the scenario deviates from the prescribed trace.

On the other hand, when Scenario B's accepting states and Scenario A's hot violation states are intersected, then the two scenarios are always compatible, because for every trace that is accepted by Scenario B is also accepted by Scenario A as depicted in Figure 6.3b. Thus there is no warning marker for the scenarios, but an info-level marker is put on both scenarios to indicate this fact.

As it is depicted in Figure 6.3, the back-annotation of the error trace into the editor is easily done for elements that do not have to be unfolded, unlike *UnorderedCombinedFragment* and *ParallelCombinedFragment*. However, for more complicated elements, the back-annotation is more difficult as it is detailed in the corresponding example Section 6.1.3.

Scenarios derived from statechart

```
import TwoSection
port section1.ProvidedCCW

scenario A {
  [
    cold sends Protocol.reserve
    cold receives Protocol.cannotGo
  ]
  {
    cold receives Protocol.reserve
    hot sends Protocol.cannotGo
  }
}

scenario B {
  [
    cold receives Protocol.reserve
    cold sends Protocol.canGo
  ]
  {
    hot receives Protocol.reserve
    hot sends Protocol.canGo
  }
}

scenario C {
  [
    cold receives Protocol.release
    cold sends Protocol.reserve
  ]
  {
    hot receives Protocol.canGo
    hot sends Protocol.cannotGo
  }
}

scenario D {
  [
    cold receives Protocol.cannotGo
    cold receives Protocol.reserve
  ]
  {
    hot sends Protocol.cannotGo
  }
}
```

Listing 6.3: Scenarios derived from statechart

By inspecting the *SectionDeclaration* statechart with respect to the *ProvidedCCW* port, several scenario definitions can be derived. The procedure of inspection is by looking for references of the respective port inside the statechart. The port is usually mentioned in transitions' triggers, actions or in states' entry and exit actions. By checking consecutive runs, event sequences can be collected from which the scenario definitions can be designed.

However, doing this process manually is error prone and it is not formally proved that every possible consecutive run is found in this way. Moreover, in YAKINDU statecharts have *interfaces* instead of ports, so name of the interface must be unambiguously mappable to the name of the ports and vice versa.

Despite the drawbacks mentioned before, the questions I try to answer in this case are the followings:

Q1: How meaningful scenarios can be derived from statecharts by manual inspection?
Q2: How does the validation algorithm handle if the precharts are different?
Q3: How does the error trace back-annotation scale when many scenarios are validated against each other?

Accepting trace in \ Error trace in	$Scenario_A$	$Scenario_B$	$Scenario_C$	$Scenario_D$
$Scenario_A$	—	✗	✗	✗
$Scenario_B$	✗	—	✗	✗
$Scenario_C$	✗	✗	—	✗
$Scenario_D$	✔	✗	✗	—

Table 6.1: Multiple scenario validation results

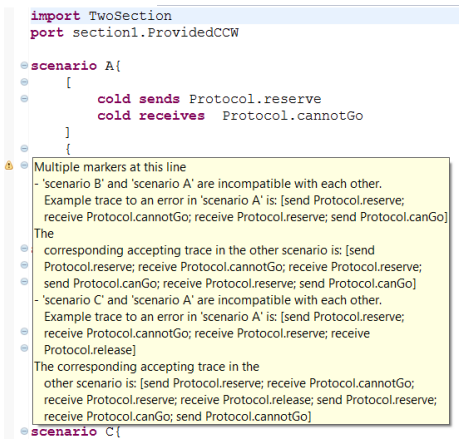


Figure 6.4: Multiple warning markers on the same element

Several scenario definitions were derived based on the *SectionDeclaration* statechart and the *ProvidedCCW* port. The definitions, which are illustrated in Listing 6.3, are much simpler than the ones in Section 6.1.3. It is due to the fact that I could only derive such from the event sequences in the statechart.

On one hand these scenarios are simpler, on the other hand their meaning, with respect to the statechart, is sometimes questionable, e.g. prechart and mainchart of Scenario B are repetitive, and a similar property holds for Scenario A as well. An automatic transition traversal and scenario derivation would address this problem, because then the scenarios would surely be derived from event sequences which occur in the statechart.

In order to avoid the unnecessary complexity of illustrating every incompatible scenario combination with an error trace, I would like to highlight only that combination which is compatible with each other, at least from one side. The validation results are in Table 6.1. The rows of the table show the scenarios reaching an accepting state, and in the columns the scenarios are shown that reach a hot violation. Every scenario is validated against every other scenario except from itself, thus the 'main diagonal' is empty. Compatible combinations are illustrated by a ✔ in the corresponding cell, incompatible ones are illustrated by a ✗ in the corresponding cell.

So the compatible scenario combination is the Scenario D vs Scenario A, because for every trace that passes their precharts and is accepted by Scenario D is also accepted by Scenario A. Such trace is for example, send Protocol.reserve; receive Protocol.cannotGo; receive Protocol.reserve; send Protocol.cannotGo.

On the other hand, the pair of Scenario A vs Scenario D is incompatible, because receive Protocol.cannotGo; receive Protocol.reserve; send Protocol.reserve leads to a hot violation in Scenario D, but the error trace's

continuation RECEIVE PROTOCOL.CANNOTGO; RECEIVE PROTOCOL.RESERVE; SEND PROTOCOL.CANNOTGO leads to an accepting state in SCENARIO A.

As it can be seen in the previous paragraph, the different prefixes of the scenarios are combined in such a way that in the error traces the *hot violation scenario*'s prechart is usually the prefix of the trace. As soon as the prechart is succeeded then the *accepting scenario*'s prechart is combined into the trace so that the *hot violation scenario* gets to the error state as soon as possible. To conclude, always the shortest possible error trace is generated.

The back-annotation of the scenario compatibility validation results to the editor is overlapping. Hence the individual combinations are not always visible, if two warning markers from two different error traces are put on the same interaction as depicted in Figure 6.4, or two info markers are put on the same scenarios.

Complex scenarios

```
import TwoSection
port section1.ProvidedCCW

scenario A {
  [
    cold receives Protocol.release
    cold sends Protocol.canGo
  ]
  {
    hot sends Protocol.cannotGo
    alternative {
      cold receives Protocol.canGo
      cold sends Protocol.canGo
    } or {
      cold receives Protocol.cannotGo
      hot sends Protocol.canGo
    } or {
      unordered {
        cold receives Protocol.canGo
      } and {
        cold receives Protocol.release
      }
    }
    hot sends Protocol.canGo
  }
}
```

```
scenario B {
  [
    cold receives Protocol.release
    cold sends Protocol.canGo
  ]
  {
    hot sends Protocol.cannotGo
    parallel {
      cold receives Protocol.canGo
      cold sends Protocol.canGo
    } and {
      cold receives Protocol.cannotGo
      hot sends Protocol.canGo
    } and {
      unordered {
        cold receives Protocol.canGo
      } and {
        cold receives Protocol.release
      }
    }
    hot sends Protocol.canGo
  }
}
```

Listing 6.4: Complex scenario definitions

In this example I would like to illustrate the expressive power of GSL by combining the *complicated elements* (e.g. UnorderedCombinedFragment, ParallelCombinedFragment) of the language. The sample scenario definition is illustrated in Listing 6.4. With this example I would like to answer the following questions:

Q1: How does the validation algorithm handle multiple *complicated elements* that are embedded into each other?

Q2: How does the error trace back-annotation look when such scenario definitions are validated?

In order to reduce complexity and to obey length limitations, the two scenario definitions are very similar to each other. The only difference between them is that

in Scenario A the mainchart contains an *AlternativeCombinedFragment*, while in Scenario B the mainchart contains a *ParallelCombinedFragment* (PCF), that has to be unfolded to an *AlternativeCombinedFragment* according to Section 5.3.2. However, the *UnorderedCombinedFragment* inside the PCF in Scenario B adds an extra complexity in unfolding.

The result of the Scenario B vs Scenario A validation is that they are incompatible, Scenario A reaches a hot violation state through the error trace: receive Protocol.release; send Protocol.canGo; send Protocol.cannotGo; receive Protocol.cannotGo; send Protocol.canGo; receive Protocol.cannotGo (the last interaction in the mainchart is violated); and the following continuation of the trace is accepted by Scenario B: send Protocol.canGo; send Protocol.canGo.

The result of the Scenario A vs Scenario B validation is that they are always compatible with each other, because every trace that is accepted by Scenario A is also accepted by Scenario B.

Although, the validation algorithm handles *complicated elements* that are embedded into each other, their representation in the editor is simplified. It is because unfolding a PCF implies creating new virtual elements that are originally not present in the scenario, thus warning markers cannot be placed on them. Hence as a simplification, the warning marker is put on the first interaction of the mainchart along with the error trace feedback.

6.2 Preliminary performance evaluation

In order to get a preliminary overview for the runtime performance of the scenario compatibility validation workflow, I carried out several measurements. In this section the research questions and the measurement environment will be introduced first. Then I will evaluate the results. Finally the threats to validity will be addressed.

6.2.1 Research questions

The purpose of the Gamma Scenario Language (GSL) is to support engineers in creating communication scenarios for a given port of a component.

Similarly to Section 6.1, the proposed use case is that the engineer creates several scenarios, each scenario has 10–100 modal interactions at most. Besides simple modal interactions in the interaction fragment, the engineer may would like to use alternative, unordered, or parallel combined fragments as well. Parallel or unordered combined fragments in the scenarios may influence the runtime significantly, because they have to be unfolded for the corresponding alternative combined fragments.

Thus the research questions which the preliminary measurements try to address are:

Q3.1: How does the naïve implementation of the proposed workflow scale for 10–100 modal interactions **without** any combined fragment?

Q3.2: How does the naïve implementation of the proposed workflow scale for 10–100 modal interactions **with** combined fragment?

Q3.3: How much does the continuous validation of automata via VIATRA influence the runtime?

6.2.2 Measurement planning

I wanted to measure the runtime of the two main phases of the validation workflow: first, the runtime of scenario to automata transformation, starting from the scenario ending

up with the Synchronous Product Automata (SFA). Second, the runtime of finding the accepting-error trace in the SFA.

Environment

The measurements were carried out on a Lenovo™ IdeaPad Y510P notebook with a 4-core Intel® Core™ i7 4700MQ CPU, 8 GB RAM and Windows® 8.1 operating system. During the measurements the internet was blocked on the machine, the anti-virus scanner and every user application was shut down except from the measurement-runner Eclipses.

Due to technical difficulties with resolving cross-model references in the Eclipse Modeling Framework, the measurements were run in Eclipse 4.7.1.a. There were two Eclipses open, one was the *host Eclipse*, where the source codes of the plug-ins were, the other one was a *runtime Eclipse*, where the measurements were run. The host Eclipse had 1 GB heap memory, the runtime one had 4 GB.

Measurement process

The source code was instrumented by custom timer which measured the elapsed time between its start and stop invocation. There were three places where these timers were instantiated: (1) around the scenario to automata transformation (the whole transformation chain until the Synchronous Product Automaton), (2) construction of Synchronous Product Automaton (SFA) from two Minimal DFA, (3) finding the accepting-error trace in the SFA.

As it was mentioned in Section 5.4.2 each scenario to automata transformation is done on separate threads concurrently. Thus for the transformation runtime, the construction time was added to the maximal runtime value of the concurrent transformations.

The aforementioned runtime was measured for each model size and each complexity (see Section 6.2.2) in 12 runs. The first two runs were the 'warm-up' ones for the Java Virtual Machine (JVM), the other 10 runs were measured. After each run the garbage collector (gc) was invoked three times and then the thread slept for 3 seconds before starting the next run in order to let the gc do its work and let the JVM cool down. Finally, the median of the 10 runtime measurements was considered into a data point in the figure. Data points which belong to the same model complexity fit to a hypothetical curve in the figure, from which only one of them is represented in the result figure.

Due to **Q3.3**, each measurement campaign (each model size with each complexity) was run twice. Once with having the VIATRA validations between the automata transformations enabled and once without these validations.

Models

I measured the aforementioned runtime for different model sizes (10, 50, 100, 500) with different complexities (simpler, unfolded). For each complexity there were two Scenario Definitions (scenarios) whose compatibilities were validated. The Scenario Definitions were designed in the Xtext-based editor.

In the simpler complexity each scenario contained only modal interactions. The precharts of these scenarios only contained one modal interaction which was the same for both of them and their maincharts only differed by the last modal interaction.

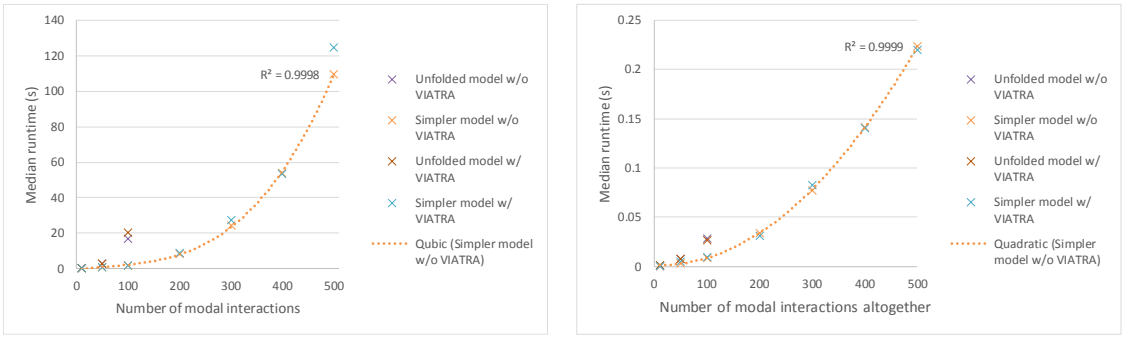
In the more complex (unfolded model) one of the scenarios contained an alternative combined fragment with three interaction fragments, the other scenario contained the same interaction fragments but in an unordered combined fragment. Both scenarios only contained the respective combined fragments in their maincharts, and their precharts were

the same as in the simpler model. The interaction fragments did not have any common prefix among each other in order to avoid merging the common prefixes.

As far as model sizes are concerned, in case of a simpler model, the number of modal interactions are the ones in the mainchart, while for the more complex models the total number of modal interactions in the mainchart should be counted as the model size. For the more complex models, these modal interactions were divided equally into the interaction fragments, sometimes +/- 1 modal interaction in each fragment, due to dividing even numbers by three.

Both in simpler and more complex models, the two scenario definitions usually differed only by the last modal interaction in their maincharts. In case of more complex models it means the last interaction fragment in the corresponding combined fragment.

6.2.3 Measurement results



(a) Scenario transformation runtime **(b)** Error trace find runtime

Figure 6.5: Preliminary runtime measurements

The results of the preliminary performance measurements are depicted in Figure 6.5 which address the research questions as follows:

A3.1: The scenario transformations run under 10 seconds for 200 modal interactions and even less for 10–100 elements (12 ms and 1.5 seconds respecitvely). Finding the accepting-error trace is performed in 3 ms for the same model size (9 μs for 100 elements).

A3.2: In the figures the transformation runtime is for the unordered combined fragment (the SFA is included in that time). The transformation runtime is 16 seconds for 100 modal interactions altogether. The error trace find runtime is 3 ms for 100 elements.

A3.3: Compared to the total runtime, the VIATRA validations have negligible influence. In case of the transformations they may have an effect, but it was only visible from 500 model elements. In case of finding the error trace, they have negligible effect since they are not used in that phase at all.

Evaluation As it can be seen in the figures, the naïve implementation performs within acceptable time for the target model sizes (10–100), both for the simple and the unfolded model. It could be used also in the proposed use case: designing several scenarios for the same port, validating them in batch and refining the scenario specifications based on the results of the validation.

6.2.4 Threats to validity

Although the VIATRA validation rules did not influence the measurements, there are several other factors which might have an influence for the results.

First, as it was mentioned in the performance environment subsection, the measurements were run in a runtime Eclipse and a host Eclipse was also running on the machine.

Second, the scenario to automata transformations were run on separate threads, until the creation of SFA automaton, since these threads were created and destroyed by the ExecutorService of Java.

Third, both the simple and the unfolded models were synthetically created by copy-pasting the same modal interaction sequences in order to design the models quickly. Moreover, scenarios with the same model sizes and same complexity only differed by the last modal interaction.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In the beginning of the Master Thesis I overviewed different scenario languages. Then I compared several model-based scenario engineering tools, which support the design, analysis, validation and further usage of scenario-based specifications.

Then I introduced the Gamma Framework that is a tool which supports the model-driven design and analysis of hierarchical, component and statechart-based reactive systems. Since the framework lacks a functionality to model and specify communication scenarios between components, I designed and developed the Gamma Scenario Language (GSL). Finally, I showed the applicability of GSL in a case study and I performed measurements on execution time with various model sizes.

I achieved several theoretical and practical results through the Thesis. These results are the followings:

Theoretical results

- The conclusion of the survey of the model-driven scenario engineering tools was that PhD dissertations and research papers have higher motivation for implementing experimental features for modeling tools, and having a custom metamodel gives higher flexibility in modeling and implementation.
- Then I designed a scenario-based specification language (GSL) by the following aspects:
  - The purpose of the language is to enable the engineer to describe communication scenarios over a port of a components.
  - The formalism is based on LSC, enriched by some elements that are used in UML Sequence Diagrams. Thus the abstract and concrete syntax are based on this formalism.
  - The formal operational semantics is defined by transforming scenarios into finite automata.
- Building upon the operational semantics, I proposed a procedure for validating scenarios against each other. The approach is based on combining the scenarios into pairs, and finding such traces which are accepted by one of the scenarios in the pairing but at the same they are rejected by the other scenario in the same pairing. If such conflicting trace exists, then it implies the two scenario definitions are incompatible with each other.

Practical results

In order to support the applicability of GSL in a model-driven framework, I implemented the abstract syntax of GSL using Eclipse Modeling Framework. To the abstract syntax I created a concrete syntax by defining its grammar in the Xtext framework. Scenarios can be designed in a textual editor that was generated by Xtext. In order to ensure the structural correctness of the model, I implemented several structural validation rules with different severities in Xtext. In this way it can be ensured that the model under design is structurally valid and meet the presumptions of the scenario validation workflow.

I also implemented the scenario validation workflow which transforms the scenarios into finite automata and find conflicting traces in these automata. These conflicting traces are then back-annotated to the editor and warning markers are put to the model to show the engineer where the scenarios end-up in an ambiguous decision for the trace.

In order to evaluate the applicability and the integrity of GSL to the Gamma Framework, I applied the language on a model railway case study. It should be admitted that these scenario definitions were neither industrially proven nor validated by railway domain experts. Moreover, theses scenarios were rather simple, since the most complicated one contained 10 signals, but in its unfolded representation there were 184 signals.

In order to get a preliminary overview for the runtime performance of the scenario compatibility validation workflow, I performed several measurements. In these experiments I measured the runtime both for the scenario to automata transformations and for finding the error trace. The recent implementation had an acceptable runtime performance for the proposed model sizes and complexities.

7.2 Future work

Although the foundations of the GSL language and the scenario validation framework are laid down, they are extensible in many ways. Regarding the GSL itself it can be extended with new elements, such as variables and constants with basic types, in order to introduce alternative fragments based on conditional guard expressions. Moreover, variable assignment and operational expressions could be also introduced. What's more, common scenario definitions could be refactored and referenced from other scenarios, so that the common parts of the behavior could be defined only once, but reused in many places.

The back-annotation that is proposed in the workflow is error prone, those scenario elements which were created by unfolding are not physically present in the model. Thus a better approach would be to have a direct traceability between the scenario and the first automaton transformation. Then in the back-annotation phase the modal interactions could be found more easily even if they are not physically present in the model.

Owing to the preliminary performance results, although the naïve implementation had an acceptable runtime performance for the proposed model sizes, it could be improved to handle auto-generated models with thousands of modal interactions and combined fragments with many hundred modal interactions. Moreover, effectively unfolding a parallel combined fragment into an alternative combined fragment can be also improved.

Acknowledgements

First of all, I would like to thank my advisors, Dr. Zoltán Micskei and András Vörös, for their guidance and continuous support. Moreover, I would like to thank Vince Molnár and Bence Graics for their valuable feedbacks on the scenario language and their help with the Gamma Framework.
I am also grateful to Oszkár Semeráth, Kristóf Marussy, Bálint Hegyi for their valuable comments and their help with the Eclipse-related technical difficulties.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science. Addison-Wesley, 1986.

[2] László Balogh, Florian Deé, and Bálint Hegyi. Hierarchical runtime verification for critical cyber-physical systems. Technical report, Scientific Students' Association, Budapest University of Technology and Economics, November 2015.

[3] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 200–236, 2004.

[4] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[5] Annette Bunker, Ganesh Gopalakrishnan, and Konrad Slind. Live sequence charts applied to hardware requirements specification and verification. *STTT*, 7(4):341–350, 2005.

[6] Luiz Fernando Capretz et al. Y: a new component-based software life cycle model. *Journal of Computer Science*, 1(1):76–82, 2005.

[7] Pierre Combes, David Harel, and Hillel Kugler. Modeling and verification of a telecommunication application using live sequence charts and the Play-Engine tool. *Software and System Modeling*, 7(2):157–175, 2008.

[8] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[9] Eclipse Papyrus Case Study Series. *A collaboration platform for avionics system*, 2016. URL: `http://www.eclipse.org/papyrus/resources/dga-usecasestory.pdf`, last access: 29/11/2017.

[10] Eclipse Papyrus Case Study Series. *Humanoid robot design*, 2016. URL: `http://www.eclipse.org/papyrus/resources/aldebaran-usecasestory.pdf`, last access: 29/11/2017.

[11] Eclipse Papyrus Case Study Series. *Transition from document-centric to model-centric design*, 2016. URL: `http://www.eclipse.org/papyrus/resources/plasticomnium-usecasestory.pdf`, last access: 29/11/2017.

[12] Sven Efftinge and Markus Völter. oAW xText: A Framework for Textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

[13] Friedenthal, Sanford and Moore, Alan and Steiner, Rick. *A practical guide to SysML: the systems modeling language*. Elsevier, 2011.

[14] Bence Graics. Model-Driven Design and Verification of Component-Based Reactive Systems. Bachelor's thesis, Budapest University of Technology and Economics, 2016.

[15] Bence Graics. Model-Driven Development of Reactive Systems with Mixed Synchronous and Asynchronous Hierarchical Composition. Technical report, Scientific Students' Association, Budapest University of Technology and Economics, November 2017.

[16] Joel Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, Paderborn, Universität Paderborn, Diss., 2011.

[17] Joel Greenyer, Christian Brenner, and Valerio Panzica La Manna. The ScenarioTools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions. *ECEASST*, 58, 2013.

[18] Joel Greenyer, Daniel Gritzner, Timo Gutjahr, Florian König, Nils Glade, Assaf Marron, and Guy Katz. ScenarioTools - A tool suite for the scenario-based modeling and analysis of reactive systems. *Sci. Comput. Program.*, 149:15–27, 2017.

[19] David Harel. From Play-In Scenarios to Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001.

[20] David Harel. Can Programming Be Liberated, Period? *IEEE Computer*, 41(1):28–37, 2008.

[21] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Int. J. Found. Comput. Sci.*, 13(1):5–51, 2002.

[22] David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart Play-out of Behavioral Requirements. In *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, pages 378–398, 2002.

[23] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*, pages 309–324, 2005.

[24] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

[25] David Harel, Shahar Maoz, and Itai Segall. Some Results on the Expressive Power and Complexity of LSCs. In *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, pages 351–366, 2008.

[26] David Harel, Shahar Maoz, Smadar Szekely, and Daniel Barkan. PlayGo: Towards a Comprehensive Tool for Scenario Based Programming. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 359–360, 2010.

[27] David Harel and Rami Marelly. *Come, let's play - scenario-based programming using LSCs and the play-engine*. Springer, 2003.

[28] David Harel and Rami Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and System Modeling*, 2(2):82–107, 2003.

[29] David Harel and PS Thiagarajan. Message sequence charts. In *UML for Real*, pages 77–105. Springer, 2003.

[30] Harry E. Crisp. Systems Engineering Vision 2020. *Seattle, Washington*, page 15, September 2007. Document No.: INCOSE-TP-2004-004-02.

[31] Gergő Horányi. Automatic synthesis of monitors for runtime verification of distributed embedded systems. Technical report, Scientific Students' Association, Budapest University of Technology and Economics, November 2011. In Hungarian, original title: Monitorok automatikus szintézise elosztott beágyazott rendszerek futásidőbeli verifikációjához.

[32] Gergő Horányi. Monitor synthesis for runtime checking of context-aware applications. Master's thesis, Budapest University of Technology and Economics, 2014.

[33] Benedek Horváth. Model-Driven Development of Distributed Safety-Critical Software Systems Based on xtUML. Bachelor's thesis in hungarian, original title: Elosztott biztonságkritikus rendszerek xtUML alapú modellvezérelt fejlesztése, Budapest University of Technology and Economics, 2016.

[34] ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language.* ITU-T, 1988.

[35] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC).* ITU-T, 1993.

[36] Jean-Louis Boulanger. *The new CENELEC EN 50128 and the used of formal method*, 2014.

[37] Judit Csima, Katalin Friedl. *Languages and automata*, 2013. Course syllabus in Hungarian at Budapest University of Technology and Economics, original title: Nyelvek és automaták.

[38] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior.* PhD thesis, Carl von Ossietzky University of Oldenburg, 2003.

[39] Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen, and Saulius Pusinskas. Verifying Real-Time Systems against Scenario-Based Requirements. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 676–691, 2009.

[40] Shahar Maoz, David Harel, and Asaf Kleinbort. A Compiler for Multimodal Scenarios: Transforming LSCs into AspectJ. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–41, 2011.

[41] Stephen J Mellor, Marc Balcer, and Ivar Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[42] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software and System Modeling*, 10(4):489–514, 2011.

[43] Model-Driven Software Engineering course. *Models and Transformations in Critical Systems*, 2015. Budapest University of Technology and Economics.

[44] Object Management Group. *OMG Object Constraint Language (OMG OCL) - Version 2.4*, 2014. `http://www.omg.org/spec/OCL/2.4/`, last access: 29/11/2017.

[45] Object Management Group. *OMG Systems Modeling Language (OMG SysML) - Version 1.4*, 2015. `http://www.omg.org/spec/SysML/1.4/`, last access: 29/11/2017.

[46] Object Management Group. *OMG Unified Modeling Language (OMG UML) - Version 2.5*, 2015. `http://www.omg.org/spec/UML/2.5/`, last access: 29/11/2017.

[47] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML) - Version 1.2.1*, 2016. `http://www.omg.org/spec/FUML/1.2.1/`, last access: 29/11/2017.

[48] Radio Technical Commission for Aeronautics (RTCA). *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, 2014.

[49] Radio Technical Commission for Aeronautics (RTCA). *DO-278A Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Manage-ment (CNS/ATM) Systems*, 2014.

[50] Radio Technical Commission for Aeronautics (RTCA). *DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A*, 2014.

[51] Roger S. Pressman. *Software Engineering: A Practitioner's Approach, Seventh Edition.* The McGraw-Hill Companies, 2010.

[52] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[53] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.*, 98:80–99, 2015.

Appendix

A.1 Gamma Scenario Language – Concrete Syntax

```
ScenarioDeclaration returns ScenarioModel::ScenarioDeclaration:
  'import' statechart = [StatechartModel::StatechartSpecification]
  port = PortReferenceDefinition
  (scenarios += ScenarioDefinition)+
;

PortReferenceDefinition returns ScenarioModel::PortReference:
  'port' component = [CompositeModel::ComponentInstance] '.' port = [
    StatechartModel::Port]
;

ScenarioDefinition returns ScenarioModel::ScenarioDefinition:
  'scenario' name = ID '{'
    prechart = PrechartDefinition
    mainchart = MainchartDefinition
  '}'
;

PrechartDefinition returns ScenarioModel::Prechart:
  '['
    ^fragment = FragmentDefinition
  ']'
;

MainchartDefinition returns ScenarioModel::Mainchart:
  '{'
    ^fragment = FragmentDefinition
  '}'
;

FragmentDefinition returns ScenarioModel::InteractionFragment:
  (interactions += AbstractInteractionDefinition)+
;

AbstractInteractionDefinition returns ScenarioModel::Interaction:
  AlternativeCombinedFragmentDefinition | UnorderedCombinedFragmentDefinition |
     ParallelCombinedFragmentDefinition | ModalInteractionDefinition
;

AlternativeCombinedFragmentDefinition returns ScenarioModel::
    AlternativeCombinedFragment:
  'alternative' '{'
    fragments += FragmentDefinition
  '}' ('or' '{'
    fragments += FragmentDefinition
  '}')+
;
```

```
UnorderedCombinedFragmentDefinition returns ScenarioModel::
    UnorderedCombinedFragment:
  'unordered' '{'
    fragments += FragmentDefinition
  '}' ('and' '{'
    fragments += FragmentDefinition
  '}')+
;

ParallelCombinedFragmentDefinition returns ScenarioModel::
    ParallelCombinedFragment:
  'parallel' '{'
    fragments += FragmentDefinition
  '}' ('and' '{'
    fragments += FragmentDefinition
  '}')+
;

ModalInteractionDefinition returns ScenarioModel::ModalInteraction:
  modality = ModalityDefinition
  interaction = SignalDefinition
;

SignalDefinition returns ScenarioModel::Signal:
  direction = DirectionDefinition
  interf = [InterfaceModel::Interface] '.' event = [InterfaceModel::Event]
;

enum DirectionDefinition returns ScenarioModel::InteractionDirection:
  SEND = 'sends' | RECEIVE = 'receives'
;

enum ModalityDefinition returns ScenarioModel::ModalityType:
  COLD = 'cold' | HOT = 'hot'
;

terminal ID : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Listing A.1: Gamma Scenario Language Grammar

A.2 Gamma Scenario Language – Semantics given by automaton

Definition 21. Some definitions which are reused in the latter algorithms. Concepts in *italics* are the same as in the Abstract syntax 5.1.

- *Direction* is a set of *InteractionDirections*, where $Direction(d)$ means $d \in \{send, receive\}$
- *Interface* is a set of *Interfaces*, where *Interface(i)* means i is an Interface
- *Event* is a set of *Event*s which belong to the same *Interface*, where $Event(e, i)$ means e is the event of Interface i
- *EverySignalExceptThis(m)* is a set of *Signals* which are constructed from the respective ModalInteraction m's *Signal* s as follows: $\forall d_k, e_k : Direction(d_k) \wedge Event(e_k, i) \wedge d_k \neq d \wedge e_k \neq e$ where d is the direction, e is the Event and i is the Interface of s;

and d_k, e_k are the respective fields of the new Signal whose Interface is the same as s's.

- *EverySignalExceptThese(Set(m))* is a set of *Signals* which are constructed from the set of $ModalInteraction(m)$. The semantics is similar to *EverySignalExceptThis(m)*, but now the generated *Signals* cannot be the ones which are referred in $Set(m)$.

- *LongestCommonPrefixesOf(Set(iFragment))* is a map of the n-wise longest common prefixes of the $iFragment$ instances of *InteractionFragment* that are in the set. Each entry of the map is a key-value pair: the key is the longest common prefix, and value is set of *InteractionFragments* who has this common prefix. Common prefix is a k-long sequence of *Interactions* which are in the same location and are the same in the corresponding *InteractionFragments* starting from the beginning. ▪

Algorithm 4: createState()

output: a new state in D NFA
1 let *state* be a new State in D;
2 let *state.name* be according to the Chart in which this method was invoked, see in Section 5.3.2;
3 **return** state;

Algorithm 5: createTransition(source, target, trigger)

input : *source*: State, *target*: State, *trigger*: InteractionDefinition
output: a new state in D NFA
1 let *transition* be a new Transition in D;
2 transition.source = source;
3 transition.target = target;
4 transition.trigger = trigger;

Algorithm 6: Transforming a ScenarioDefinition to a NFA

input : a S ScenarioDefinition
output: a D NFA accepting the traces prescribed by S

```
1  let initialState be the initial state of D;
2  let latestState be the latest location in the accepting run;
3  latestState = initialState;

4  firstModalInteraction = S.prechart.head;
5  firstInteraction = firstModalInteraction.interaction;
6  foreach signal ∈ EverySignalExceptThis(firstModalInteraction) do
7  |  createTransition(initialState, initialState, signal);
8  end
9  latestState = transformInteraction(firstModalInteraction, latestState);

10 foreach interaction ∈ S.prechart.tail do
11 |  createTransitionsToViolationStates(interaction, latestState);
12 |  latestState = transformInteraction(interaction, latestState);
13 end

14 foreach interaction ∈ S.mainchart do
15 |  createTransitionsToViolationStates(interaction, latestState);
16 |  latestState = transformInteraction(interaction, latestState);
17 end
18 convert latestState to accepting state;

19 let ViolationStates be the set of every hot or cold violation state in D;
20 foreach state ∈ {ViolationStates ∪ {latestState}} do
21 |  let interf be the Interface of the corresponding Port in S;
22 |  let Events be every Event that is available on interf;
23 |  foreach d ∈ Direction(d) and e ∈ Events do
24 |  |  create signal Signal;
25 |  |  signal.interface = interf;
26 |  |  signal.event = e;
27 |  |  signal.direction = d;
28 |  |  createTransition(state, initialState, signal);
29 |  end
30 end
```

Algorithm 7: createTransitionsToViolationStates(mi, latestState)

input : mi : ModalInteraction, $latestState$: State
output: a corresponding fragment of the D NFA

```
1  let violationState be a violation state based on mi.modality and on the Chart in
   which mi is;
2  foreach signal ∈ EverySignalExceptThis(mi) do
3  |  createTransition(latestState, violationState, signal);
4  end
```

Algorithm 8: createTransitionsToViolationStates(cf, latestState)

input : *cf* : CombinedFragment, *latestState* : State
output: a corresponding fragment of the D NFA

```
1 let firstModalInteractions be a set of the first ModalInteractions in each
  InteractionFragment of cf;
2 let firstModality be the modality of firstModalInteractions's elements;
3 let violationState be a violation state based on firstModality and on the Chart in
  which cf is;
4 foreach signal ∈ EverySignalExceptThese(firstModalInteractions) do
5 |   createTransition(latestState, violationState, signal);
6 end
```

Algorithm 9: transformInteraction(mi, latestState)

input : *mi* : ModalInteraction, *latestState* : State
output: a new state in the D NFA

```
1 newState = createState();
2 createTransition(latestState, newState, mi.interaction);
3 return newState;
```

Algorithm 10: transformInteraction(ucf, latestState)

input : *ucf* : UnorderedCombinedFragment, *latestState* : State
output: corresponding fragments of the D NFA and *newState* as a continuation of the accepting run

```
1 fragments = ucf.fragments;
2 let permutationsOfFragments be a list of InteractionFragments from fragments
  according to Definition 18

3 let acf be an AlternativeCombinedFragment whose fragments are the permutations
  in permutationsOfFragments;
4 newState = transformInteraction(acf, latestState);
5 return newState;
```

Algorithm 11: transformInteraction(pcf, latestState)

input : *pcf* : ParallelCombinedFragment, *latestState* : State
output: corresponding fragments of the D NFA and *newState* as a continuation of the accepting run

```
1 fragments = pcf.fragments;
2 let partialOrderedPermutationsOfFragments be a list of InteractionFragments from
  fragments according to Definition 19

3 let acf be an AlternativeCombinedFragment whose fragments are the permutations
  in partialOrderedPermutationsOfFragments;
4 newState = transformInteraction(acf, latestState);
5 return newState;
```

Algorithm 12: transformInteraction(acf, latestState)

```
   input  : acf : AlternativeCombinedFragment, latestState : State
   output: corresponding fragments of the D NFA and newState as a continuation of
           the accepting run
 1 fragments = acf.fragments;
 2 // collect and transform the n-wise common prefixes of fragments,
 3 // the algorithm is extracted to Algorithm 13

 4 // transform the continuation of each fragment from the common prefix by saving
   the last states of each fragment path
 5 let newStates be an empty set of States in D; foreach fragment in fragments do
 6 |  if fragment has a common prefix with any other fragment then
 7 |  |  foreach prefix in which fragment is effected do
 8 |  |  |  let state be the latest state that represents the end of prefix;
 9 |  |  |  foreach interaction ∈ fragment after the prefix do
10 |  |  |  |  if interaction is null then
11 |  |  |  |  |  newState = createState();
12 |  |  |  |  |  createTransition(state, newState, ϵ);
13 |  |  |  |  |  newStates.add(newState);
14 |  |  |  |  else
15 |  |  |  |  |  createTransitionsToViolationStates(interaction, state);
16 |  |  |  |  |  newState = transformInteraction(interaction, state);
17 |  |  |  |  |  newStates.add(newState);
18 |  |  |  |  end
19 |  |  |  end
20 |  |  end
21 |  else
22 |  |  foreach interaction ∈ fragment do
23 |  |  |  createTransitionsToViolationStates(interaction, state);
24 |  |  |  newState = transformInteraction(interaction, state);
25 |  |  |  newStates.add(newState);
26 |  |  end
27 |  end
28 end

29 newLatestState = createState();
30 foreach state ∈ newStates do
31 |  createTransition(state, newState, ϵ);
32 end

33 return newLatestState;
```

Algorithm 13: Transform common prefixes of an AlternativeCombinedFragment's InteractionFragments

```
   input  : acf: AlternativeCombinedFragment
   output: a map which contains the common prefixes and their transformed paths
           states
 1 fragments = acf.fragments;
 2 longestCommonPrefixes = LongestCommonPrefixesOf(fragments);
 3 sortedLongestCommonPrefixes = sort(longestCommonPrefixes by key.length);

 4 let transformedPrefixPaths be a map of prefixes and their intermediate states;
 5 foreach entry ∈ sortedLongestCommonPrefixes do
 6 |   prefix = entry.key;
 7 |   effectedFragments = entry.value;
 8 |   if no longer prefix is transformed yet between the effectedFragments then
 9 |   |   firstInteraction = prefix.head;
10 |   |   let intermediateStates be an empty list of States in D;
11 |   |   tempState = transformInteraction(firstInteraction, latestState);
12 |   |   intermediateStates.add(tempState);
13 |   |   foreach interaction ∈ prefix.tail do
14 |   |   |   createTransitionsToViolationStates(interaction, tempState);
15 |   |   |   tempState = transformInteraction(interaction, tempState);
16 |   |   |   intermediateStates.add(tempState);
17 |   |   end
18 |   end
19 end

20 return transformedPrefixPaths;
```