JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

Using Interactions to Validate Executing State Machines

MASTER'S THESIS

submitted in partial fulfillment of the requirements
for the academic degree

Diplom-Ingenieur

in the Master's Program

SOFTWARE ENGINEERING

Submitted by

Philipp Mitterer, BSc.

At the

Institute for Software Systems Engineering

Advisor

Univ.-Prof. Dr. Alexander Egyed, M.Sc.

Linz, September 2014

Abstract

Many developers use UML Sequence Diagrams to show scenarios of the system's behavior. Sequence diagrams are easy to read and hence also non-professionals can use them to discuss the designed systems and its requirements. Nevertheless, they are imprecise and incomplete.

In contrast to sequence diagrams, state machine diagrams model the behavior of individual components. They depict a more complete, generally valid picture of a component's behavior. However, it is not easy to understand system behavior from individual class's behavior.

Thus, these diagram types illustrate different views of the system, but still represent the same system. Therefore, it would be beneficial to use both views to leverage from their respective strenghts when designing a system. However, to be able to use both views requires tool support to validate the consistency between these two diagram types.

The goal of this thesis is to combine these two views by using sequence diagrams as constraints for state machine validation. This means that the designed state machines are tested using sequence diagrams as references for valid and invalid scenarios. The tests are executed by simulating the state machines. The simulation's input is given by the user, which has thereby the possibility to experiment with the designed model and execute different scenarios. If a scenario depicted as sequence diagram was validly executed, the approach should be able to recognize the execution and inform the user. Furthermore, if a sequence diagram is violated, the violation should also be reported to the user. Through this debugging process, the user can verify whether the modelled state machines comply to the requirements stated or not.

This process of validation is done by monitoring the communication between these state machines during simulation. The resulting stream of communication events is then handed to nondetermistic finite automata representing the individual components in a sequence diagram. Through the automata's resulting states we are able to tell if an interaction was executed validly or if it was violated. We adapted the McNaughton-Yamada-Thompson algorithm to translate the sequence diagrams to automata.

Moreover, the approach is well suited for object-oriented design. Hence, multiple state machines of the same type can be instantiated during simulation. The scenarios for the individual state machine instance are then recognized separately. The tool decides during run-time which state machine represents which components in a sequence diagram.

Kurzfassung

Viele EntwicklerInnen verwenden UML Sequenzdiagramme um Ausführungsszenarien des Systems zu modellieren und darzustellen. Sequenzdiagramme sind einfach zu lesen und dadurch auch für Laien geeignet, um über das designte System und dessen Anforderungen zu diskutieren. Jedoch sind diese ungenau und unvollständig.

Im Gegensatz zu Sequenzdiagrammen eignen sich Zustandsdiagramme zum Modellieren des Verhaltens einzelner Komponenten. Ein vollständigeres, allgemein gültiges Bild des Komponentenverhaltens wird dabei gezeigt. Allerdings ist das Begreifen des Systemverhaltens anhand von Zustandsdiagrammen alleine mühsam.

Obwohl diese beiden Diagrammtypen dasselbe System darstellen, illustrieren sie unterschiedliche Sichtweisen auf dieses. Es wäre daher vorteilhaft beide Sichtweisen beim Design von Systemen heranzuziehen, um deren unterschiedliche Stärken zu nutzen. Allerdings ist dafür Werkzeugunterstützung zur Validierung der Konsistenz dieser Diagramme notwendig.

Das Ziel dieser Arbeit ist die Verbindung dieser unterschiedlichen Systemansichten, indem Sequenzdiagramme als Constraints für die Validierung von Zustandsautomaten verwendet werden. Dies bedeutet, dass die erstellten Zustandsautomaten getestet werden und dabei die Sequenzdiagramme als Referenz für gültige und ungültige Szenarien dienen. Dabei werden die Tests mittels Simulation der Zustandsautomaten durchgeführt. BenutzerInnen treiben die Simulation durch das Auslösen diverser Events voran und haben dadurch die Möglichkeit mit dem konzipierten Modell zu experimentieren und unterschiedliche Szenarien auszuprobieren. Wird ein Szenario, welches in einem Sequenzdiagramm abgebildet ist, korrekt ausgeführt, soll das System dies erkennen und die BenutzerInnen darüber informieren. Genauso soll das System Verletzungen von Sequenzdiagrammen melden. Mithilfe dieser Vorgehensweise kann festgestellt werden, ob die modellierten Zustandsautomaten den Anforderungen ensprechen.

Der Validierungsprozess wird während der Simulation durchgeführt, indem die Kommunikation der Zustandsautomaten überwacht wird. Die resultierenden Kommunikationsevents werden dann nichtdeterministischen endlichen Automaten, welche die einzelnen Komponenten in einem Sequenczdiagramm repräsentieren, übergeben. An den resultierenden Zuständen der Automaten kann das System dann erkennen, ob ein Szenario gültig ausgeführt oder verletzt worden ist. Für die Übersetzung von Sequenzdiagrammen zu Automaten wurde der McNaughton-Yamada-Thompson-Algorithmus adaptiert.

Darüber hinaus ist der Prozess gut für objektorientiertes Design geeignet. Das bedeutet, dass während der Simulation mehrere Zustandautomaten vom selben Typ instanziert werden können. Die Szenarien werden dann für die einzelnen Zustandautomaten getrennt erkannt. Die Entscheidung, welcher Zustandsautomat welcher Komponente im Sequenzdiagramm entspricht wird zur Laufzeit getroffen.

Contents

Chapter 1

Introduction

The Unified Modeling Language (UML [1]) is a powerful and widely used language for describing and depicting the model of a software system. Its Sequence Diagrams are used to describe intended and forbidden behavior of the system. They are easy to read, even for non-professionals. Therefore they can be used to state high-level requirements and use cases for the system, which can be understood by all stakeholders. This helps to build up a common knowledge about the specification and therefore reduces the risk of building the wrong system.

With these sequence diagram specifications the system is designed. Developers create models (e.g. state machines) of the system which they think meet the previously stated requirements. However as [2] pointed out, state machines represent intra-object communication – i.e. the control flow within a certain component or module – while sequence diagrams show inter-object communication – i.e. the messages and signals sent between certain components of the system. These different views are decoupled from each other, although they describe the same system. Especially in a modern iterative development process, changing the model repeatedly can quickly lead to inconsistencies between the stated requirements and the design. Thus we need a mechanism to check if the designed model satisfies the requirements.

In this paper we present a solution to dynamically validate state machines using sequence diagrams. The approach observes communication of a state machine simulator and performs validation during simulation using stated sequence diagrams as constraints. The validation uses non-deterministic final automata representing the lifelines of the sequence diagrams.

Even though we used the validation to verify state machine communication, the generic nature of this approach allows it to be easily extensible for other simulations or even actual system executions to be validated.

Chapter 2 shows an illustrative example, with which we will demonstrate the goal and our approach, and gives a short introduction into the two addressed diagram types - statecharts and sequence diagrams. In Chapter 3 we will analyze the general problem of sequence diagram and state machine validation. Chapter 4 describes what our approach does to tackle the issues stated in Chapter 3. In Chapter 5 we explain how we resolved these issues and which algorithms we used. Chapter 6 then shows how we implemented the proposed approach in a reference implementation. Next, Chapter 7 assesses the correctness and performance of the proposed approach. The subsequent Chapter 8 analyzes other papers related to sequence diagram and state machine validation and compares them to the approach in this paper. Lastly, Chapter 9 summarizes this thesis and gives an outlook on future work.

Chapter 2

Illustration

To demonstrate our approach we use an *automatic light* system. The system controls a light within a room. It turns the light on for some time, when someone enters the room and is detected by a motion detector. To save energy, the light is not turned on during the day, when the natural sun light illuminates the room sufficiently. In addition, the user can manually switch on the light with a manual light switch. This switch is independent of the daylight. Furthermore, the user can choose to deactivate the system and therefore prohibit the activation of the light by pressing a main switch.

Figure 2.1 shows the class diagram for the system. Most of the classes are actually representing the hardware of the system. The only software components are *ControlUnit* and *LightController*. The *ControlUnit* class is the central software component which decides when to turn on the light. *LightController* is responsible for the communication with the light, i.e. it turns the light on and off again after a specified time interval. *MainSwitch* and *ManualLightSwitch* are the switches to de-/activate the system and manually turn on the light respectively. The *DaylightSensor* tells the system if the natural light is enough to illuminate the room. *MotionDetector* detects people entering the room. Lastly, the class *Light* represents the light which the system controls.

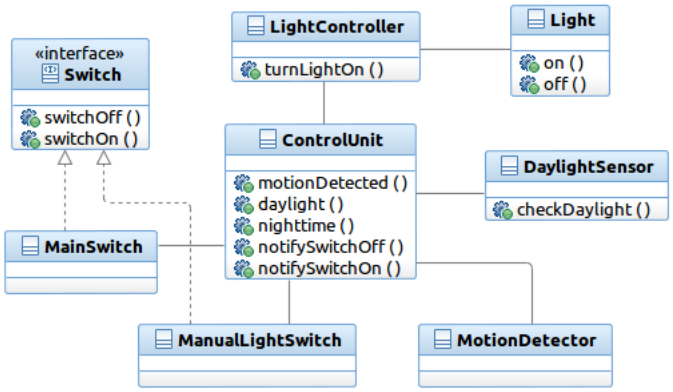

Figure 2.1: Class diagram of the light system

For our illustration, we picked two requirements to model as sequence diagrams. The first

requirement is, that the control unit should switch on the light when a motion is detected and the natural light is not sufficient. This requirement is shown in Figure 2.2 as sequence diagram.

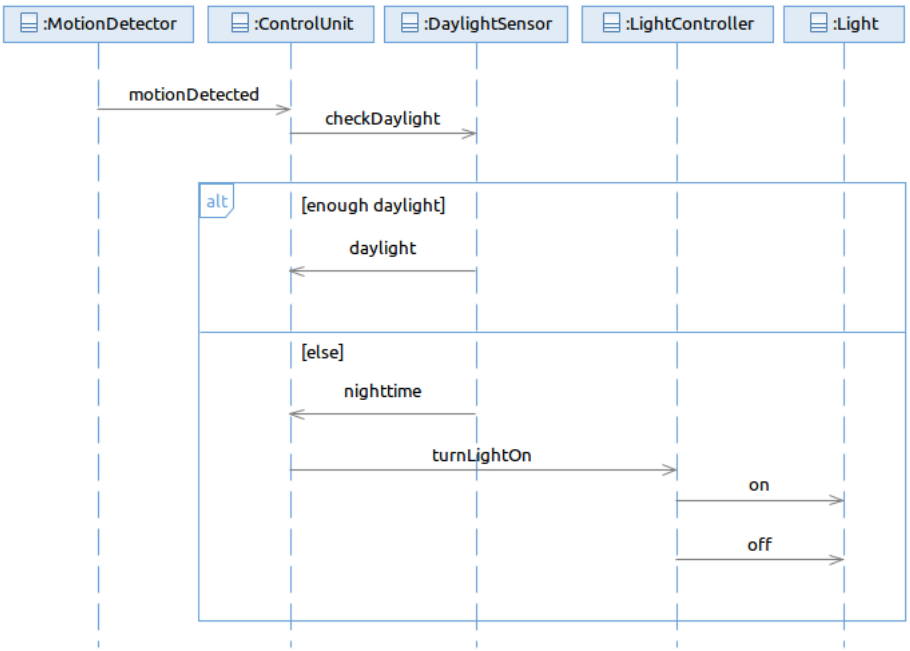

Figure 2.2: Sequence diagram showing the *motion detected* scenario

Sequence diagrams depict the communication of certain components. The time is shown on the y-axis, while the x-axis resemble different components participating in the interaction. The vertical line of each component is called lifeline. The arrows between the lifelines represent messages sent from one component to another component. These communication events are partially ordered by the time of their occurrence. Partially ordered here means that independent message can come in any order. We will discuss this property in more detail in Section 5.2.

Hence, the sequence diagram in Figure 2.2 contains lifelines for the components *MotionDetector*, *ControlUnit*, *DaylightSensor*, *LightController*, and *Light*. When the motion detector perceives a movement, it sends a signal to the control unit. Then the control unit queries the daylight sensor if enough natural light is present. If this is the case, the control unit is done. Otherwise, it tells the light controller to turn on the light. This conditional branching is represented by the alternative combined fragment in the sequence diagram. Combined fragments were introduced in UML 2 and are shown as a box over some of the lifelines. A combined fragment covers the lifelines intersecting it, which means that it affects them depending on its interaction operator. They can have one or more operands divided by a horizontal line. Our combined fragment has an alternative interaction operator, and two operands, representing the two alternatives *daylight* and *nighttime* and covers the components *ControlUnit*, *DaylightSensor*, *LightController*, and *Light*. Each operand contains a guard shown in brackets. If a guard is evaluated to true, the corresponding operand is executed, otherwise it is skipped. We will see more combined fragments in Section 5.2. After the controller turns the light on, it waits for some time and turns it off again.

As it is often the case with sequence diagrams or similar diagrams, the depicted diagram

shows only one scenario of the system's behavior. It only applies in certain situations and is not globally valid. In our case the scenario does not apply if the main switch is turned off.

The second requirement for our example system is shown in Figure 2.3. It shows the constraint, that after the main switch was turned off, the light should not be switched on until the main switch is turned on again. Here we see three more combined fragment types. The process of turning off and on again can be repeated and the constraint should hold for subsequent off switches as well, so we use a *loop* combined fragment to signal that the sequence diagram can be repeated over and over again. Since we are only interested in a certain subset of communicated messages, we filter the irrelevant messages for the diagram. This filtering process is done by the *consider* combined fragments. These combined fragments indicate that we are only concerned about certain messages, namely `switchOff`, `notifySwitchOff` and `turnLightOn`, `on`, `switchOn`, `notifySwitchOn` respectively, and ignore all other messages within these combined fragments. Other messages might include `off` messages sent to the light. We do not care about these messages here, since they do not affect our constraint. The last new combined fragment is a negative combined fragment, which indicates that `turnLightOn` and `on` messages are not allowed after the main switch was switched off. This condition only applies until the main switch is switched on. In contrast to the scenario of our first requirement, this constraint should always hold. It should be valid for every execution of the system.

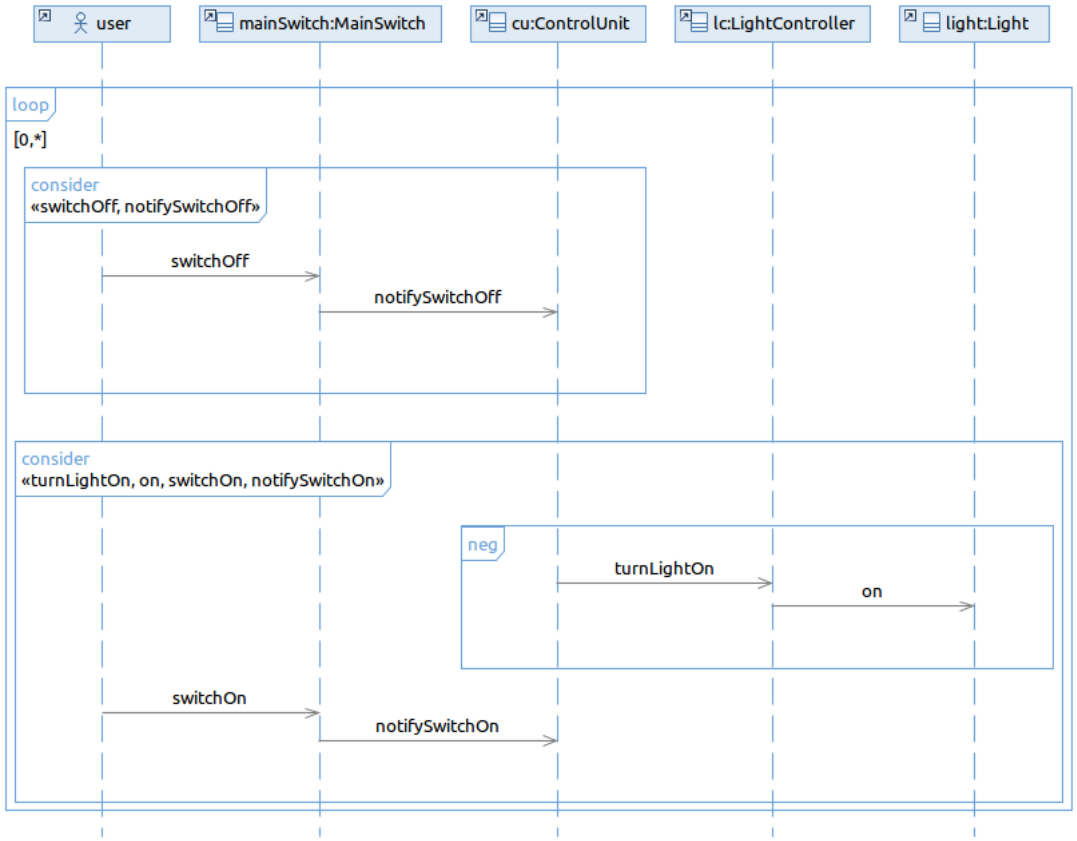

Figure 2.3: Sequence diagram showing the *main switch* constraint

Figure 2.4 shows the state machine for the control unit of our light system as a UML statechart. Statecharts illustrate the behaviour of one component or class of the system. Their main structures are the states of the component, which give the diagram its name. States are illustrated as boxes with rounded edges in the diagram and normally contain a name which identifies the state. The initial state of the component is shown as a little circle. These states are connected with one another via transitions, shown as arrows. A transition defines from which state to which state and under which circumstances the component can transition and what happends if it does so. Therefore, transitions can specify three optional properties: a trigger, a guard, and an action. The trigger defines the event which will trigger the transition, like the detection of a motion. The guard specifies the condition under which the transition can be triggered. Lastly, the action specifies what happens, when the transition is executed. In the statechart these informations are written at the arrow with the following syntax: `trigger [guard] / action`

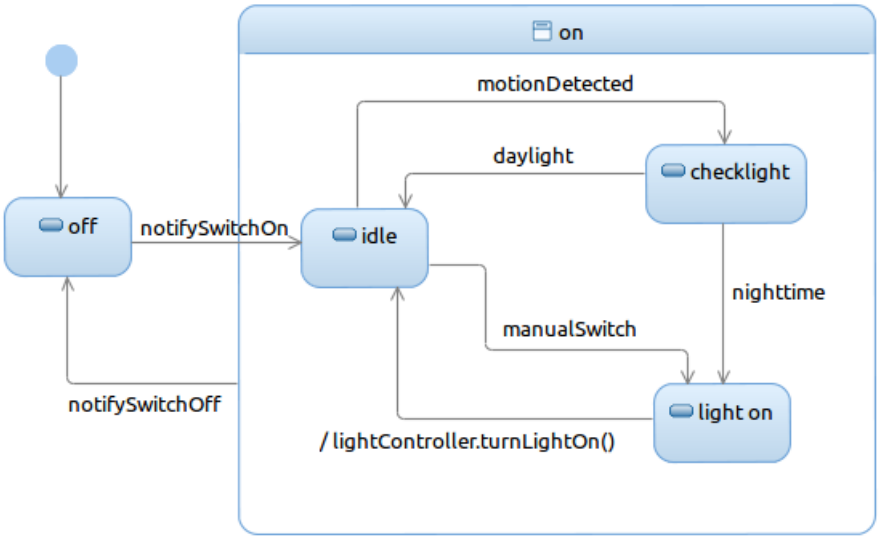

Figure 2.4: State chart showing the control unit

In our example the control unit is initially switched off. When it is turned on (main switch sends a `notifySwitchOn` message), it transitions into the *idle* state. In case of motion being detected it first checks the daylight in the *checkLight* state, before either going back to the *idle* state directly or first signaling the light controller to turn the light on before coming back. When manual switch is activated, the control unit immediately transitions to the *light on* state, without checking the daylight in advance. When the control unit is switched off it transitions back to the *off* state from any other state it is currently in. This behavior is achieved by the composite state *on* which contains the states *idle*, *checklight* and *light on*.

The state chart of the light controller is shown in Figure 2.5. It only consists of two states. When the controller gets the signal to turn on the light it turns on the light and transitions into the *light on* state. After 90 seconds it turns the light off again and transitions back to the *light off* state. If it gets the signal to turn on the light while already in the *light on* state it stays in this state but resets the timer. This ensures that the light is turned off exactly 90 seconds after the last command to turn on the light was received.

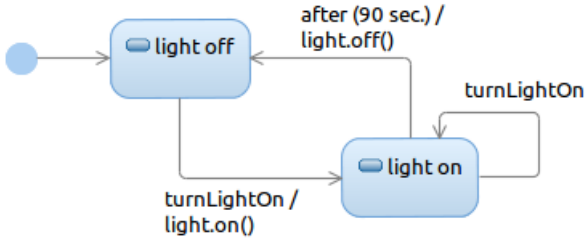

Figure 2.5: State chart showing the light controller

For the purpose of illustration, it suffices to only show the state charts of these two components. In fact, all other components are just simple hardware components and therefore the whole business logic resides within these two components.

Chapter 3

Background & Problem Statement

The two diagram types shown in the illustration depict two different views of the system. While the sequence diagrams show the communication between components of the system – i.e. inter-object communication – with only little focus on what happens within the components, the state charts show the internal behavior of individual components – i.e. intra-object communication.

Even though the diagrams illustrate different views, they still describe the same system. Hence, it is possible to recognize the alternative combined fragment in the two outgoing transitions of the control unit's *checkLight* state and therefore deduct the consistency of the diagrams. However, this manual deduction is significantly more difficult with more state machines of higher complexity.

For now let us analyze the two views in more detail. State machines are useful for precise and comprehensive behavior descriptions. A lot of powerful simulation languages and model checkers are available for state machines. Therefore, nearly all behavioral modeling focuses on state machines. Nevertheless, they are also complex to write. Especially since each state machine tends to model independent component behavior and inter-component communication is hard to see and express. This may easily lead to errors in the model and inconsistencies with the requirements, which are usually defined as inter-object communication (e.g. sequence diagrams).

To cope with these deficiencies, message sequence charts (MSC) [3] and later UML sequence diagrams [1] were developed. These diagrams are useful to express specific interactions patterns, for instance what should happen, when a specific event occurs. They are much easier to write but by their very definition incomplete, since they only show certain scenarios of the system's execution. This can also be seen in our automatic light system. The *motion detected* scenario does not show what happens, when a motion was detected and the main switch was switched off. It does not even state the condition of the main switch at all. With just this diagram we have no way of knowing how the main switch contributes to the scenario. Moreover, the diagram does not state when the scenario is applicable but only that it may apply at some point.

This incompleteness is one of the reasons why only a few tools exist that focus on event scenarios (e.g. Harel's play engine as described in [2]). But these tools are flawed in that it is nearly impossible to enumerate all the corner cases to build a complete model of the system. Furthermore, adding a new scenario may affect existing scenarios. For example, if a new scenario implies that there are two possible responses to something that happens than both scenarios need to be refined to account for the difference. Another issue is the ambiguity of sequence

diagrams. Thus, the interpretation and formalization of sequence diagrams has been widely discussed (e.g. [4, 5, 6, 7]). This ambiguity is a significant hindrance when trying to synthesize sequence diagrams.

These circumstances obviously lead to the idea of combining both approaches. Hence, a whole community (SCESM) focused on how to merge event scenarios and state machines or how to generate state machines from scenarios [8]. In essence this community found this problem unsolvable with traditional state machine and scenario languages. Only by making scenarios semantically much more complex it was possible to achieve this goal. However, these languages became unusable as they merged the two different constructs into one model which consequently lacked the "ease of understanding" traditionally designed state machines possess [9]. Even if the these issues could be overlooked, the issue of completeness could never be satisfactory addressed. From inherently incomplete sequence diagrams it is impossible to generate complete state machines. This incompleteness might be detectable in some cases but not automatically fixable. Thus manual tweaking would be necessary to correctly derive a state machine (see [9]) which brings us back to the original problem: Are the state machines correct?

In conclusion the synthesis of event scenarios into state machines is insufficient. Nevertheless, we still want to utilize the benefits of sequence diagrams and state machines. Hence, instead of synthesizing state machines from sequence diagrams, perhaps we can validate the correctness of state machines using sequence diagrams. Naturally, we are not the first ones to come up with this idea. Other researchers also validated state machines or some other artifact using sequence diagrams (e.g. [10, 11]). They create models from the diagrams, which are used to explore the state space for inconsistencies and errors. Most of these approaches use model checking for their validation. However, model checking suffers from certain deficiencies. Firstly, model checking of larger models can easily lead to a state explosion. This is especially true for sequence diagrams due to the partial order of their events. Even more complicating is the fact that the language of sequence diagrams is not context-free [12, 13]. Furthermore, model checking is not well suited for polymorphism and other dynamic features of object-oriented design [14, 15]. Another disadvantage of model checking approaches is that both parts of the system (state machines and sequence diagrams) must be translated into a model which the model checker is capable to understand and interpret. Lastly, purely checking if a use case is executable at all does not suffice for our validation. As previously stated, sequence diagrams do not define, in which cases their scenario is applicable. Hence, conventional model checkers only give counter-examples, when one interaction cannot be satisfied at all. However, we want to know if the interaction was performed if a certain event happens at a certain point of the system's execution.

In addition to the drawbacks of traditional model checking, most approaches are not applicable with standard UML sequence diagrams. The user must enhance and prepare the sequence diagrams for the validation. This fact burdens the user with additional work for the approaches to be usable and hence is a major hindrance when it comes to validating already existing legacy models which were not built for these validation approaches.

Chapter 4

Goal

The goal of this thesis is to explore if sequence diagrams can be used as constraints for state machine validation. In essence, it should allow us to test the designed state machines with the help of sequence diagrams. Sequence diagrams help us model common scenarios as well as corner cases. The approach should be flexible enough to cope with the incompleteness of these sequence diagrams. In fact, incomplete scenarios only imply incomplete test coverage. Incompleteness can be even beneficial in model design, since it hides details which are not relevant for the model and therefore hide complexity. The details of the implementation should be decided by the implementing developer and not the requirements engineer or architect.

Due to the limitations of conventional model checking processes, we pursue a more dynamic, exploratory approach in our work. Hence, the approach should be able to decide during the execution of state machines if they comply with the sequence diagrams. The tool should recognize valid executions (valid traces) and violations (invalid traces) of scenarios described as sequence diagrams. As already discussed in the previous chapter, state machines and sequence diagrams have their strengths and weaknesses. We want to utilize both views to be able to express behavior more richly. Nevertheless, our main goal is not the validation of state machines, but rather the use of sequence diagrams for validations. Thus, our approach should put emphasis on the interchangeability (low coupling) of the communication source – i.e. the validation target. It should be as independent as possible, so there is little effort to integrate another source, e.g. a different model execution or even an actual system execution. To achieve this goal our approach should only use the stream of communication messages as source for its validation.

Furthermore, we want to keep the syntax of sequence diagrams as simple as it is. This ensures that anybody familiar with sequence diagrams can use this tool and existing models can be validated using the tool without any further customization. UML is primarily used for object-oriented design and our tool should be capable of coping with the dynamic nature of object-oriented languages (e.g. polymorphism). Although the approach is applicable to standard UML sequence diagrams, we will see some extensions to UML to ease the use of this work. However, they just simplify constraint definitions, but are not mandatory for the approach to be applicable.

Chapter 5

Approach

The following chapter describes the approach we used, what problems we came across and how we resolved them. The architecture is outlined in Figure 5.1.

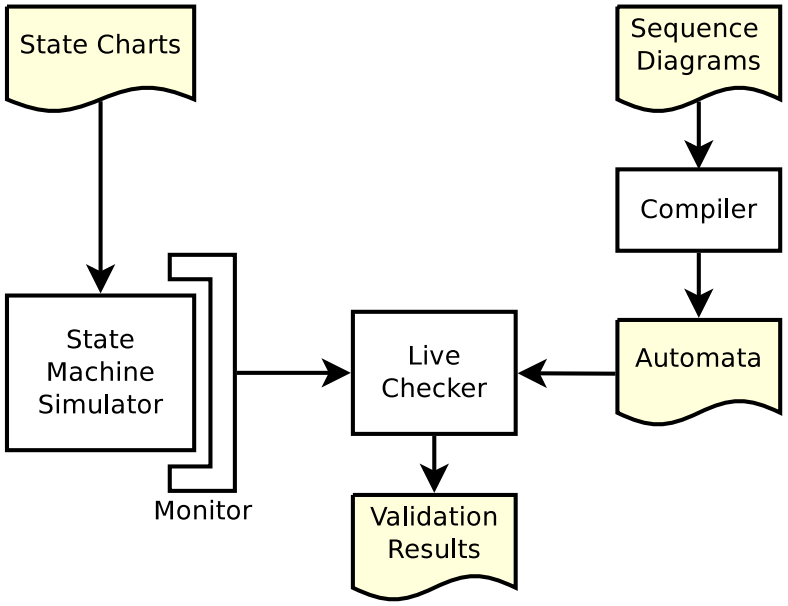

Figure 5.1: Architecture overview

We have got a state machine simulator on one side. The simulator processes state transitions and serves as medium for the individual state machines to communicate with each other by sending messages. Meanwhile, the monitor observes the running simulation and hands the observed communication to the live checker. More about the simulation and monitoring will be covered in Section 5.1.

On the other side, we have got sequence diagrams which supposedly outline the communication within the system. UML sequence diagrams illustrate sequences of communication events across several participating components. The shown interactions are scenarios, i. e. they are just partial examples of system execution and therefore the shown

interactions are incomplete. UML sequence diagrams depict interactions. Interactions are described by a standardized data structure defined in the UML standard [1]. So, rather than using sequence diagrams, we are actually using interactions for the validation. However, it should be noted here, that these two terms are used synonymously for most parts of this thesis. We could use this data structure to interpret the sequence diagrams directly during validation. Unfortunately the data structure is not well suited for our validation purpose. Hence, instead of interpreting the interactions, they are transformed into automata by the depicted compiler. These automata are then used to validate the simulated execution. The compilation process can be done beforehand or upon start-up and is described in Section 5.2.

The actual validation is done live during the simulation. The observed messages are handed from the monitor to the live checker where they are processed. This list of message is called a trace. A trace identifies the communication within the system. The live checker then decides if the sequence diagrams where executed validly or invalidly, when they were executed and which components were involved. A valid execution of a sequence diagram is present, when the messages shown in the diagram are executed in the specified order. An invalid execution on the other hand can be only achieved with negative and assert combined fragments, like the one shown in the *main switch* constraint shown in Figure 2.3. In this example the trace is invalid, if the control unit tells the light controller to turn the light on which in return sends a message with the name *on* to the *light* component after the main switch was switched off. We will see in more detail how this process works in Section 5.3.

5.1 State Machines, Simulation and Event Monitoring

The state machines modeling the system's behavior are similar to those shown in the illustration in Chapter 2. A model consists of multiple state machines, each representing a single component or class, communicating with one another. A state machine simulator takes these state machines and simulates them. One state machine describes the behavior of one class and like classes in an object-oriented language, state machines can be instantiated. Thus, the simulator actually simulates instances of state machines and it is possible that one state machine is instantiated multiple times. During simulation the state machine instances communicate with one another to trigger state changes similar to the *ControlUnit* calling the *turnLightOn* method on the *LightController*. A user controlling the simulation may provide additional input by sending messages to state machine instances. Furthermore, the user can see which state machine instance is in which state and thereby observe the whole simulation.

During this simulation process, the monitor observes the communication between the state machine instances. This observation can be done via observer pattern as described in [16] or by listening to a communication bus, through which the state machines may communicate with each other. This procedure guarantees that the monitoring does not interfere with the simulation. Apart from listening to the communication, the monitor does never query the simulator.

With the observed communication the monitor then produces a stream of communication events. These communication events or messages contain the following information:

- Which state machine instance sent the message
- Which state machine instance is the recipient of the message
- What is the name of the message

To avoid suspending the simulation, the monitor puts the message into a queue. From there the live checker can take the message and process it asynchronously.

5.2 Compiling Sequence Diagrams to Automata

Sequence diagrams are used to show us the ordering of events in a certain scenario. Automata are an an efficient way of representing event ordering for later live checking. We use enhanced non-deterministic final automata (NFA) for this purpose. An NFA is formally defined as a 5-tuple $(S, \Sigma, \Delta, s_0, F)$ with [17]

- finite set of states S
- finite set of input symbols Σ
- the transition function $\Delta \subseteq S \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(S)$
- the initial state from where the execution starts $s_0 \in S$
- finite set of final (accepting) states $F \subseteq S$

In our case the alphabet Σ of the NFA is the set of all messages in the system. A message is a 3-tuple (r, r, m) with

- a sender $s \in L$
- a receiver $r \in L$
- a message name m

where L is the set of lifelines. The automata can do ε-transitions (empty word or empty message in our case), have multiple current states and each current state has a small memory to remember its history (execution context). Furthermore, we introduce a couple of special states, which provide additional logic to the NFA, as we will see later. This enhancement is necessary to be able express the intention of sequence diagrams.

The automata created from the sequence diagrams depend on the defined semantics. Though the parsing process can be changed to represent different semantics we used the following semantics for our implementation.

The following combined fragments are used for the validation:

OPT The operand is only executed, if the operand's guard evaluates to true, otherwise it is skipped.

LOOP The operand is executed zero, one or multiple times.

ALT The *alternative* combined fragment is similar to the *optional* combined fragment with multiple choices. At least one of the operands is chosen to execute. An example can be seen in the *automatic light* illustration.

BREAK If the operand's guard evaluates to true, the operand is executed and the enclosing combined fragment is exited afterwards. Otherwise, the combined fragment is skipped.

NEG If the operand is executed validly, the trace is invalid. An example can be seen in the *automatic light* illustration.

ASSERT If the operand is not executed validly, the trace is invalid.

CONSIDER A consider combined fragment additionally state a list of message names. All messages with names not contained in the stated list are not considered. This combined fragment combined with an *assert* or *ignore* combined fragment is a common pattern to describe validation. An example can be seen in the *automatic light* illustration.

IGNORE The *ignore* combined fragment is similar to a *consider* combined fragment. The only difference is that it ignores the stated message, instead of the not stated messages.

Since sequence diagrams only show certain scenarios and therefore only a part of the communication within the system, prefixes and suffixes are allowed. This means that messages can sent before or after the messages shown in the diagram do not change the validity of the interaction.

Furthermore the sequence of messages can interleave with messages from or to objects not stated in the diagram. This behavior makes it easier to avoid revealing too much implementation details in the diagram. However, messages whose sender and receiver are stated in the diagram are considered for the validation process. Thus including empty lifelines, which do not interact with other lifelines, can make sense, when one wants to state, that no messages should be sent to the component with the empty lifeline.

Sequence diagrams are a high-level view of certain scenarios the system should be capable of handling. They are often stated in an early phase of the project and lack implementation details. Hence the guards stated in these sequence diagrams are in many cases informal and not suited for formal evaluation. We wanted to include these diagrams into the verification as well and therefore do not evaluate guards. Instead, we use nondeterministic automata and take every possible path. The only exception is a *else* guard in an alternative combined fragment. The UML specification states that "at most one of the operands will be chosen" [1]. However, if a guard with else is stated, exactly one of the operands will be chosen. This handling of guards comes with another advantage. UML sequence diagrams do not assign a guard to lifelines but to their interaction operand in general. Therefore, it is not clear which lifeline evaluates the guard during execution. This issue is furthermore complicated by the fact that due to the partial order of events, lifelines might enter the interaction operand at different times (see [4] for more details). With not evaluating guards, we avoid this non-local choice problem.

A negative combined fragment can be seen as a sub diagram, which, if executed validly, makes the containing interaction invalid. Micskei et al. described in [4] the problems of nesting negative fragments. While nesting of assertions in negative combined fragments is counter-intuitive and should not be used, nesting of negative fragments in one another defies its purpose. Thus the parser will not allow such sequence diagrams.

The UML specification states that "OccurrenceSpecifications on different lifelines from different operands may come in any order" [1]. In the illustration example the messages `switchOn` and `turnLightOn` are independent messages and thus can come in any order. For our example, this means that the `switchOn` message might be sent before the `turnLightOn` message, but as long as `notifySwitchOn` is sent after `turnLightOn`, the constraint is violated. This partial order of messages makes it particularly difficult for a single automaton to capture all traces of an interaction. One of the reasons is that the language of sequence diagrams is not regular [12, 13]. Furthermore multiple independent messages might lead to a state explosion, because the automaton must capture all possible permutations of message occurrences. To cope with this property, instead of using a single automaton for a whole sequence diagram, our approach uses one automaton for each lifeline. This makes the lifelines independent enough to easily maintain partial order.

The automata for each lifeline is created using an algorithm based on the McNaughton-Yamada-Thomspon construction as described in [18]. The Thompson construction is normally used to transform regular expressions into nondeterministic finite automata (NFA). However, the sequence of messages on a lifeline are similar to regular expressions. Therefore this algorithm can be used for our purpose.

Before creating NFAs out of a sequence diagram, the interaction is parsed into intermediate abstract syntax trees (AST). In order to provide a lifeline-centered view, the interaction is split

up into one AST per lifeline. The AST has three different kind of nodes:

Message: a leaf node which represents a message occurrence

CombinedFragment: a node representing a combined fragment. The children represent the operands.

Sequence: a node representing the sequence of multiple interaction fragments. The children represent the sequenced fragments.

One thing to point out is that there is no difference between a sending and a receiving message event on the lifeline. The message event checked by the validation encodes the sender and receiver of the message, though this is not shown in the figures depicting ASTs in order to keep them small.

The NFA construction algorithm is recursive with every call having the following properties:

Input: An AST node resembling the interaction fragment r

Output: An NFA $N(r)$ accepting the interaction fragment r

It traverses the AST in post-order. For each node it constructs a new NFA with its child NFAs.

```
public NFA constructNFA(AstNode x) {
    NFA nfa1 = constructNFA(x.leftChild);
    NFA nfa2 = constructNFA(x.rightChild);
    return applyConstructionRule(x.nodeKind, nfa1, nfa2);
}
```

The individual construction rules for each node kind are described in the Sections 5.2.1 to 5.2.9. The created NFA $N(r)$ has the same properties as the NFAs created with the original Thompson construction:

- $N(r)$ has only one initial state. This state has no incoming transitions.
- $N(r)$ has only one final state. This state has no outgoing transitions.
- All states have either one outgoing transition with a message or multiple ε-transitions.

These properties are maintained for each construction step of the algorithm. The only exceptions here are the negative combined fragment and break combined fragment. However, we will see in Chapter 7 that this does not impede the validity of the algorithm.

5.2.1 MessageOccurrence

A MessageOccurrence defines the event of sending or receiving a message on a lifeline. It is parsed to a simple transition from an initial state i to a final state f (as shown in Figure 5.2). The transition is only executed if the actual message observed conforms to the message specified. For more details on message conformance see Section 5.3.3.

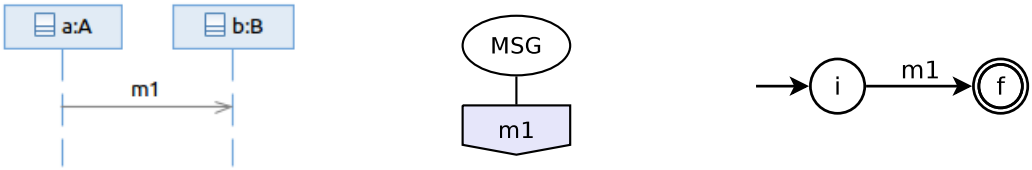

Figure 5.2: Sequence diagram (left), AST (middle) and NFA (right) for a message occurrence *m1*

5.2.2 Sequence

Sequences of occurrence specifications (interaction fragments and message occurrences) are parsed similar to sequences in the Thompson construction. The NFA $N(r)$ of a sequence of two interaction fragments s and t is created as shown in Figure 5.3. The initial state of $N(r)$ is the initial state of $N(s)$ and the final state of $N(r)$ is the final state of $N(t)$. The final state of $N(s)$ and the initial state of $N(t)$ are combined into one state. Thus the final state of $N(s)$ is no longer a final state in $N(r)$. An example of this construction is shown in Figure 5.4.

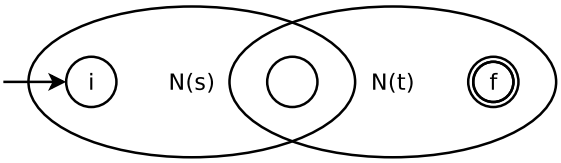

Figure 5.3: NFA for the sequence of s and t. Adapted from [18].

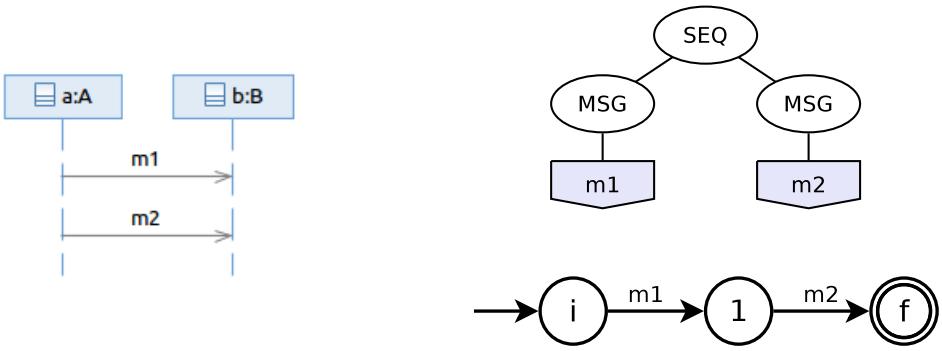

Figure 5.4: Example showing the sequence construction with sequence diagram (left), AST (top right) and NFA (bottom right)

5.2.3 Alternative

As with sequences the construction of alternatives is taken from the Thompson construction. The NFA $N(r)$ of an alternative combined fragment with the two operands s and t is created as shown in Figure 5.5. The new initial state i transitions to the initial states of $N(s)$ and $N(t)$,

while the final states of the two operands transitions both to the same new final state f. The new transitions are ε-transition (empty message). In contrast to normal transitions, ε-transition can (and will) be executed without any input. We will see how ε-transitions are handled in Section 5.3.1. An example of the alternative construction is shown in Figure 5.6.

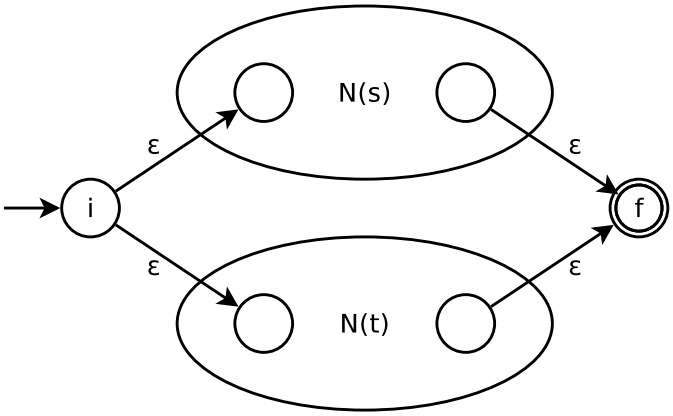

Figure 5.5: NFA for an alternative combined fragment with operands s and t. Adapted from [18].

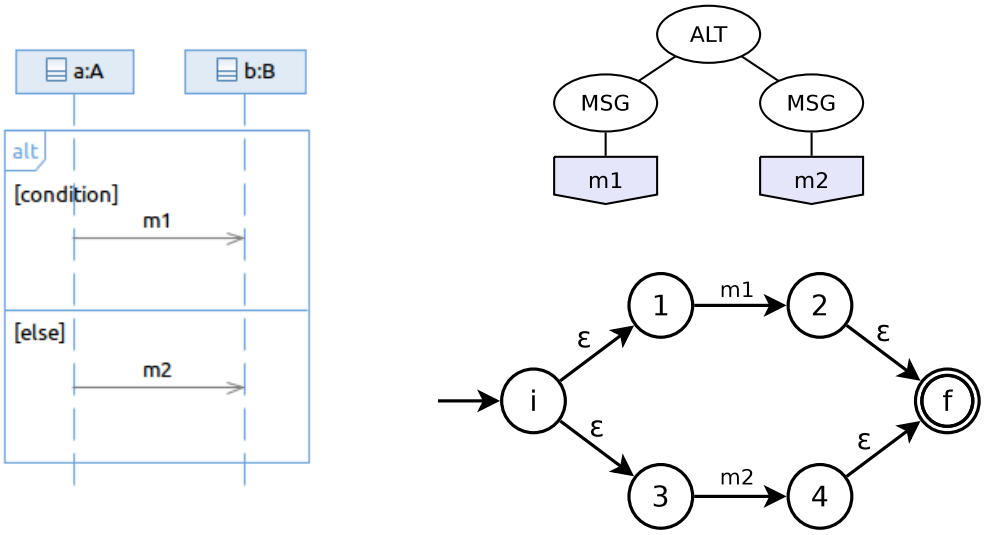

Figure 5.6: Example showing the alternative combined fragment construction with sequence diagram (left), AST (top right) and NFA (bottom right)

Here it should be noted, that the alternative semantics used here differs from the semantics of the alternative combined fragment as stated by the UML Specification [1]. The alternative used here allows exactly one operand to be executed instead of at most one. To comply to the specification the AST created for alternative combined fragments (without an else guard) has an optional node as root. An example for such a case is shown in Figure 5.7.

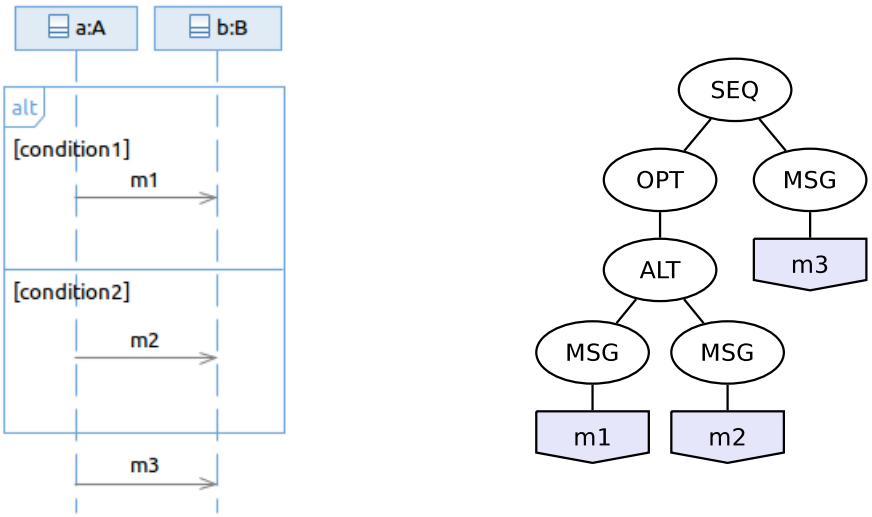

Figure 5.7: Sequence diagram (left) with valid traces *(m1,m3)*, *(m2,m3)* and *(m3)* and the corresponding AST (right) with inserted OPT node.

5.2.4 Optional

The NFA $N(r)$ of an optional combined fragment with the operand s is created like an alternative combined fragment with an empty second operand. The construction is shown in Figure 5.8. The initial state i and final state f are new states. From the initial state i an empty transition leads either to the initial state of $N(s)$ or to the final state of $N(r)$. The final state of $N(s)$ transitions to the final state of $N(r)$ f with an empty message. An example of this construction is shown in Figure 5.9.

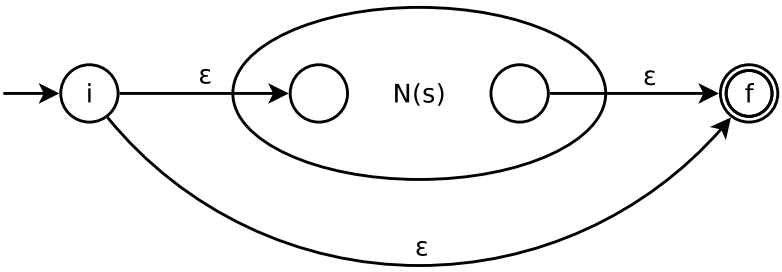

Figure 5.8: NFA for an optional combined fragment with operand s

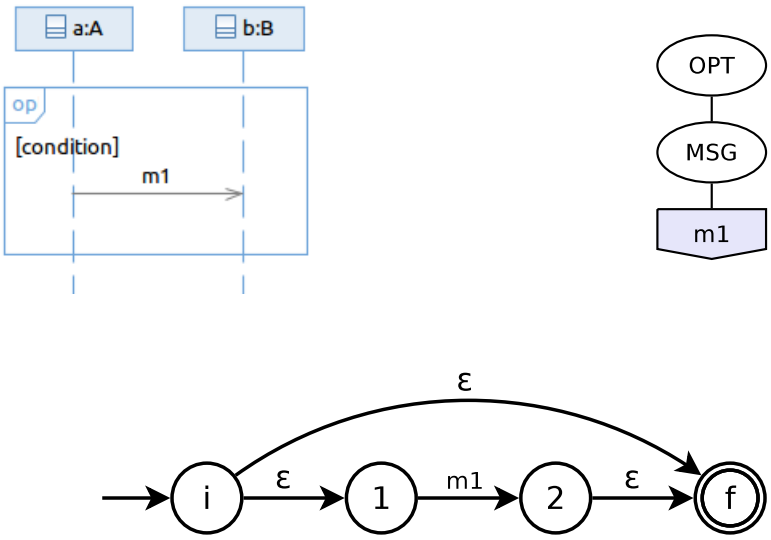

Figure 5.9: Example showing the optional combined fragment construction with sequence diagram (top left), AST (top right) and NFA (bottom)

5.2.5 Loop

The construction of NFA $N(r)$ for a loop combined fragment with the operand s is similar to the optional construction. The only difference is the loop transition from the final state of $N(s)$ to the initial state of $N(s)$. The result is shown in Figure 5.10. This construction corresponds to the loop construction of the original Thompson construction. An example of this construction is shown in Figure 5.11.

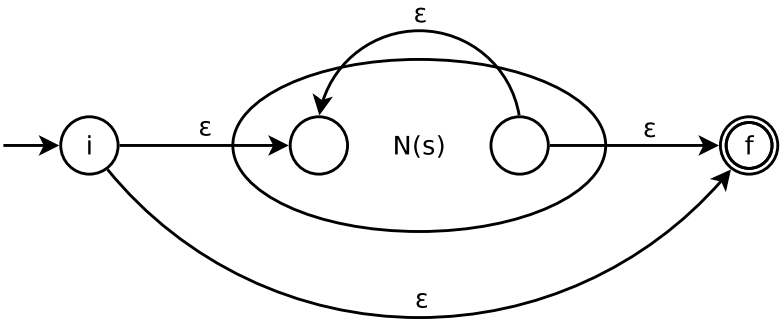

Figure 5.10: NFA for a loop combined fragment with operand s. Adapted from [18].

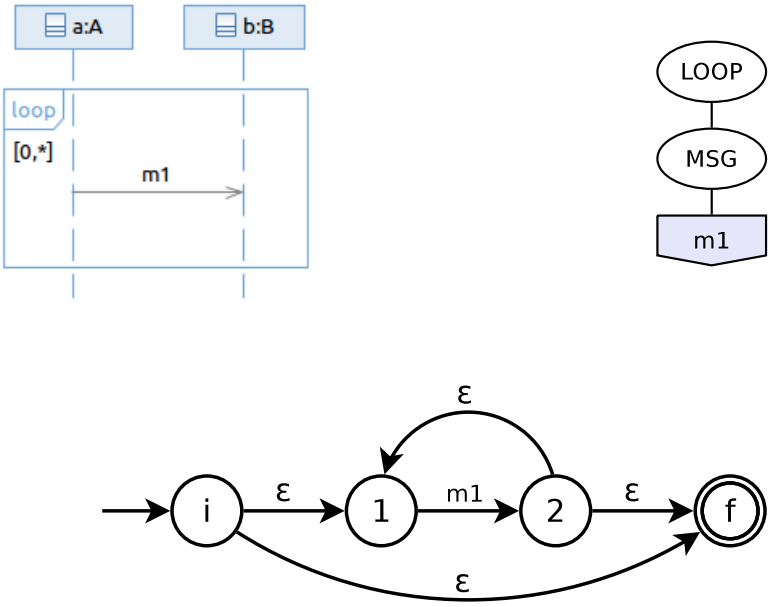

Figure 5.11: Example showing the loop combined fragment construction with sequence diagram (top left), AST (top right) and NFA (bottom)

5.2.6 Break

The break NFA $N(r)$ of a break combined fragment with operand s shown in Figure 5.12 shows a significant difference from the previous constructs. It has two states without outgoing transitions: the final state f and the break state b. The break state b needs to be connected to the final state of the enclosing interaction fragment f'. Since we do not know the enclosing interaction fragment and its final state, we have to remember the break state and connect it when processing the enclosing interaction fragment. An interaction fragment might contain multiple break fragments, in which case all their break states must be connected to the enclosing final state f'. The enclosing final state f' is shown in dotted lines in Figure 5.12. In case the enclosing combined fragment is a break combined fragment as well, the enclosing break would be left but not the enclosing combined fragment of the enclosing break. An example of the break construction is shown in Figure 5.13, where state 4 resembles the break state, which gets attached to the final state of the loop fragment.

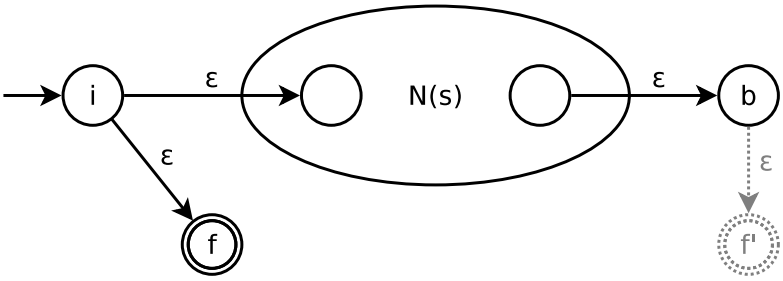

Figure 5.12: NFA for a break combined fragment with operand s

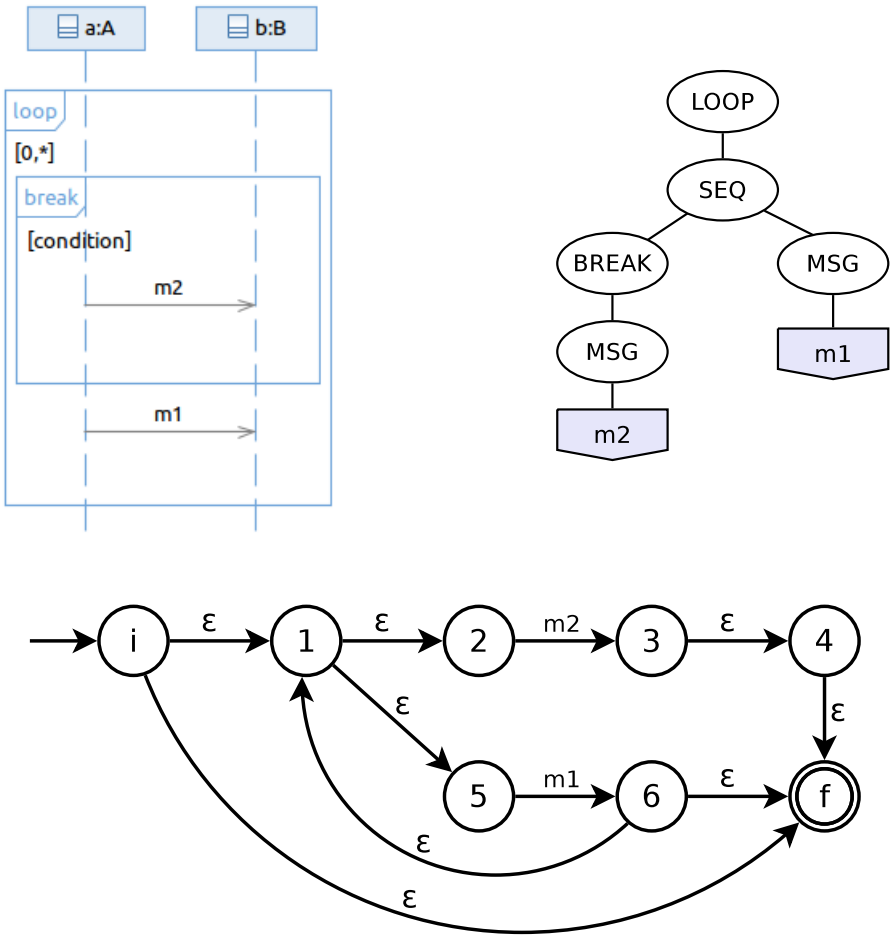

Figure 5.13: Example showing the break combined fragment construction with sequence diagram (top left), AST (top right) and NFA (bottom)

5.2.7 Consider/Ignore

Conventional NFAs are not suited for filtering messages. Therefore, the NFA $N(r)$ for a consider or ignore combined fragment with operand s uses two new state types. Figure 5.14 illustrates the construction of $N(r)$. Before entering $N(s)$ the filter begin state b signals that subsequent states only consider certain messages. After execution of $N(s)$ the filter end state e signals that subsequent states are no long affected by the filter before transitioning to the final state f. The execution is sure to pass the filter end state since all NFAs produced only contain one final state. The only exception here is a NFA representing a break combined fragment. When a break combined fragment is enclosed in a consider or ignore combined fragment, the the break state cannot transition to the final state as with the other combined fragments. The break state here needs to be attached to the filter end state, in order to make sure the execution traverses the filter end state. How the new state types work in detail is described in 5.3.2. An example of the filter construction is shown in Figure 5.15.

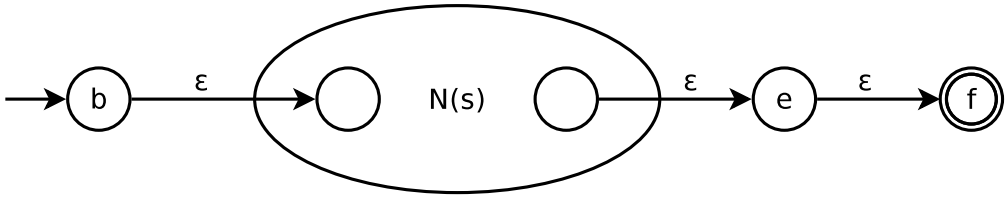

Figure 5.14: NFA for a consider/ignore combined fragment with operand s

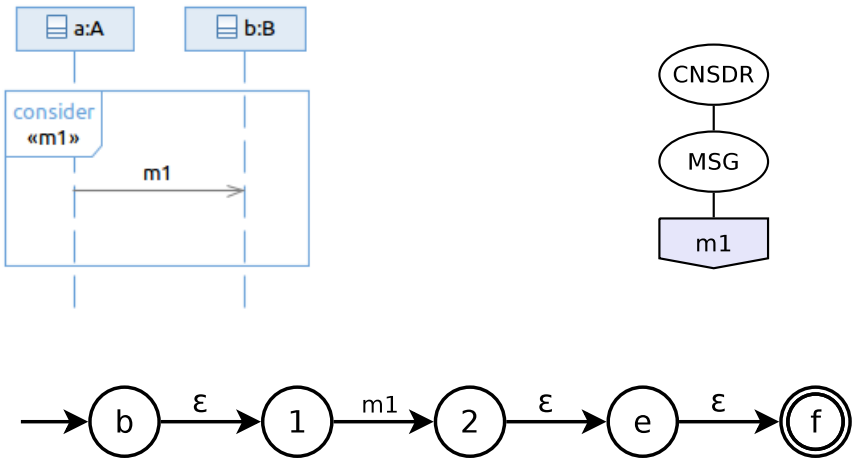

Figure 5.15: Example showing the consider combined fragment construction with sequence diagram (top left), AST (top right) and NFA (bottom)

5.2.8 Negative

The negative combined fragment states that if its operand is executed validly, the trace is invalid. To simplify the design for this fragment's NFA, we can look at it as a sub-interaction. So basically, if this sub-interaction is executed validly, the whole trace becomes invalid. A valid execution is

achieved by reaching the final state on each lifeline. Therefore, the negative combined fragment marks the trace as invalid if the containing lifelines reach the negative final state.

However, what happens when the negative combined fragment is not executed validly? How should we proceed with the rest of the diagram? For this case let us look at the example shown in Figure 5.16.

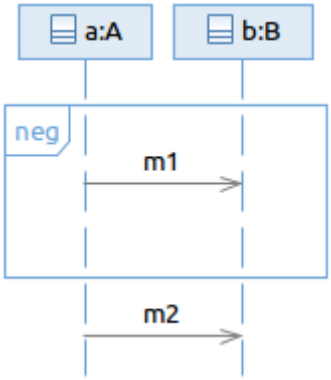

Figure 5.16: Simple sequence diagram with negative combined fragment

Obviously, the trace *(m1)* with any suffix is invalid, since the negative combined fragment states that *m1* must not occur. Nevertheless, the question arises, what is a valid trace? One possible interpretation of this diagram would be, that anything but *m1* can occur, but eventually *m2* must be sent. With this interpretation in mind, sending *m3* before *m2* would be allowed, making the trace *(m3,m2)* valid. This interpretation could be broadened by interpreting "anything" as "any trace" instead of "any message". In that case, the trace *(m3,m4,m3,m2)* would be valid too. However, we chose an interpretation which is more restrictive. The negative combined fragment was built to show unintended behavior, and therefore should not affect the valid traces. Hence, the valid traces stay the same, whether the negative combined fragment is present or not. Therefore, the only valid trace in this example is *(m2)*. If someone want to achieve the behavior discussed before – valid traces *(m3,m4,m3,m2)* –, a consider combined fragment can be used, as shown in the *automatic light* example (see Figure 2.3).

Thus, the NFA $N(r)$ for a negative combined fragment with operand s looks as shown in Figure 5.17. We have one ε-transition form the initial state i to the final state f. This path ensures that the negative combined fragment does not affect the valid traces. The other path leads through $N(s)$ and transitions with an empty message to the negative final state nf. When a trace reaches this state it validly executed the negative combined fragment. In case the other covered lifelines reach their negative final state as well, the trace becomes invalid. We will see in more detail how this works in Section 5.3.2. The resulting AST and NFA for the sequence diagram in Figure 5.16 is shown in Figure 5.18.

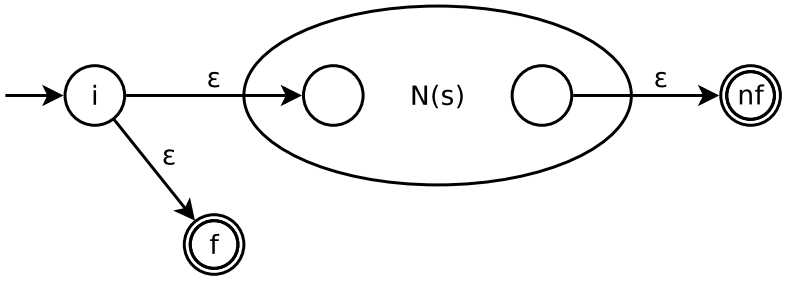

Figure 5.17: NFA for a negative combined fragment with operand s

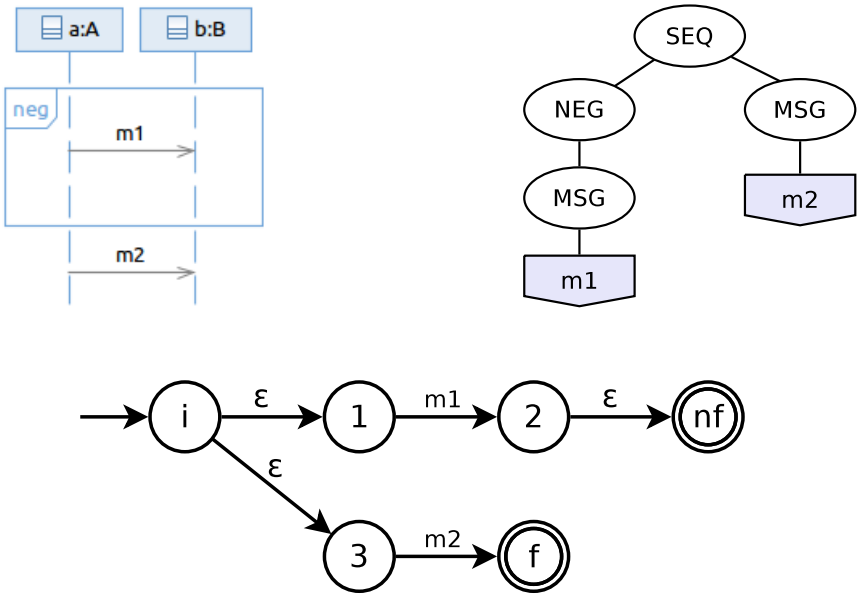

Figure 5.18: Example showing the negative combined fragment construction with sequence diagram (top left), AST (top right) and NFA (bottom)

5.2.9 Assert

In contrast to negative combined fragments, assert combined fragments do affect the valid traces. The operand of an assert combined fragment is part of the valid trace. Nevertheless, the set of valid traces would not change if the operand was not enclosed by an assert. The assert just states that in this situation the stated sequence of messages must be followed. In other words no trace is allowed to "die" within an assert combined fragment. So how can we recognize when a trace "died". In conventional NFAs a trace just ceases to exist, when it has no allowed transition at its current state. Therefore, we have to somehow monitor if a trace ceased. Similar to the consider and ignore combined fragment, we need to tell the execution, when it entered an assert fragment. Hence, we use two new state types: Assert Begin State and Assert End State. The construction, as shown in Figure 5.19, is equal to the construction with consider combined fragments. In the figure, State b is the assert begin state and e is the assert end state. When an execution enters

the assert fragment, we remember it. In the end, it has to leave it again for the assert fragment to be executed validly. So if non of the current states entered the assert fragment, but we know an execution entered it and did not leave it, the trace died within the assert fragment. How this remembering and checking process works, is part of the discussion in Section 5.3.2. An example of the assert construction is shown in Figure 5.20.

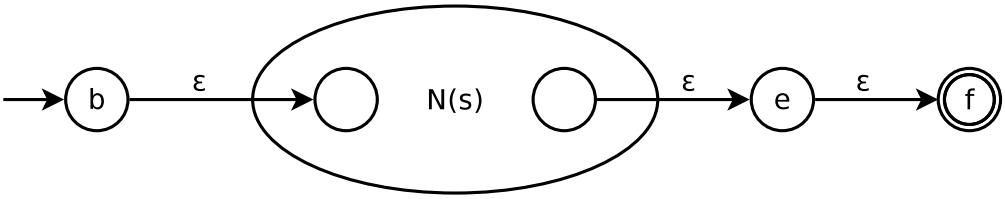

Figure 5.19: NFA for an assert combined fragment with operand *s*

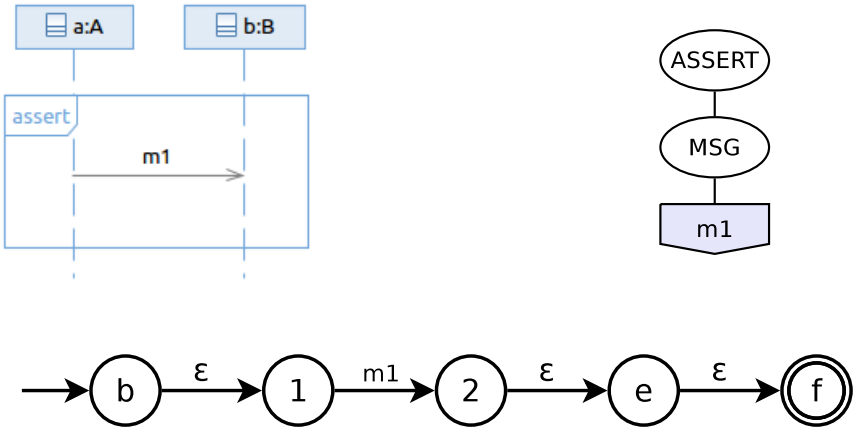

Figure 5.20: Example showing the assert combined fragment construction with sequence diagram (top left), AST (top right) and NFA (bottom)

5.2.10 Special Cases

Empty Operands

The constructions for the combined fragments mentioned above depend on at least one operand being present. However, UML allows a combined fragment to be empty. In this case a simple NOP automaton is created. The automaton is shown in Figure 5.21.

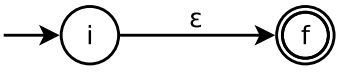

Figure 5.21: NOP NFA used for empty operands

Recursive Application of Construction Rules

Although sequences and combined fragments can have more than two children, the AST is kept binary to simplify the construction of NFAs. If they have more than two children, the node is applied recursively as shown in Figure 5.22. This property results in a recursive application of the construction rules in such cases. The reason why this simplification is possible is the associativity of alternative combined fragments and sequences of fragments in the construction process.

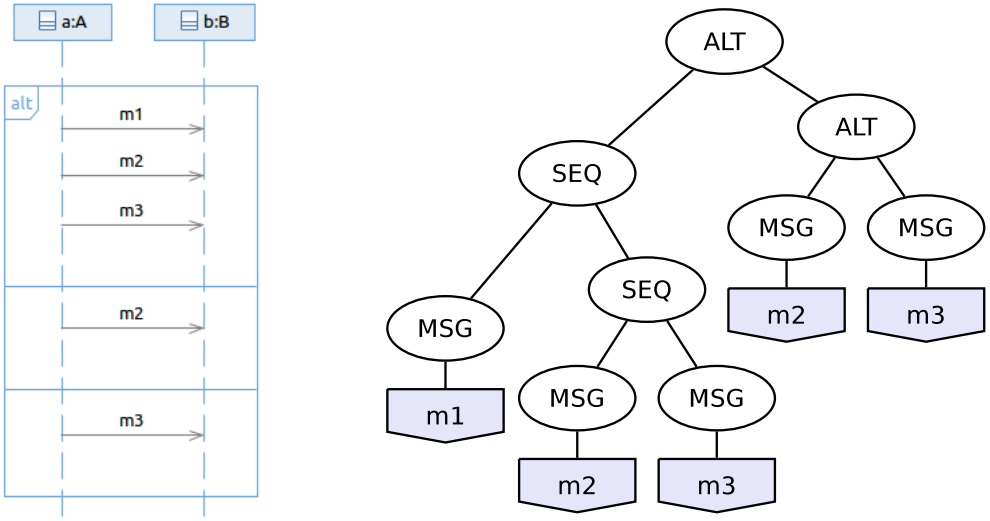

Figure 5.22: Example AST with recursively applied ALT and SEQ nodes on the right side and the corresponding sequence diagram on the left side

5.2.11 Examples

Following the defined algorithm, we can create ASTs and NFAs for our illustration example. Figures 5.23 and Figure 5.24 show the sequence diagram of the *main switch* constraint and the corresponding AST for the *ControlUnit* lifeline Figures 5.25 to 5.29 show the intermediate NFAs created while traversing through the AST. Creation of simple message occurrence NFAs is not shown in the figures but bare in mind that these NFAs are created in a separate step for each occurrence. All other construction steps are shown in the figures. The end result is shown in Figure 5.30. The states labelled *bc, bc', ec* and *ec'* are the begin consider states and the end consider states for the consider combined fragments. The initial state *i* has one empty transition to the final state. This path resembles the possibility that a loop can have zero iterations. The transition from state *8* to state *bc* illustrates the reentering of the loop after an iteration. State *nf* is the negative final state, which can only be reached, when a `turnLighOn` message is sent.

Figure 5.32 and Figure 5.33 show the AST and NFA respectively for the *ControlUnit* lifeline in the *motion detected* scenario. The NFA of the motion detected scenario clearly shows the alternative combined fragment and its branching behavior in state *2*, which splits the execution path into the two alternative paths. The two paths are combined again in the final state *f*.

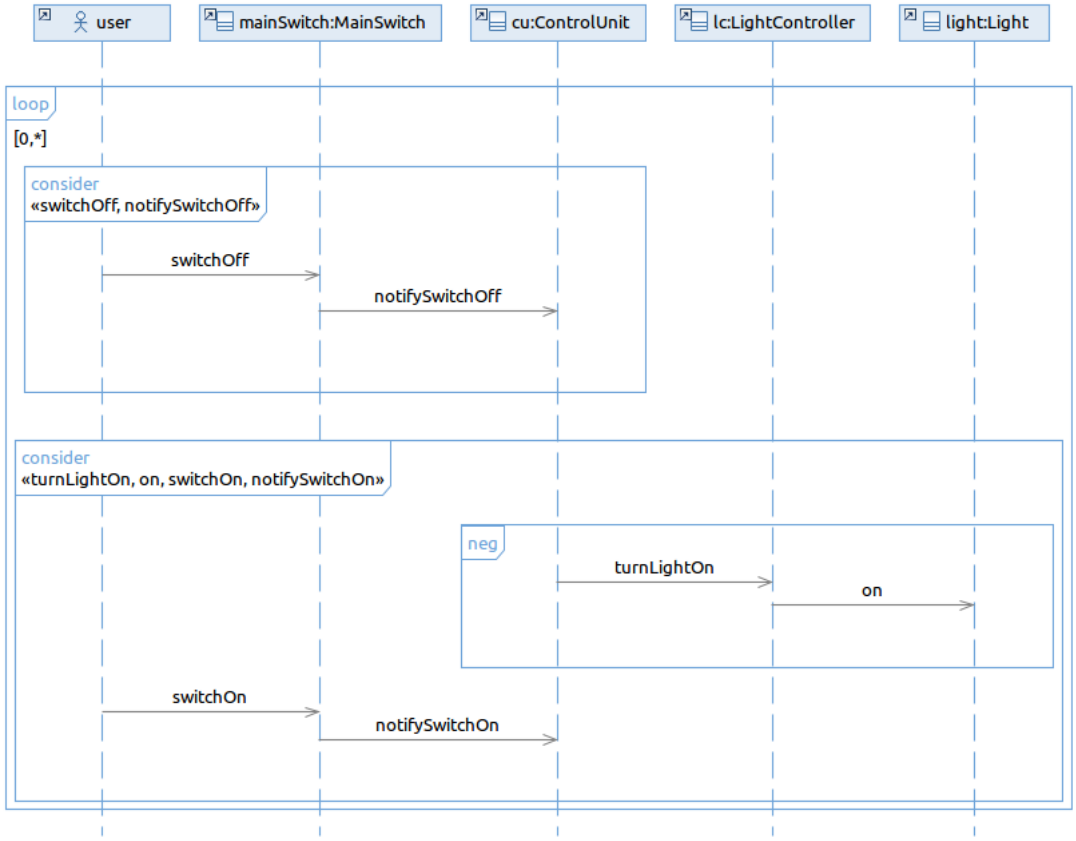

Figure 5.23: Sequence diagram showing the *main switch* constraint

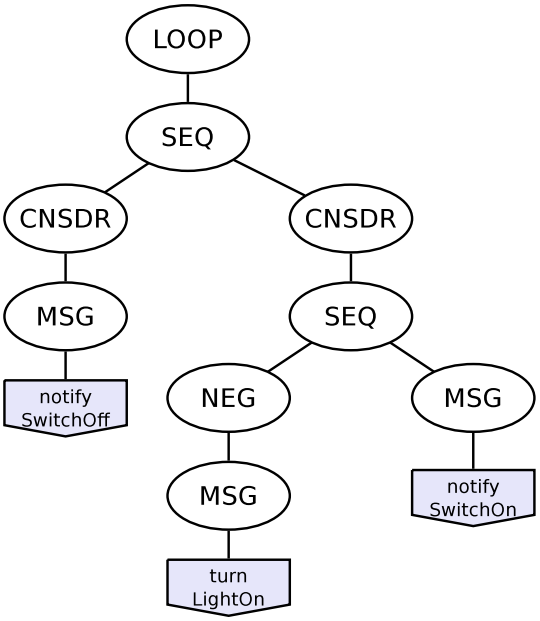

Figure 5.24: AST for the *ControlUnit* lifeline in the *main switch* constraint depicted in Figure 5.23

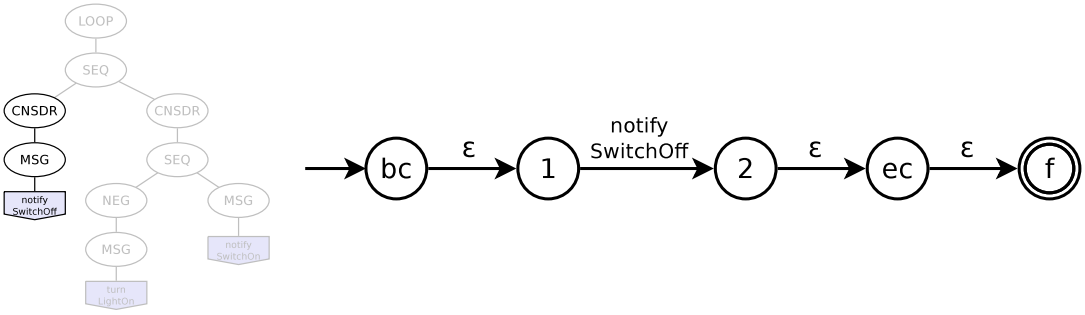

Figure 5.25: NFA constructed from the highlighted subtree on the left side of the AST covering the first consider fragment

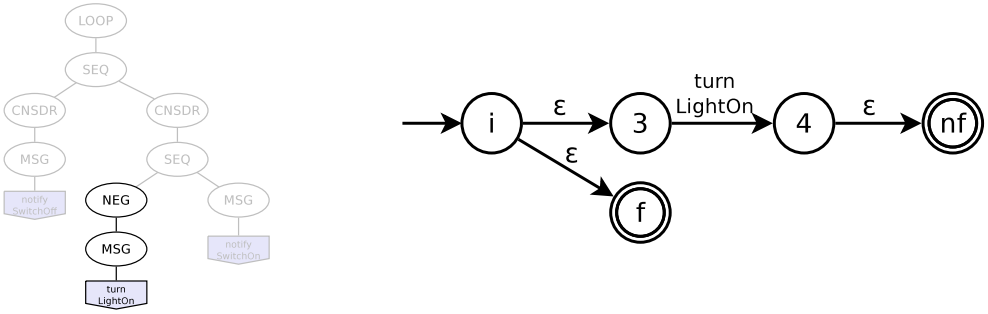

Figure 5.26: NFA constructed from the highlighted subtree at the bottom of the AST covering the negative combined fragment

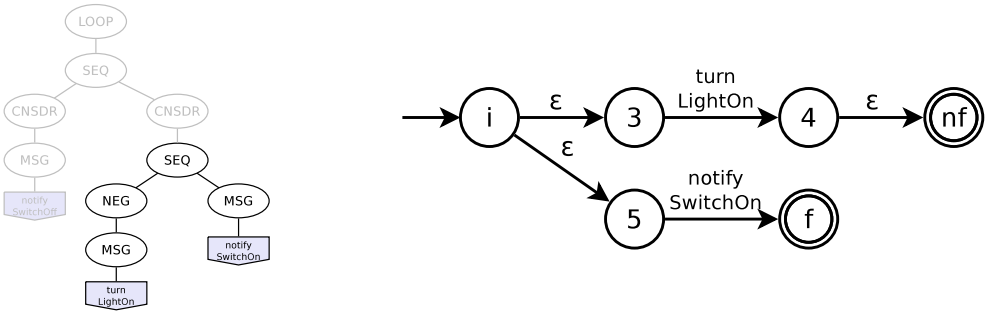

Figure 5.27: NFA constructed from sequencing the NFA from Figure 5.26 with a `notifySwitchOn` message occurrence

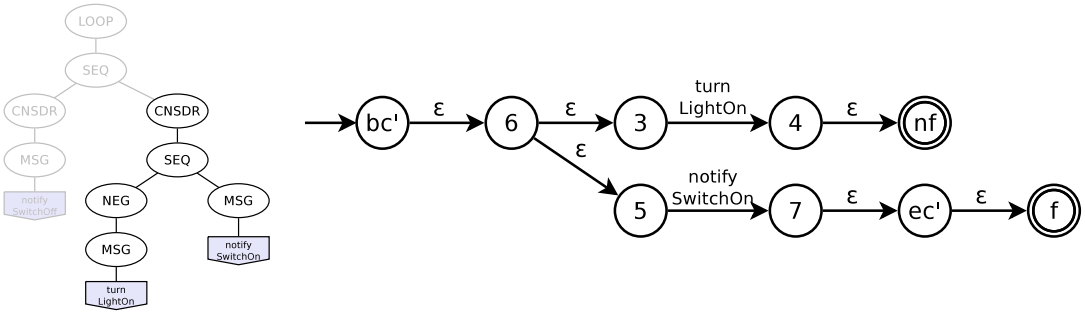

Figure 5.28: Wrapping the NFA from Figure 5.27 with a consider combined fragment

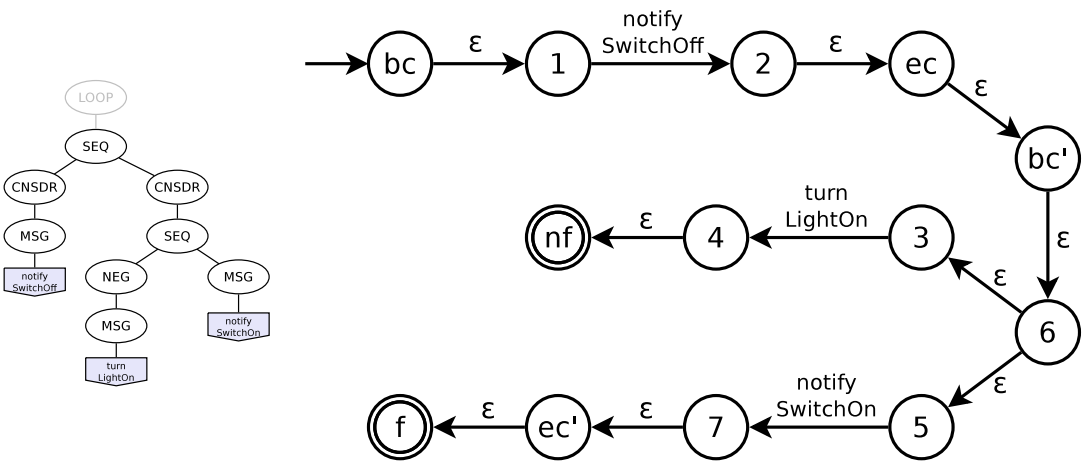

Figure 5.29: Combining the NFAs constructed in Figure 5.25 and Figure 5.28 as a sequence

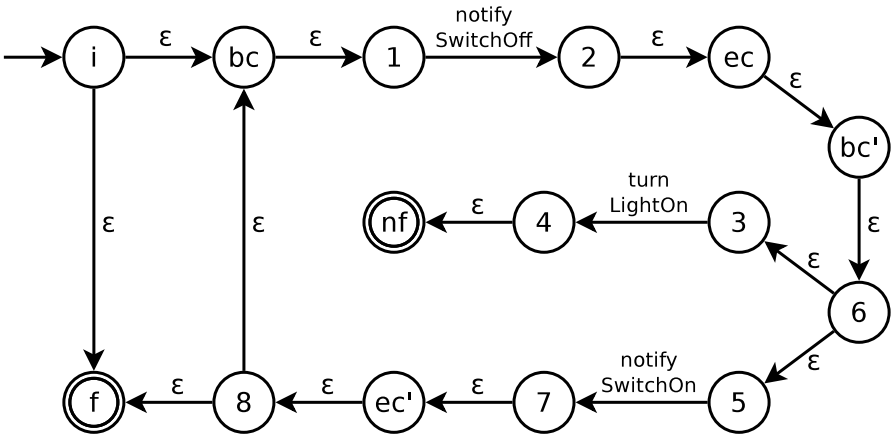

Figure 5.30: Final result of the NFA construction for the *ControlUnit* lifeline depicted in Figure 5.23

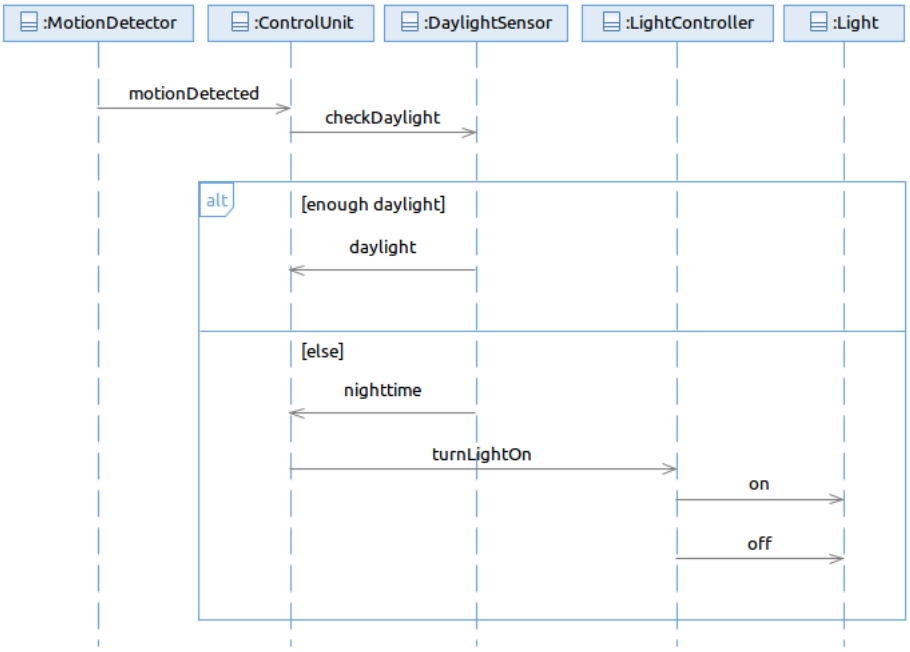

Figure 5.31: Sequence diagram showing the *motion detected* scenario

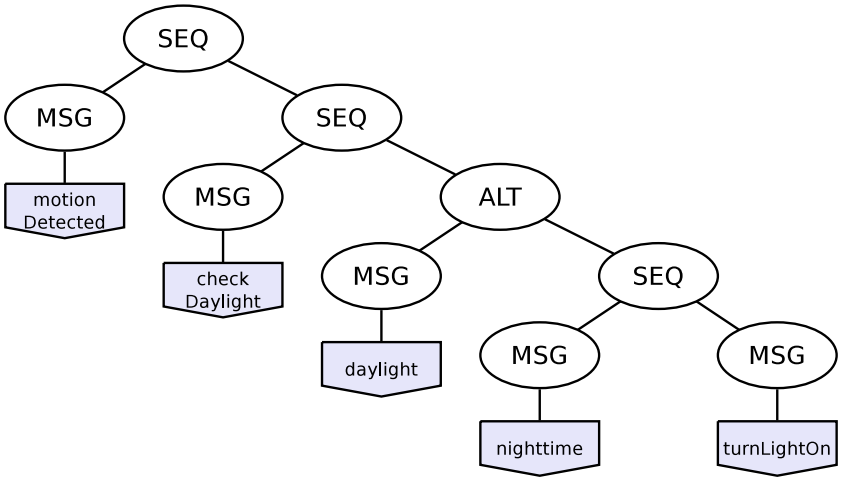

Figure 5.32: AST for the *ControlUnit* lifeline in the *motion detected* scenario depicted in Figure 5.31

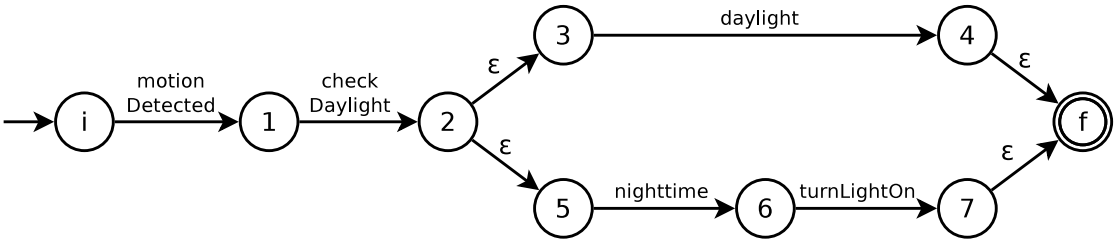

Figure 5.33: NFA for the *ControlUnit* lifeline in the *motion detected* scenario depicted in Figure 5.31

5.3 Live Message Checking

We have seen so far, how to create the NFAs for our validation. In this section we will see, how these NFAs are used for validation during the runtime of the simulation. Each sequence diagram is validated by a component called *Validator*. Figure 5.34 shows the validator and its relationship to other components of the validation process. A validator represents one possible execution of

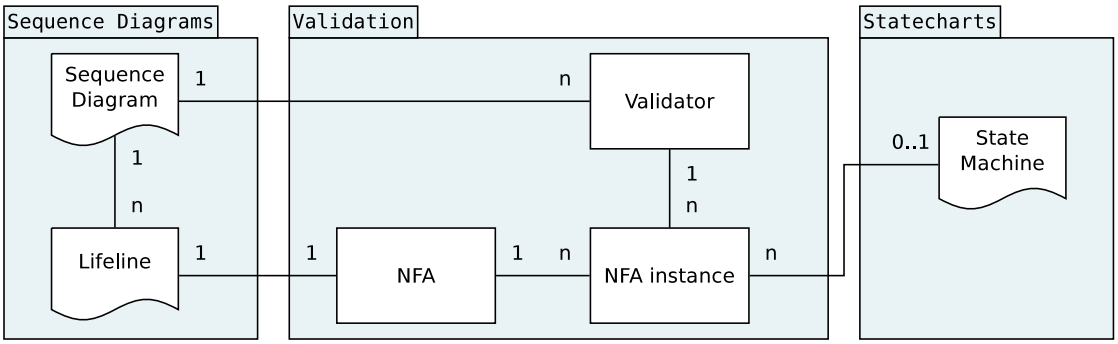

Figure 5.34: The checking components and their relationship

a sequence diagram's interaction. Since a sequence diagram might be executed multiple times, and those executions might interleave, multiple validators can exist for one sequence diagram. Therefore, the validator does not use the NFAs directly, but instances of them. These instances hold the current state of the NFAs for this interaction execution along with some additional information which we will discuss in later sections.

In the following Section 5.3.1 we will first discuss how the transitioning of a single NFA instance works. Thus, the algorithm introduced in this section only affect the NFA and NFA instance. Section 5.3.2 describes how the current states of individual NFA instances are combined to detect valid and invalid executions of sequence diagrams. Hence, this section discusses the validation process from the validators point of view. The subsequent Section 5.3.3 explains how the mapping from state machines to lifelines works. The instantiation and termination of NFA instances is analyzed in Section 5.3.4. The last section of the live checking part, Section 5.3.5, then combines the findings of the preceding sections to explain how the live checking works as a whole.

5.3.1 Transitioning through a Single NFA Instance

Let us assume, the *MotionDetector* detects a motion in the room and thus sends a *motionDetected* message to the *ControlUnit*. The validator takes the message and hands it to the NFA instances of the *MotionDetector* and *ControlUnit* lifelines. The NFAs then process the received message according to the following algorithm:

```
public void transitionNFA(NFAInstance i, Message m) {
    Set<State> newCurrentStates = new Set<State>();
    Set<State> currenStates = i.currentStates;
    for(State state : currenStates) {
        Set<Transition> transitions = state.outgoingTransitions;
        for(Transition t : transitions) {
            if(t.message == m) {
                newCurrentStates.add(t.targetState);
            }
        }
    }
    i.currentStates = newCurrentStates;
    transitionEpsilonTransitions(i);
}
```

For the motion detector NFA, this is a rather simple process. The NFA (shown in Figure 5.35) goes through all current state. At the beginning the only current state is the initial state.

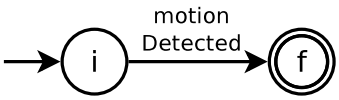

Figure 5.35: NFA for the *MotionDetector* lifeline in the *motion detected* scenario

For each current state, the NFA looks for transitions which contain the received message. In our case the only transition has the message *motionDetected*. Hence, the NFA transitions from the initial state to the final state. The current states after the processing is a set with the final state as its only element. Since the NFA reached a final state, from the lifelines point of view, the sequence diagram was executed validly. However, this does not mean, that the whole sequence diagram was executed validly, but only that this lifeline is done with the validation.

The NFA for the *ControlUnit* lifeline looks a bit more complicated (see Figure 5.33). Nevertheless, the processing for this message works quite similar. It transitions from the initial state to the state with the number 1. Thus, after processing the *motionDetected* message, the new set of current states is {1}. This ends the processing of message *motionDetected*.

If control unit was still switched off, the motion in the room is ignored and nothing happens. So the next message the control unit receives will not match the expected *checkDaylight* message on the transition from state 1 to state 2. When the NFA cannot find any matching transitions for a certain state, this execution path ceases. So after the unexpected message being received, the NFA for the control unit has no current states left. Therefore, no final state can be reached anymore and the whole interaction cannot become valid. We call such an interaction or trace inconclusive (in contrast to valid and invalid states). In our case the interaction done within the state machine simulator just was not the motion detected scenario.

Now let us analyze what happens, when the control unit was not switched off at the time

of perceiving the motion. In this case, the control unit sends a *checkDaylight* message to the daylight sensor to check if the daylight is sufficient enought to light the room. Similar to before, the validator perceives this communication and hands the message to the NFA instances of the sender, the control unit, and the receiver, the daylight sensor. When processing this message, the control unit's NFA will come to state 2. State 2 has two outgoing transition, both of them with ε-transitions. As stated in Section 5.2.3, ε-transitions do not need any input for transitioning. Therefore ε-transitions are transitioned immediately after transitioning the normal message transitions. The following transitioning algorithm only works when the NFA satisfies the following constraint: All states must either have one non-ε-transition or an arbitrary number of ε-transitions. This property allows us to transition the ε-transitions as far as we get without cutting off any possible paths, and thereby improving performance. Luckily, our construction algorithm ensures this property. The ε-transitioning algorithm is repeated until the NFA reaches a stable state, i.e. no transitions are possible anymore. For each current state the algorithm checks if ε-transitions are present. If so, all ε-transitions are taken. This means that the current state might be split up into two ore more current states. In our example the control unit's state 2 has two outgoing ε-transitions. Hence, both of them are taken resulting in the current states being 3 and 5. Since the state changed in this step, the algorithm is repeated. States 3 and 5 have no outgoing ε-transitions. In this case, in contrast to the normal message transition, no transition is taken and 3 and 5 stay current states.

With the algorithm as described we face one problem: In case the NFA contains circles only consisting of ε-transitions (e.g. two states with ε-transitions to one another), this algorithm would never reach a stable state. To avoid this problem, we keep track of all the states we already visited during the ε-transitioning process. When after one transitioning iteration a current state is an already visited state, then this states is removed from the set of current states. The ε-transitioning algorithm is summarized in the following pseudo-code snippet:

```
public void transitionEpsilonTransitions(NFAInstance i) {
    Set<State> currenStates = i.currentStates;
    Set<State> visitedStates = new Set<State>(currentStates);
    boolean newStatesFound = true;
    while (newStatesFound) {
        Set<State> newCurrentStates = new Set<State>();
        for(State state : currenStates) {

            // get epsilon transitions
            Set<State> nextStates = new Set<State>();
            Set<Transition> transitions = state.outgoingTransitions;
            for(Transition t : transitions) {
                if(t.message == EPSILON) {
                    nextStates.add(t.targetState);
                }
            }

            if(nextStates.isEmpty()) {
                // no epsilon transition, keep state
                newCurrentStates.add(s);
            } else {
                // only add unvisited states
                nextStates.removeAll(visitedStates);
                newCurrentStates.addAll(nextUnvisitedStates);
                visitedStates.addAll(nextUnvisitedStates);
            }
        }

        if (newCurrentStates == currentStates)) {
            newStatesFound = false;
        } else {
            currentStates = newCurrentStates;
        }
    }
    i.currentStates = currentStates;
}
```

One interesting property of the algorithm is its idempotence. It allows us to execute it after every received message. This is especially useful in consideration of ignore and consider combined fragments, because they might filter a received message and the current states will not change at all. Furthermore, it allows us to transition ε-transitions after the initialization of the NFA. This initial transitioning is important since some NFAs intial states have outgoing ε-transitions.

In some cases it is possible, that two ore more execution paths meet at the same state. In these cases the approach is similar to ε-transitions leading to already visited states. The two equal current state are merged into one current state. This can be done simply by remembering current states as a set.

Let us get back to our example. The control unit's NFA is now in states 3 and 5. Depending on the current daytime, the daylight sensor will either respond with a *daylight* or a *nighttime* message. As soon as the validator receives this message and the NFA processes it, one of the current states will cease because it does not have any valid transitions. This leaves only one current state. When the interaction took place during the night, the daylight sensor repsonds

with *nighttime*, and the control unit sends a *turnLightOn* signal to the light controller. In this case the NFA transitions through the states 6 and 7 to the final state. In case the daylight sensor responded *daylight* the control unit does not send a signal, and the NFA transition through the state 4 to the final state.

5.3.2 Interaction Detection

Valid Executions

So now the control unit's NFA is in a final state, indicating that this might be a valid execution of the motion detected sequence diagram. However, one NFA alone is not enough to declare an interaction valid. The execution of a sequence diagram is valid, if all its lifelines' NFAs reach a final state. Though this simple condition is sufficient for most cases, there are some cases where this definition is not strong enough. One of these examples is illustrated in Figure 5.36. The

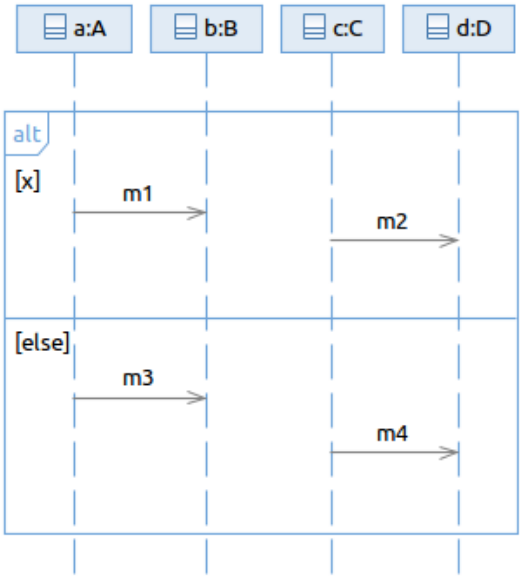

Figure 5.36: Illustration of the global decision problem

problem here is that each lifeline makes its own decisions independent of the other lifelines. There is no component telling the lifelines to either take the first or the second alternative operand and since guards are not evaluated, the lifelines cannot make this decision alone correctly. What does this mean for the depicted example? The valid traces for this sequence diagram are easy to find. There are two alternatives, each representing two valid traces. The first operand is taken when x is true. The obvious valid trace resulting from taking the first operand is $(m1, m2)$. Furthermore, the trace $(m2, m1)$ is also valid because the two messages are independent of each other and can therefore appear in any order. If x is false, the second operand is taken, resulting in the valid traces $(m3, m4)$ and $(m4, m3)$. Although the valid traces are easy to find for a human, the lifelines' NFAs tell a different story. Figure 5.37 shows the NFA for the lifelines *a:A* and *b:B*, while Figure 5.38 shows the NFA for the lifelines *c:C* and *d:D*. Obviously, each lifeline has its own NFA, but in this case the NFAs for a and b and the NFAs for c and d are alike, because

NFAs do not distinguish if they are the receiving or sending participant but always check both sender and receiver of the message.

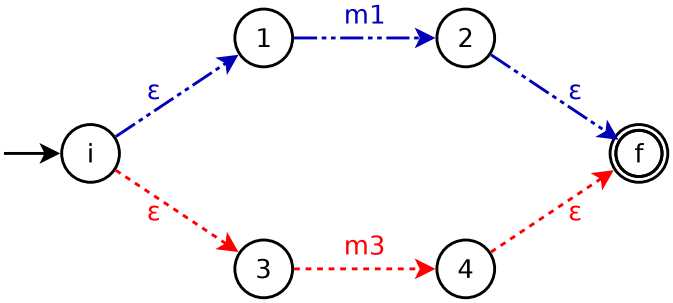

Figure 5.37: NFA for lifelines a:A and b:B in Figure 5.36

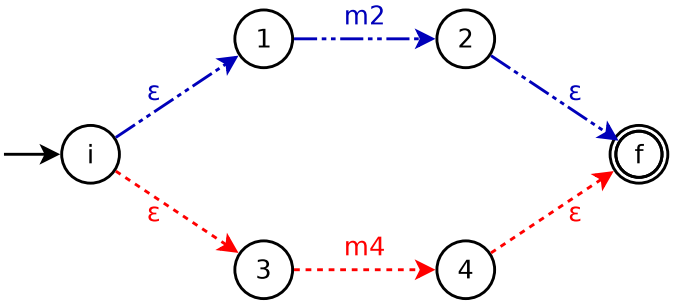

Figure 5.38: NFA for lifelines c:C and d:D in Figure 5.36

For the valid traces discussed above, the four NFAs will transition into their final states, resulting in the valid execution of the interaction. However, this is also true for the traces $(m1, m4)$, $(m4, m1)$, $(m2, m3)$ and $(m3, m2)$. The reason why this is possible is that the NFAs made different decisions which path to take. Normally, the NFAs should unanimously either choose the first operand's path (depicted as dash-dotted lines in the Figures) or the second operand's path (depicted as dotted lines in the Figures). So how can we enforce the same path on every NFA? The simple answer here is, we cannot. The inherent behavior of the designed NFAs is to not make decisions. On every possible turn, every path is taken simultaneously. This behavior prevents us from making wrong decisions. If we would enforce a certain decision on the NFA and later realize that this decision might have been wrong, we would have no way to undo the decision and take the other path. Therefore, we cannot solve the problem by enforcing the NFAs to make a decision. What we can do is check if they made the same decision after they reached the final state. Hence, the definition when an execution is valid is extended to be valid when all NFAs reached a final state and made the same decisions. For the purpose of being able to reconstruct the path the NFA took, a *TraceID* needs to be remembered with the execution of the NFA. A TraceID identifies which path was taken. It is formally defined as a finite sequence of PathIDs $(p_0, p_1, p_2, \dots p_n)$. A PathID denotes a decisions the NFA made during the execution and is defined as $p_x \in N \times D$, where N is a finite set of nodes, where decisions are made and D is a finite set of choices for the nodes. For example, an optional

combined fragment $opt_1 \in N$ can make a decision to enter the operand or to skip it. Hence, the corresponding traces will have TraceIDs with the PathIDs $(opt_1, operand)$ and $(opt_1, skip)$ respectively, where $\{operand, skip\} \subseteq D$. Since an NFA can take multiple paths at the same time, it may also have multiple TraceIDs. Therefore the TraceIDs needs to be attached to the execution which traversed the NFA. Thus, instead of simply storing the current states of an NFA, a bit more information on the execution is stored. Besides the current state, the TraceID of the execution is stored. Whenever the NFA can transition into multiple other states, the execution and its TraceID is split up. Furthermore, one execution might contain multiple TraceIDs since multiple executions with different decisions might end up in the same state and therefore get merged. The sequence diagram's execution is only valid if the TraceIDs of the involved NFAs match. TraceIDs match when they made the same decisions on the same nodes in the same order. The number of nodes may differ, because not all combined fragments are covering all lifelines. Hence, when checking if two TraceIDs match, only the intersection of their sets of passed nodes is taken into account. The following code snippet summarizes the matching algorithm:

```
public boolean matches(TraceId traceId1, TraceId traceId2) {
    // pathMap is a Map<NodeId, PathId>
    //   which stores the sequence of decisions made for each node
    for (NodeId nodeId : NODE_IDS) {
        List<PathId> pathList1 = traceId1.pathMap.get(nodeId);
        List<PathId> pathList2 = traceId2.pathMap.get(nodeId);
        if(pathList1 != null && pathList2 != null) {
            // if both trace IDs passed the same node,
            //   they must have made the same decisions to match
            if(pathList1 != pathList2)) {
                return false;
            }
        }
    }
    return true;
}
```

The creation of TraceIDs within the NFA is done at each state with multiple outgoing ε-transitions. These path splitting states receive the node identifier $n \in N$ of the combined fragment they were designed from. Furthermore each ε-transition receives one decision marker $d \in D$. The resulting PathIDs are assigned to the TraceID when the NFA passes these states and transitions.

One question which remains to be answered is what happens when the TraceIDs do not match. In this case the execution is obviously not valid. However, the TraceIDs of the finished executions are not thrown away. Since the NFAs may be traversable through multiple paths and some paths might take longer than others, it is still possible that an execution with matching TraceID reaches the final state. Hence, if an NFA reaches the final state, all finished executions are searched for matching TraceIDs. If executions with matching TraceIDs for every lifeline are found, these executions and their TraceIDs are discarded. This procedure prevents the validator to produce multiple valid results for closely related paths through the sequence diagram. The algorithm is outlined in the following code.

```
public boolean checkValidResults(Set<NFAInstance> instances) {
    for(NFAInstance instance : instances) {
        for(Execution e : instance.currentExecutions) {
            if(e.isInFinalState()) {
                validTraces.add(new ValidResult(instance, e.traceID));
            }
        }
    }
    // find matching TraceID for every lifeline in validTraces
    Set<ValidResult> results = findMatchingTraces(validTraces);
    if(results != null) {
        // matching TraceIDs found
        validTraces.remove(results);
        return true;
    } else {
        // no matching TraceIDs found
        return false;
    }
}
```

Invalid Executions

UML Sequence Diagrams offer two possibilities to state invalid traces: negative combined fragments and assert combined fragments. What these combined fragments represent and how they behave was discussed in Section 5.2. This section explains how invalid executions are perceived with the use of the constructed NFAs.

Let us start off with the negative combined fragment. A negative combined fragment can be seen as a sub-sequence diagram. If it is executed validly, the trace becomes invalid. This rule also involves the issues discussed in the previous section. The negative combined fragment is only valid if all covered lifelines' NFAs reach the negative final state with matching TraceIDs. The algorithm looks similar to the algorithm for checking valid results.

```
public boolean checkNegResults(Set<NFAInstance> instances) {
    for(NFAInstance instance : instances) {
        for(Execution e : instance.currentExecutions) {
            if(e.isInNegFinalState()) {
                negTraceMap.put(e.negCombinedFragment,
                    new NegResult(instance, e.traceID));
            }
        }
    }
    for(NegCombinedFragment ncf : negTraceMap.keys()) {
        // get the traces of this negative combined fragment
        traces = negTraceMap.get(ncf);
        // find matching TraceID for every covered lifeline of the neg combined fragment
        Set<NegResult> results = findMatchingTraces(traces);
        if(results != null) {
            // matching TraceIDs found
            negTraceMap.remove(results);
            return true;
        }
    }
    return false;
}
```

Assert fragments are a bit trickier. As discussed in Section 5.2, the NFA for an assert combined fragment involves an assert begin state and an assert end state. Upon traversing the assert begin state, the entered assert fragment (or some unique identifier) is remembered at two places. Firstly, it is remembered at the execution, where the current state and the TraceID is already stored. Attaching an assert fragment to an execution means that this execution has entered the assert fragment. Secondly, the assert fragment is stored globally at the NFA instance. Via this reference we get the means to check if an execution ceased. After each processing of a message and the ε-transitioning, the validator verifies if any assert fragment was violated. This is done by checking if all assert fragments stored at the NFA instance are still attached to any current execution. If the stored assert fragment is no longer attached to any execution, then the executions currently residing in the assert fragment ceased, and the assert fragment was violated. When an execution reaches the assert end state, the attached assert fragment is taken from the execution and from the NFA instance. This results in the assert fragment being viewed as executed correctly.

Though the trace should be invalid if only one automaton fails to execute the whole assert fragment, we cannot be sure the path this automaton took was legitimate. Figure 5.39 shows an example where such a path is illegitimate. The problem here becomes clearer when we look at the NFA of lifeline a:A shown in Figure 5.40. Through the states i, 3, 4, ba (the begin assert state) and 5, the NFA transitions into the assert fragment immediately. If the monitor now observes a message m1 sent from a:A to b:B, which is a legitimate execution according to the diagram, the execution currently at state 5 will cease and the validator would render the assert fragment as violated. Therefore the trace is only invalid if every covered lifeline automaton entered the assert fragment with matching TraceIDs and one of them results in an invalid execution. The checking process is outlined in the following code snippet.

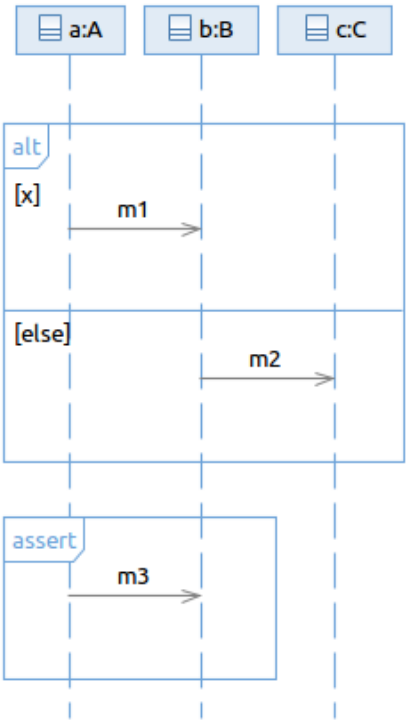

Figure 5.39: Global decision problem with an assert combined fragment

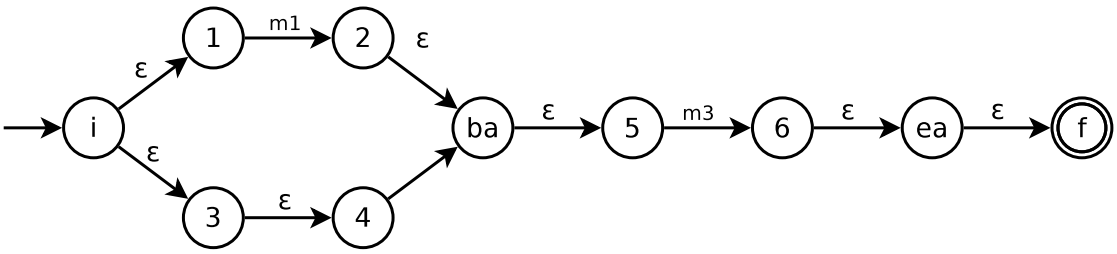

Figure 5.40: NFA for lifeline a:A in Figure 5.39

```
public boolean checkAssertResults(Set<NFAInstance> instances) {
    for(NFAInstance instance : instances) {
        for(AssertFragment af : instance.assertFragments) {
            boolean executionInAssert = false;
            for(Execution e : instance.currentExecutions) {
                if(e.assertFragments.contains(af)) {
                    executionInAssert = true;
                }
            }
            if(!executionInAssert) {
                // all executions within this assert fragment ceased
                if(af.wasEnteredWithMatchingTraceIDs()) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Filtering Messages

As discussed in Section 5.2, the consider and ignore combined fragments also introduce new state types similar to the ones used by assert fragments. When an execution enters such a filtering combined fragment by traversing the filter begin state, it remembers the filter for this fragment until it leaves the combined fragment again by traversing the filter end state. The alternative would be to copy the filter to every state enclosed by the combined fragment, therefore avoiding the need to remember the filter during execution. However, we decided to store the filter execution at runtime because the traversal of all sub-states during compile time might be computationally expensive. There are two types of filters, each used by one of the combined fragment types. When inside an ignore combined fragment messages in the list are filtered. When inside a consider combined fragment messages not in the list are filtered. If a message is filtered, it will not trigger any transitions in the NFA.

Ambiguity

As [4] showed, some UML Sequence Diagrams can be ambiguous. One such diagram is shown in Figure 5.41.

The example is contradicting, since message *m1* leads to both the final and negative final state at the same time. Moreover, some sequence diagrams can create multiple valid and invalid results through different paths. A simple example for this behavior can be created using the sequence diagram shown in Figure 5.42. For this sequence diagram the validation will produce a valid result on every message *m1* sent from *a* to *b*.

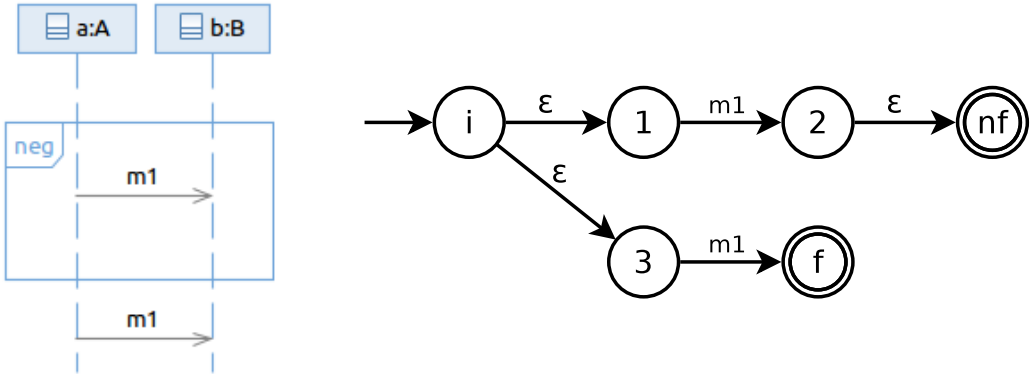

Figure 5.41: Example of an ambiguous sequence diagram and corresponding NFA. Sequence diagram adapted from [4].

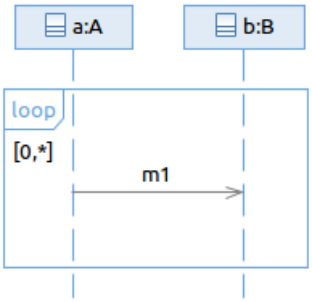

Figure 5.42: Example of an sequence diagram producing multiple valid results

Due to these circumstances, the validator cannot decide which result is "the right result". Therefore, the users have to make this decision, because only they know what the sequence diagram is supposed to mean. The validation process in this paper only points out possible inconsistencies between the sequence diagrams and the state machines.

5.3.3 State Machine Assignment and Binding

We have seen in the previous section, how observed messages are processed by the validator and its NFA instances. However, thus far we ignored the fact that observed messages represent messages sent from state machine to state machine. Instead we assumed that the observed messages contained already the information which lifeline was the sender and which lifeline was the receiver. In reality, this is not the case. Therefore, we need a mapping from state machines to lifelines. This mapping is what we call *binding*. Firstly, we need to know which lifeline is able to represent which state machine. In other words, when is a state machine assignable to a lifeline? How this type conformance is handled is part of the implementation and will not be discussed here. Similarly, the conformance of messages is handled. When is an observed message conforming to a stated message in the sequence diagram? Again this issue can be handled differently by the implementation, but for simplicity let us assume the following rules. Firstly, observed sender and receiver must conform to expected sender and receiver types

respectively. Secondly, the message names must be the same. The rest of the method signature (e.g. parameters, return value) is ignored. A more sophisticated mapping is possible, e.g. via using same method models.

Now that we know which state machine is assignable to which lifeline, we can discuss how the mapping of state machines to lifelines is created. At the beginning, the validator's lifeline are not bound to any state machines. When the first message arrives, the sequence diagram is searched for lifelines which the sender and receiver conform to, respectively. In most cases the state machine will only be assignable to one lifeline. However, the matter is a bit more complicated, when the state machine conforms to multiple lifelines. In this case we try each combination and see if it works out. Therefore, the validator is cloned and the binding is established with different combinations on the cloned (and the original) validators. The binding algorithm is outlined in the following code snippets.

```
/**
 * Binds the state machines sending and receiving the message to appropriate lifelines.
 * If multiple lifelines are possible to bind to, the validator is cloned and
 * the execution will be bound to a different lifeline on every clone.
 *
 * Returns a set containing
 *   - only the original validator if the state machine was already bound
 *        or the mapping was unambiguous
 *   - the original validator and every created clone if the the mapping was ambigous
 *   - no elements if no appropriate lifeline for binding could be found
 */
Set<Validator> bind(Validator v, Message msg) {
    StateMachine sender = msg.getSender();
    StateMachine receiver = msg.getReceiver();
    boolean senderBound = v.hasStateMachineBound(sender);
    boolean receiverBound = v.hasStateMachineBound(receiver);
    if (senderBound && receiverBound) {
        return Collections.singleton(v);
    }

    Set<Lifeline> bindingCandidates = null;
    if (!senderBound) {
        bindingCandidates = v.getUnboundLifelines(sender);
        if (bindingCandidates.isEmpty()) {
            // no candidates for sender, cannot bind message
            return Collections.emptySet();
        }
    }

    Set<Lifeline> bindingCandidates2 = null;
    ... // find binding candidates for receiver similar to above

    if (!senderBound && !receiverBound
            && bindingCandidates.union(bindingCandidates2).size() <= 1) {
        // only one or less candidates for 2 participants, cannot bind message
        return Collections.emptySet();
    }

    // bind candidates and create clones
    Set<Validator> boundClones = v.bindStateMachines(sender, bindingCandidates);
    Set<Validator> allBoundClones = new Set<>();
    for (Validator c : boundClones) {
        allBoundChildren.addAll(c.bindStateMachines(receiver,
                bindingCandidates2));
    }
    return allBoundClones;
}
```

```
private Set<Validator> bindStateMachines(Validator v,
        StateMachine sm, Collection<Lifeline> candidates) {
    Set<Validator> boundValidators = new Set<>();
    for (Iterator<Lifeline> iterator = candidates.iterator(); iterator.hasNext();) {
        Lifeline lifeline = iterator.next();
        if (iterator.hasNext()) {
            // not last candidate -> clone and bind
            Validator child = v.clone();
            child.bind(sm, lifeline);
            boundValidators.add(child);
        } else {
            // last candidate -> bind to original validator
            v.bind(sm, lifeline);
            boundValidators.add(v);
        }
    }
    return boundCheckers;
}
```

If the validator turns out to resemble the wrong binding, the execution will most likely cease and no harm was done. However, if the wrong binding detects a valid or invalid or valid trace the user has to realize this binding as being wrong. Therefore, additional information on how the state machines were bound to the lifelines is provided, when a trace is perceived as valid or invalid. This information gives further insight on how and what went wrong.

Once a state machine is bound to a certain lifeline, the validator remembers this binding throughout its lifetime. After all, a lifeline represents exactly one object. It would not make sense to switch the represented object during the execution of the sequence diagram. However, other validators might have different bound state machines on the same lifelines and its NFA instances might be in different states. This also means that validators with different binding may create different results on the same sequence diagram execution.

When the next message arrives, the live-checker hands it to all validators, which the message concerns. Concerned validators are those, who have a bound or a bindable lifeline for each of the sending and the receiving state machines. The following algorithm summarizes how concerned mappers are looked up.

```
public Set<Validator> getConcernedValidators(Message message) {
    Set<Validator> validatorsContainingSender = getConcernedValidators(message
            .getSender());
    Set<Validator> validatorsContainingReceiver = getConcernedValidators(message
            .getReceiver());
    Set<Validator> freeValidators = getUnboundValidatorsFor(message);
    Set<Validator> concernedValidators = validatorsContainingSender
        .union(validatorsContainingReceiver)
        .union(freeValidators);
    return concernedValidators;
    }

private Set<Validator> getConcernedValidators(StateMachine sm) {
    // validatorMap is a Map<StateMachine>, Set<Validator>>
    //   mapping state machines to validators which have a binding with them
    Set<Validator> boundValidators = validatorMap.get(sm);
    return boundValidators;
}

private Set<Validator> getUnboundValidatorsFor(Message message) {
    Set<Validator> freeValidators = new HashSet<>();
    for (Validator c : unboundValidators) {
        if (c.hasAssignableLifelines(message.getSender(),
                message.getReceiver())) {
            freeValidators.add(c);
        }
    }
    return freeValidators;
}
```

If no concerned validators are present, a new validator is instantiated (see Section 5.3.4).

5.3.4 Instantiation and Termination

Instantiation

As mentioned in the previous section, new validators are instantiated when none of the present validators provide a fitting binding. Moreover, no validator is instantiated for any sequence diagram initially and only instantiated on demand. This behavior provides performance benefits. In large systems, we do not want validators checking all the time when the messages sent do not concern the validator. Hence, validators are only instantiated, when a message arrives which might concern the validator. A message triggers an instantiation of a validator if the sequence diagram defines a message the observed message conforms to (according to the definition of message conformance described in Section 5.3.3). The following code outlines the algorithm for checking the instantiation terms.

```
private boolean shouldBeInstantiated(final Message message) {
    Set<InteractionMessage> definedMessages = constraint.getDefinedMessages();
    for(InteractionMessage im : definedMessages) {
        if(conformsTo(message, im) {
            return true;
        }
    }
    return false;
}
```

One could reason that it is sufficient to instantiate when on of the first messages stated in the sequence diagram is sent. However, this solution would make it impossible to state constraints which look into the past. An example for such a constraint is shown in Figure 5.43. In this example we extend our light example by the control unit's neccessity to register with the main switch in order to receive notifications. Thus, before sending a `notifySwitchOn` or `notifySwitchOff` message to the control unit, the control unit must send a `register` message to the main switch. The depicted sequence diagram models a constraint to ensure this behavior. The problem with instantiating only on the first stated message becomes clear, because if this sequence diagram's validator would only be instantiated on an *register* message, sending an illegal `notifySwitchOn` or `notifySwitchOff` message beforehand would be ignored. Therefore, the validation would not reflect the sequence diagram's intent.

Termination

Since validators are instantiated during runtime, they must be terminated as well, when their job is done. Otherwise, the increasing number of validators would make the validation impossible for larger systems. A validator is done as soon as none of its NFAs contain current states anymore.

5.3.5 Putting the Pieces Together

The previous sections described different parts of the validation process. The flowchart shown in Figure 5.44 sums up the individual parts.

1. A new message was observed by the monitor.
2. Find validators which have bound or bindable lifelines for this message's participants.
3. Check if such validators exists.
4. If no such validator was found, check if the sequence diagram states a conforming message and we should therefore instantiate a new validator.
5. Instantiate and initialize a new validator if the previous check was positive.
6. Bind the message's participants to all concerned validators or the newly instantiated validator. This process might clone validators.
7. Hand the message to the newly bound validators for them to process it
8. Check the results of the validators. If a new result is found, the checker can hand it to other components to present it to the user or add it to a validation result file.

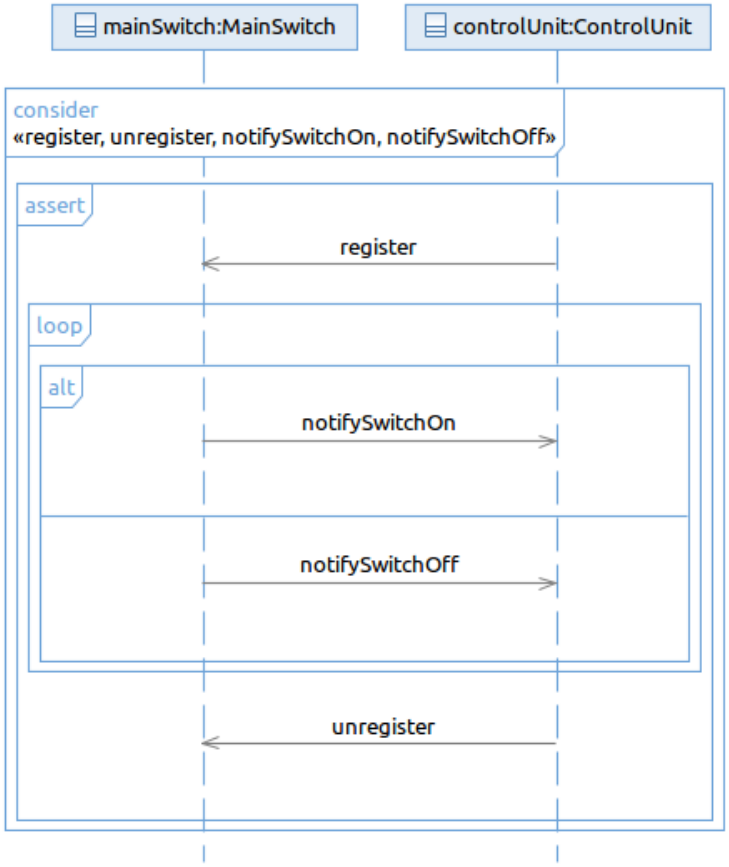

Figure 5.43: Sequence diagram depicting the registration of the control unit with the main switch

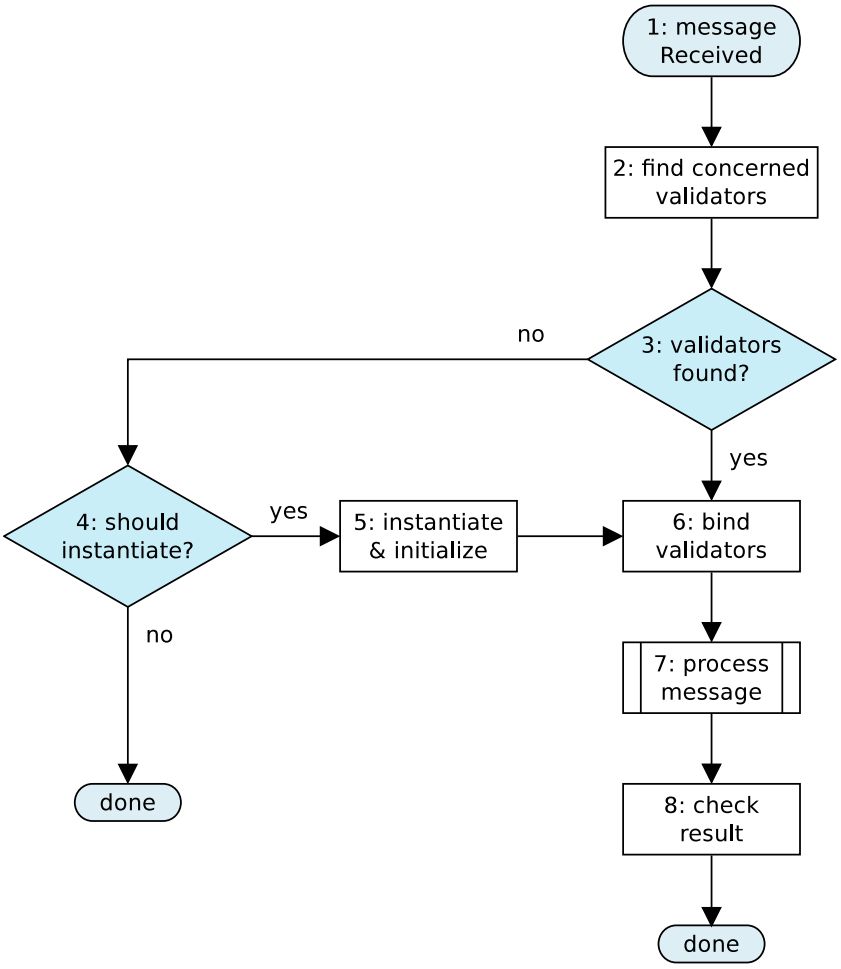

Figure 5.44: Flowchart of the checking algorithm

5.3.6 Example

In this section we will look at an live-checking example to illustrate the introduced concepts. We will use the motion detected scenario for our example. Its sequence diagram is depicted in Figure 5.45. The corresponding NFAs for the lifelines are shown in Figures 5.46, 5.47, 5.48, 5.49, and 5.50.

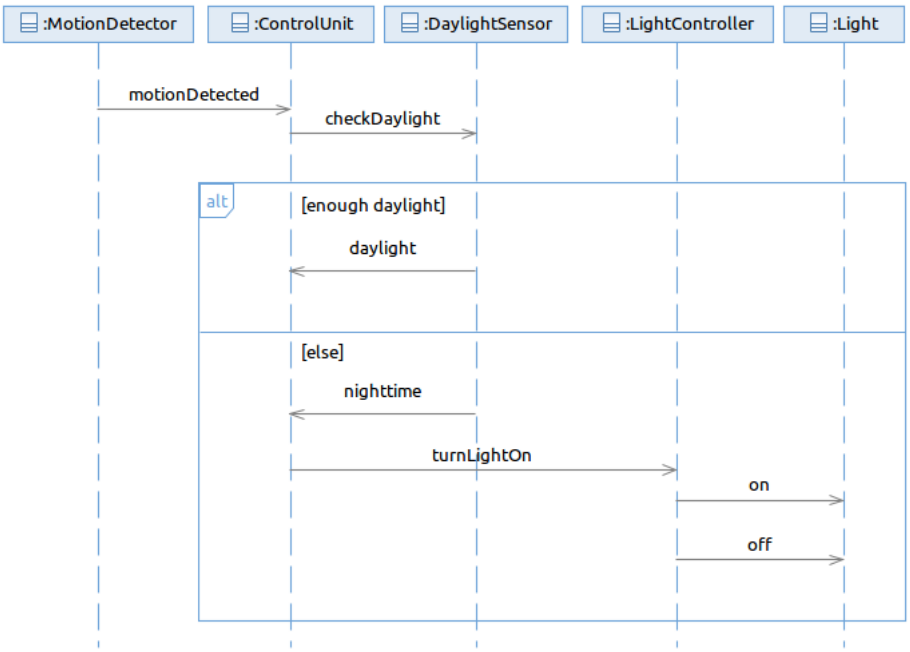

Figure 5.45: Sequence diagram showing the *motion detected* scenario

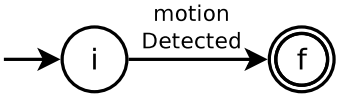

Figure 5.46: NFA for the *MotionDetector* lifeline in Figure 5.45

Our example consists of of the following state machines: A MotionDetector *MD*, a ControlUnit *CU*, a DaylightSensor *DS*, a LightController *LC* and a Light *L*. Table 5.1 and 5.2 show what happens when these state machines communicate with one another. The table resembles a single validator for the motion detected scenario. This example only needs one validator but in other scenarios more validators might get instantiated. The tables contain information on what happens inside the simulation and how the validation module processes the communication as well as the current state of the validator and its NFA instances. The TraceID for each current state is shown in brackets after the state, where “op1” is the first operand of the alternative combined fragment and “op2” is the second operand.

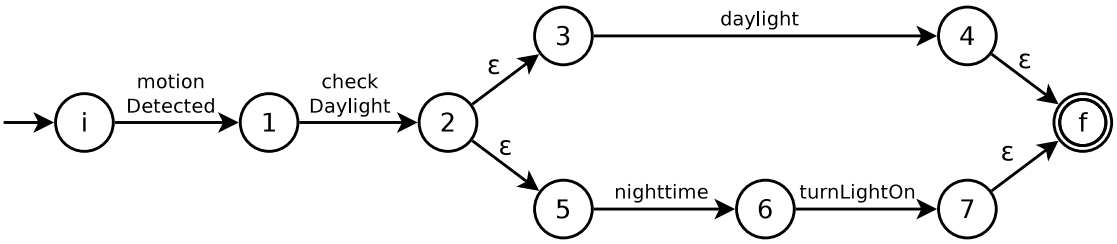

Figure 5.47: NFA for the *ControlUnit* lifeline in Figure 5.45

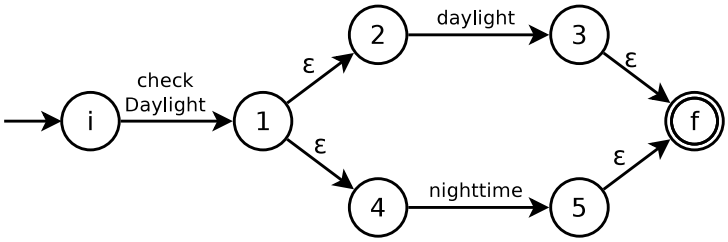

Figure 5.48: NFA for the *DaylightSensor* lifeline in Figure 5.45

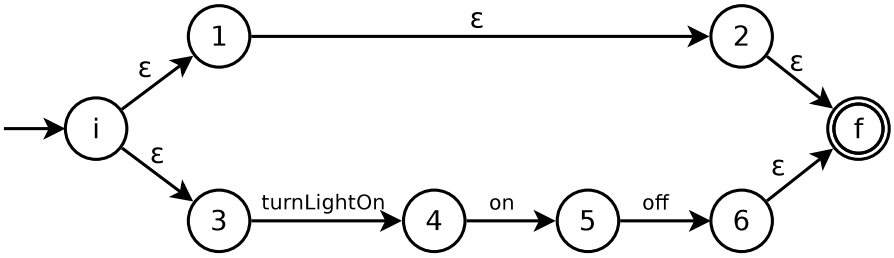

Figure 5.49: NFA for the *LightController* lifeline in Figure 5.45

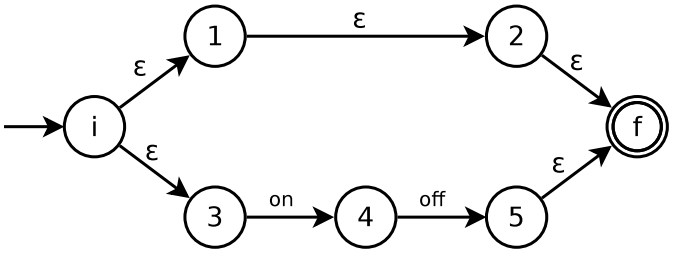

Figure 5.50: NFA for the *Light* lifeline in Figure 5.45

Instruction		**Motion Detector**	**Control Unit**	**Daylight Sensor**	**Light Controller**	**Light**
Simulation: *MD* sends *motionDetected* to *CU* no concerned validators *motionDetected* is a stated message, therefore instantiate validator						
instantiate	States: Binding:	i(-) -	i(-) -	i(-) -	i(-) -	i(-) -
initialize (ε-transitions)	States: Binding:	i(-) -	i(-) -	i(-) -	**f(op1)** **3(op2)** -	**f(op1)** **3(op2)** -
bind	States: Binding:	i(-) ***MD***	i(-) ***CU***	i(-) -	f(op1) 3(op2) -	f(op1) 3(op2) -
transition *motionDetected*	States: Binding:	**f(-)** *MD*	**1(-)** *CU*	i(-) -	f(op1) 3(op2) -	f(op1) 3(op2) -
transition ε-transitions	States: Binding:	f(-) *MD*	1(-) *CU*	i(-) -	f(op1) 3(op2) -	f(op1) 3(op2) -
not all lifelines reached a final state						
Simulation: *CU* sends *checkDaylight* to *DS*						
bind	States: Binding:	f(-) *MD*	1(-) *CU*	i(-) ***DS***	f(op1) 3(op2) -	f(op1) 3(op2) -
transition *checkDaylight*	States: Binding:	f(-) *MD*	**2(-)** *CU*	**1(-)** *DS*	f(op1) 3(op2) -	f(op1) 3(op2) -
transition ε-transitions	States: Binding:	f(-) *MD*	**3(op1)** **5(op2)** *CU*	**2(op1)** **4(op2)** *DS*	f(op1) 3(op2) -	f(op1) 3(op2) -
not all lifelines reached a final state						
Simulation: *DS* sends *nighttime* to *CU* no binding necessary						
transition *nighttime*	States: Binding:	f(-) *MD*	**6(op2)** *CU*	**5(op2)** *DS*	f(op1) 3(op2) -	f(op1) 3(op2) -
transition ε-transitions	States: Binding:	f(-) *MD*	6(op2) *CU*	**f(op2)** *DS*	f(op1) 3(op2) -	f(op1) 3(op2) -
not all lifelines reached a final state						

Table 5.1: Part 1 of the example validation step by step

Instruction		**Motion Detector**	**Control Unit**	**Daylight Sensor**	**Light Controller**	**Light**
Simulation: *CU* sends *turnLightOn* to *LC*						
bind	States:	f(-)	6(op2)	f(op2)	f(op1) 3(op2)	f(op1) 3(op2)
	Binding:	*MD*	*CU*	*DS*	***LC***	-
transition *turnLightOn*	States:	f(-)	**7(op2)**	f(op2)	f(op1) **4(op2)**	f(op1) 3(op2)
	Binding:	*MD*	*CU*	*DS*	*LC*	-
transition ε-transitions	States:	f(-)	**f(op2)**	f(op2)	f(op1) 4(op2)	f(op1) 3(op2)
	Binding:	*MD*	*CU*	*DS*	*LC*	-
all NFAs reached a final state but their TraceIDs are not matching						
Simulation: *LC* sends *on* to *L*						
bind	States:	f(-)	f(op2)	f(op2)	f(op1) 4(op2)	f(op1) 3(op2)
	Binding:	*MD*	*CU*	*DS*	*LC*	***L***
transition *on*	States:	f(-)	f(op2)	f(op2)	f(op1) **5(op2)**	f(op1) **4(op2)**
	Binding:	*MD*	*CU*	*DS*	*LC*	*L*
transition ε-transitions	States:	f(-)	f(op2)	f(op2)	f(op1) 5(op2)	f(op1) 4(op2)
	Binding:	*MD*	*CU*	*DS*	*LC*	*L*
still no matching TraceIDs						
Simulation: after some time *LC* sends *off* to *L* no binding necessary						
transition *off*	States:	f(-)	f(op2)	f(op2)	f(op1) **6(op2)**	f(op1) **5(op2)**
	Binding:	*MD*	*CU*	*DS*	*LC*	*L*
transition ε-transitions	States:	f(-)	f(op2)	f(op2)	f(op1) **f(op2)**	f(op1) **f(op2)**
	Binding:	*MD*	*CU*	*DS*	*LC*	*L*
check result	States:	**f(-)**	**f(op2)**	**f(op2)**	f(op1) **f(op2)**	f(op1) **f(op2)**
	Binding:	*MD*	*CU*	*DS*	*LC*	*L*
every NFA reached a final state with matching TraceIDs *motion detected* scenario was performed validly						

Table 5.2: Part 2 of the example validation step by step

5.4 Advanced Features

Additionally to the general validation process described in the sections above, we designed some more advanced features. The following sections describe these features.

5.4.1 Supertype Binding

As we already mentioned in Section 5.3.3, we use the same type model for state machines and lifelines. Furthermore, this allows us to use polymorphism in our binding. In other words a state machine can be assigned to a lifeline, if the lifeline's type is a generalization of the state machine's type. We could for example use the type *Switch* for the lifeline *mainSwitch* in the main switch constraint (Figure 2.3) because *MainSwitch* is a subclass of *Switch* and its state machine can therefore be assigned. The only algorithmic change necessary for this feature is to change the conformation check.

5.4.2 Wildcard Type

When we take the idea of assigning state machines to supertypes a step further, we can introduce a type to which any other type is assignable (similar to java.lang.Object in Java). This type is designated as wildcard type and it is referred to with an asterisk ('*'). The example shown in Figure 5.51 uses such a type in its first lifeline. The figure shows a simple interaction stating that after a light was switched on by anything, it should be switched off again eventually. It should be noted that the second asterisk in the lifeline header, the one after the colon, denotes the wildcard type. The other asterisk denotes a different type of wildcard (see next section).

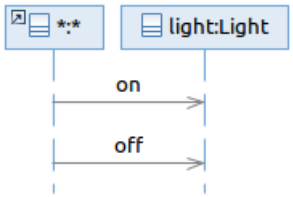

Figure 5.51: Simple sequence diagram using wildcards

Again, the only change to be made to the algorithms is to adapt the conformation check accordingly. This feature can also be used to hide implementation details. If, for instance, the design changes and the control unit controls the lights directly instead of the light controller, then this sequence diagram is still valid. Hence, the constraint is more robust by not explicitly mentioning the particular classes. The same is valid for using a supertype as described in the previous section.

5.4.3 Wildcard Lifelines

A lifeline always represents a certain object or attribute of the system and possesses a certain type. This property makes it difficult to state constraint like, "No object should ...". Let us take the light handling interaction depicted in Figure 5.51 as example. We want to specify that after the light was switched on, it must be switched off again. It does not matter whether the object turning the light off was also the one who turned it on. To be able to express this, we

use a wildcard as lifeline identifiers (denoted with '*'). The first asterisk in the first lifeline of the example sequence diagram marks the lifeline as a wildcard lifeline. This means that no particular state machine is bound to this lifeline. It can represent multiple state machines at the same time. Therefore, the binding algorithm checks if the lifeline is bindable. If this is not the case it establishes a temporary binding, which is used for the message translation. The temporary binding is removed after the observed message has been processed.

Since the state machine is changing and only the messages stated in the sequence diagram are actually recorded, we can only observe a fraction of the communication done by the state machines assigned to the wildcard lifeline. Thus, a validation of a wildcard lifeline does not make any sense. Hence, the NFA created for wildcard lifelines is a NOP NFA (similar to the one shown in Figure 5.21). Nevertheless, the lifelines interacting with the wildcard lifeline are still validated. Moreover, all other rules how the other lifelines are bound and validated still apply.

Since this feature worked so well for the light handling example, we might also be able to use it for the main switch constraint as well. This enhancement is quite reasonable because the only important parts of the constraint are when the user switches the main switch and if the light was lit. It does not state anything about the control unit or the light controller. Therefore, we should hide these implementation details behind a wildcard lifeline. The result of this modification is shown in Figure 5.52. As we can see the control unit and light controller are no longer mentioned and the sequence diagram was simplified.

Unfortunately, this is not how wildcard lifelines work. We mentioned earlier that wildcard lifelines are not validated. This lack of validation leads to a decoupling of the sequence diagram's parts. Figure 5.53 illustrates this behavior by splitting the wildcard lifeline. This sequence diagram will create the same validation results as the one with just one wildcard lifeline. However, in this diagram we can clearly see that the interaction was split into two independent parts. The check if the light is allowed to be lit does no longer depend on the switching of the main switch. Therefore the validation would yield an error whenever the light is lit. The consequence of this is that we should not use this feature to connect parts of the diagram, but only to communicate with the "outside world" (at the edge of the diagram) like the light handling handling example.

5.4.4 Actor Lifelines

Another special lifeline depicted in the main switch constraint example is an actor lifeline. According to [1], "an actor specifies a role played by a user or any other system that interacts with the subject." A developer can use such a lifeline to model the user of the simulator or other external systems. Actor lifelines are similar to wildcard lifelines in that they are not evaluated, because an actor is not part of the system and therefore outside the validation scope. Nevertheless, messages sent from an actor lifeline to another lifeline are still validated to be from an actor in the receiving lifeline's NFA.

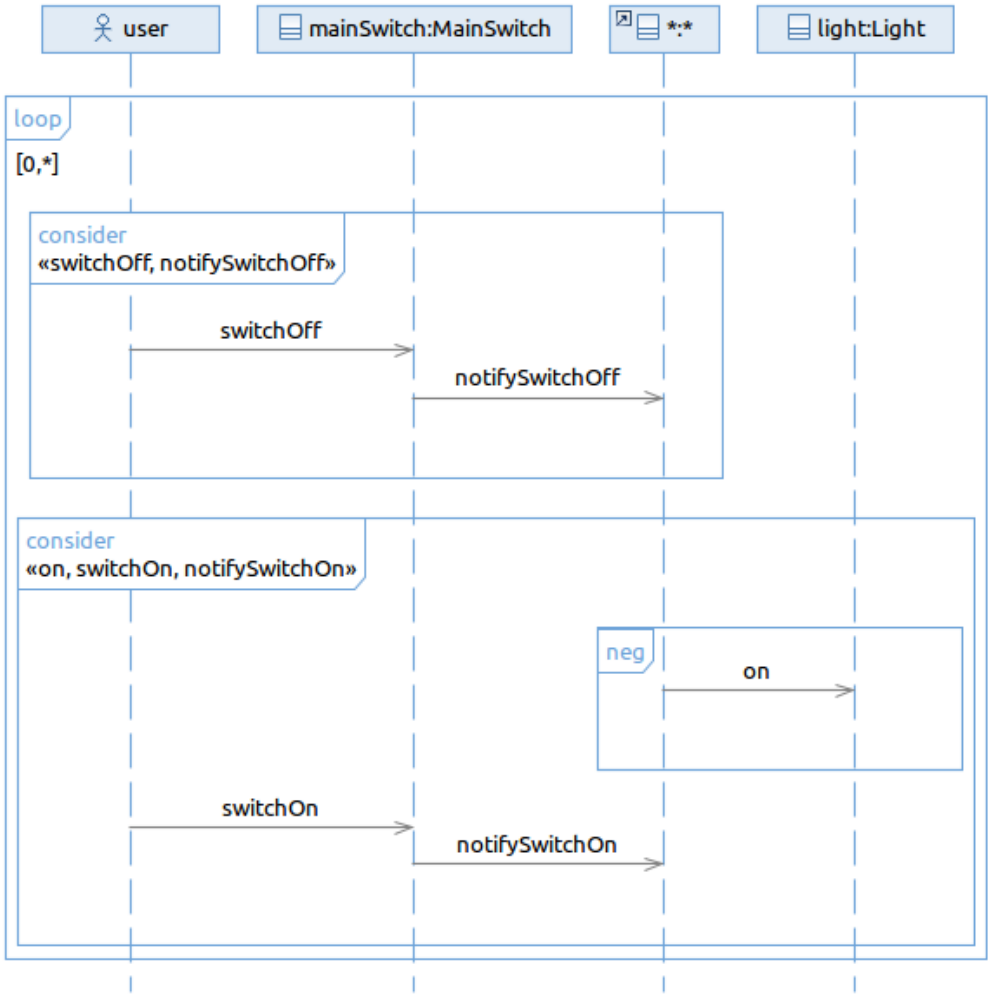

Figure 5.52: *Main switch* constraint with wildcard lifeline

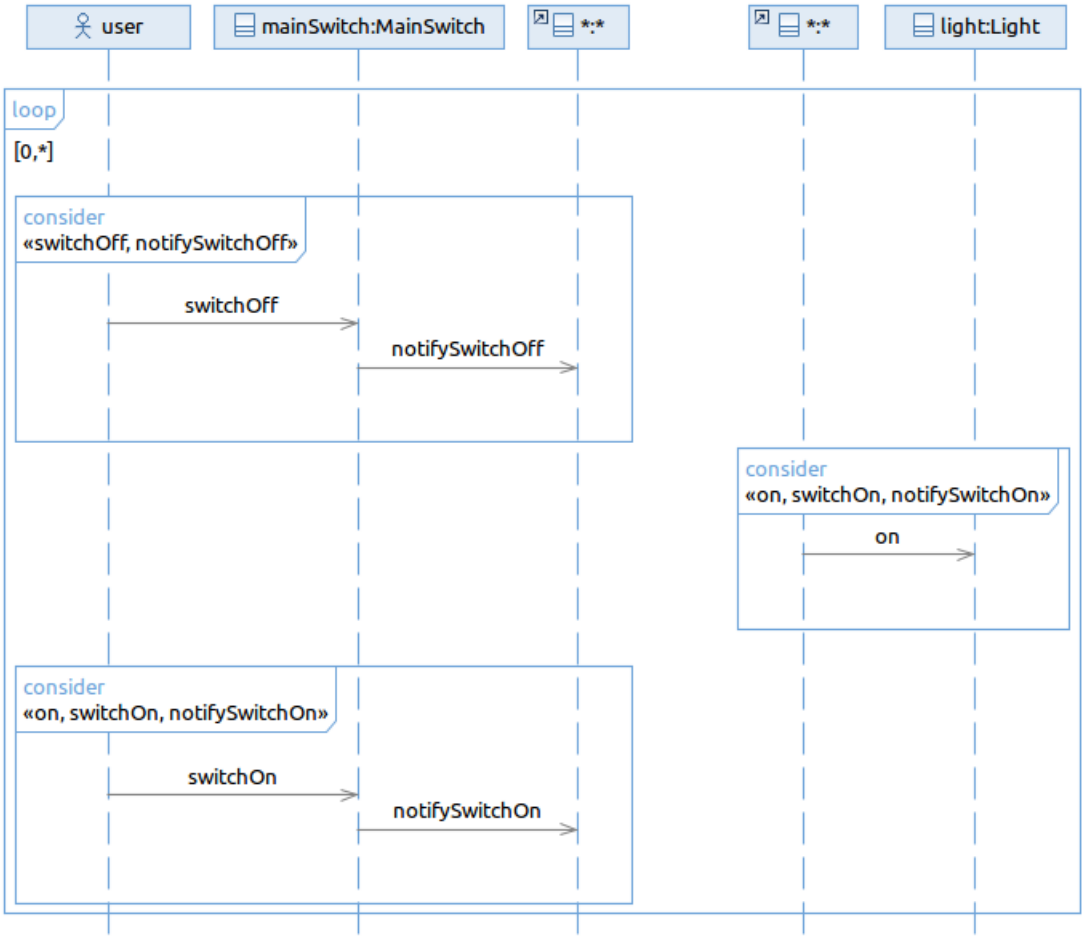

Figure 5.53: *Main switch* constraint with split wildcard lifelines

5.5 Sequence Diagram Semantics

As [4] pointed out, sequence diagrams are not fully formalized. There is room for interpretation, which lead to different semantics used for sequence diagrams. We used semantics which we found were intuitive for most users and fit our purpose. Nevertheless, these semantics can be adapted or extended by changing the constructed NFAs or the algorithms accordingly. Which part needs to be altered depends on how the semantics is changed. If for instance the interpretation of negative combined fragments is changed to allowing any message but the stated one (see Section 5.2.8), a change of the NFA construction rule for negative combined fragments is sufficient. However, changing the semantics to allowing intermediate messages between stated messages would lead to major changes in the NFA construction and checking algorithms. In this case especially the decision on whether a message is an intermediate or a stated one is a difficult task, since all NFAs must agree on one decision.

Chapter 6

Implementation

The following chapter describes the implementation of a tool utilizing the proposed approach. The tool is implemented as a plugin for IBM's Rational Software Architect. It is written in Java and based on the SDS described in [19].

6.1 IBM Rational Software Architect Plugin

IBM Rational Software Architect is an integrated development environment developed and maintained by IBM [20]. It has good modeling capabilities and supports UML 2. This feature makes it an attractive base for our tool.

Rational Software Architect is built on the Eclipse IDE [21]. Therefore, it uses the same OSGi-based plugin mechanism. The plugin, its capabilities and dependencies are described in three files: MANIFEST.mf, plugin.xml and build.properties. The plugin can be deployed as a simple JAR file. While build.properties only describes the building process of the plugin – i.e. which files to compile and include in the final plugin –, MANIFEST.mf and plugin.xml are encapsulated within the JAR. For installing the plugin it is sufficient to copy the JAR to the plugins folder of the Rational Software Architect's installation directory. On the next start, the plugin is automatically loaded and can be used by the developer.

The most interesting features of Rational Software Architect is the capability for modeling UML state and sequence diagrams. These diagrams are translated into Eclipse UML2 models, which is an EMF-based implementation of UML [22]. These models can be easily accessed by the plugin for the simulation of state machines and the validation of sequence diagrams.

6.2 Simulator for Dynamic Statecharts

The state machine simulator used in our implementation is the Simulator for Dynamic Statecharts (SDS) as described in [19]. The term "dynamic" refers to the fact, that state machines can be dynamically instantiated and terminated, similar to the instantiation of classes in object-oriented languages. The original SDS was developed in J# for Rational Rose. A screen shot of it is shown in Figure 6.1. In order to be able to use it for our purpose, it was migrated to Java and integrated into our plugin.

The most important step upon integrating SDS into the tool was to connect it to the UML models created with Rational Software Architect. However, migrated SDS version does not use Eclipse UML2 directly. An abstraction layer lies between the simulator and Eclipse Uml2. This

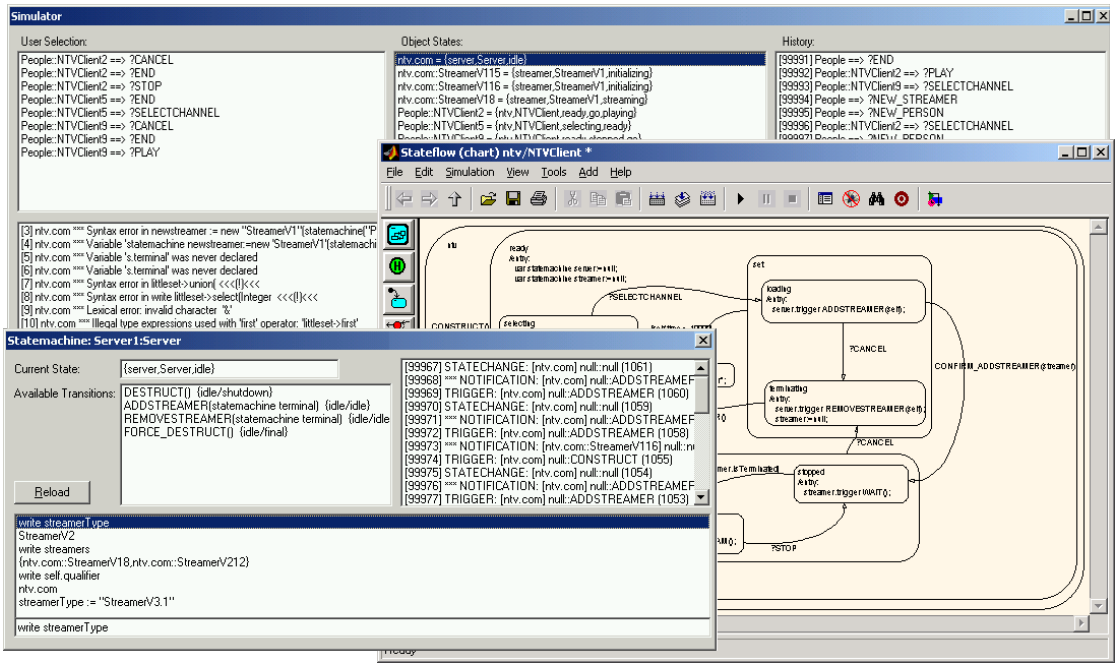

Figure 6.1: Screenshot of the original SDS [19]

layer allows us to change the used model easily and therefore integrate SDS into other tools with different models. The layer consists of a couple of interfaces representing the elements used by the simulator. The tool plugin contains an implementation of these interfaces for Eclipse UML2. One way to accomplish the connection would be to transform the Eclipse UML2 model into a model implementing the mentioned interfaces. However, the Eclipse UML2 is quite large and the transformation would be therefore computational expensive. Especially since we probably will not need the whole model but only a subset of it. Thus, we instead access the Eclipse UML2 model on demand and translate the requests and results. This kind of design pattern is commonly known as adapter pattern as described in [23]. Therefore, the actual UML2 elements are wrapped by the implementing adapter classes. An example of this wrapping can be seen in Figure 6.2. In many cases the result of the accessed method are again Eclipse UML2 classes. An example of such a case is the operation *getAllOperations()* by the interface *org.eclipse.uml2.uml.Class*. It returns a list of *org.eclipse.uml2.uml.Operation* objects. These have to be wrapped themselves before they can be returned as the result of the *getOperations()* operation. So to avoid wrapping the same Eclipse UML2 objects repeatedly, all wrapped objects are stored in a registry. Before wrapping a new object, the tool first checks if the registry already contains a wrapper for the object. If the registry does not yet contain such a wrapper, a new wrapper will be created and added to the registry.

SDS uses its own language called Statecharts for Dynamic Systems Language (SDSL) to describe guards, triggers and actions in the statechart. Figure 6.3 depicts the light controller state machine from the automatic light example, enhanced with SDSL statements. It shows some of the features of SDSL including state machine instantiation (`light := new("Light");`), sending messages from one state machine to another (`light.trigger(on);`), and time-triggered transitions (`self.time > 90000`). Additionally to the current state, the state machine can

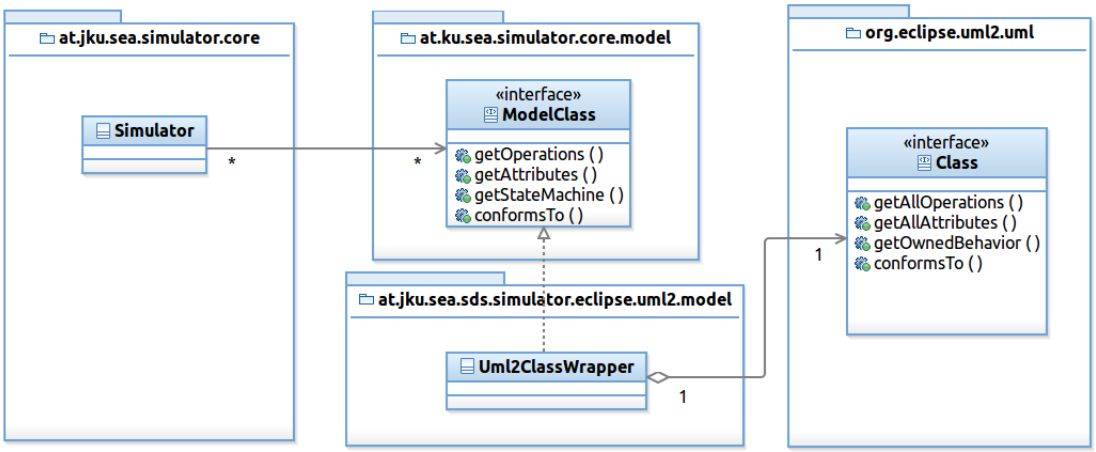

Figure 6.2: Abstraction layer between SDS and Eclipse UML2 at the example of *Class*

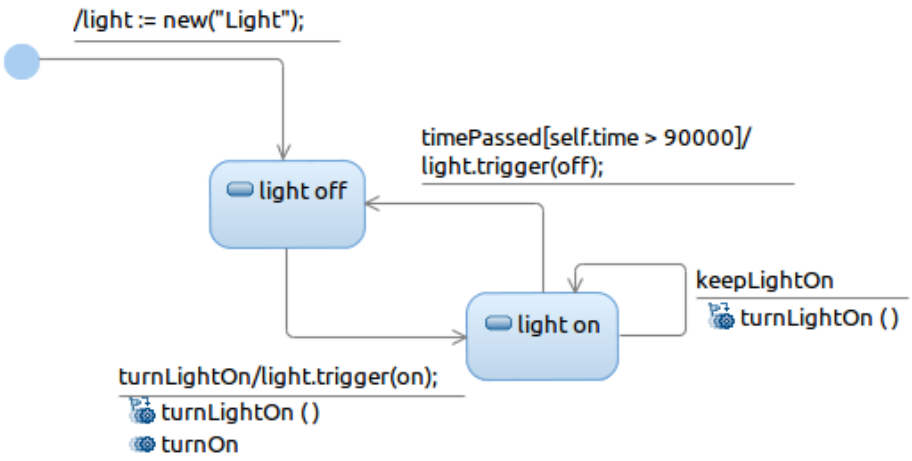

Figure 6.3: Light Controller state machine with SDSL statements

remember a set of member variables called attributes. Further information about the current state and references to other state machine instances can be stored in attributes. The *light* variable in the example is such an attribute. At the initial transition, a new light state machine is created and assigned to the attribute. Subsequently, it is used to send *on* and *off* messages to the light state machine.

SDSL is a dynamically-typed language. Therefore, all attributes have a type. The type is defined when the attribute is defined with one of these two possibilities:

- By defining the attribute in an SDSL statement programmatically: `var <type> <name>;`, optionally with an initial value.
- By defining an attribute in the state machine's class in the UML model. This feature was added when migrating SDS to the tool plugin. Since the state machine is modelled within a larger UML model, each state machine has a class whose behavior the state machine models. Hence, the state machine adopts all attributes of its class.

SDSL specifies its own set of types:

integer data type for representing integers

boolean data type for representing boolean values, i.e. the values `true` and `false`

double data type for representing floating-point numbers

string data type for representing character sequences or texts

set data type representing a collection of unique unordered values

bag data type representing a collection of unordered values; duplicates are allowed

sequence data type representing a collection of ordered values; duplicates are allowed

port data type representing the reference to another state machine

object unspecific data type

In order to be able to use these types in the UML model, a SDSL type library was implemented for Rational Software Architect. Any project importing this library is able to use the SDSL types. However, this limits the possibility to use already present models with other types. To cope with this limitation other types used are automatically translated to SDSL types for the simulation. The mapping of common UML types to SDSL types is based on the name of the types. If their names match, they are treated as the same type. Any other type is mapped to *object*.

6.3 Event Processing

The communication between state machines during the simulation is purely event-based. This means that instead of sending the message directly to the other state machine, an event containing the messages is sent to an event processor, which will dispatch the events accordingly. In addition to the communication between state machines, the communication of the user with the state machines as well as notifying events, like state changes, are also sent through this system. Therefore, the event system is the heart of the simulation and all relevant information passes through the event processor. The propagation of events works in two ways. Firstly, events

can be directed to one or more targets, like the communication between state machines. The targets are specified with the event and the event processor sends the events to these targets. Secondly, any object can subscribe itself to receive events according to the observer design pattern [16]. The event processor checks the list of subscribers when he handles an event and notifies them of the event. This properties makes it simple for the validation monitor to observe the communication in the simulator. All it has to do is to subscribe itself for communication events and pass incoming events to the validation module.

6.4 Validation module

6.4.1 Message Checking

One of our main requirements for the validation module was to be highly independent of the simulator. This independently would allow us to change the underlying simulator or even exchange it with an actual system execution. On the other side the simulator should not have any dependencies to the validation module at all. This goal is achieved by the event processing as described in the section above. The class *ConstraintManagerConnector* represents the monitor of the system. It implements the *SimulatorEventListener* and therefore receives all event sent through the system, including the communication events between the state machines. It then translates the messages observed into validation messages and wraps the state machines which sent and received the message. This reduces the dependency to a point, where only the connector one other class (the state machine wrapper) have a direct dependency to the state machine simulator. Hence, migrating the validation module for a different simulator or another communication source (e.g. an actual system execution) can be done by simply exchanging these two classes.

Figure 6.5 depicts the structure of the primary message processing classes. These classes roughly represent the components depicted in Figure 6.4 as described in Section 5.3. The

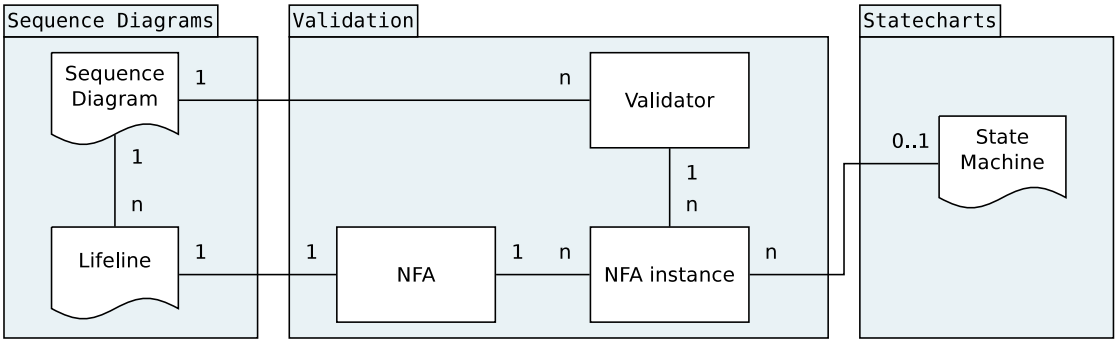

Figure 6.4: The checking components and their relationship

Validator component is represented by the class *ConstraintChecker*, while *LifelineAutomaton* and *AutomatonExecution* represent *NFA* and *NFA instance*, respectively. *ConstraintSupervisor* is responsible for managing all *ConstraintChecker* instances for one sequence diagram. Hence, it is also responsible for instantiating and terminating *ConstraintCheckers* and the binding of state machines to a lifeline. Nonetheless, each *ConstraintCheckers* holds its own set of bindings. The *ConstraintManager* is a singleton and maintains one *ConstraintSupervisor* instance for each sequence diagram in the model.

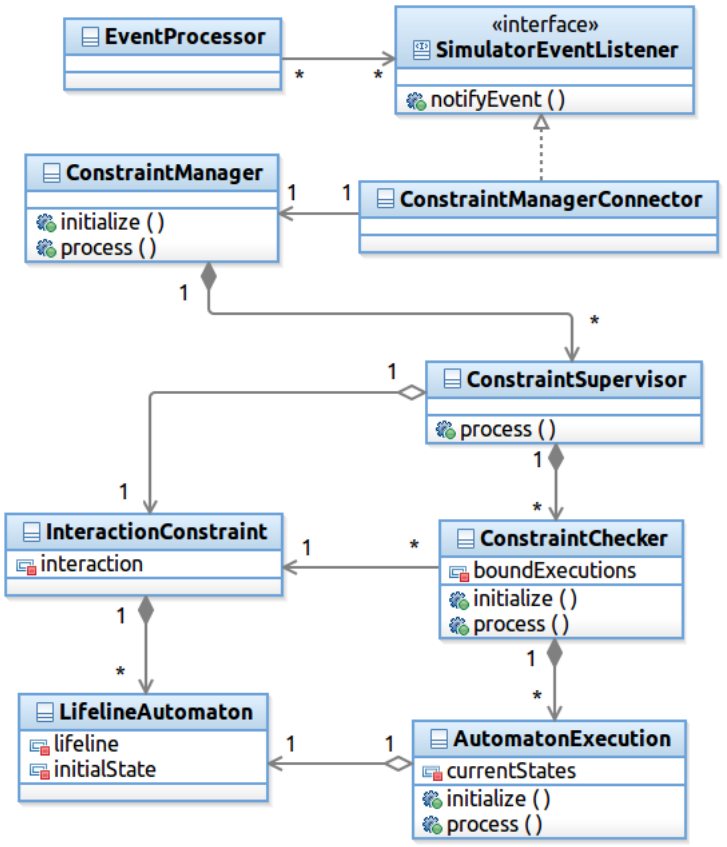

Figure 6.5: Classes responsible for the validation process

Whenever a new message is observed, the *ConstraintManagerConnector* hands the translated message to the *ConstraintManager*, which will lets every supervisor process the newly received message. The supervisor checks for concerned checkers and instantiates a new one if none can be found. The state machines are then bound to the matching lifelines at each concerned checker. In the next step the messages are processed by each concerned checker, which hand the message to the affected automaton executions. After the processing the results are checked. The whole process is described in more detail in Section 5.3 and depicted as sequence diagrams in Figures 6.6 and 6.7.

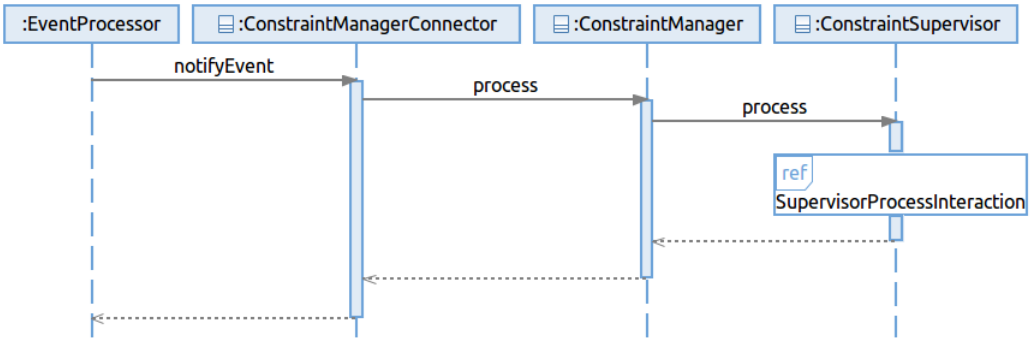

Figure 6.6: Processing of an observed message

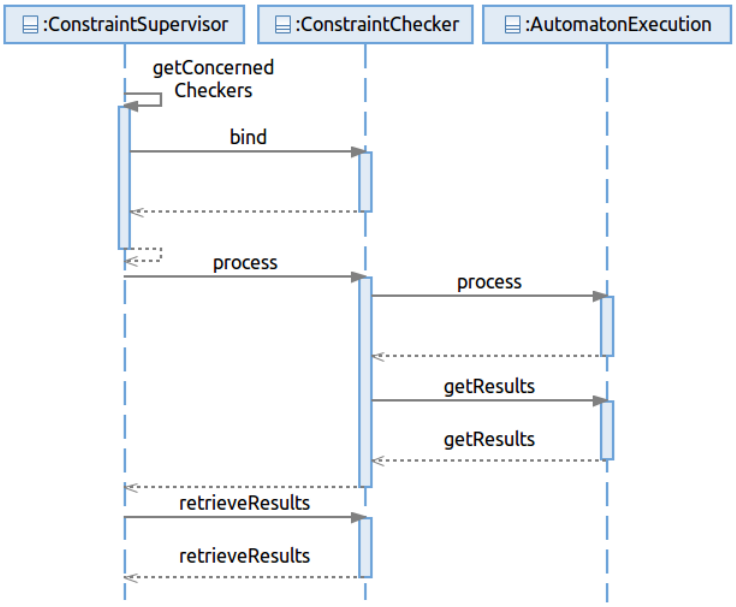

Figure 6.7: ProcessSupervisorInteraction as referenced in 6.6

6.4.2 Initialization and Compilation

Now we have seen a lot about how the processing of observed messages works. The other side of the approach is the compiling of sequence diagrams into NFAs. This process happens before the simulator is started by calling the *initialize* method on the *ConstraintManager*. Figure 6.8 shows this process. For each sequence diagram (i.e. interaction) in the model a *ConstraintSupervisor* instance is created. The instance contains the NFA which was compiled by the *ConstraintParser*. Note that parser is actually not the right term here since it does not parse any text but transforms one data structure into a different one. The *InteractionAstParser* creates the ASTs as described in Section 5.2. It separates the interaction into one AST for each lifeline. The *ConstraintParser* than transforms these ASTs into NFAs according to the approach described in Section 5.2.

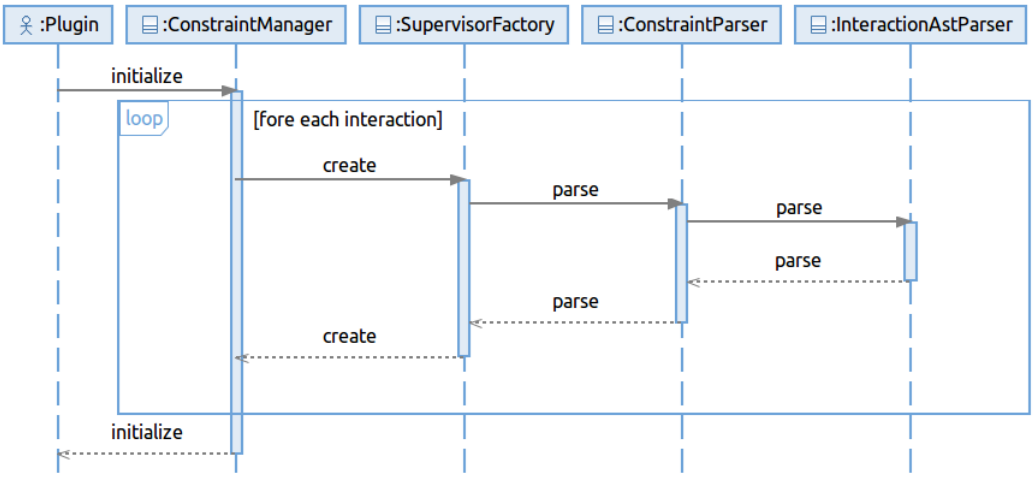

Figure 6.8: Initialization of the validation module

Unfortunately, Rational Software Architect is incapable of defining the lists of pertinent messages for consider and ignore combined fragments. Hence, we defined these messages as keywords for the combined fragments and extract them during the compilation process accordingly. This workaround can also be seen in the illustration of this thesis (see Figure 2.3).

6.4.3 Automata

The data structure resulting from the compiling primarily consists of the classes shown in Figure 6.9. The central class is the *AutomatonState* interface. As the name suggests, instances of this interface represent one state in the resulting NFA. Furthermore automaton state remember their incoming and outgoing transitions to other states. Thus, the *LifelineAutomaton* only needs to remember the initial state of the NFA. When executed the *AutomatonExecution* remembers its current states and some meta-information about the execution. This information is stored in the *AutomatonStatus* class. The *AutomatonState* interface contains one operation named *transform*. This operation performs the actual transition from one state into another one, although the way it does it and which information it alters on its way depends on the implementation. *Transform* takes one automaton status and transforms it into one or more other statuses. Most implementations of the interface look up their outgoing transitions and create a new automaton

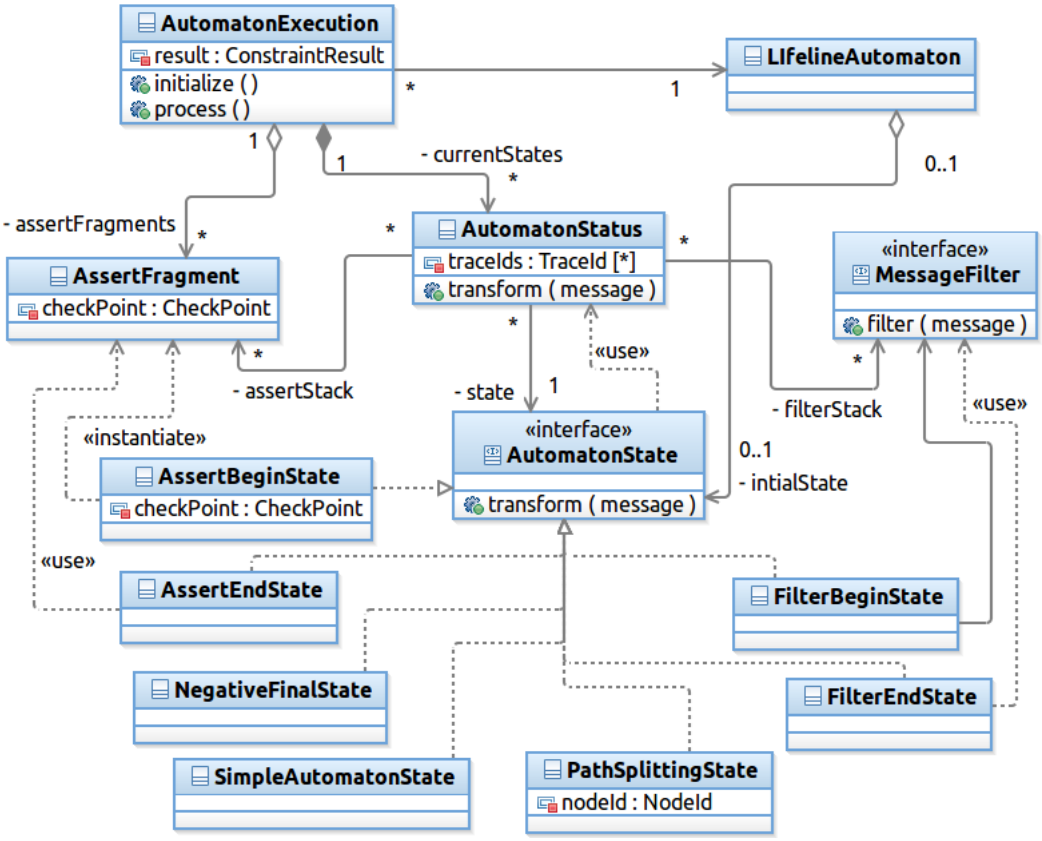

Figure 6.9: Classes building the NFA and its execution environment

status for each target state of the transitions. The *SimpleAutomatonState* does not alter any meta-information and therefore simply copies the original automaton status except for the state. The *PathSplittingState* is used for states with multiple outgoing transitions to contribute to the *TraceID*, which identifies the path taken through the NFA. Each target status receives a different TraceId, so we can later distinguish them. A *AssertBeginState* creates a new *AssertFragment* upon transition and pushes it onto the assert stack of the resulting automaton statuses. With the instantiation it also passes a *CheckPoint* instance to the assert fragment and registers the TraceIds with the check point. CheckPoints are shared among every by the assert fragment covered lifeline for each assert fragment. This means that one check point has a complete knowledge about which execution entered the assert fragment with which TraceId. Thus, we can identify if an execution with matching TraceId entered the assert fragment on every covered lifeline in case one execution violates an assert fragment (see Section 5.3.2). The other implementations of *AutomatonStatus* behave according to their descriptions in Chapter 5.

When the transition for the observed message is done, the ε-transitions are processed for each current state. Then the assert fragments are validated to ensure no assert fragment was violated before the results are checked. The whole transitioning process is depicted in Figure 6.10.

6.5 User Interface

A proper simulation and validation tool is only convenient with a decent user interface to manage it. Therefore our plugin contains several views and dialog to control the simulation and present the validation results.

The main view of the simulator is the "State Machine Instances" view shown in Figure 6.11. The view shows all state machine in the model. For each state machine, the user can create, rename and delete instances. An example is the *ControlUnitStateMachine* instance "CU" marked blue. The blue highlighting signifies that this instance is the currently selected instance for this state machine. The current states of the selected instances are highlighted in the editor during the simulation (see Figure 6.12).

The created instances are remembered between Rational Software Architect executions. This is done by storing the information about the instances in the plugin preferences. The resulting XML file is located at at `.metadata/.plugins/at.jku.sea.sds.simulator/instances.xml` in the workspace directory and looks like this example file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Instances>
  <Instance Name="CU"
    StateMachine="AutomaticLight::ControlUnit::ControlUnitStateMachine"/>
  <Instance ... />
</Instances>
```

Only state machines created by the user are persisted this way. Dynamic state machines – i.e. state machines created by other state machines during the simulation – are not persisted. They are not even kept between simulation executions but destroyed at the end of the simulation.

The view also provides buttons and context menu entries to start and stop the simulation. The start uses an Eclipse run configuration. This means that the simulation can also be started by creating a run configuration similar to starting a normal application (see Figure 6.13). When started from the "State Machine Instances" view, the plugin looks for an existing SDS simulator run configuration. If it finds one, it uses the present run configuration, otherwise it creates a new one. The run configuration provides one parameter which can be adjusted. The simulation

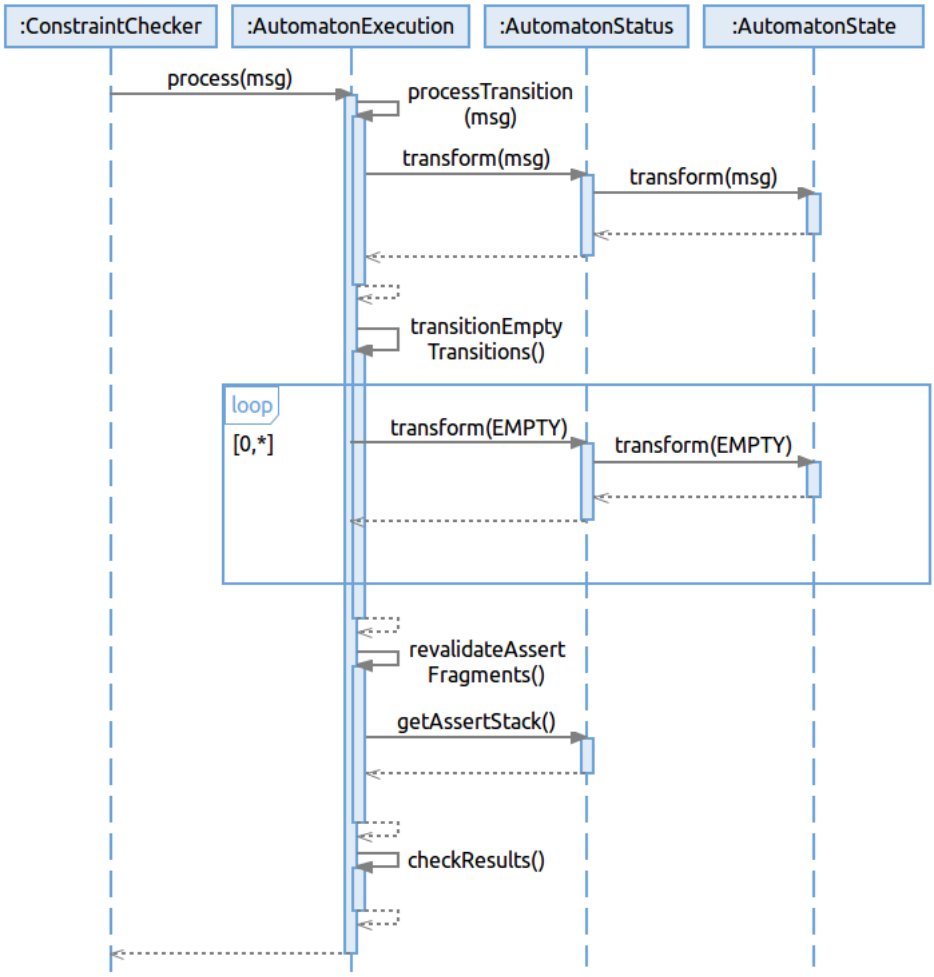

Figure 6.10: Processing of a message in the NFA

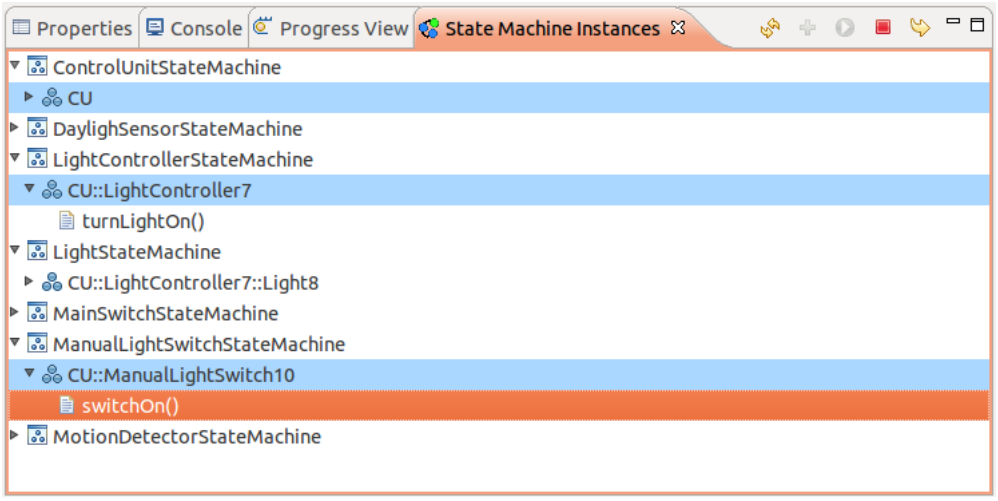

Figure 6.11: State Machine Instances View

rate specifies how fast the simulation is executed. What really happens under the hood is that the event processor waits the specified time between distributing the events. Increasing the simulation rate can help to see the state machines transitioning through intermediate states.

To communicate and trigger events on state machine instances, the user can send events to each instance. The view shows all trigger events as children of the instances. A user can select one of these events and send it to the state machine instance. For parameterized events the dialog shown in Figure 6.14 is shown, where the user can specify the parameter values.

As mentioned in Section 6.2 each state machine instance stores its own set of attributes. The values of these attributes can be inquired by opening the inspection dialog shown in Figure 6.15 from the instances view. It shows all attributes, their types and values.

Furthermore, the instances view provides the convenience functionality to open the editor when double-clicking on a state machine or instance.

The other important view of the plugin is the “Interaction Checking” view shown in Figure 6.16. This view represents the validation module. Hence, it shows new validation results when they are produced. A red result indicates an invalid execution, while a green one indicates a valid execution. Moreover, it shows which state machine was bound to which lifeline to produce the result. Apart from the results produced, the view also shows the current checkers and their binding for each sequence diagram. This can help understand why the validation module produced a certain result. However the checkers are not automatically refreshed but only when the user manually refreshes them or a new result was produced. Similar to the state machine instances view, this view opens the editor when an interaction is double-clicked.

The last view our plugin uses is the standard Eclipse console. Most developers know the console from showing the standard output of normal programs. We use the console to show more details about what happens within the system. An example output is shown in Figure 6.17.

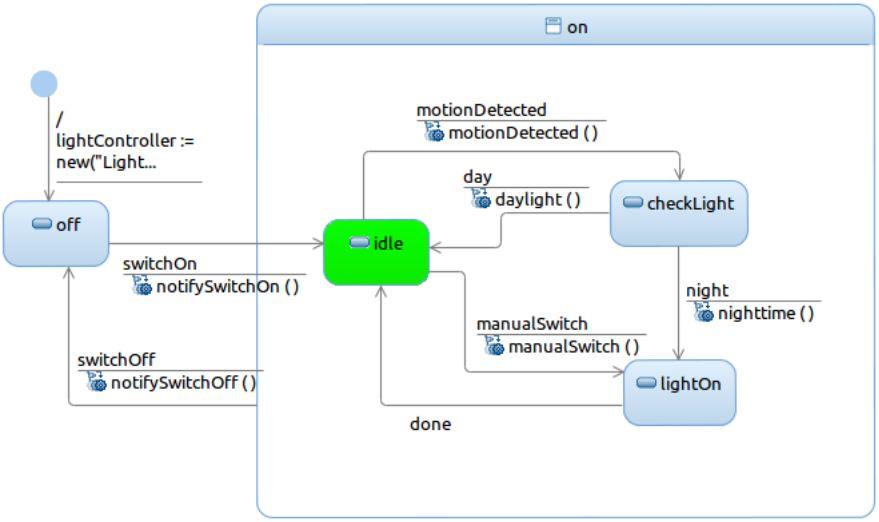

Figure 6.12: SDS highlights current states in state charts

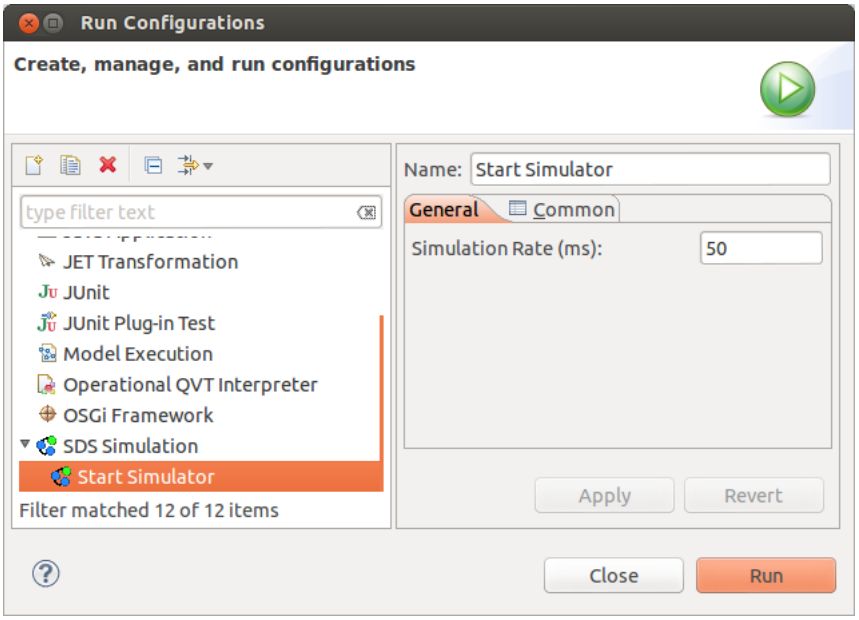

Figure 6.13: SDS run configuration

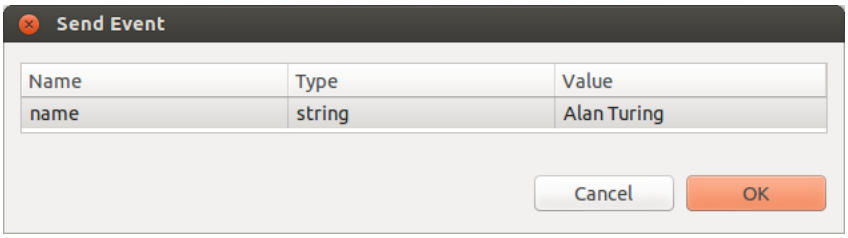

Figure 6.14: Send Event Dialog

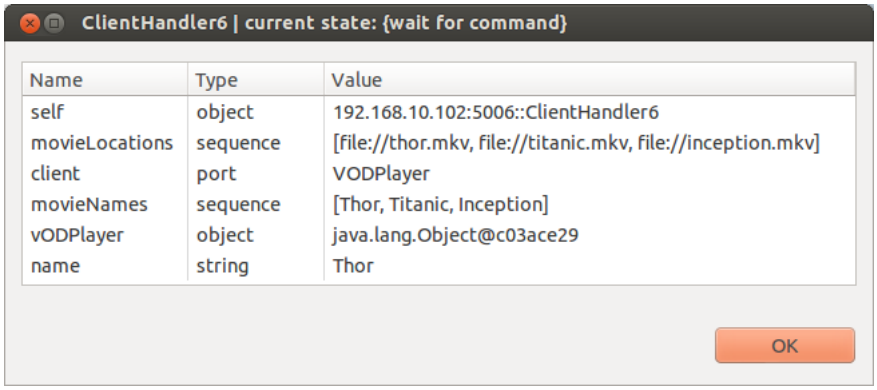

Figure 6.15: Inspection Dialog

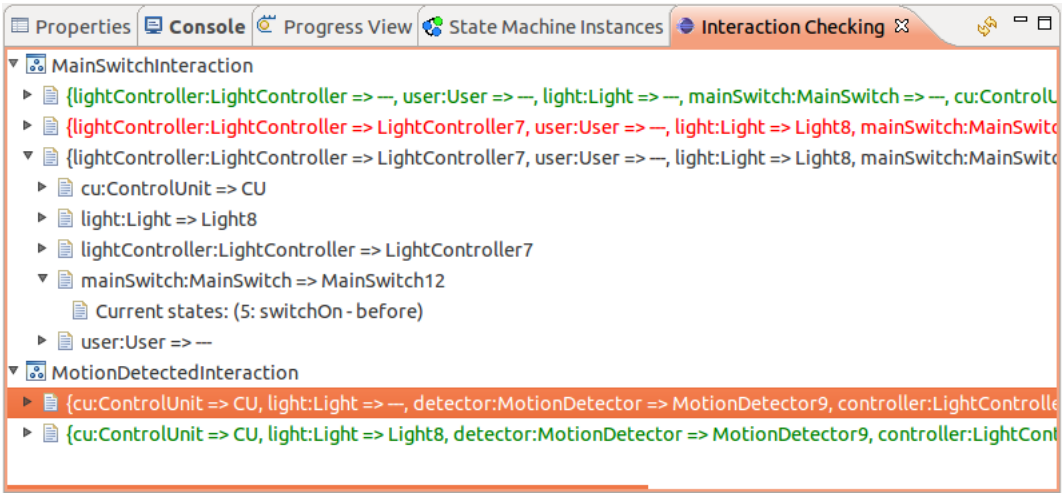

Figure 6.16: Validation Result View

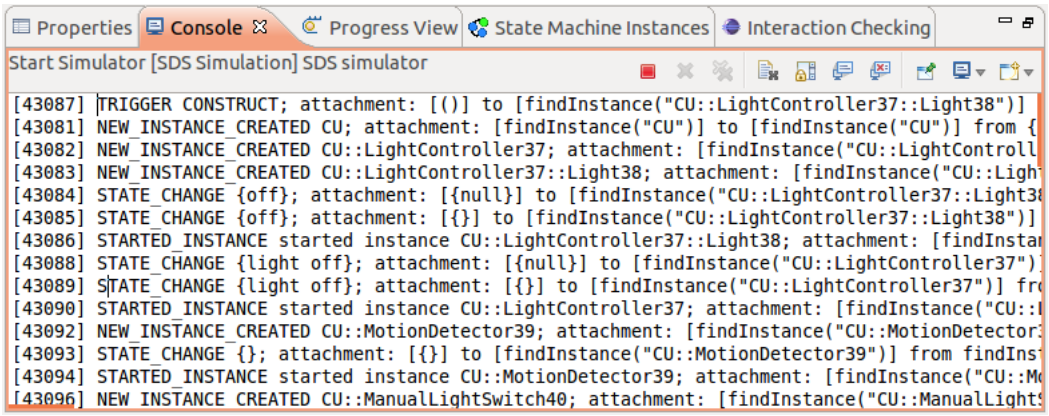

Figure 6.17: Console Output

Chapter 7

Evaluation

The evaluation of our approach was partly done theoretically, partly empirically. The following section describe this evaluation process and its results.

Note that the state machine simulation is not the focus of this work and therefore was not evaluated. Actually, the state machine simulation is irrelevant. For as long as there is a system that allows us to monitor events we can validate it. Indeed, this approach is not limited to state machine simulation. We could even use it to validate code (see Section 7.5.1).

7.1 Sequence Diagram Semantics

As we already discussed in previous chapters, sequence diagrams are inherently ambiguous. This is the reason why a lot of research is done concerning sequence diagram semantics (e.g. [4, 5, 6, 7]). Hence, we tried to provide semantics which are intuitive for the designer of the sequence diagrams and suit the creation of constraints, since these differ from ordinary scenarios. However, different domains require different semantics and the user's concept of what is intuitive might differ from ours. Our general approach is generic enough to allow semantics to be changed. Different sequence diagram semantics primarily affects automata generation. Although some case might not be easily adjustable, like using strict sequencing and therefore creating total order instead of partial order. The independence of the NFAs prevents total order.

7.2 Case Study

Even a well-designed approach is insufficient if it does not prove to be usable in real life situations. Therefore, our approach was evaluated on several models, taken from other projects, by using the reference implementation.

The first model is the automatic light example illustrated in this thesis in Chapter 2. Although it is a rather small model it uses many of our approach's features.

Still, one could argue that this model was created by ourselves and therefore designed to work well with our approach. Hence, we used models from other sources as well.

The first external model was taken from a different IBM development tool. IBM's Rational Rhapsody is a modeling tool for UML [24]. We did not use Rational Rhapsody itself, but it comes with several example UML models. The example we used for our evaluation was the dishwasher example. It models a dishwasher with tank, heater, and jet as components. Unfortunately, the model only contains class and state machine diagrams. Therefore, we had to design appropriate

sequence diagrams ourselves. Additionally, the existing state machines were enhanced with SDSL statements to be executable with our simulator.

The next model we used had the opposite problem. This model was taken from [2]. It is an example of a control system for ovens. Harel and Marelly used this example to illustrate their play-engine. They synthesize an executable model from Live Sequence Charts (LSC). LSCs are quite similar to UML Sequence Diagrams. Especially version 2 of UML was influenced by LSCs. Therefore we were able to translate LSC to UML Sequence Diagrams. However, since the play-engine only needs LSCs for their model, we had to create the executable state machines ourselves.

The last model of our case study is a video on demand system. It was originally created for the SDS in [19]. In contrast to the other models, this model is a bit larger. This model contains both sequence diagrams and state machines which are already executable by SDS. Though the state machines were made with SDSL in mind, the model was initially not designed for our validation. Therefore, it proves a well-suited case.

Table 7.1 summarizes the models and the size of each model.

	State Machines	**Sequence Diagrams**	**Lifelines per SD**	**Ref.**
Automatic Light	4	2	4.5	
Dishwascher	4	1	5	[24]
Bakery	6	3	5.67	[2]
Video on Demand	6	2	3.5	[19]

Table 7.1: Results of the performance measurements

We simulated the state machines with the SDS simulator presented in Chapter 6 and executed the scenarios depicted in the sequence diagrams among others. Our tool proved sufficient for all models from the case study. It was able to recognize all sequence diagrams, when the state machines behaved accordingly.

7.3 Assessing Correctness

Though the case study showed that our approach works correctly for the presented models, several other features and corner cases are not tested with those models. Therefore, we will present a more theoretical assessment of the approaches correctness.

7.3.1 NFA Construction

Firstly, let us analyze the NFA construction algorithm. AST creation is straightforward and just transforms one data structure into another one. The only major difference is the fact that our ASTs are binary. However, this does not alter the result of the algorithm, since sequences and alternative combined fragments are associative.

For the Thompson construction to be applicable we assume a large similarity between regular expressions and sequence diagrams. The original Thompson construction was proven to be correct by [18]. Therefore the construction for message occurrences, sequences, alternative, and loop combined fragments are correct as well, because they are created as in the original Thompson algorithm. Furthermore, all additionally added construction rules mainly maintain the following properties of the original algorithm [18]:

- $N(r)$ has one start state and one accepting state. The accepting state has no outgoing transitions, and the start state has no incoming transitions.

- Each state of $N(r)$ other than the accepting state has either one outgoing transition on a symbol in Σ or two outgoing transitions, both on ε.

However, two assert fragments violate these properties slightly. The first one is the break fragment. Since a break introduces a break state, it creates another state without any outgoing transitions. Nevertheless, this property only holds as long as the break state is not encapsulated by another combined fragments, which should always be the case.

The other violation is produced by negative combined fragments. Again, it introduces another state without outgoing transitions. In spite of this, the negative final state has unique properties and differ significantly from normal final states.

7.3.2 Live Event Checking

Next, we will assess the correctness of the live event checking. The algorithm for execution NFAs is a simple implementation similar to the one proposed in [25]. Therefore, we will not analyze this algorithm in more detail.

The more interesting part of the checking process is the binding and instantation. Current binding and instantiation procedure does not always find the right solution. It is a heuristic with some issues like what if instantiation is done too early and the actual execution of the sequence diagram is then missed? What if a lifeline should have been bound with another state machine which will send its message later? There is still room for improvement, however the case study showed, it works well in most cases. Especially when the binding of state machines is unambiguous, which is the case for most real life sequence diagrams.

To furthermore assure the correctness of our approach, we created 186 automated tests using 60 different sequence diagrams. The test cases include simple and complex sequence diagrams as well as interesting corner cases, like ambiguity. The corner cases are difficult to create in real life examples like the models used in the case study, therefore they give further confidence in the correctness of our approach.

7.4 Assessing Scalability & Performance

The second important aspect investigate in our evaluation is scalability and performance. The main questions addressed in this section are whether the approach is fast enough for real-time evaluation and if it scales on even large models.

7.4.1 NFA Construction

Firstly, we will evaluate the construction algorithm again. This algorithm is executed once before starting the validation. Thus, it comes with a one time cost.

During AST creation each interaction fragment is visited once, though one fragment can cover multiple lifelines. For each covered lifeline AST nodes have to be created. Hence, it has a time complexity of $O(f \times lps)$, with f being the number of interaction fragments (includes message occurrences and combined fragments) and lps being the number of lifelines per sequence diagram (which is considered small even for large diagrams).

The NFA construction is based on the McNaughton-Yamada-Thompson algorithm described in [18] and has a time complexity of $O(f)$ (see [26]). In the original algorithm, f is the number of operators and operands in a regular expression. In our approach, these operators and operands are interaction fragments. Nevertheless, this time complexity is not immediately clear for break assert fragments, since an enclosing combined fragment might need to attach multiple break

states in one step. However, each break combined fragment only produces one break state and each break state is attached to only one other state (the one of the enclosing fragment). So each step produces and consumes at most one break state, which leads to the conclusion that this does not violate the linear time complexity of the original Thompson construciton.

To summarize, the overall complexity for NFA compilation per sequence diagram is as follows (definitions of f and lps as above):

$$createAST + lps \times createNFA = compile$$
$$O(lps \times f) + lps \times O(f) = O(lps \times f)$$

For the subsequent algorithms we also need to know how large the resulting NFAs are. [18] points out an important property of the algorithm to identify its space complexity:

> $N(r)$ has at most twice as many states as there are operators and operands in r. This bound follows from the fact that each step of the algorithm creates at most two new states.

In our enhanced algorithm, each step produces at most three new state. Thus, $N(r)$ has at most thrice as many states as there are interaction fragments in r. This means that the Thompson construction has a linear space complexity.

7.4.2 Live Event Checking

Next, let us analyze the live checking scalability. The live checking is done whenever a message is received, so the cost must be low to be scalable for larger models. Like at the construction algorithm, we will identify the cost per sequence diagram. First, we will look into the binding algorithm. The binding algorithm has no means to identify the right choice when a binding is ambiguous. Therefore it simply tries every possible combination. This leads to an exponential time complexity $O(2^l ps)$ with lps being the number of lifelines per sequence diagram. Nevertheless, this exponential complexity is only reached if all lifelines in a sequence diagram have the same type, which is a very rare case in real life. Furthermore, the number of lifelines is small, which makes the exponential time complexity in fact practically irrelevant. In conclusion, if all lifelines have different types and all state machines can be assigned to only one lifeline – which is a quite common case – the complexity is reduced to a linear one, since in this case all the algorithm has to do is find the fitting lifelines for the two state machines (sender and receiver). The measurements in Section 7.4.3 support this hypothesis.

In addition to binding state machines to unbound validators, the algorithm also needs to identify the concerned validators to let each of them process the message. The number of concerned validators depends on result of the binding algorithm. Thus, in the worst case this number will also increase exponential with the number of lifelines. However, our measurements showed that the system maintains 1.51 validators per sequence diagram on average. Therefore, the number of concerned validators is small as well.

The processing of one message for an NFA has a time complexity of $O(f)$ [26]. This complexity results from the fact, that in the worst case, each state of the NFA is in the set of current states. Furthermore, we process each message on two NFAs – the sender and the receiver. Therefore the time complexity for processing one message on a validator is as follows:

$$2 \times O(f) = O(f)$$

However, measurements with the models from the case study showed that one NFA on average possessed 1.23 current states during processing. Hence, in reality the cost is much lower than linear.

In conclusion, the scalability for the checking algorithm is sufficient. Indeed, it is linear with the number of lifelines for most cases. Furthermore, not all validators are called at every message, since they are only instantiated on demand.

7.4.3 Measurements

Table 7.2 shows execution times for different algorithms or part of algorithms. The Table consists of the following values

maximum The maximum time any execution took

p99 99^{th} percentile of the execution times, i.e. 99% of all executions are faster than this value

mean The mean execution time

n The number of sample executions measured

The measurements were performed on a computer with an Intel Core 2 Duo P8700 CPU with 2 x 2.53 GHz using the reference implementation described in Chapter 6. Since the implementation is coded in Java, the maximum values are typically reached at the first execution, when the overhead of loading the classes significantly slows the execution. Hence, the more interesting factor here is the 99^{th} percentile, which in most cases is considerably lower than the maximum value. Furthermore, this circumstance leads to a decline of the mean value with increasing samples. Nevertheless, bare in mind that some samples might still reach high values when Java's Garbage Collector or JIT interferes the normal execution.

With less than 1 ms time cost per sequence diagram, the process is fast enough for validation during runtime. Though the measurements were performed with the model from the case study, the execution times are considerably low and thus we believe that it will perform well even for large models.

Combined	**maximum**	**p99**	**mean**	**n**
NFA Construction	10.52 ms	10.52 ms	1.60 ms	74
Overall Live Checking	27.29 ms	19.50 ms	3.70 ms	224
Checking per SD	16.50 ms	5.01 ms	0.50 ms	653
Binding	2.28 ms	1.13 ms	0.11 ms	482
Validator Process	4.62 ms	1.43 ms	0.19 ms	725
NFA Transitioning	3.70 ms	0.35 ms	0.05 ms	1450

Table 7.2: Results of the performance measurements with a combination of all models

Automatic Light	**maximum**	**p99**	**mean**	**n**
NFA Construction	55.39 ms	55.39 ms	6.04 ms	12
Overall Live Checking	50.29 ms	50.29 ms	4.34 ms	83
Checking per SD	8.50 ms	3.91 ms	0.33 ms	332
Binding	7.26 ms	0.39 ms	0.12 ms	347
Validator Process	1.87 ms	1.56 ms	0.15 ms	124
NFA Transitioning	1.61 ms	0.28 ms	0.04 ms	248

Table 7.3: Results of the performance measurements with the automatic light model

Dishwasher	**maximum**	**p99**	**mean**	**n**
NFA Construction	7.32 ms	7.32 ms	1.50 ms	6
Overall Live Checking	17.47 ms	17.27 ms	1.99 ms	105
Checking per SD	12.51 ms	8.18 ms	0.37 ms	210
Binding	3.57 ms	1.25 ms	0.08 ms	219
Validator Process	3.25 ms	3.25 ms	0.17 ms	99
NFA Transitioning	0.17 ms	0.16 ms	0.04 ms	198

Table 7.4: Results of the performance measurements with the dishwasher model

Bakery	**maximum**	**p99**	**mean**	**n**
NFA Construction	32.81 ms	32.81 ms	4.61 ms	9
Overall Live Checking	43.57 ms	41.43 ms	5.37 ms	118
Checking per SD	25.62 ms	18.74 ms	1.19 ms	354
Binding	21.81 ms	5.16 ms	0.35 ms	369
Validator Process	24.63 ms	0.40 ms	0.12 ms	910
NFA Transitioning	1.65 ms	0.17 ms	0.02 ms	1820

Table 7.5: Results of the performance measurements with the bakery model

Video on Demand	**maximum**	**p99**	**mean**	**n**
NFA Construction	8.65 ms	8.65 ms	2.33 ms	9
Overall Live Checking	33.43 ms	31.86 ms	3.12 ms	113
Checking per SD	29.00 ms	12.65 ms	0.48 ms	334
Binding	1.52 ms	0.44 ms	0.07 ms	346
Validator Process	24.26 ms	13.88 ms	0.34 ms	235
NFA Transitioning	12.37 ms	0.26 ms	0.07 ms	470

Table 7.6: Results of the performance measurements with the video on demand model

7.5 Discussion

Though the approach was tested with several model, these models were relatively small and therefore only provide verification for the features used by those models. The reason for this insufficient tests were the lack of larger projects and the fact that making state charts simulatable is a considerable amount of work.

Another issue arises due to the number of features implemented in our approach. The interaction of features is difficult and expensive to test, since the number of combinations, like nesting combined fragments, is vast.

Furthermore, we did not perform any usability tests. The exploration of the wildcard feature showed, that it is a powerful but dangerous feature which is difficult to understand. Moreover, how validation came up with results is sometimes hard to comprehend. Thus, we added a lot of debugging and inspection features to our implementations to help the designer. Nevertheless, these features still require a solid knowledge about the validation process. Hence, there is still room for improvement like highlighting current states in sequence diagrams.

7.5.1 Validating Code

As we already mentioned in previous chapters, our work is not limited to state machine simulation. Although the approach was originally designed to work with state machine simulation it is generic enough to handle other execution environments as well. Therefore it should even be possible to validate executions of written software, due to the slim connection to the simulation. All we need is something which provides us with a stream of messages containing the information about who sent who a message with which name.

Chapter 8

Related Work

In the past, there were several approaches on state machine validation. One such approach is described in [27]. In their approach Lilius and Paltor translate statecharts into PROMELA code and use SPIN to model check the resulting code. However, they only checks for errors like deadlocks, lifelocks, and buffer overflows. Though they present the resulting counter examples as sequence diagrams, they do not incorporate sequence diagrams designed by the user for the same model. Therefore, this approach does not meet our goal of connecting sequence diagrams and statecharts.

Other approaches try to integrate sequence diagrams by using them for synthesis. Especially the field of model-driven engineering uses synthesis to generate code from sequence diagrams.

For this purpose, many researches proposed formal semantics for sequence diagrams. Notably here is [4], which gives a good overview about different semantics used in the literature. Furthermore, it discussed numerous issues concerning sequence diagram semantics (some of them were mentioned in our work).

In contrast to code synthesis, other approaches synthesize statecharts out of sequence diagrams, which is more closely related to our goal. [28] and [8] compare several such approaches. However, the resulting statecharts are hard to read and describe the system's behavior incompletely, due to the scenario nature of sequence diagrams. Therefore, [9] proposed a semi-automatic generation of statecharts. The need for a human input in the synthesis of statecharts out of sequence diagrams prove that sequence diagrams were not designed for creating a whole model.

A different approach is followed by Harel in [2]. He omits statecharts at all and creates a running system just with Live Sequence Charts – which are closely related to sequence diagrams. For the execution an int internal model of the system is created, which is not visible to the designer. Therefore, this approach is only usable for visual prototyping. Furthermore, the incompleteness of sequence diagrams (and LSCs) leads to problems here as well, since they do not cover all corner cases.

However, as we already discussed in Chapter 4 sequence diagrams are well suited for validation. This leads us to the approaches which verify sequence diagrams or use them for verification. which check against SDs against one another. [5, 29, 30, 31] describe approaches which are used for refinement in incremental software development. Starting with a rough high-level specifications of interactions, with each iteration the sequence diagrams are refined to contain more details. These approaches check if such a more detailed, newer version still complies to the original high-level version or in the case of Lu and Kim [31], if they represent a certain design pattern or aspect. Grosu et al. [5] create hierarchic non-deterministic büchi

automata. They use a single automaton for one sequence diagram and model check liveness and safety properties with these. The problem with these approaches include the need for the user to explicitly state this refinement relation between the sequence diagrams. Therefore, they are not directly usable for existing models.

[32] and [33] combine two other diagram types: State machine diagrams and collaboration diagrams. Their tool HUGO/RT translates state machines into PROMELA code and collaboration diagrams into Büchi automata. It then uses the resulting artifacts to model check the diagrams using SPIN.

Though this work does not use sequence diagrams, it uses the closely related collaboration diagrams. Their follow-up work [10] integrates sequence diagrams into HUGO/RT. Thus, this work follows the same goal as our approach, namely to check state machines with sequence diagrams. Nevertheless Knapp and Wuttke's approach differs from ours. They use a single automaton for one sequence diagram. This decision leads to the necessity to simplify handling of loops and alternatives, due to the fact that the language of sequence diagrams is neither regular nor context-free [12, 13]. Furthermore, Knapp and Wuttke do not consider inconclusive traces. Each interaction is either satisfiable or not. Similar to [32], the resulting automata are used with SPIN for model checking.

Another work about consistency checking between sequence diagrams and statecharts is described in [11]. Graaf and van Deursen synthesize sequence diagrams to statecharts and then compare them to the user-designed statecharts. They interprets all sequence diagrams as universal, i.e. if the first message and state invariant of a sequence diagram matches, the whole sequence diagram must match. This behavior is quite strict and therefore in our opinion only usable for designing sequence diagrams with this rule in mind. Additionally sequence diagrams need further "normalization" in which they are adjusted to work properly with the tool. Hence, there is no support for sequence diagrams of existing projects. Furthermore, this normalization introduces tight coupling between sequence diagrams and state machines. Lastly, the tool does not feature the ability to automatically compare the designed and generated statecharts. The user has to manually compare the state machines.

The discussed approaches use static model checking. However, model checking has several issues as already discussed in Chapter 3. We do not seek to replace model checking with our approach, because model checking is well suited for finding invalid behavior. We rather see our approach as an additional tool for verification. It is more capable of debugging and exploration if something happens at the right moment.

A rather similar approach was chosen by Gan et al. in [34]. They chose to validate the interactions of web services using UML 2 sequence diagrams during run-time. Their motivation for choosing run-time validation was the dynamic nature of web services. As a consequence of the similarity of our approaches, the architectures are quite similar as well. They first compile sequence diagrams to NFAs, monitor the system and use those NFAs (translated to deterministic finite automata) for validation. However, they only use a subset of the sequence diagram specification they think is fitting for their purpose and do not exploit the full potential of sequence diagrams. This simplification lets them use one NFA per sequence diagram. Like other approaches they unwind the partial order and enumerate every possible order of events. This leads to a significantly higher number of state than our approach. Lastly, Gan et al. are currently not able to handle multiple services of the same type. Therefore, they avoid the binding issue our work explicitly addresses. In general, all analyzed approaches do not address this issue either. This might due to the fact, that model checking is not well suited to resolve these issues. Nevertheless, we think that more research must be done to solve this nontrivial matter.

Chapter 9

Conclusion

UML specifies sequence diagram to show the interaction between components of the system for certain scenarios. They are used to define intended and forbidden behavior. Thereby, they are easy to read, even for non-professionals, so they can be used to express the requirements of the system and create a common understanding of the intended behavior.

State machine diagrams, on the other hand, are created by specialist to precisely specify the behavior of one such component. Therefore, they can become quite complex and their connection to certain interaction scenarios is not immediately clear.

The goal of this work was to combine these two views to validate if the designed model of the system (i.e. the state machines) comply to the requirements stated as sequence diagrams. This goal was achieved by simulating the state machines, monitoring their communication and validating this communication using the designed sequence diagrams. For the validation, we translate the sequence diagrams into nondeterministic finite automata, one automaton per lifeline. The translation uses an adapted McNaughton-Yamada-Thompson algorithm. Thereby, many features of UML 2 sequence diagrams were implemented to use for the validation. State machines are assigned to instances of these NFAs during runtime. Whenever a message is sent between state machines, the NFAs to which the participating state machines are assigned process this message. If the NFAs land in a final or illegal state, the interaction is considered valid or invalid.

Even though many features of sequence diagrams were implemented in our approach, several are still missing. One of these missing features are state invariants. It seems obvious to implement such a feature as well, since we are dealing with state machines. However, we chose not to implement state invariants, because it would violate the general idea, that the validation just observes communication. Since we do want to keep the coupling between the simulation and validation as weak as possible, we wanted to avoid asking objects, which state they are in. Another possibility to solve this issue, would be to observe and remember state changes as well.

Furthermore, strict sequencing and parallel combined fragments were not incorporated. Our approach is designed to work well with weak sequencing and the consequential partial order of events. This decision comes to the expense of being able to easily implement strict sequencing and parallel execution. One possible solution would be to introduce some kind of synchronization between NFAs, like a rendezvous state.

Apart from missing features, the process itself has some issues as well. One notable issue is the current binding process, as was already discussed in Chapter 7. A possible solution would be manual binding, i.e. the user tells the model which lifeline represents which state machine instance. A similar solution was also proposed in [34]. However, this would involve the user to

explicitly specify this relation and therefore increase coupling between the diagrams.

Another feature with room for improvement is the wildcard lifeline feature. It is currently not working as initially intended, since it cannot be used to link or synchronize parts of a diagram.

The main issue concerning the implementation in our work is its relatively poor usability. It is sometimes hard to comprehend how the validation came up with a certain result. Possible improvements involve a better debugging support, like showing the current states of NFAs on their lifelines in the diagram.

One of our main ambitions was to be able to work with standard sequence diagrams, so the user does not need to amend the diagrams for our approach. However, during our work we faced difficulties on designing constraints with standard UML sequence diagrams. Designer have to make a trade-off between accuracy and stability of the constraints – i.e. do changes to the model break the validation? UML 2 sequence diagrams are not suited for constraints. Other works, like [5, 7, 29, 34], address this issue extensively. They realized that sequence diagram lack a notion of mandatory interaction, i.e. interaction which must be executed. In our case we distinguish two kinds of interactions: scenarios and constraints. Scenarios are executed at some point in the system, but only show a fraction of the communication going on between certain components. On the other hand, constraints should always hold and therefore should always be validated. Moreover, constraint interactions should be instantiated just once for a set of state machines, while scenario interaction might be instantiated anytime and even multiple times, when the scenario is executed multiple times. Currently, our approach does not distinguish these interactions in such a way, but treats every interaction as a scenario. This behavior leads to difficulties when designing constraints. A possible solution for this issue would be a concept of global interactions. By introducing a *global* stereotype for interactions, our tool would be able to distinguish between global interactions, which are instantiated just once, and normal interaction, which are instantiated any time possible. However such a solution is up to future work.

Nevertheless, in our opinion this work provides a valuable tool for exploring the behavior of a designed model and checking if it meets the requirements.

Bibliography

[1] Object Management Group. OMG Unified Modeling Language, Superstructure, Version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, 2011. accessed 2014-05-07.

[2] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, New York, NJ, USA, 2003.

[3] ITU-T. Message sequence chart (MSC). Standard, ITU-T, 2011.

[4] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 sequence diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, 2011.

[5] Radu Grosu and Scott A Smolka. Safety-liveness semantics for UML 2.0 sequence diagrams. In *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*, pages 6–14. IEEE, 2005.

[6] Harald Storrle. Semantics of interactions in UML 2.0. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, pages 129–136. IEEE, 2003.

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

[8] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12. ACM, 2006.

[9] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 314–323. IEEE, 2000.

[10] Alexander Knapp and Jochen Wuttke. Model checking of uml 2.0 interactions. In *Models in Software Engineering*, pages 42–51. Springer, 2007.

[11] Bas Graaf and Arie van Deursen. Model-driven consistency checking of behavioural specifications. In *Model-Based Methodologies for Pervasive and Embedded Software, 2007. MOMPES'07. Fourth International Workshop on*, pages 115–126. IEEE, 2007.

[12] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *CONCUR'99 Concurrency Theory*, pages 114–129. Springer, 1999.

[13] Anca Muscholl, Doron Peled, and Zhendong Su. Deciding properties for message sequence charts. In *Foundations of Software Science and Computation Structures*, pages 226–242. Springer, 1998.

[14] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Model Checking Software*, pages 80–102. Springer, 2001.

[15] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276. Springer, 1999.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*, section Observer, pages 293–303. Addision-Wesley, 1995.

[17] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, section Nondeterministic Finite Automata, pages 147–148. Addison-Wesley, Boston, MA, USA, 2nd edition, 2006.

[18] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, section Construction of an NFA from a Regular Expression, pages 159–163. Addison-Wesley, Boston, MA, USA, 2nd edition, 2006.

[19] Alexander Egyed and Dave Wile. Statechart simulator for modeling architectural dynamics. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 87–96. IEEE, 2001.

[20] IBM. Rational Software Architect. `http://www-03.ibm.com/software/products/en/ratisoftarch`. accessed 2014-06-23.

[21] Eclipse IDE. `http://www.eclipse.org/ide/`. accessed 2014-06-23.

[22] Eclipse UML2. `http://www.eclipse.org/modeling/mdt/?project=uml2`. accessed 2014-06-23.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*, section Adapter, pages 139–150. Addision-Wesley, 1995.

[24] IBM. Rational Rhapsody. `http://www-03.ibm.com/software/products/en/ratirhapfami`. accessed 2014-07-23.

[25] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, section Simulation of an NFA, pages 156–159. Addison-Wesley, Boston, MA, USA, 2nd edition, 2006.

[26] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, section Efficiency of String-Processing Algorithms, pages 163–166. Addison-Wesley, Boston, MA, USA, 2nd edition, 2006.

[27] Johan Lilius and I Porres Paltor. vuml: A tool for verifying uml models. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 255–258. IEEE, 1999.

[28] Daniel Amyot and Armin Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):61–94, 2003.

[29] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Software & Systems Modeling*, 4(4):355–357, 2005.

[30] Mass Soldal Lund. *Operational analysis of sequence diagram specifications.* PhD thesis, University of Oslo, 2007.

[31] Lunjin Lu and Dae-Kyoo Kim. Required behavior of sequence diagrams: Semantics and refinement. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 127–136. IEEE, 2011.

[32] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357–369, 2001.

[33] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking timed UML state machines and collaborations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 395–414. Springer, 2002.

[34] Yuan Gan, Marsha Chechik, Shiva Nejati, Jon Bennett, Bill O'Farrell, and Julie Waterhouse. Runtime monitoring of web service conversations. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 42–57. IBM Corp., 2007.

Curriculum Vitae

Philipp Mitterer, BSc.

Date of Birth: 1988-05-03
Residence: Tyrol, Austria
E-Mail: ph.mitterer@gmail.com

Education

HTL Saalfelden *2002–2007*
College of Electrical Engineering Specialising in Information Technology

Johannes Kepler University Linz *2008–2012*
Bachelor's Degree in Computer Science

Johannes Kepler University Linz *since 2012*
Master's Degree in Software Engineering

Oxford Brookes University *2012*
Study Abroad in the Field of Computer Science

Professional Experience

Various Smaller Jobs and Internships *2004–2011*
Mostly IT jobs at different companies including Egger Fritz GmbH & Co OG, Hutchison Drei Austria GmbH, and RHI AG.

eMundo GmbH *since 2011*
Software Engineer

Sworn Declaration

I hereby declare under oath that the submitted Master's thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited. The submitted document here present is identical to the electronically submitted text document.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Date

Signature