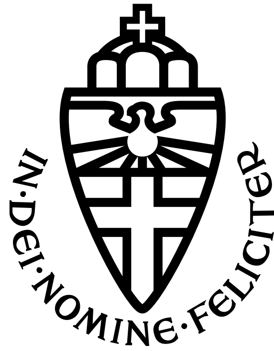
Radboud University Nijmegen

Faculty of Science



Automated Testing: from Sequence Diagrams to Model-Based Testing

Master's Thesis

Author:
Bram Petersen
s4786025
brampetersen@gmail.com

Internal supervisor:
Prof. dr. Frits W. Vaandrager
F.Vaandrager@cs.ru.nl

External supervisor:
Ir. Wietse Poiesz
wietse.poiesz@inspiro.nl

Second reader:
Dr. ir. G.J. Tretmans
tretmans@cs.ru.nl

Programme:
Computing Science (Software Science)

February 12, 2020

Abstract

In this thesis, we see that the increasing complexity and scope of testing critical (embedded) software systems, as well as relying on error-prone test and input selection by testers, calls for a new way of testing, namely *model-based testing*. This technique is slowly gaining ground, mostly in larger companies and longer running projects, but is not yet widely adopted. In this thesis, we argue that this problem exists due to the time and effort that has to be invested in creating and maintaining the models suitable for model-based testing. This is why this thesis proposes a solution in the form of a tool, which can generate models suitable for model-based testing from Unified Modelling Language (UML) diagrams to make model-based testing more accessible. UML is often already used in projects to document what has to be implemented, improve communication, reduce the number of software defects, and improve the quality of the software. In this thesis, we researched the syntax and semantics of sequence diagrams, as well as the transformation process from these diagrams to labelled or symbolic transition system to create a tool capable of transforming UML diagrams to TorXakis input models. This tool was then used in a case study, to generate models from sequence and class diagrams from which we were able to test an embedded system. Future work should research the possibility to generate models for a real-time model-based testing tool to be able to test (embedded) software systems subject to timing constraints below 100 milliseconds.

Acknowledgements

During my thesis and masters's, I have received a lot of support and guidance from others, for which I wish to thank them. I would like to express my very great appreciation to my supervisor prof. dr. Frits W. Vaandrager for his patience, time, guidance, advice and constructive feedback, to make the most out of my thesis. Second reader dr. ir. Jan Tretmans for his willingness to provide feedback on short notice, enabling me to finish my thesis in time.

I would also like to offer my special thanks to my external supervisor ir. Wietse Poiesz, for spending his valuable time on highly appreciated suggestions, feedback, and steering me in the right direction. Boudewijn Kroeze and Maarten Angenent for giving me the opportunity to perform my research at Inspiro, as well as my other colleagues for their assistance and pleasant conversations during coffee and lunch breaks.

On a more personal note, I would like to thank my (student) friends for helping me get through the master's and being there for me to occasionally escape from my thesis work or other obligations. My family for all of their support throughout the years, and last but not least, my girlfriend, Carmen, for her encouragement and unconditional support throughout my master's. Every single one of you gave me the support and motivation I needed to achieve this milestone, and without you all, the journey would have been a lot harder.

Contents

1 Introduction **1**

2 Related Work **4**

3 Syntax and Semantics of Sequence Diagrams **6**
- 3.1 Syntax 7
  - 3.1.1 Concrete syntax 7
  - 3.1.2 Abstract syntax 8
  - 3.1.3 Relation between concrete and abstract syntax 10
- 3.2 XMI file format 10
- 3.3 Semantics 11
  - 3.3.1 Semantics of basic interactions 11
  - 3.3.2 Semantics of combined fragments 13

4 Model-based Testing **15**
- 4.1 Labelled transition systems 15
- 4.2 The implementation relation *ioco* 17
- 4.3 Symbolic transition systems 18
- 4.4 TorXakis 20
  - 4.4.1 Channel definition (CHANDEF) 20
  - 4.4.2 Model definition (MODELDEF) 21
  - 4.4.3 Connection definition (CNECTDEF) 21
  - 4.4.4 Type definition (TYPEDEF) 22
  - 4.4.5 Function definition (FUNCDEF) 22
  - 4.4.6 Process definition (PROCDEF) 22
  - 4.4.7 State automaton definition (STAUTDEF) 22

5 From Sequence Diagrams to Model-Based Testing **24**
- 5.1 From model (XMI) to Symbolic Transition System (STS) 24
- 5.2 From STS to model-based testing with TorXakis 29
- 5.3 Extra features 32
  - 5.3.1 Visualize STS with Graphviz 33
  - 5.3.2 User-defined typed message formats, variables and expressions 33
  - 5.3.3 Callable (user-defined) TorXakis library functions 35

6 Practical Example: Calculator **36**

7 Case Study: Stairlift Remote and Receiver **40**
- 7.1 Protocol 41
- 7.2 Different modelling strategy due to timing constraints 42
- 7.3 Requirements 44
- 7.4 Testable UML sequence and class diagrams 45

8 Results and Discussion **48**
8.1 Results . . . 48
8.2 Discussion . . . 50
8.2.1 Test results . . . 50
8.2.2 Sequence diagrams as specification . . . 50
8.2.3 Verbose messages . . . 51
8.2.4 Limitations . . . 51

9 Conclusion and Future Work **52**
9.1 Conclusion . . . 52
9.2 Future work . . . 53

1 Introduction

With the increasing complexity and scope of software systems, the probability that such a system contains bugs or faults also increases. Especially in embedded software, applications can potentially be dangerous when the software (or hardware) contains a fault. The impact of this can be seen by the current developments in the field of autonomous vehicles, where collisions have happened due to software faults, which have resulted in casualties [1]. Another concern is often the cost of a software bug. It is not always possible to perform over-the-air updates for embedded systems that have already been released, as they are not always connected to the internet. This means that fixing bugs when the product is already released, may require performing updates in the field or even a recall, which can be costly[1]. Moreover, fixing a bug after release may not be possible in some cases, as such a bug could destroy the system. This was the case for the flight failure 501 of the Ariane 5 launch vehicle, which exploded 40 seconds after initiation of its flight sequence due to a software fault [2].

There are multiple techniques available to check the quality of software and spot and prevent all kinds of failures. Probably the most well-known technique is testing, which evaluates the software by observing its executions on actual valued inputs [3]. It is estimated that, during the development of a system, 50% of the time and cost is used for testing the system [4]. There are many different approaches available when it comes to testing. *Systematic software testing* belongs to one of the most important and widely used techniques for testing software [5]. However, creating test cases for this approach is often a tedious and lengthy process. Even though the execution of the test cases can be automated, it cannot be guaranteed that there are no faults left in the system, as it is impossible to test with all possible input combinations. Moreover, most test cases are created to make sure that each feature of a system under test (SUT) works properly. However, these test cases are usually not capable of finding faults in sequences of features [6]. The creation of the test cases therefore depends too much on the person that selects the input, which is why this process is often error prone [7].

This calls for a new way of testing, namely *model-based testing* [8]. In model-based testing, test cases are generated from a model, which is a specification that is based on the implementation of the SUT. These test cases can be generated and automatically executed by a model-based testing tool (e.g.UPPAAL TRON [9], TorXakis [10], JTorX [11], GraphWalker [12]), to find faults that only show up when a sequence of test cases is performed [5]. The tool achieves this by stepping through the model, sending the specified input to the SUT and checking if the output matches with expected output. When the tool encounters different output than expected, it will stop and report that a potential fault is found. Then, either a fault in the system is found, or the created specification model does not correspond to the desired behaviour of the system. Mohacsi et al. [13] states that other benefits may include:

- Improved quality of specification artefacts.
- More of the defects are found at an early stage of the project due to model creation.
- Improved test case quality due to systematic model coverage.
- Improved quality of SUT due to faster and more efficient testing.

Model-based testing is used in the industry but does not seem widely adopted. However, the technique does seem to get some grip, as some commercial companies have spawned,

[1] https://www.celerity.com/the-true-cost-of-a-software-bug/

such as Axini[2] and Tricentis[3], who provide model-based testing solutions as their primary business model. Related work shows that model-based testing is currently mostly being used in the transportation and automotive software domain [8, 13]. Gustafsson states that one of the reasons for the low adoption rate could be that a lot of effort and time needs to be invested in creating and maintaining the models that are needed for the model-based testing tools [6]. This is plausible, as most model-based testing tools require input models based on the specification of the SUT, which often have to be created with a specific modelling tool or syntax. Mohacsi et al. [13] strengthens this argument by stating that, due to the initial cost, the project needs to have a certain number of test cycles before model-based testing pays off. This means that in small or low budget projects, model-based testing does not seem like a viable option.

A solution to this problem may come in the form of Unified Modeling Language (UML) diagrams. UML has a status as an industry standard for modelling and is frequently used by the industry, to the point that it is part of many undergraduate university curricula in Information Technology fields [14, 15, 16]. UML diagrams are often already created in the projects design phases. The reason for this is that these diagrams make it easier for the developers to understand what has to be implemented, improves communication, reduces the number of software defects, improves the quality of the software, and provides valuable documentation [16]. These diagrams would also improve the testing process if they could be used to generate input models for a model-based testing tool.

In this work, UML sequence diagrams, in combination with UML class diagrams, have been chosen as the diagrams that will be supported. The reason for this is that the research is performed at the company Inspiro, which mainly uses sequence and class diagrams to model their systems. Also, TorXakis will be used as the model-based testing tool of choice, as part of the development of this tool is performed at the Radboud University. The main research question this work aims to answer is:

- Can we generate input models for the model-based testing tool TorXakis from UML sequence and class diagrams to improve on the testing process of embedded software?

The main research question can be answered by answering the following sub-questions:

- What are the syntax and semantics of sequence diagrams?
- Which type of model is most suitable as an intermediate model representation, from which an input model for the model-based testing tool TorXakis can be generated?
- What are the options to add types and variables to the UML diagrams, such that TorXakis can generate input based on these types?
- What are the options to add constraints to parameters, such that TorXakis can generate input based on these constraints?

In this thesis, Chapter 2 describes the related work that exists in this field of research. Chapter 3 explains the syntax and semantics of sequence diagrams, which is needed to extract the necessary information from the diagram to generate an input model for a model-based testing tool. Chapter 4 explains the theoretical background of model-based testing with labelled transition systems and TorXakis. Chapter 5 describes what is needed to convert

[2] `https://www.axini.com/nl/`

[3] `https://www.tricentis.com/`

sequence diagrams to a model suitable for model-based testing. Chapter 6 gives an example of how the proposed solution can be used in a practical setting. Chapter 7 describes the case study executed on some real hardware and software made by Inspiro. Chapter 8 explains the obtained results and contains the discussion. Finally, Chapter 9 provides the conclusion and future work on this research.

2 Related Work

This chapter describes some of the research and techniques related to this work, which can be categorized into three groups. The first group of related work researched ways to convert or transform UML (sequence) diagrams into other (mathematical) representations. However, they differ from our approach, as these intermediate representations are mostly used to translate UML diagrams into other diagrams, or for formal verification techniques such as model checking. The second group of related work researched how to generated or derive test cases from UML (sequence) diagrams, that often make use of some intermediate model representation from which the test cases can be generated. The final group contains some research that is closely related to this work, where UML diagrams are used to create an intermediate model-representation, from which test cases can be derived and automatically executed.

In the field of converting sequence diagrams into other (mathematical) representations, Grønno et al. [17] proposed a technique to transform sequence diagrams to state machines by using graph transformation. The transformation technique utilizes concrete syntax-based graph transformation rules, which are implemented in a tool called the attributed graph grammar system (AGG)[4]. This approach differs from other approaches, as the other mentioned approaches all use abstract syntax, instead of concrete syntax, to translate to other representations. Kundu et al. [18] introduced an approach to convert the XML Metadata Interchange (XMI) representation of UML interaction diagrams into control flow graphs. The paper identifies some difficulties that can occur when creating control flow graphs and proposes a conversion procedure to overcomes these difficulties. Brasil et al. [19] transformed behavioral diagrams, including: state machines, sequence diagrams, activity diagrams, etc. into transition systems for formal verification with model checking. A slightly more advanced approach is taken by Cartaxo et al. [20] which proposes a technique to convert sequence diagrams into labelled transition systems, instead of regular transition systems.

In the field of extracting test cases from UML diagrams, Muthusamy et al. [21] proposed a new approach to derive test cases from sequence diagrams. In their work, they first convert the sequence diagram from to the XMI format, also used in this work and explained in chapter 3.2, from which a sequence dependency graph is created. Then, test cases are generated from this graph, based on path coverage by using the "iterative deepening" depth-first search algorithm. Oluwagbemi et al. [22] proposed another technique to automatically generate test cases from UML diagrams. This technique also utilizes the XMI format to extract the necessary information from the sequence diagrams. However, instead of a sequence dependency graph, a dependency flow tree is used as the intermediate model representation. The test cases are then generated from the dependency flow tree based on coverage criteria such as model-flow, conditional-flow, element-flow and data-flow coverage. Panthi et al. [23] also proposed a technique to automatically generate test cases from sequence diagrams. They first convert the sequence diagram in a sequence graph, after which the sequence graph is traversed to collect the predicate functions. These predicate functions are transformed into source code, from which an extended finite state machine (EFSM) is created. The resulting state machine is then used to generate test cases.

In the field of executing the derived test cases from UML, Dahlweid et al. [24, 25] developed a tool called RT-Tester to transform a subset of UML/SysML diagrams into an intermediate model-representation, from which automatically executable test cases are derived. This subset

[4] http://www.user.tu-berlin.de/o.runge/agg/

includes composite structures or block diagrams to express the structure of the SUT, as well as state machines and operations to express the behaviour of the SUT. Similar to our work, the information of a sequence diagram is extracted from the XMI representation. From this information, an internal model representation is created, where instead of labelled transition systems, Kripke structures are utilized. The Kripke structures allow for the generation of the test cases and model checking. The main difference between Kripke structures and labelled transition systems is that Kripke structures have labelled states instead of labelled transitions. RT-Tester has a build-in test executor which can be seen as a model-based testing tool such as TorXakis.

3 Syntax and Semantics of Sequence Diagrams

A sequence diagram is a variant of an interaction diagram, which is part of the Unified Modeling Language (UML) standard. UML was created by the Object Management Group (OMG) to provide the tools for analysis, design, and implementation for system architects and software developers [26]. Sequence diagrams give a specification of a system, which shows how objects or other instances behave and communicate with each other. Sequence diagrams can have multiple purposes [27]. For instance, they can be useful for design and documentation, to graphically show the communication flow to a developer, which is often easier to understand than a textual explanation. Sequence diagrams are often intuitive enough that stakeholders can understand them without the need for background knowledge in computer science, which makes them very useful in explaining the communication flow of a system.

As mentioned before, sequence diagrams are a variant of the interaction diagram, of which there exist other forms, namely: communication diagrams, interaction overview diagrams, and timing diagrams. However, in this research, we will only focus on sequence diagrams. The syntax and semantics of UML diagrams are described in the OMG UML specification [26]. With the update to version 2.0 of this specification, the expressiveness of the language was highly increased, but the semantics were not precisely defined. This allows for the creation of hard to interpret diagrams, which leads to some problems [28].

Page 595 of the UML specification describes sequence diagrams as follows: "A sequence diagram is the most common kind of interaction diagram, which focuses on the message interchange between several lifelines" [26]. In the purest form, a sequence diagram consists out of lifelines and messages between these lifelines. The lifelines represent the objects or other instances that communicate with each other. The messages represent the actual communication that takes place between the lifelines. Figure 3.1 illustrates a simple sequence diagram, where a client sends a request message to the server, which is replied to with a reply message. This is an example of synchronous communication between two instances, where the sender of the message waits for the reply before sending another message. Other forms, such as asynchronous communication, are also included in the specification.

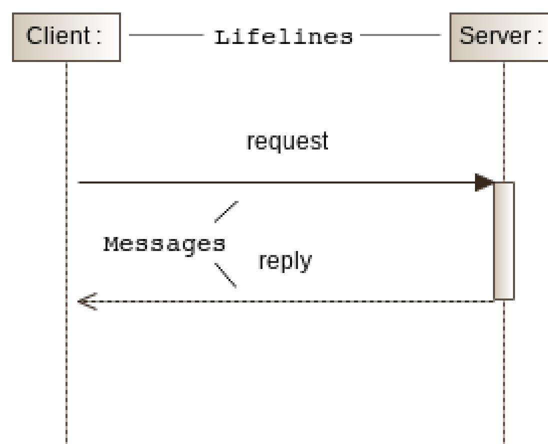


Figure 3.1: Example of a sequence diagram

There are a lot of tools available that are able to model sequence diagrams[5]. However, most modelling tools only implement a subset of the full UML specification. This can be seen when we compare the UML modelling tools Modelio[6] and StarUML[7]. StarUML includes the concept of lost and found messages, specified on page 574 of the UML specification [26], whereas Modelio does not. Related work also often includes only a subset of the full specification. The reason for this is that considering the full specification is often not feasible. Micskei et al. [28] researched this by comparing the elements that get mentioned in multiple research papers about sequence diagrams, which shows that certain elements do not seem important or used much. In our research, where we model the communication between two instances, certain elements such as the creation and deletion of objects are not relevant. This is why, in this work, we will also only focus on a subset of the sequence diagram specification. The sequence diagram notation consists of graphic nodes and graphic paths. The subset of graphic nodes includes lifelines and combined fragments (optional, alternative and loop). The subset of graphic paths includes: synchronous, asynchronous and reply messages.

3.1 Syntax

3.1.1 Concrete syntax

The *concrete syntax* of an interaction are the elements and their notations in sequence diagrams [28]. An example of the concrete syntax of messages and lifelines can be seen in Figure 3.2, where *Lifeline 1* and *Lifeline 2* represent the lifelines of the sequence diagram. The vertical dotted line describes the time-line for a process. Time increases down the line, but the distance between events does not describe any unit of actual time. It only tells us that a non-zero amount of time has passed [26]. The arrows between the lifelines represent the messages. Here, the arrows with filled arrowheads represent synchronous messages, which always wait for a reply. A line with an unfilled arrowhead represents the asynchronous messages. In the example in Figure 3.2, the *asynchronous message* is only sent after the *reply* to the *synchronous message* as been received.

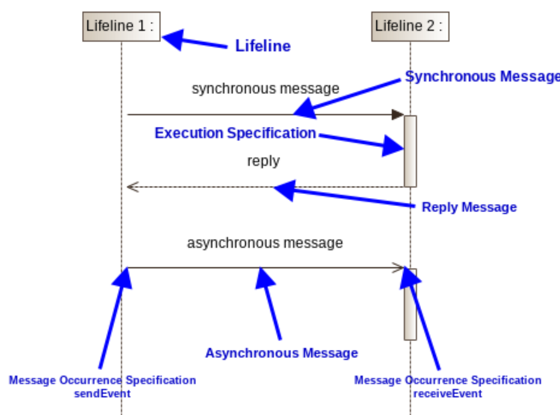


Figure 3.2: Lifelines with messages

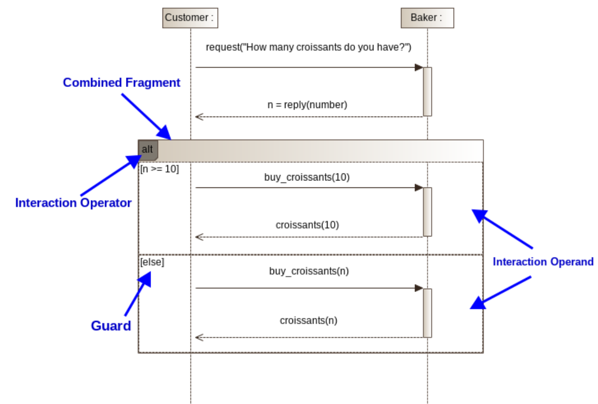


Figure 3.3: Combined fragments

[5] https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools
[6] https://www.modelio.org/
[7] http://staruml.io/

A *combined fragment* can be seen in Figure 3.3, where the box that contains the *alt* interaction operator represents the concrete syntax of a combined fragment. The interaction operands separate the different sequences of events that can be executed. If the guard in the interaction operand is satisfied, the sequence of events in this interaction operand will be executed. There are multiple types of combined fragments, which are explained in Section 3.3.2.

3.1.2 Abstract syntax

The *abstract syntax* of interaction diagrams, given in chapter 17 of the UML specification, contains the information of the elements in the diagram and the relations between them [26]. Each element and their relations that make up the interaction diagram is illustrated by meta-modelling with class diagrams throughout the chapter. The top-level abstract syntax of an interaction diagram is shown in Figure 3.4. We can see here that an interaction diagram can contain multiple interaction fragments. An InteractionFragment can exist in the form of Interaction, OccurrenceSpecification, ExecutionSpecification or StateInvariant [28]. A CombinedFragment, shown in Figure 3.3, is also an InteractionFragment, which can contain multiple InteractionFragments in its InteractionOperand. This shows that there can be multiple levels of nested InteractionFragments within an interaction diagram [26]. The semantics of some of these fragments are explained in Section 3.3.

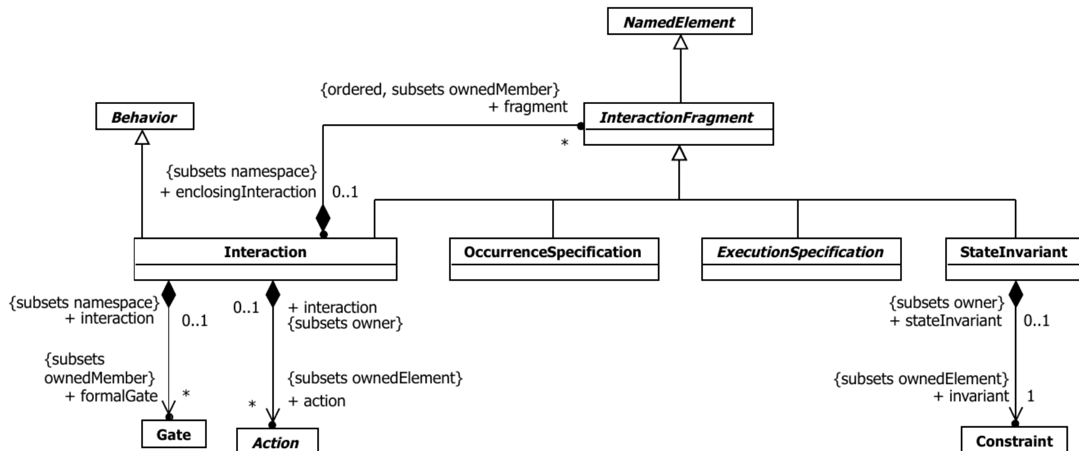


Figure 3.4: Abstract syntax interaction [26]

The sequence diagram variant of an interaction diagram also contains lifelines and messages along with the interaction fragments. The abstract syntax for lifelines is shown in Figure 3.5. This figure shows that, among other things, a lifeline is named, is contained in an interaction diagram, and is covered by InteractionFragments. These fragments contain the necessary information about the occurring events on a lifeline.

The abstract syntax for messages is shown in Figure 3.6. This abstract syntax diagram shows, among other things, that messages are of a certain sort and kind, is contained in an interaction diagram and can have as much as two MessageEnds. These MessageEnds have to be either a sendEvent or receiveEvent, but only one is allowed. The sendEvent contains the link to the sender of the message, e.g., a lifeline. The receiveEvent contains

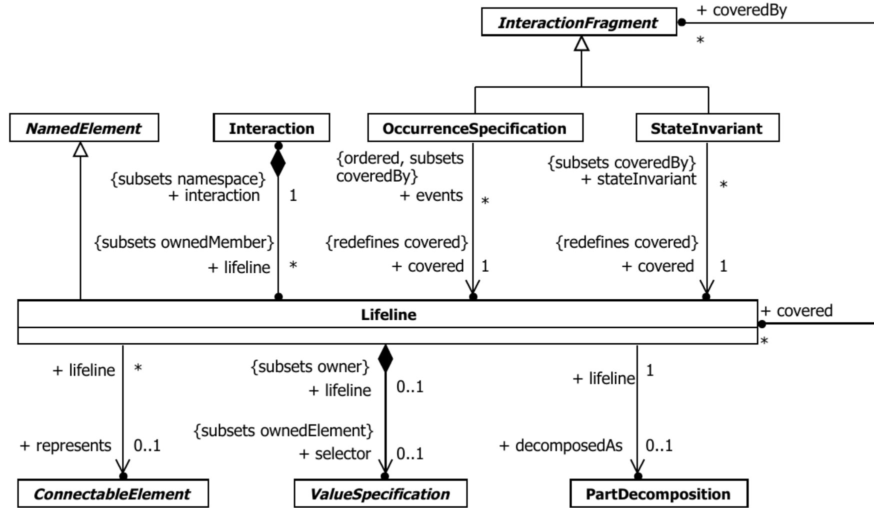


Figure 3.5: Abstract syntax lifelines [26]

the link to the receiver. The MessageEnd is a MessageOccurrenceSpecification, which is an OccurrenceSpecification fragment, which in turn is an InteractionFragment.

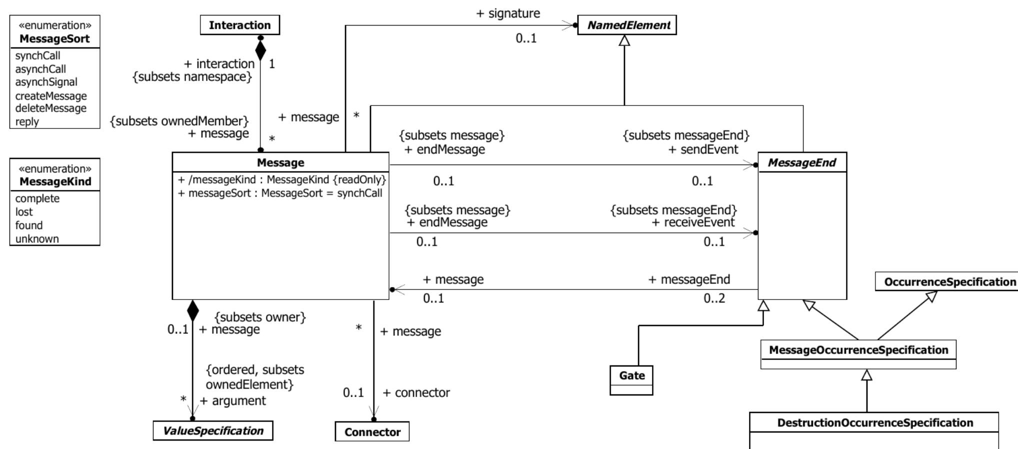


Figure 3.6: Abstract syntax messages [26]

If we omit some information and extra features and only show the lifelines, messages and fragments that hold the necessary information about the relations, we end up with the diagram shown in Figure 3.7. Here we can easily see that an interaction diagram consists out of lifelines, messages and (interaction)fragments. The messages have MessageEnds, which are a variant of a MessageOccurrenceSpecification. This MessageOccurrenceSpecification inherits from OccurrenceSpecification, which inherits from InteractionFragment. The relation between the lifelines and messages can be seen in the relation between OccurrenceSpecification and Lifeline. Here, the lifeline covers a list of OccurrenceSpecifications, which can be the MessageEnds, which is either a sendEvent or receiveEvent.

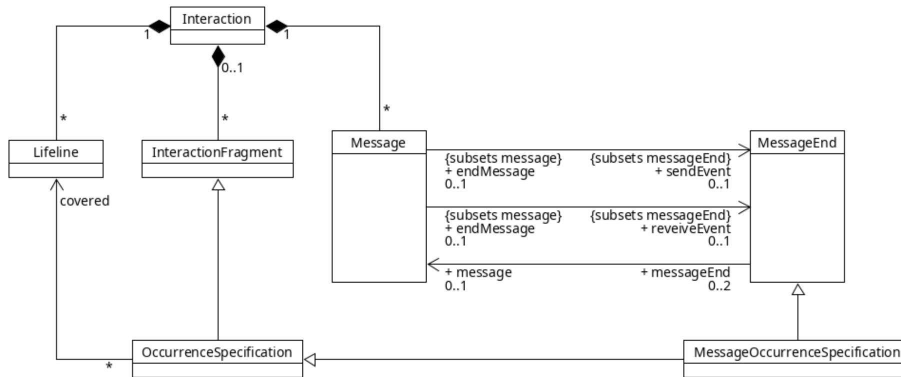


Figure 3.7: Abstract syntax simplified

3.1.3 Relation between concrete and abstract syntax

Figure 3.8 illustrates the relation between concrete and abstract syntax. The left side contains the concrete syntax of a sequence diagram. This sequence diagram contains two lifelines: `a` and `b`, where `a` sends an asynchronous message `m1` to `b`. The right side illustrates the abstract syntax, which shows that every element is contained in an interaction diagram. The elements are the lifelines, message and MessageOccurrenceSpecifications (fragments) that contain the information about the sender and recipient of the message.

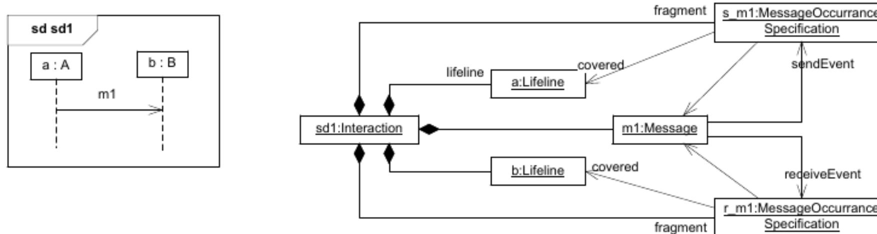


Figure 3.8: Relation between concrete and abstract syntax [28]

3.2 XMI file format

Modelling tools often use different data structures and file formats to save and interpret UML diagrams. The OMG group tried to combat this by developing a standard that should make it possible to exchange models between different tools [29]. This standard encodes the abstract syntax of a diagram to an XML document or schema, which allows for post-processing in the form of validation, code generation and much more. Many tools implement this standard or have plugins available to export to and import from the XMI file format[8]. However, not all tools generate the same XMI output for the same model [30], which introduces compatibility issues.

[8] `https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools`

If we model the sequence diagram from figure 3.8 with the tool Modelio and export it to XMI, it will generate the XMI output shown in Figure 3.9. Here, some of the randomly generated identifiers have been replaced with more readable identifiers. The packagedElement contains the information about the type of the diagram, which is `uml:Interaction`. Within this packagedElement, the two lifelines, the message and fragments are visible. For example, the *lifeline_a's* coveredBy attribute is linked to the *fragment_1*. This fragment's message attribute is linked to *message_1* and finally, the sendEvent of this message is linked back to *fragment_1*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://
www.eclipse.org/uml2/3.0.0/UML" xmi:id="_3jwb4ON5Eemd2I3geJGY9A" name="project4">
  <eAnnotations xmi:id="_3jwb4eN5Eemd2I3geJGY9A" source="Objing">
    <contents xmi:type="uml:Property" xmi:id="_3jwb4uN5Eemd2I3geJGY9A" name="exporterVersion">
      <defaultValue xmi:type="uml:LiteralString" xmi:id="_3jxC8ON5Eemd2I3geJGY9A" value="3.0.0"/>
    </contents>
  </eAnnotations>
  <packagedElement xmi:type="uml:Interaction" xmi:id="_3jxC8eN5Eemd2I3geJGY9A" name="Interaction">
    <nestedClassifier xmi:type="uml:Collaboration" xmi:id="_3jxC8uN5Eemd2I3geJGY9A" name="locals"/>
    <lifeline xmi:id="lifeline_a" name="a" coveredBy="fragment_1"/>
    <lifeline xmi:id="lifeline_b" name="b" coveredBy="fragment_2"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="fragment_1" covered="lifeline_a"
    message="message_1"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="fragment_2" covered="lifeline_b"
    message="message_1"/>
    <message xmi:id="message_1" name="m1" messageSort="asynchCall" receiveEvent="fragment_2"
    sendEvent="fragment_1"/>
  </packagedElement>
</uml:Model>
```

Figure 3.9: XMI output of sequence diagram

3.3 Semantics

The semantics of sequence diagrams are also given in the UML specification, though only informally [26]. Also, with these given semantics, Micskei et al. [28] states that there are two significant challenges. Firstly, the descriptions of the semantics are scattered throughout the text, which makes it hard to determine the precise semantics. Secondly, parts of the semantics are not specified in detail on purpose to allow the use of UML in many domains. This means that the modeller has to choose the variation of semantics for his/her particular domain, which is not always explicitly defined. This is why other research often proposes its own semantic interpretation for sequence diagrams [28]. For simplicity, only the semantics of the selected elements will be explained. The following sections give a summary of the semantics of basic interactions, fragments and combined fragments, based on the UML specification [26]. For some of the elements, a decision had to be made on what semantic interpretation would be adopted in this work. These decisions are based on the comparison of the proposed semantics in Micskei et al. [28].

3.3.1 Semantics of basic interactions

Interactions illustrate the behaviour of a system, where the communication between participants exists in the form of messages. The central concept of the semantics of an interaction can be defined as a pair of sets of traces, represented by the set of valid and invalid traces [28].

A trace can be seen as a sequence of event occurrences, described by OccurrenceSpecifications. The invalid set of traces can only be present if messages are contained in a combined fragment that has the Negative (Neg), or are on the outside of a combined fragment with the Assertion (Assert) operator. These operators are not included in our selection, so the set of invalid traces can be ignored in this work. All other included traces are part of the set of valid traces.

The only form of OccurrenceSpecifications in this work are MessageOccurrenceSpecifications. These MessageOccurrenceSpecifications represent either the sendEvent or receiveEvent of a message, where the sendEvent should always occur before the receiveEvent. The signature of a message can either refer to an operation or a signal. In this work, we will only consider the operation signature. With this signature, the messageSort can be either synchCall, asynchCall or reply. When the messageSort is a reply, it is the return of a synchCall. An operation can also carry arguments, which are described by ValueSpecifications.

For a message to be able to exist, the sendEvent and receiveEvent should be linked to participants, represented by lifelines. As mentioned before, these lifelines describe the timeline of a process, where time increases down the line. The events that occur on a lifeline are executed in order. For example, if two messages occur on a lifeline, and the second message is further down the line, it will be sent after the first message is sent and received. The ordering is determined based on the order of OccurrenceSpecifications events.

Consider the example sequence diagram in Figure 3.10. The events of message *m1* are $!m1$ and $?m1$, where $!m1$ denotes the sendEvent of the message, and $?m1$ denotes the receiveEvent of the message. However, since there can be more lifelines, some of the proposed semantics propose that the elements of the message should be encoded in tuples in the form of $(sender, receiver, message)$ [28]. Also, to differentiate between two messages that have the same name, a unique identifier should be added to the tuple. Now message *m1* can be encoded in tuple $(0, A, B, m1)$ and message *m2* can be encoded in tuple $(1, C, B, m2)$.

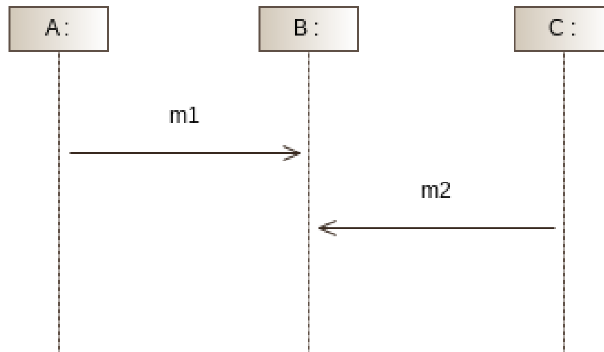


Figure 3.10: Sequence diagram sd_1

The UML specification states that occurrences on the same lifeline must occur in the same order as they are specified, even for the receiving of messages sent by different objects [26]. This why when only the valid traces are considered, the standard interpretation of the sequence diagram sd_1 in Figure 3.10 is $!m1 \cdot ?m1 \cdot !m2 \cdot ?m2$. The order of the sendEvents does not matter because they are not related, but $?m1$ has to be received before $?m2$ [28].

3.3.2 Semantics of combined fragments

Combined fragments are different from traces, in the sense that they do not add traces themselves, but only alter the ordering of the already present traces. As mentioned before, combined fragments are regions highlighted with an InteractionOperator such as *alt*. Depending on the operator, the combined fragment also contains one or more InteractionOperands. Each of these InteractionOperands can be seen as a full sub-sequence diagram. Moreover, a sequence diagram itself can also be seen as a CombinedFragment with the "seq" operator. This operator applies sequential composition, also known as weak sequencing, to all fragments in the diagram. The rules of weak sequencing are such that, when OccurrenceSpecifications share the same lifeline, the ordering is maintained. However, OccurrenceSpecifications on different lifelines from different operands may come in any order [28].

For readability, the tuple notation is omitted in this example of weak sequencing in Figure 3.11, as well as future examples. Here, the messages *m1* and *m2* share lifeline *b*. However message *m3* does not share lifelines with the other messages. This means that the send and receive events of message *m3* can come at any time, as long as the events of *m2* come after the events of *m1*. As an example, the traces $!m3 \cdot ?m3 \cdot !m1 \cdot ?m1 \cdot !m2 \cdot ?m2$, $!m1 \cdot ?m1 \cdot !m2 \cdot ?m2 \cdot !m3 \cdot ?m3$, and $!m1 \cdot !m3 \cdot ?m1 \cdot !m2 \cdot ?m3 \cdot ?m2$ are all valid.

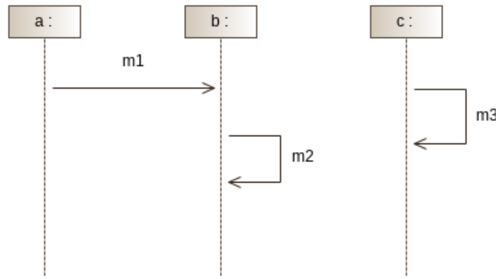


Figure 3.11: Weak sequencing example

The opposite of weak sequencing is strict sequencing. With strict sequencing, the order of sending and receiving messages is strictly maintained. While weak sequencing is the default, it introduces decidability issues, which is why some of the proposed semantics in Micskei et al. [28] introduce synchronization on entering and exiting fragments to eliminate these decidability issues. Most of the time, synchronization on entering and exiting fragments is adopted when sequence diagrams are used for verification purposes. In this work, weak sequencing will not be an issue, as lifelines will always be shared when communicating with the environment, maintaining the ordering. This is explained further in Section 5.1.

The interaction operators for combined fragments considered in this work include: *opt*, *alt* and *loop*. Both *opt* and *loop* can only contain a single interaction operand. However, *alt* may contains one or more interaction operands. Each operand also contains a guard. If none of the guards are satisfied, none of the interaction operands in the combined fragment will be executed. Otherwise, if the guard of an operand is satisfied, this operand will be executed. If multiple operands have guards that are satisfied, which operand is executed will be determined non-deterministically. Some proposed semantics remove the non-determinism by executing the first operand guard that evaluates to true [28]. However, in this work, the non-deterministic choice is maintained.

- The *opt* operator is short for option and acts as a single if-then statement.
- The *alt* operator is short for alternatives and acts as a switch or if/else statement where the guards are the conditions. If multiple guards are satisfied, at most, one of the operands will be executed non-deterministically [26].
- The *loop* operator will repeat over its events until the condition of the guard is unsatisfied. The guard includes three values. The first value describes the minimum number of iterations, the second value the maximum number of iterations, and the third value describes the loop guard. This means that when the minimum number of iterations is not yet reached, even if the loop guard is not satisfied, the loop will continue its iterations. The loop will terminate if the loop guard is not satisfied or the maximum number of iterations is reached. The OMG specification states that weak sequencing is in effect between the iterations of the loop [26]. This means that the send events of the messages will always be in order. However, it is not known in what order the messages will be received, or even if they will be received before the next iteration starts. This introduces the same issues as mentioned before, which is why strict sequencing will be applied between loop iterations in this work [28]. This results in an interpretation of a loop that is similar to loops in programming languages.

Consider sequence diagram sd_2 in Figure 3.12. To easily calculate the set of traces in this sequence diagram, the guards in the combined fragment hold a *True* or *False* value to decide whether the guard is satisfied or not. In a regular sequence diagram, the guard contains a value that is not yet evaluated. This means that to calculate the set of traces in a diagram where the satisfiability of the guards is not yet known, all possible combinations of guard evaluations have to be included in the set. In this case, the satisfiability of the guards is known, meaning that the standard interpretation of sequence diagram sd_2 is: $!m1 \cdot ?m1 \cdot !m2 \cdot ?m2 \cdot !r2 \cdot ?r2 \cdot !m2 \cdot ?m2 \cdot !r2 \cdot ?r2 \cdot !m3 \cdot ?m3 \cdot !m4 \cdot ?m4 \cdot !r4 \cdot ?r4 \cdot !m5 \cdot ?m5$.

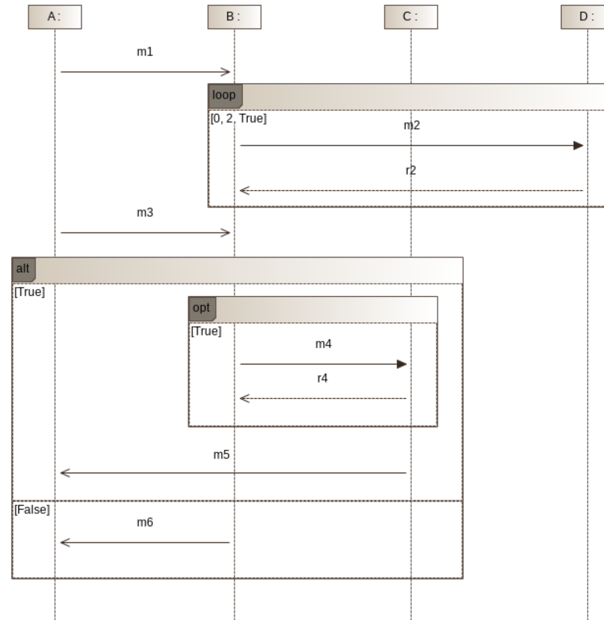


Figure 3.12: Sequence diagram sd_2

4 Model-based Testing

Model-based testing is a technique that is used to test a system in search of bugs [6]. In model-based testing, a model has to be created, which should be based on a specification of the system under test (SUT) [5]. This specification describes the desired behaviour of the SUT. From this model, a model-based testing tool can generate a large number of test cases, which can then be executed automatically on the SUT. Separately performing the generation and execution steps is called batch test case derivation or off-line testing. There is also a method available that combines these steps, which is called on-the-fly test case generation [31] or on-line testing.

There are different approaches possible when it comes to model-based testing. These different approaches depend on the kind of model used, which quality aspects are being tested, and what the level of accessibility and observability of the system is [5]. In model-based testing, the SUT is often regarded as a *black box*, which means that no internal details of the SUT can be seen. Moreover, the SUT is only accessible by providing some input and observing the output of the SUT, which can often be tested based on the requirement specification knowledge. The opposite of *black-box* testing is *white-box* testing, also called structural or glass box testing. White-box testing is a technique that uses the source code as a basis for generating the test cases [32]. Here, the internals are known, and the test cases are designed to structurally test, e.g. all different branches in a piece of code.

4.1 Labelled transition systems

A modelling formalism is needed to describe a system with a model. Such formalisms can exist in the form of specification languages or mathematical structures. A UML sequence diagram, as described in chapter 3, is an example of a graphical specification language. An example of a model formalism that is described by a mathematical structure is a Labelled Transition System (LTS) [31]. The formal definition of labelled transition systems, taken from Tretmans and Stoelinga et al. [5, 31] is given below.

Definition 4.1. A *Labelled Transition System* (LTS) is a tuple $\langle S, s_0, L, T\rangle$, where

- S is a non-empty set of states;
- $s_0 \in S$ is the initial state;
- L is a set of labels;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$, with $\tau \notin L$, is the transition relation.

We write $s \xrightarrow{\mu} s'$ if there exists a transition labelled μ, from state s to state s', or formally $(s, \mu, s') \in T$. If no such transition exists for any s, we write $s \overset{\mu}{\nrightarrow}$. Moreover, if we take the previous transition and there exists another transition $s' \xrightarrow{\mu'} s''$, we can compose transitions by writing $s \xrightarrow{\mu \cdot \mu'} s''$. In general for a transition from state s_0 to, for example state s_2 and an arbitrary number of transitions, we write $s_0 \xrightarrow{\mu_0 \cdot \mu_1 \cdot \ldots \cdot \mu_n} s_2$. A transition labelled τ, describes an internal action in the system that is not observable from the environment. Suppose $\sigma \in L^*$ is a sequence of labels, then we write $s \overset{\sigma}{\Longrightarrow} s'$ if there exists a sequence $\varrho \in (L \cup \tau)^*$ such that $s \xrightarrow{\varrho} s'$, where σ can be obtained from ϱ by omitting all $\tau's$.

We write $s \overset{\sigma}{\Longrightarrow}$ if there exists a state s' such that $s \overset{\sigma}{\Longrightarrow} s'$. When reasoning about labelled transition systems, e.g. $A = \langle S, s_0, L, T\rangle$, we write $A \overset{\sigma}{\Longrightarrow}$ instead of $s_0 \overset{\sigma}{\Longrightarrow}$. The *traces* of a

labelled transition system are all possible sequences of labels. For the traces of $LTS\ A$, we write $traces(A)$, of which the definition is $traces(A) = \{\sigma \in L^* | s \stackrel{\sigma}{\Longrightarrow}\}$. An example of a simple labelled transition system A can be seen in Figure 4.1. If we take definition 4.1 of an LTS, we can explicitly write $LTS\ A$ and the $traces(A)$ as:

$$A = \langle \{s_0, s_1, s_2, s_3\}, s_0, \{a, b, c\}, \{(s_0, a, s_1), (s_1, b, s_2), (s_1, c, s_3)\} \rangle$$
$$traces(A) = \{\epsilon, a, a \cdot b, a \cdot c\}$$

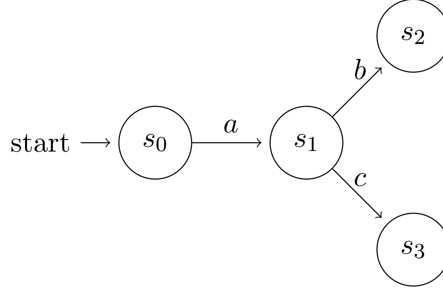


Figure 4.1: Labelled transition system A

Each transition in a Labelled Transition System contains a label. These labels do not yet describe which transitions model input to our SUT, nor which transitions model output from our SUT [5]. We can extend the labelled transition system with inputs and outputs by splitting the set of labels L in two sets: L_I and L_O. Here, the set of input actions are contained in L_I, and the set of output actions are contained in the set L_O. The following definition of labelled transition systems with inputs and output originates from Tretmans and van den Bos et al. [5, 33].

Definition 4.2. A *Labelled Transition System* (LTS) with inputs and outputs is a tuple $\langle S, s_0, L_I, L_O, T \rangle$, where

- $\langle S, s_0, L_I \cup L_O, T \rangle$ is a labelled transition system in $\mathcal{LTS}(L_I \cup L_O)$, where $\mathcal{LTS}(L)$ is the class of all image finite and strongly converging labelled transition systems with labels in L;
- L_I and L_O are sets of input labels and output labels, which are disjoint $L_I \cap L_O = \emptyset$.

In labelled transition systems with inputs and outputs $\mathcal{LTS}(L_I, L_O)$, transitions that denote inputs are usually prefixed with '?' and transitions that denote outputs are prefixed with '!'. If a state $s \in S$ has no outgoing transition with an output label, the state is quiescent (δ), written by $s \xrightarrow{\delta} s$. to compute the $traces(p)$ that include quiescence, where p is a labelled transition system $\langle S, s_0, L_I, L_O, T \rangle$, we write $Straces(p) = \{\ \sigma \in (L_I \cup L_O \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Longrightarrow}\ \}$. Here, $Straces$ stands for suspension traces.

Consider LTS B with inputs and outputs, shown in Figure 4.2. The set of possible inputs L_I consists of $\{?a, ?b, ?c\}$ and the set of possible outputs L_O consists of $\{!d\}$. The system only produces an output when the transition $s_2 \xrightarrow{!d} s_0$ is taken. This means that states s_0 and s_1 are quiescent, shown with an outgoing transition labelled with δ. Quiescent states in future examples of transition systems will not include an outgoing transition labelled with δ, to increase readability.

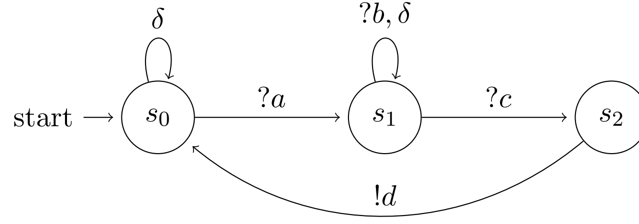


Figure 4.2: Labelled transition system B

To be able to describe the *out*puts an LTS produces **after** a number of transitions are taken, we write $out(p \textbf{ after } \sigma)$. Here, p is the transition system, e.g. LTS B, and σ is the sequence of labels along a path that is taken through the system, which can contain zero or more transitions. Some examples for LTS B are given below:

$$
\begin{aligned}
out(B \textbf{ after } \epsilon) &= \{\delta\} \\
out(B \textbf{ after } ?a) &= \{\delta\} \\
out(B \textbf{ after } ?a{\cdot}?a) &= \emptyset \\
out(B \textbf{ after } ?a{\cdot}?b \cdot \delta{\cdot}?b) &= \{\delta\} \\
out(B \textbf{ after } ?a{\cdot}?b{\cdot}?c) &= \{!d\} \\
out(B \textbf{ after } ?a{\cdot}?b{\cdot}?c{\cdot}!d{\cdot}?a{\cdot}?c) &= \{!d\}
\end{aligned}
$$

4.2 The implementation relation *ioco*

In model-based testing, the goal is to have a specification (model) of the system and compare this specification to the actual implementation. This comparison between specification and implementation is also called the implementation relation. The **ioco** implementation relation is an example of such a relation, and tells us that there should be an **i**nput-**o**utput **co**nformance between an implementation and specification. This means that any output that an implementation i produces should be specified in specification s. The specification s may specify a larger set of outputs, but as long as the implementation produces a subset of this set of outputs, i **ioco** s.

Implementation i requires a different transition system, namely an Input-Output Transition system (IOTS), where $IOTS(L_I, L_O) \subseteq LTS(L_I, L_U)$. These do not differ from labelled transition systems other than that they are input enabled. This means that in any state of the system, it is always prepared to receive any of the specified input actions. From a real-world standpoint, this makes sense, as a user is always able to press any button on, e.g. a coffee machine, even when the machine is in the process of making coffee. Definition 4.3 of the **ioco** implementation relation originates from Tretmans and Stoelinga et al [5, 31].

Definition 4.3. *Given a set of input labels L_I and a set of output labels L_O, the relation* **ioco** $\subseteq IOTS(L_I, L_O) \times LTS(L_I, L_O)$ *is defined as follows:*

$$i \textbf{ ioco } s \;\; \Leftrightarrow_{def} \;\; \forall \sigma \in Straces(s): \quad out(i \textbf{ after } \sigma) \;\subseteq\; out(s \textbf{ after } \sigma)$$

Figure 4.3 illustrates an example of two transition systems i and s where **ioco** holds. Here i **ioco** s because for all possible $Straces(s)$, the out set of i **after** *?a* is a subset of the out set of s **after** *?a*, as $\{!b\}$ is a subset of $\{!b, !c\}$.

$$\begin{aligned} out(i \textbf{ after } \epsilon) = \{\delta\} \;&\subseteq\; out(s \textbf{ after } \epsilon) = \{\delta\} \\ out(i \textbf{ after } ?a) = \{!b\} \;&\subseteq\; out(s \textbf{ after } ?a) = \{!b, !c\} \\ out(i \textbf{ after } ?a{\cdot}!c) = \emptyset \;&\subseteq\; out(s \textbf{ after } ?a{\cdot}!c) = \{\delta\} \end{aligned}$$

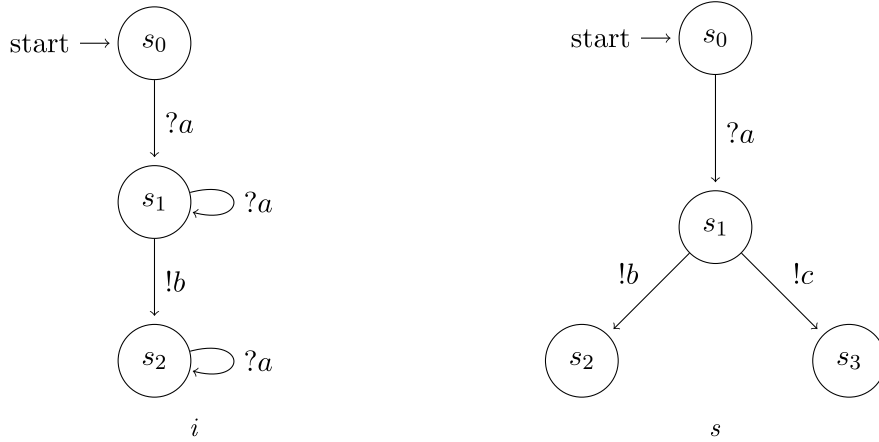


Figure 4.3: Implementation i and specification s with **ioco**

4.3 Symbolic transition systems

Symbolic Transition Systems (STS) add data, variables and data-dependent conditions to labelled transition systems [33]. These additions are needed when dealing with, e.g. loops that need a loop counter to count the number of iterations that the loop has executed. Data-dependent conditions can be seen as guards, which are also supported in sequence diagrams. The following definition originates from van den Bos et al. [33].

Definition 4.4. A *Symbolic Transition System* (STS) with inputs and outputs is a tuple $\langle \mathcal{L}, l_0, \mathcal{V}_l, m_{ini}, \mathcal{V}_p, \Gamma_I, \Gamma_O, \mathcal{R} \rangle$, where:

- $\mathcal{L}$ is a set of locations;
- $l_0 \in \mathcal{L}$ is the initial location;
- $\mathcal{V}_l$ is a set of location variables;
- $m_{ini} \in \mathcal{T}(\emptyset)^{\mathcal{V}_l}$ is the initialization;

- $\mathcal{V}_p$ is a set of gate parameters such that $\mathcal{V}_p \cap \mathcal{V}_l = \emptyset$;
- Γ_I is a set of input gates;
- Γ_O is a set of output gates;
- $\mathcal{R} \subseteq \mathcal{L} \times (\Gamma_I \cup \Gamma_O \cup \{\tau\}) \times \mathcal{V}_p^* \times \mathcal{T}_{Bool}(\mathcal{V}_l \cup \mathcal{V}_p) \times \mathcal{T}(\mathcal{V}_l \cup \mathcal{V}_p)^{\mathcal{V}_l} \times \mathcal{L}$ is the switch relation.

We require that $\Gamma_I \cap \Gamma_O = \emptyset$ and denote $\Gamma = \Gamma_I \cup \Gamma_O$. Terms are defined with $\mathcal{T}$, where $\mathcal{T}_s(X)$ means the set over all terms over variables $X \subseteq \mathcal{X}$ with sort $s \in S$. Here, S is a nonempty set of sort names, $\mathcal{X}_s$ is a set of variables of sort s and $\mathcal{X}$ is the set of all variables. This means that $\mathcal{T}_{Bool}(X)$ is the set of all terms of sort *Bool*, which can be *True* or *False*, over variables X. The function $\texttt{sort}_\texttt{t} : \mathcal{T}(\mathcal{X}) \to S$ gives the sort of a term. The function $\texttt{sort}_\texttt{g} : \Gamma \to S^*$, associates a sequence of sorts to a gate. $(l_1, \lambda, p_0 \dots p_k, \phi, \psi, l_2) \in \mathcal{R}$ are the elements of a switch, with source location, gate, parameters, guard, assignment, and destination location, respectively, and we require that:

- $p_0 \dots p_k$ is a sequence of distinct variables
- $\texttt{sort}_\texttt{g}(\lambda) = \texttt{sort}_\texttt{t}(p_0 \dots p_k)$
- $\phi \in \mathcal{T}_{Bool}(\mathcal{V}_l \cup \{p_0, \dots, p_k\})$
- $\psi \in \mathcal{T}(\mathcal{V}_l \cup \{p_0, \dots, p_k\})^{\mathcal{V}_l}$

In Figure 4.4 STS C is shown, which illustrates an example with variables and data-dependant conditions. The STS first initializes variable v_0 to 0 with m_{ini} $\{ \ v_0 := 0 \ \}$. Then input *increment* is sent with parameter n. This transition also assigns the value of this parameter to variable v_0. Finally, output *result* is expected where a data-dependant condition is used to check that the parameter r is incremented by one. This is done by checking if r equals $v_0 + 1$.

$$
\begin{aligned}
C &= (\{s_0, s_1, s_2\}, s_0, \{v_0\}, \{v_0 := 0\}, \{n, r\}, \{?increment\}, \{!result\}, \{r_0, r_1\}) \\
\texttt{Int} &= \texttt{sort}_\texttt{g}(?increment) = \texttt{sort}_\texttt{g}(!result) \\
r_0 &= (s_0, ?increment, n, True, \{ \ v_0 := n \ \}, s_1); \\
r_1 &= (s_1, ?result, r, [[\ r == v_0 + 1 \ ]], id, s_2);
\end{aligned}
$$

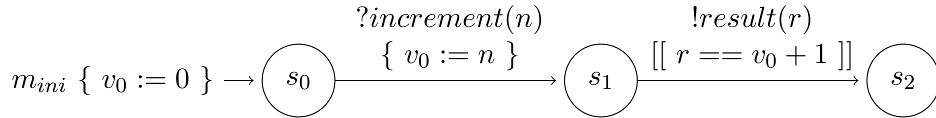


Figure 4.4: Symbolic Transition System C

4.4 TorXakis

TorXakis is a model-based testing tool of which the research and development are carried out as part of the NWO-TTW project 13859: SUMBAT: Supersizing Model-Based Testing, as well as part of the Enable-S3 program, which is under the responsibility of ESI (TNO) with Philips as the carrying industrial partner. TorXakis is open-source, available on GitHub[9] and utilizes on-the-fly (on-line) test generation. Figure 4.5 illustrates the required architecture for testing with the model-based testing tool TorXakis. The tool requires a `.txs` input model file which contains the information it needs for which channels (a gate that includes a network address) can be used for connecting to the test harness, as well as the actual model of the system under test. The test harness is an adapter that forwards the inputs that are being sent by TorXakis via sockets to the SUT, and sends the output that the SUT produces via sockets back to TorXakis for evaluation. This harness is needed as not all applications communicate via sockets, which means a translation step is required.

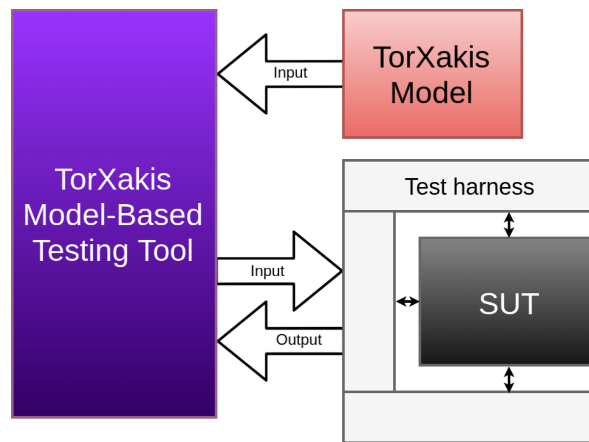


Figure 4.5: Model-based testing architecture

The input model file has to be written in specific syntax, as TorXakis has a built-in compiler. The model file has at least each of the following segments: CHANDEF, MODELDEF and CNECTDEF, which define the connection to the SUT. There can also be a multiple of other segments: TYPEDEF, FUNCDEF, PROCDEF and STAUTDEF that hold the information about the types used in the model and the model of the SUT itself. More information can be found on the TorXakis wiki[10].

4.4.1 Channel definition (CHANDEF)

The channel definition segment contains the information of all channels that are available for communication between TorXakis and the SUT. This segment does not yet specify the direction of defined channel (input or output). Also, when defining a channel, a type has to be supplied. These types can be one of the standard types: Int, Bool, String or Regex, but can also be user-defined by using the TYPEDEF keyword (see Section 4.4.4). During testing, TorXakis will automatically generate a value of the specified type. In the example

[9]https://github.com/TorXakis/TorXakis
[10]https://github.com/TorXakis/TorXakis/wiki

below, the channel definition named "ChanDefs" defines two channels: *InputChannel* and *OutputChannel*, that both carry type Int.

```
CHANDEF ChanDefs ::=
    InputChannel    :: Int;
    OutputChannel   :: Int;
ENDDEF
```

4.4.2 Model definition (MODELDEF)

The model definition segment defines the direction of the available channels, as well as the behaviour of the model. Here, the numerous process definitions (see Section 4.4.6) and state automaton definitions (see Section 4.4.7) can be assigned to the model to describe the behaviour of the SUT. In the example below, the model definition named "Model" assigns the direction of the previously defined channels. Also, a process definition called "process", is appended to the behaviour of the model, where the communication channels are supplied to the process definition between the square brackets. Here, optional arguments can also be supplied between the parentheses.

```
MODELDEF Model ::=
    CHAN IN     InputChannel
    CHAN OUT    OutputChannel

    BEHAVIOUR   process [ InputChannel, OutputChannel ] ( )
ENDDEF
```

4.4.3 Connection definition (CNECTDEF)

The connection definition segment contains the information to set up the socket connections to the test harness of the SUT. Here, each channel is assigned an IP-address and port. Also, channels that are defined as input channels send input to the SUT. This means that from the perspective of the model-based testing tool, these channels output to the SUT. This is why they are prefixed with they keywords CHAN OUT and vice versa for output channels. The ENCODE and DECODE keywords specify how data should be encoded or decoded to make sure that the recipient is able to interpret the data. In the example below, the output of TorXakis is assigned to the input channel of the SUT, which is defined to be on IP-address `localhost` and port `9999`. Also, all input data is encoded to a string before it gets sent to the SUT. The opposite holds true for the input to TorXakis, where the output of the SUT is received from the previously defined ip-address and port, and the data is decoded to a string.

```
CNECTDEF Sut ::=
    CLIENTSOCK

    CHAN OUT    InputChannel    HOST "localhost" PORT 9999
    ENCODE      InputChannel    ?x -> !toString(x)

    CHAN IN     OutputChannel   HOST "localhost" PORT 9999
    DECODE      OutputChannel   !fromString(x) <- ?x
ENDDEF
```

4.4.4 Type definition (TYPEDEF)

A type definition allows for user-defined types. As mentioned before, the standard available types are Int, Bool, String and Regex. Possible type definitions are, for instance: records, recursive data types or enums. TorXakis automatically generates type checking and predefined functions such as equality on user-defined types. In the example below, a recursive list of integers is defined.

```
TYPEDEF List ::=
        Nil | Cons { head::Int; tail::List }
ENDDEF
```

4.4.5 Function definition (FUNCDEF)

Function definitions allow for user-defined functions to prevent duplicate code. In the example below the function checks if the supplied argument x is a valid 32bit integer by checking if its value is in between the maximum and minimum allowed values.

```
FUNCDEF isValid_int32 ( x :: Int ) :: Bool ::=
    (-2147483648 <= x) /\ (x <= 2147483647)
ENDDEF
```

4.4.6 Process definition (PROCDEF)

The process definitions can describe an actual process in the model of the SUT. In the example below, the channels are passed to the process definition named “test”. Then TorXakis generates an integer for input x for which the guard between the square brackets holds, and sends it on the InputChannel to the SUT. The sequence operator >-> denotes that after the input is sent, the next step is performed. Here, the process waits for output on the OutputChannel. If the output is received, the process checks if the value of the output value of x equals the input value of x, and if the guard holds. If both hold, the process calls itself to repeat the process. If any steps in the sequence trigger a fault, TorXakis gives an error that the output was not as expected, or no output was received. The process example below describes a system that returns the supplied input as output. Often, a process definition is defined for each state in a labelled or symbolic transition system.

```
PROCDEF test [ InputChannel, OutputChannel :: Int ] ( ) ::=
        InputChannel  ?x [[isValid_int32(x)]]
    >-> OutputChannel !x [[isValid_int32(x)]]
    >-> test [ InputChannel, OutputChannel ] ()
ENDDEF
```

4.4.7 State automaton definition (STAUTDEF)

The state automaton definition allows for modelling both labelled and symbolic transition systems. This means that the STAUTDEF is able to keep track of variables. Each symbolic transition system can be defined within a single STAUTDEF. The following example describes the same system as the previous process definition example (see Section 4.4.6), but also keeps track of the variable y, which can be manipulated by the expression between the curly brackets.

```
STAUTDEF test [ InputChannel, OutputChannel :: Int ] ( ) ::=
    STATE    s0, s1
    VAR      y :: Int
    INIT     s0

    TRANS    s0 -> InputChannel  ?x [[isValid_int32(x)]] {y := x}
                 -> s1
             s1 -> OutputChannel !y [[isValid_int32(y)]]
                 -> s0
ENDDEF
```

5 From Sequence Diagrams to Model-Based Testing

We developed the tool sd2txs to transform UML sequence diagrams to an input model that is suitable for model-based testing with TorXakis. The tool performs some steps that are required for the transformation, which can be seen in Figure 5.1. The first step extracts the semantics from the abstract syntax. This is achieved by exporting the sequence diagram to an XMI file, which is then parsed to get access to the objects and relations between them. From these objects and relations, an intermediate model representation in the form of a symbolic transition system is created. Chapter 2 shows that related work also often makes use of intermediate model representations. A reason for this is that sequence diagrams do not have a good representation of repetitive, recursive, or conditional sequences, suppressing many control and interface details. On the contrary, an STS provides a global, monolithic description of the set of all possible behaviours of the system. This results in STSs being highly testable models, where a path on the STS can be taken as a test sequence [20]. Moreover, an STS can easily be visualized with a tool like GraphViz[11] to spot possible mistakes in the modelling process, and can be used to generate the TorXakis input model file. It is also possible to perform formal verification such as model checking on the intermediate model representation, as when the model does not contain data and variables, it is based on the labelled transition system mathematical structure that can be translated to, e.g., a Kripke Structure. In this section, all the mentioned required steps are explained in detail.

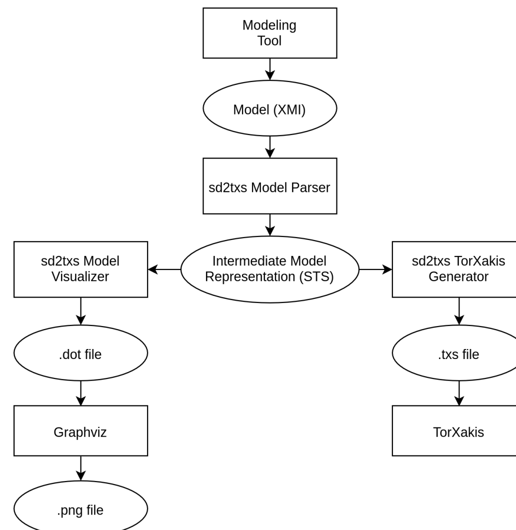


Figure 5.1: Architectural Diagram sd2txs

5.1 From model (XMI) to Symbolic Transition System (STS)

The first step is to extract the information about the objects and the relations between them from the sequence diagram by using the exported XMI file. An example of such an XMI file can be seen in Figure 3.9. The extracted information can then be used to build the STS, for which most of the steps with some minor changes, could be used directly from Cartaxo et al. [20]. We also introduce the concept of Host in this section. This concept can be seen as the

[11] `https://www.graphviz.org/`

environment that is able to send input and receive output from the SUT. The environment can be thought of as a user that is interacting with interfaces of the SUT by, e.g., pressing buttons. These interfaces are represented in a sequence diagram by all lifelines other than Host, which together form the SUT. This enables us to make a closed system, which does not communicate with the outside world, an open system that is able to communicate via multiple interfaces.

1. The states of the STS are natural numbers prefixed with s, numbered in increasing order according to the transitions that need to be added;
2. State s_0 represents the initial state of the system, i.e., the state that represents the initial requirements for the scenario represented in the sequence diagram to occur. The initial state has an input arrow to mark the start;
3. For each new transition, a new state is created;
4. For each message (in the order they appear) of the sequence diagram, if the message is:
   (a) From Host to SUT, a transition with a switch containing which interface of the SUT is communicating and the message content prefixed with '?' (input) is created;
   (b) From SUT to Host, a transition with a switch containing which interface of the SUT is communicating, and the message content prefixed with '!' (output) is created;

A difference with the approach of Cartaxo et al. [20] is that their approach tries to translate sequence diagrams to labelled transition systems, whereas our approach uses symbolic transition systems. However, symbolic transition systems are just labelled transition systems with data, data-dependent conditions, and variables added. When combined fragments, data, and variables are not present in the sequence diagram, the resulting intermediate model representation is just an LTS. This means that the steps remain mostly the same, except for a difference which can be spotted in step 4, where the proposed translation by Cartaxo et al. [20] contains extra edges and states in the form of *steps* and *expectedResults*. These edges carry the information of input and output, where the message edges occurring after *steps* are input, and the edges that occur after *expectedResults* are output.

An example of these extra edges can be seen in LTS lts_{m0}, in Figure 5.2, translated from sequence diagram sd_{m0}. However, in the case of a labelled transition system with inputs and outputs, explained in Section 4.1, these extra edges are not needed because the direction of the message can be encoded directly in the edge by using the prefixes ? or !. Also, as explained in Section 3.3.1, the lifeline that receives the input or sends the output should be encoded in the edge. Since all modelled communication is to or from the model-based testing tool (Host), the Host does not have to be encoded in the edge. When taking these described properties into account, it results in STS sts_{m0}, which can also be seen in Figure 5.2. The sequence diagram sd_{m0} in this figure shows lifeline a represented by class A, which is a class in a class diagram. The reasoning behind this will be explained in Section 5.3.2.

As explained in Section 3.3, the semantics of sequence diagrams boils down to a set of traces. However, as can be seen in step 4, the only traces that are included in the STS are the input traces that get sent by the Host, and the output traces that are received by the Host. This means that not all traces in the sequence diagram should be considered, but only the $ObservableTraces$, which are the sequences of messages sent or received by the Host, as these are the only events that can be observed from the environment.

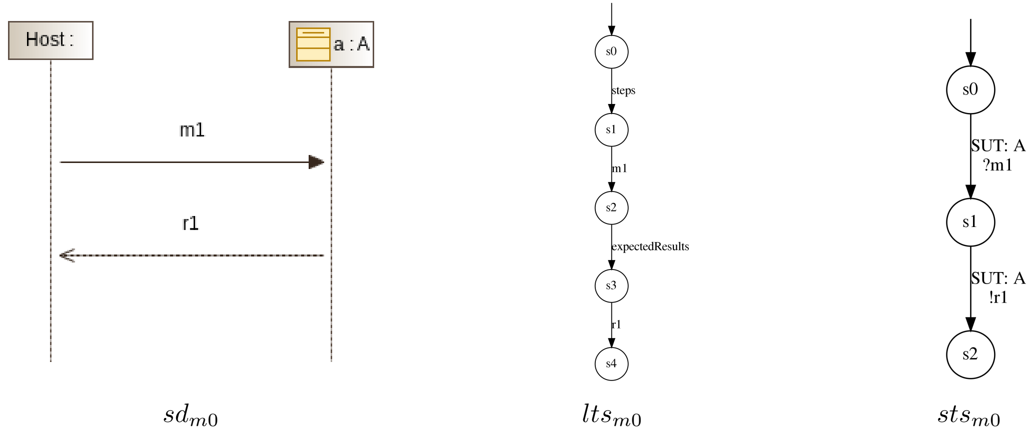


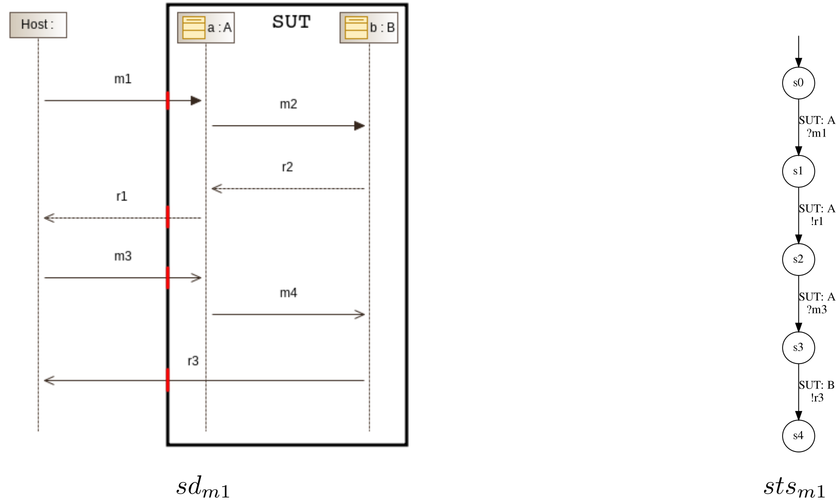
Figure 5.2: Transformation of sequence diagram sd_{m0} to lts_{m0} as proposed by Cartaxo et al. [20] and to sts_{m0} as proposed in this thesis

All messages in $ObservableTraces$ are either sent or received by the Host. Section 3.3.1 explains that all messages must occur ordered on the same lifeline, which is why we can abstract away from the sendEvents and receiveEvents and just write the message name instead of the events. Also, for readability, the examples have different message contents, such that it is possible to abstract away from the tuple notation, and the prefixes that describe input and output. Now the $ObservableTraces$ of sd_{m0}, written as $ObservableTraces(sd_{m0})$, is the set: $\{m1 \cdot r1\}$. The resulting STS sts_{m0} should be able to traverse the same paths that are present in the sequence diagram.

Figure 5.3 illustrates that only the observable traces are considered, where lifelines A and B are present in the sequence diagram sd_{m1}. These lifelines resemble the different interfaces of a SUT, but the collection of these interfaces should still be seen as a single black box. This means that there is no way to observe communication between the lifelines themselves. Only the messages $m1$, $r1$, $m3$, and $r3$ that pass through the border of the black box can be observed. The Host can still send and receive messages from all present interfaces, but cannot observe the internal communication: $m2$, $r2$ and $m4$, between the lifelines. This results in sts_{m1}, which produces completed traces $\{m1 \cdot r1 \cdot m3 \cdot r3\}$ that are observable for the Host. It is up to the modeller to decide whether to model internal communication or not. Omitting the internal communication reduces the amount of work to create a testable model. However, this internal communication may provide valuable documentation to developers or other stakeholders.

As explained in Section 3.3.2, combined fragments do not add new traces to the sequence diagrams, but can alter the ordering of the traces. If combined fragments are considered in the subset of used components in sequence diagrams, each interaction operator: *alt*, *loop* and *opt* require an extra rule.

5. For each interaction operand in a *alternative flow*, a transition must be created with label τ and the corresponding guard, as well as a transition where every guard is included in a negation of a disjunction, e.g., $(\neg A) \wedge (\neg B) \wedge (\neg C)$, to skip the combined fragment if none of the guards are satisfied. Also, when the combined fragment exits, a transition labelled τ for each operand is added to merge the multiple interaction operands.



sd_{m1} sts_{m1}

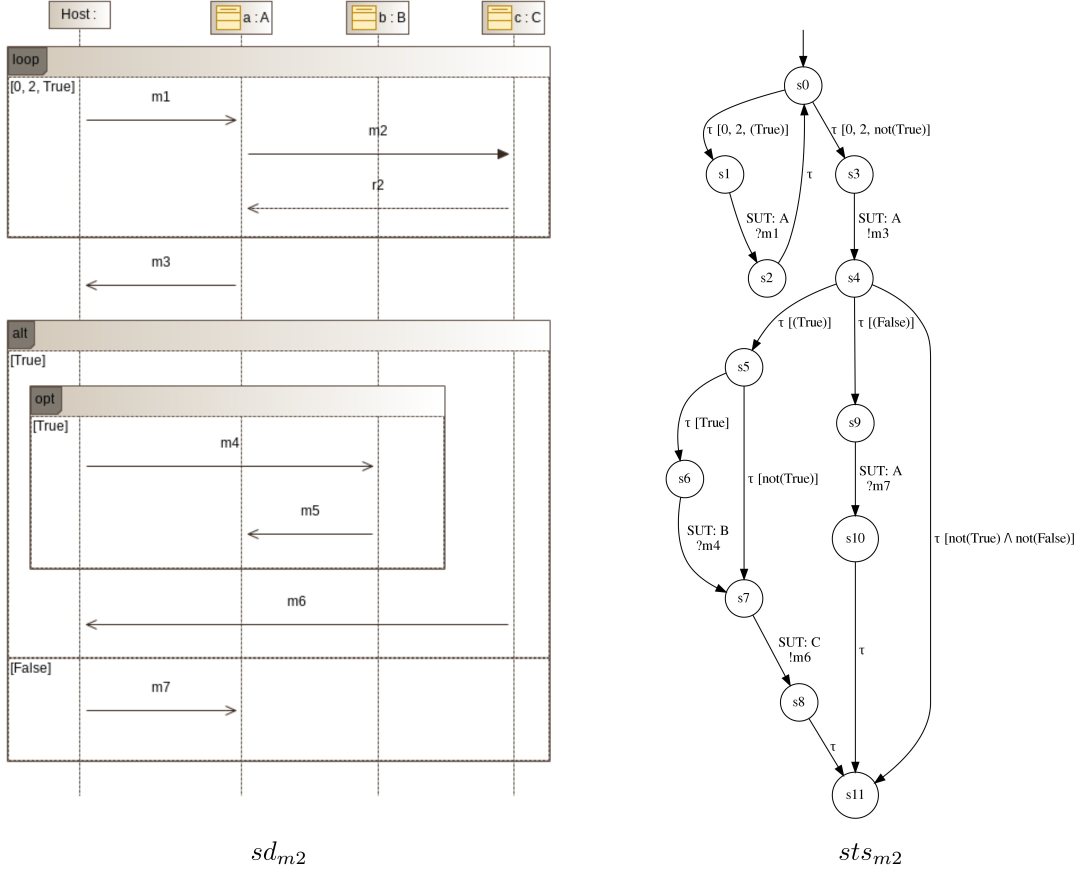
Figure 5.3: STS sts_{m1} generated from sequence diagram sd_{m1} with multiple interfaces (lifelines)

6. For each *conditional loop*, a transition labelled τ must be created, which includes the proper loop condition at the beginning of the loop. Another τ-transition must be created to go back to the initial state of the loop. Also, a τ-transition must be created to exit the loop, which includes the loop condition where the guard is negated.
7. For each *optional flow*, a transition labelled τ must be created, which includes the guard, as well as a τ-transition with a negated guard.

Figure 5.4 shows the complex sequence diagram sd_{m2} that contains multiple lifelines and several (nested) combined fragments. When creating the STS by applying the rules stated above, sts_{m2} is the result. Here, sts_{m2} illustrates that only the $ObservableTraces$ were included.

Cartaxo et al. [20] proposes a similar solution that only included the rules for the *alt* and *loop* operators. In their proposal, the condition is encoded in the label instead of in the guard. In our solution, the guards are separated from the labels, such that the τ-transitions can be eliminated in future work. If τ-transitions can be avoided by introducing more guards, the model gets smaller, which means that fewer transitions have to be traversed when using a model-based testing tool. Also, τ-transitions could add unwanted quiescence behaviour, which we may want to avoid. However, the 7-step algorithm to build an STS with the use of τ-transitions is easier to implement, especially when nested combined fragments with different operators are included in the sequence diagram.

For a sequence diagram that contains combined fragments, all possible $ObservableTraces$ in sequence diagram sd_{m2} can be found by computing the partial orders on its set of events [28]. This means that all possible combinations of satisfied guards have to be considered for each interaction operand in the combined fragments. This results in the following set of



sd_{m2} sts_{m2}

Figure 5.4: STS sts_{m2} generated from sequence diagram sd_{m2} with Combined Fragments

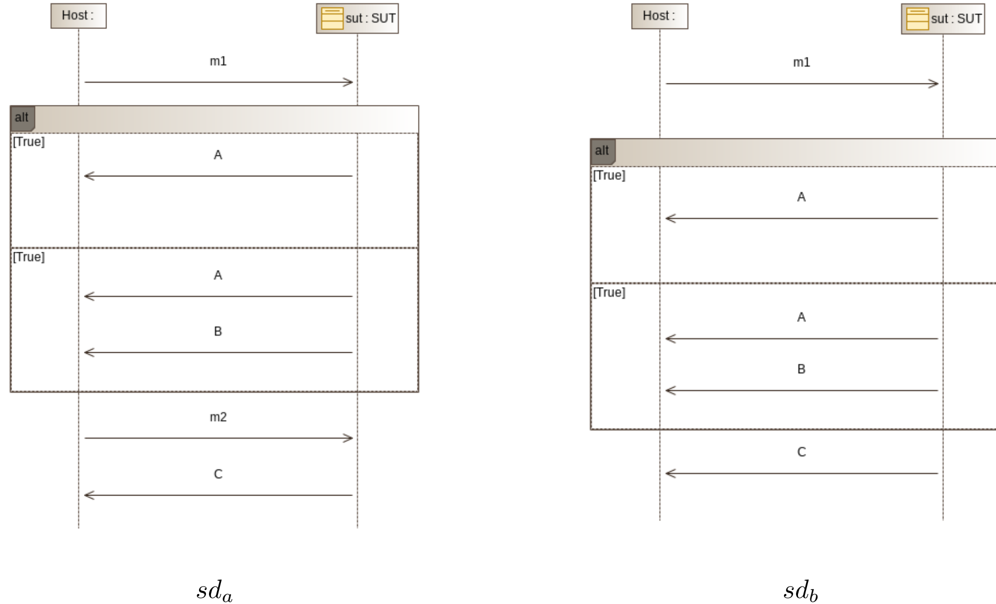
ObservableTraces, which should be the same as the τ-abstracted paths of sts_{m2}:

$$\begin{aligned} ObservableTraces(sd_{m2}) = \{\ & m1 \cdot m1 \cdot m3 \cdot m4 \cdot m6,\ m1 \cdot m3 \cdot m4 \cdot m6,\ m3 \cdot m4 \cdot m6, \\ & m1 \cdot m1 \cdot m3 \cdot m6,\ m1 \cdot m3 \cdot m6,\ m3 \cdot m6, \\ & m1 \cdot m1 \cdot m3 \cdot m7,\ m1 \cdot m3 \cdot m7,\ m3 \cdot m7, \\ & m1 \cdot m1 \cdot m3,\ m1 \cdot m3,\ m3\ \} \end{aligned}$$

As explained in Section 4.1, *Straces* include quiescence. However, sequence diagrams do not model quiescence explicitly. In sequence diagrams, input or output is always eventually expected, except for the final state. However, the final state will always exit the model. Thus, if quiescence is observed from an implementation, the timeout is set too short due to the system still being busy, or the system is unresponsive, resulting in an implementation is not input-output conform with the specification.

Sequence diagrams allow for modelling non-determinism. However, TorXakis is not always able to handle non-determinism. This means that it is possible to model a specification that conforms to the implementation, whereas TorXakis will report a failure during testing. An example of this can be seen in Figure 5.5, where each sequence diagram contains a combined fragment with the *alt* operator. The interaction operands in these combined fragments both have a guard that evaluates to True, meaning that one of the interaction operands will be

executed non-deterministically. However, this can result in race conditions. This can be seen in sequence diagram sd_a, which shows a diagram that is faulty. Here, TorXakis will first send a message $?m1$, which will be answered by either $!A$ or $!A \cdot !B$. However, as soon as $!A$ has been received, TorXakis will send $?m2$, because the first interaction operand conforms to this behaviour. If the SUT then still replies with $!B$, it will trigger a fail in TorXakis because a $!C$ was expected. sd_b does not trigger a race condition, as TorXakis expects only output, which is why TorXakis will not send an unexpected message to the SUT.



sd_a sd_b

Figure 5.5: Example race condition with non-deterministic sequence diagrams

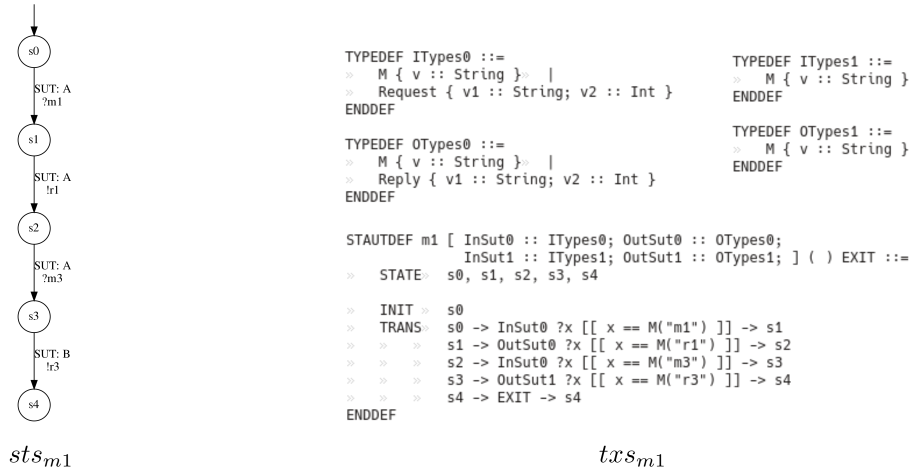
This example shows that it is possible to model incorrect models. TorXakis expects synchronous communication between Host and SUT. However, it is possible to model asynchronous communication. Often this will work fine, but when combined fragments with multiple interaction operands are present, and the model then offers a choice between input and output, race conditions can occur when multiple guards evaluate to True. There are multiple ways to combat this. An option is to enforce that multiple guards can never evaluate to True, to eliminate non-determinism that leads to race conditions. Another option is to build in a check in the sd2txs tool, which monitors if, at any time, a choice between input and output is offered. An option would also be to explicitly model asynchronous communication, by e.g., queues in the TorXakis model [34]. When these non-deterministic combined fragments are not used, all events occur ordered, deterministic, and without quiescence. This means that if the model is modelled correctly and TorXakis reports that a test failed, the implementation is not input-output conform with the specification.

5.2 From STS to model-based testing with TorXakis

It is now possible to use the created STS for the generation of a `.txs` input file, which can be used with the model-based testing tool TorXakis. As mentioned in Section 4.4.7, TorXakis supports the state automation definition (STAUTDEF). The STS is used to translate to such a STAUTDEF, along with some types and channels. Figure 5.6 shows such a translation from

sts_{m1} to txs_{m1}, where txs_{m1} includes some TYPEDEF and a STAUTDEF. The STAUTDEF consists out of the STATE, INIT and TRANS segment where:

- The STATE definition in txs_{m1} includes every state present in sts_{m1};
- The INIT definition shows the initial state s_0 and holds the initialization of the variables;
- The TRANS definition contains the transitions of the model with format: *state* $\rightarrow$ *communication* $\rightarrow$ *state*;



sts_{m1} txs_{m1}

Figure 5.6: TorXakis STAUTDEF txs_{m1} generated from STS sts_{m1}

Here, the left-most *state* resembles the state connected to the beginning of a transition in sts_{m1}, and the right-most *state* section resembles the state connected to the end of this transition. The *communication* section resembles what communication should occur when the transition is traversed. The communication section consists of:

- The name of the channel, e.g. *InSut0*;
- A value prefixed with the '?' or '!' operator where:
  - The operator ! denotes that the data item is fully specified: a known value is communicated;
  - The operator ? denotes that the data item is not fully specified: (part of) the value that is communicated is unknown;
- A condition (optional), encompassed within square brackets '[[ ]]';
- A variable update section (optional), encompassed within curly brackets '{ }', where the values of variables can be altered;

As can be seen in txs_{m1}, the communication section can also hold keyword EXIT, which exits this current state automation definition. In the example, no variables are present, which is why the INIT section does not contain any initialization of variables. Also, only the prefix '?' will ever be used, as most of the time, the data items are not fully specified. When they are fully specified, it is shown in the condition, where $x == M(''m1'')$ means that the variable

x should equal $M("m1")$, which results in the string $M("m1")$ being sent as input on channel `InSut0`.

In txs_{m1}, multiple channels are shown: `InSut0`, `OutSut0`, `InSut1` and `OutSut1`. These channels map to the lifelines in sd_{m1} in Figure 5.3, where every lifeline functions as an interface that is able to communicate by using their input and output channels, coupled to a specific IP-address and port. To support multiple different message formats on a channel, each channel must have its own type definition. This is where the TYPEDEFs in txs_{m1} come in. When types are not explicitly specified for messages in sequence diagrams, the default type $M(v)$ is used, where v is a string that contains the message as a string. The *Request* and *Reply* fields in the TYPEDEFs are examples of other message types. Section 5.3.2 explains how these other message types can be defined.

Figure 5.7 illustrates STS sts_{m2} with combined fragments, generated from sd_{m2} (see Figure 5.4). Combined fragments in a sequence diagram introduce τ-transitions in the resulting STS. It is possible to represent these τ-transitions TorXakis model by using the ISTEP keyword. The keyword ISTEP only replaces the name of the channel and the value. This means that it is possible to append both conditions and variables to an ISTEP transition. Sequence diagram sd_{m2} also contains an extra lifeline. This means that two more channels are added: `InSut2` and `OutSut2`. These channels also have its own type definitions, namely: `ITypes2` and `OTypes2`. Other than these two extra channels, the Host in sd_{m2} communicates with the same lifelines as in txs_{m1}, which means that the channels and types defined in txs_{m1} have to be used.

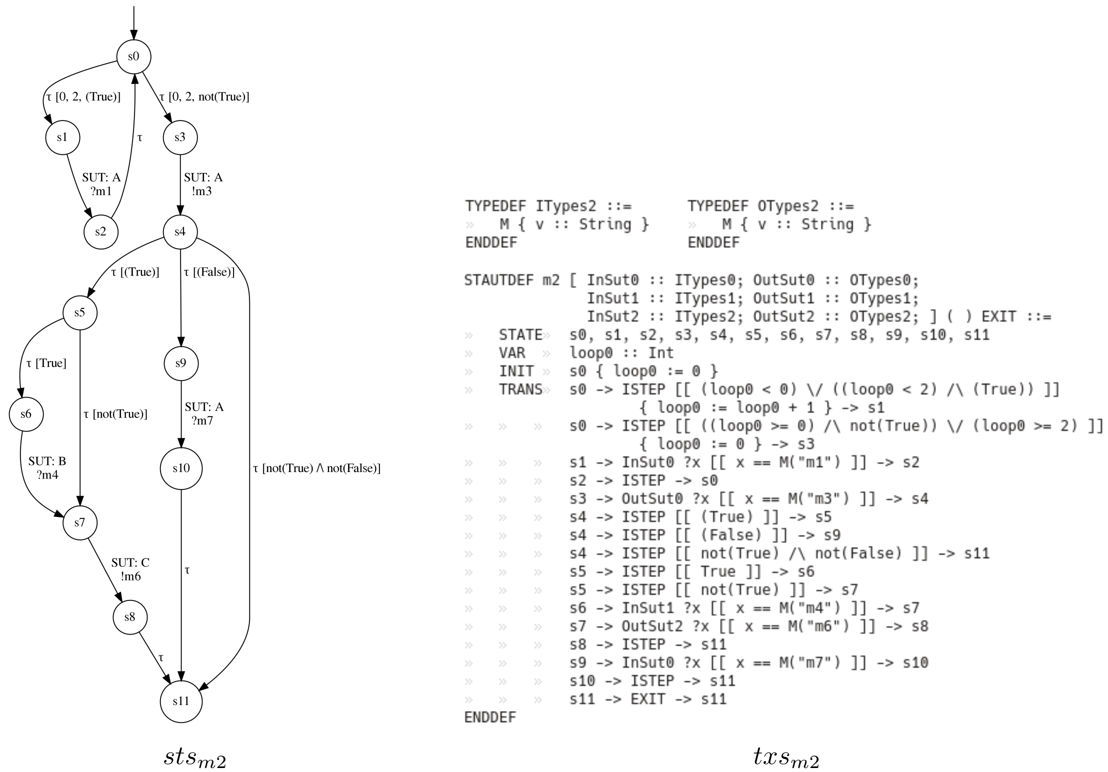


Figure 5.7: TorXakis STAUTDEF txs_{m2} generated from STS sts_{m2}

Just generating the STAUTDEFs and TYPEDEFs is not enough to compile the `.txs` file and test with the model-based testing tool TorXakis. For this, three more segments have to be generated, namely: CHANDEF, MODELDEF, and CNECTDEF. CHANDEF is explained in Section 4.4.1 and describes the mapping between the channels and types. MODELDEF is explained in Section 4.4.2 and contains all channels that are included in the model and which direction they have (input or output). CNECTDEF is explained in Section 4.4.3 and contains the IP addresses and ports of the SUT(s), and which input and output channels belong to which SUT. Figure 5.8 shows the generated CHANDEF, MODELDEF, and CNECTDEF.

```
CHANDEF ChanDefs ::=
    InSut0  ::  ITypes0;
    InSut1  ::  ITypes1;
    InSut2  ::  ITypes2;
    OutSut0 ::  OTypes0;
    OutSut1 ::  OTypes1;
    OutSut2 ::  OTypes2;
ENDDEF
```

```
MODELDEF Model ::=
    CHAN IN     InSut0, InSut1, InSut2
    CHAN OUT    OutSut0, OutSut1, OutSut2

    BEHAVIOUR   overarching_proc [
        InSut0, OutSut0,
        InSut1, OutSut1,
        InSut2, OutSut2 ] ( )
ENDDEF
```

```
CNECTDEF Sut ::=
    CLIENTSOCK

    CHAN OUT    InSut0  HOST "localhost" PORT 9990
    ENCODE      InSut0  ?x -> !toString(x)
    CHAN IN     OutSut0 HOST "localhost" PORT 9990
    DECODE      OutSut0 !fromString(x) <- ?x

    CHAN OUT    InSut1  HOST "localhost" PORT 9991
    ENCODE      InSut1  ?x -> !toString(x)
    CHAN IN     OutSut1 HOST "localhost" PORT 9991
    DECODE      OutSut1 !fromString(x) <- ?x

    CHAN OUT    InSut2  HOST "localhost" PORT 9992
    ENCODE      InSut2  ?x -> !toString(x)
    CHAN IN     OutSut2 HOST "localhost" PORT 9992
    DECODE      OutSut2 !fromString(x) <- ?x
ENDDEF
```

Figure 5.8: CHANDEF, MODELDEF and CNECTDEF

Note that BEHAVIOUR field in the MODELDEF calls a process definition call *overarching_proc*. The reason for this is that every sequence diagram translates to a single STAUTDEF in the full `.txs` model file. To support being able to test multiple sequence diagrams, an overarching process definition has to be defined with a recursive implementation. Its implementation chooses any of the models non-deterministically and executes it. When the model has finished executing and has exited, the overarching process definition recursively calls itself, enabling it to choose another model to execute. This process repeats itself until it is manually halted. Figure 5.9 shows such an overarching process definition where all the previously defined models $m0$, $m1$ and $m2$ are included in the non-deterministic choice options.

```
PROCDEF overarching_proc  [ InSut0 :: ITypes0; OutSut0 :: OTypes0;
                            InSut1 :: ITypes1; OutSut1 :: OTypes1;
                            InSut2 :: ITypes2; OutSut2 :: OTypes2; ] ( ) ::=
    (
        m0 [ InSut0, OutSut0 ] ( )
    ##
        m1 [ InSut0, OutSut0, InSut1, OutSut1 ] ( )
    ##
        m2 [ InSut0, OutSut0, InSut1, OutSut1, InSut2, OutSut2 ] ( )
    ) >>> overarching_proc [ InSut0, OutSut0, InSut1, OutSut1, InSut2, OutSut2 ] ( )
ENDDEF
```

Figure 5.9: Sequence diagram nested combined fragments

5.3 Extra features

This section explains the extra features added to the tool sd2txs. These features range from graph visualization to support for user-defined typed message formats, variables, and expressions.

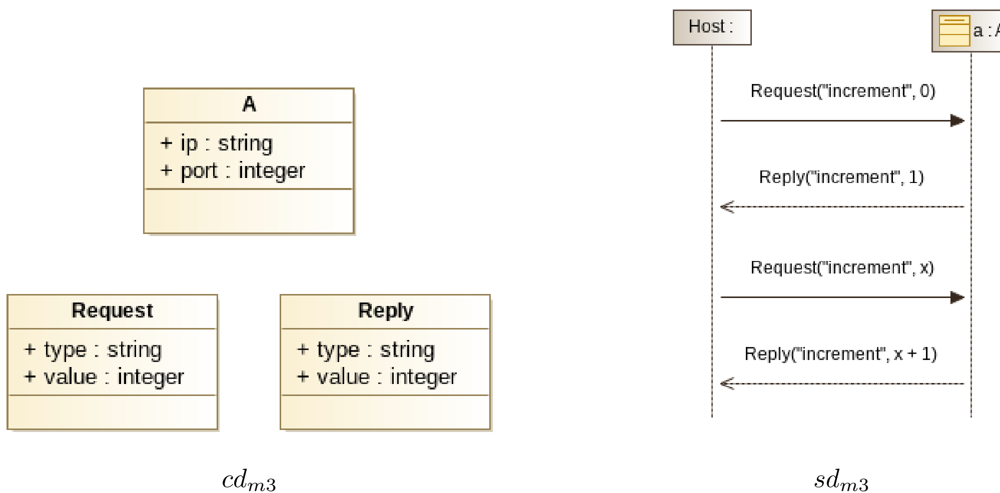
5.3.1 Visualize STS with Graphviz

The tool sd2txs has built-in support to visualize the generated STSs. Each separate STS is used as input to generate a `.dot` file. The dot format is one of the supported formats by the open-source graph visualization software Graphviz[12], which represents the structural information as diagrams of abstract graphs and networks. Most figures with labelled or symbolic transition systems in this thesis were generated from sequence diagrams by using the built-in visualization feature.

5.3.2 User-defined typed message formats, variables and expressions

TorXakis communicates with sockets, which is why the CNECTDEF segment in a TorXakis model file requires an IP-address and port to know how to communicate with a SUT. At first, the idea was to supply this information as arguments when executing the tool. However, the opportunity to supply the IP-address and port to a class came with the introduction of class diagrams. Figure 5.10 shows class diagram cd_{m3}. The class diagram contains class A, with attributes *ip* and *port*. This is also why all the lifelines in this chapter, other than Host, have a yellow box next to their name, which illustrates that the lifeline represents a class in a class diagram.

Along with the IP-addresses and ports, class diagrams also allow for user-defined message formats. TorXakis is able to automatically generate values for the basic data types Bool, Int and String. This means that when the type is known by using a typed attribute in a class, the user can supply a static value or define a variable to take advantage of this. Figure 5.10 shows such an example, where the user-defined message formats *Request* and *Reply* are defined within class diagram cd_{m3}. Both message formats have the same two attributes: *type* with type string and *value* with type integer, resulting in message formats $Request(string, integer)$ and $Reply(string, integer)$.



cd_{m3} sd_{m3}

Figure 5.10: Class diagram cd_{m3} and sequence diagram sd_{m3}

The use of these message formats is illustrated in sd_{m3}, where the first synchronous request and reply show the static use of the message formats. Here the *type* parameter of the *Request*

[12] https://www.graphviz.org/

message has value “increment”. The quotes show that a string value is supplied instead of a variable. The *value* parameter also has a static value of 0. The *Reply* message illustrates the expected result, where that the “increment” string should be echoed back, and that the *value* is incremented by 1. The second synchronous request and reply show the combined use of static and dynamically valued parameters. Here the *type* parameter still shows use of a static value “increment”. However, the *value* parameter does not contain an integer but a variable. This lets TorXakis know that a random integer value has to be generated for this variable.

When defining a variable in a sequence diagram, it gets saved in the TorXakis model. This means that the variable can be reused in other message parameters that have the same type. These variables can also be used in expressions. The second *Reply* message in sd_{m3} illustrates this, where the expected integer output parameter *value* should contain value: $x+1$, meaning that in the expected output, the randomly generated variable x should have been incremented with 1.

Figure 5.11 shows the TYPEDEFS with types *Request* and *Reply* generated from class diagram cd_{m3} and the STAUTDEF generated from sequence diagram sd_{m3}. Here, the VAR segment shows that a variable $vi0$ has been defined with type Int, denoting the *type* variable x. In the INIT section, the variable $vi0$ gets initialized to value 0. On the third line of the TRANS segment is generated from $Request(\text{“}increment\text{”}, x)$. Here, the variable $vi0$ gets initialized to the value of $v2(x)$, where it looks at which type in which channel is being used and returns the value of the variable with that name. In this example this is channel `InSut0` which has type `ITypes0`. The guard `isRequest(x)` denotes that type *Request* is used. In this type, variable *v2* exists, which is then returned. Also note that the guard $v1(x) == \text{“}increment\text{”}$ enforces that the variable *v1* in type *Request* has to contain value *“increment”*.

The fourth line of the TRANS segment is generated from $Reply(\text{“}increment\text{”}, x+1)$. Here, the guard checks that variable *v1* in type *Reply* contains value *“increment”*. The guard checks that variable *v2* in type *Reply* equals value $vi0+1$, where $vi0$ is the variable previously assigned to the value of x.

```
TYPEDEF ITypes0 ::=
»   M { v :: String }»  |
»   Request { v1 :: String; v2 :: Int }
ENDDEF

TYPEDEF OTypes0 ::=
»   M { v :: String }»  |
»   Reply { v1 :: String; v2 :: Int }
ENDDEF

STAUTDEF m3 [ InSut0 :: ITypes0; OutSut0 :: OTypes0; ] ( ) EXIT ::=
»   STATE»  s0, s1, s2, s3, s4
»   VAR  »  vi0 :: Int
»   INIT »  s0 { vi0 := 0 }
»   TRANS»  s0 -> InSut0 ?x [[ x == Request("increment", 0) ]] -> s1
»   »    »  s1 -> OutSut0 ?x [[ x == Reply("increment", 1) ]] -> s2
»   »    »  s2 -> InSut0 ?x [[ IF isRequest(x) THEN (v1(x) == "increment") ELSE False FI ]] { vi0 := v2(x) }  -> s3
»   »    »  s3 -> OutSut0 ?x [[ IF isReply(x) THEN (v1(x) == "increment") /\ (vi0 + 1 == v2(x)) ELSE False FI ]] -> s4
»   »    »  s4 -> EXIT -> s4
ENDDEF
```

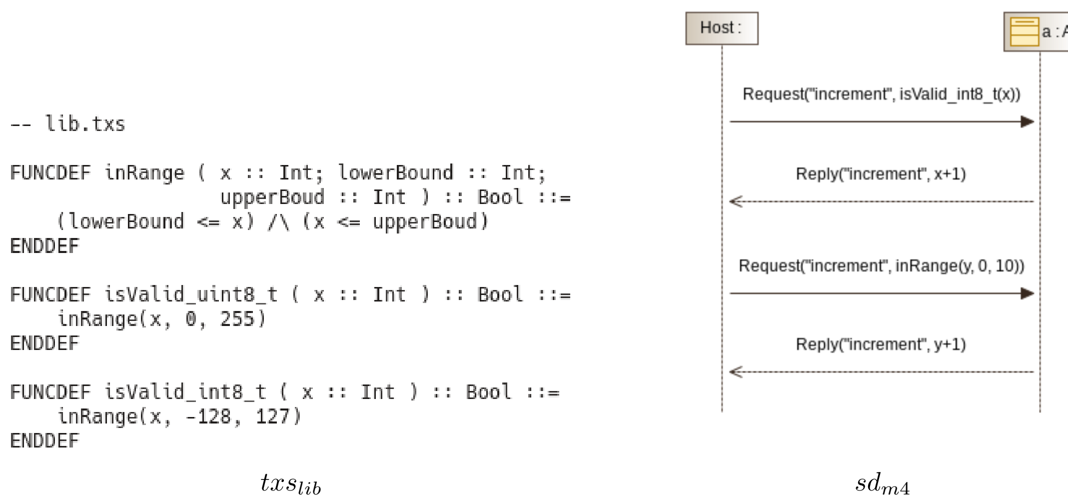
Figure 5.11: TorXakis input model txs_{m3} with types

The UML specification mentions an alternative way to assign values to variables [26]. This alternative assignment would work for static value assignment, e.g. *Request(“increment”, x = 1)*, where variable x will now hold value 1. However, when we want to let TorXakis generate a value, this syntax does not suffice. This is why the current syntax was adopted. We could still adopt the UML specification syntax by specifying what kind of variable we want TorXakis to generate. For example, if TorXakis has to generate an integer, we could define *Request(“increment”, x = Int)*. This enables reassignment of variables and makes it more clear to a user if a variable is fresh or used. A problem still remains with scoping of

variables. For example, if a variable is declared in a combined fragment and used outside of this fragment, it is possible that the combined fragment was skipped. This means that the variable could have been referenced before it was assigned. To solve this, a compiler would need to be added to the sd2txs tool that checks for scoping issues when variables are referenced before assignment.

5.3.3 Callable (user-defined) TorXakis library functions

To allow for constraints on generated parameters, support for callable (user-defined) TorXakis library functions has been added. This means a user can call a function, defined in a different `.txs` files, from a sequence diagram by satisfying its signature. These different `.txs` files can either be predefined libraries of functions or files with specialized user-defined functions. Figure 5.12 gives an example of a few library functions in txs_{lib}, which are included in the `lib.txs` file, which can be found in the tool. The user is completely free to add more functions to this library or other self-defined libraries. Sequence diagram sd_{m4} shows how the library functions can be called. For TorXakis to find the library functions, TorXakis has to be started with the library file included in the command, e.g., `torxakis lib.txs model.txs`.



txs_{lib} sd_{m4}

Figure 5.12: Call (user-defined) TorXakis library functions *inRange* and *isValid_int8_t* in txs_{lib} from sequence diagram sd_{m4}

6 Practical Example: Calculator

This chapter will give a practical example to illustrate how an actual application can be tested by using sequence diagrams, class diagrams and the model-based testing tool TorXakis. The practical example comes in the form of a Python calculator application that supports the following four operations:

```
1  def add(x, y):
2      return x + y
3
4  def subtract(x, y):
5      return x - y
6
7  def multiply(x, y):
8      return x * y
9
10 def divide(x, y):
11     if y != 0:
12         return int(x // y)
13     return 'division by zero'
```

The supported operations are modelled in the class diagram $cd_{calculator}$ in Figure 6.1. To call an operation, the user-defined message format *Expression* has been created that contains a parameter *lhs* with type integer, a parameter *operator* with type string and a parameter *rhs* with type integer. Examples of the messages can be seen in $sd_{multiply}$. Here, the operations themselves are internally modelled, which means that they cannot be observed from the outside. However, the results are still observable because the adapter encodes the result in the user-defined message format *Result* that contains an integer value as a parameter.

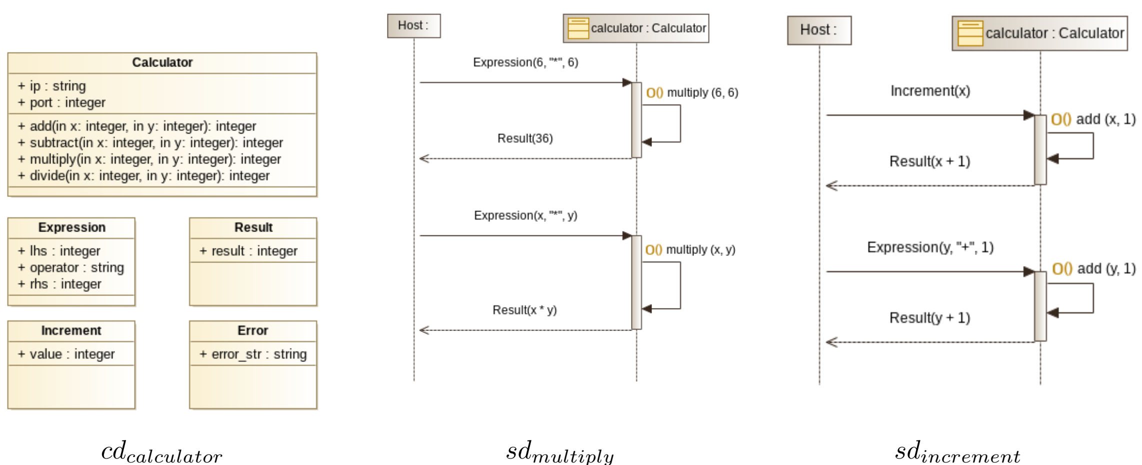


Figure 6.1: Calculator application modelled in UML

The sequence diagram $sd_{multiply}$ shows two input messages with format *Expression*. The first one illustrates a message with only static values, where the goal is to test if the calculator

produces the correct result 36, when it calculates $6 * 6$. The second message with format *Expression* shows a message with static and dynamic values, where the operator is statically defined, and the values will be dynamically generated. This example tests if the calculator returns the value of $x \times y$ for any values of x and y.

Sequence diagram $sd_{increment}$ shows that the same thing can be modelled in different ways. Here, an extra message format *Increment*, containing a parameter *value* with type integer is used, which can also be seen in $cd_{calculator}$. In $sd_{increment}$ The first input message uses the *Increment* message format, where the result should contain input value incremented by 1. However, it is also possible to model this by using the *Expression* message format, where only one of the parameters is dynamically generated. This example shows that a user is free to choose his/her own message formats.

The divide operation has to be modelled differently, as division by 0 is not allowed. Sequence diagram $sd_{division}$ in Figure 6.2 shows this, where the first *Expression* message explicitly tests division by zero. Here, the expected output is a message *Error*, with *error_str* "division by zero". The second *Expression* message has dynamic variables y and z. This means that most of the time, the calculator will return the value of $\frac{y}{z}$. However, the slight chance exists that the generated value of z will equal 0. This is where a combined fragment with the interaction operator *alt* can be used. Here, we model that we expect the error message as output when $z == 0$, and the result message with value $\frac{y}{z}$ when $not(z == 0)$.

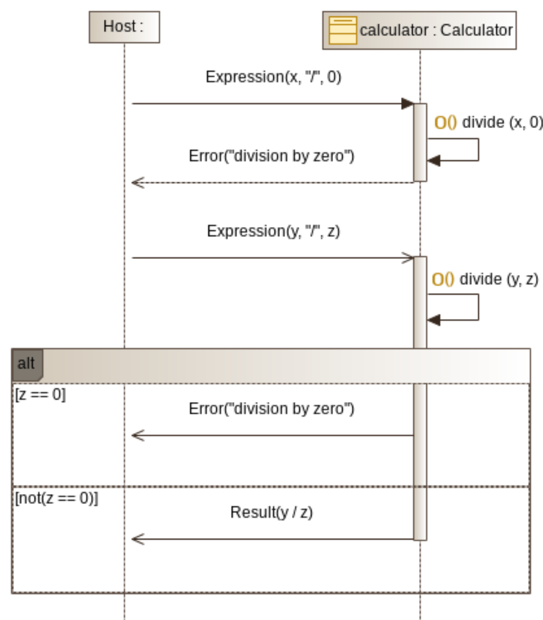


Figure 6.2: Sequence diagram $sd_{division}$

For the model-based testing tool to be able to communicate with the calculator application, an adapter is needed. The adapter contains the code that communicates with sockets, parses the messages, calls the corresponding functions and builds the reply messages. The code below shows how the messages are parsed, how the functions are called, and how the reply messages are being built for the calculator application.

```python
def handle(self, current_test):
    # Extract function name before brackets
    message_name = current_test[:current_test.index('(')]
    # Extract parameters
    arguments = current_test[current_test.index('(')+1:-1]
    # Remove whitespace
    arguments = arguments.replace(' ', '')
    # Split on comma's
    arguments = arguments.split(',')
    if message_name == 'Expression':
        # Convert to tuple
        x, op, y = tuple(arguments)
        # Convert x and y to integers
        x = int(x)
        y = int(y)
        if op == '"+"':
            return 'Result(' + str(calculator.add(x, y)) + ')'
        elif op == '"-"':
            return 'Result(' + str(calculator.subtract(x, y)) + ')'
        elif op == '"*"':
            return 'Result(' + str(calculator.multiply(x, y)) + ')'
        elif op == '"/"':
            r = calculator.divide(x, y)
            if type(r) == int:
                return 'Result(' + str(r) + ')'
            else:
                return 'Error("' + str(r) + '")'
    elif message_name == 'Increment':
        x = int(arguments[0])
        return 'Result(' + str(calculator.add(x, 1)) + ')'
    else:
        raise LookupError('Message: ' + message_name + ' not supported')
```

Now everything is ready to start with the model-based testing process. First, the calculator application has to be started, which waits for a connection with the model-based testing tool. Then, when the model-based testing tool is started with the corresponding model file, the command `tester Model Sut` can be executed to create the connection between the calculator and model-based testing tool. Finally, the command `test` n can be executed, where n is the number of test steps. The output of TorXakis is shown below:

```
TXS >> tester Model Sut
TXS >>  Tester started
TXS >> test 10
TXS >>  .....1: IN: Act { { ( InSut0, [ Expression(-69,"/",0) ] ) } }
TXS >>  .....2: OUT: Act { { ( OutSut0, [ Error("division by zero") ] ) } }
TXS >>  .....3: IN: Act { { ( InSut0, [ Expression(51,"/",53) ] ) } }
```

```
TXS >>  .....4: OUT: Act { { ( OutSut0, [ Result(0) ] ) } }
TXS >>  .....5: IN: Act { { ( InSut0, [ Increment(-93) ] ) } }
TXS >>  .....6: OUT: Act { { ( OutSut0, [ Result(-92) ] ) } }
TXS >>  .....7: IN: Act { { ( InSut0, [ Expression(69,"+",1) ] ) } }
TXS >>  .....8: OUT: Act { { ( OutSut0, [ Result(70) ] ) } }
TXS >>  .....9: IN: Act { { ( InSut0, [ Expression(6,"*",6) ] ) } }
TXS >>  ....10: OUT: Act { { ( OutSut0, [ Result(36) ] ) } }
TXS >>  PASS
```

The output of the calculator adapter is shown below, which shows that the ten tests (inputs and outputs) have passed.

```
Starting listening on localhost port 9990
1: IN: Calculator, Expression(-69,"/",0)
2: OUT: Calculator, Error("division by zero")
3: IN: Calculator, Expression(51,"/",53)
4: OUT: Calculator, Result(0)
5: IN: Calculator, Increment(-93)
6: OUT: Calculator, Result(-92)
7: IN: Calculator, Expression(69,"+",1)
8: OUT: Calculator, Result(70)
9: IN: Calculator, Expression(6,"*",6)
10: OUT: Calculator, Result(36)
Other side closed connection, closing socket...
[Pass] -> Passed 10 tests
```

7 Case Study: Stairlift Remote and Receiver

This chapter describes the case study that has been performed, to test the created tool on a real embedded product. The system under test is a radio frequency (RF) remote and receiver module, used to control a stairlift. The remote has three buttons: up, down and park, which will act as the inputs of the SUT. In the tested configuration, the RF remote sends commands on a frequency of 915MHz when a button is pressed. The remote will repeat the commands with an interval of 60 milliseconds for as long as the button is pressed, after which at most 11 stop commands will be sent with the same interval.

Figure 7.1 shows the architectural diagram that illustrates the connectivity between all components. Here it can be seen that the remote is controlled by the USB-OPTO-RLY88 relay module to automate the button presses. The relay module can be controlled via serial communication by using a Universal Asynchronous Receiver-Transmitter (UART). The receiver is connected to an FTDI USB serial cable to read the received commands. Both the relay module and FTDI cable can be accessed by using the pyserial[13] Python library, which is why both adapters have been created with the Python programming language.

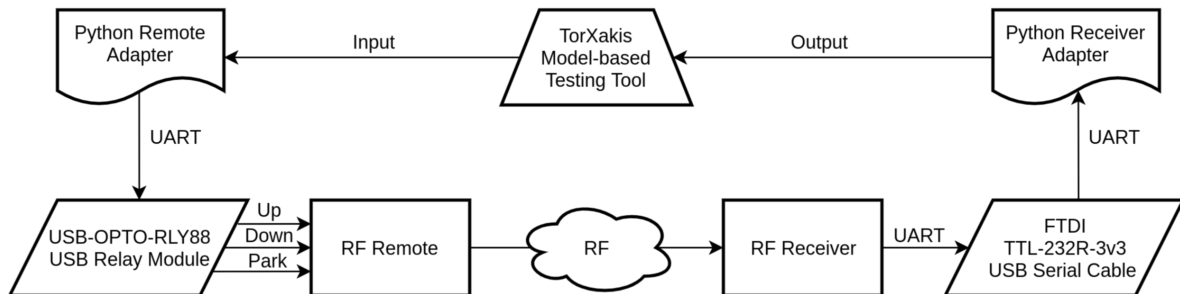


Figure 7.1: Architectural Diagram Case Study

Figure 7.2 shows the hardware setup with, from left to right: RF Receiver, USB Relay Module and the RF Remote.

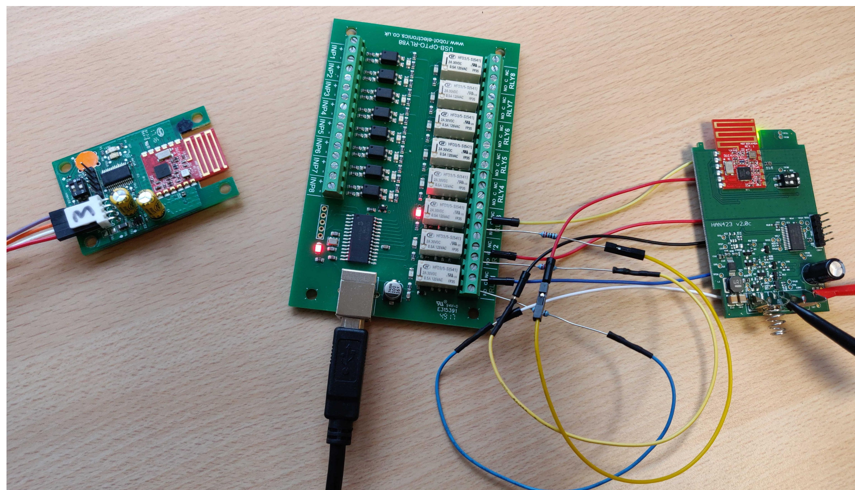


Figure 7.2: Hardware Setup Case Study

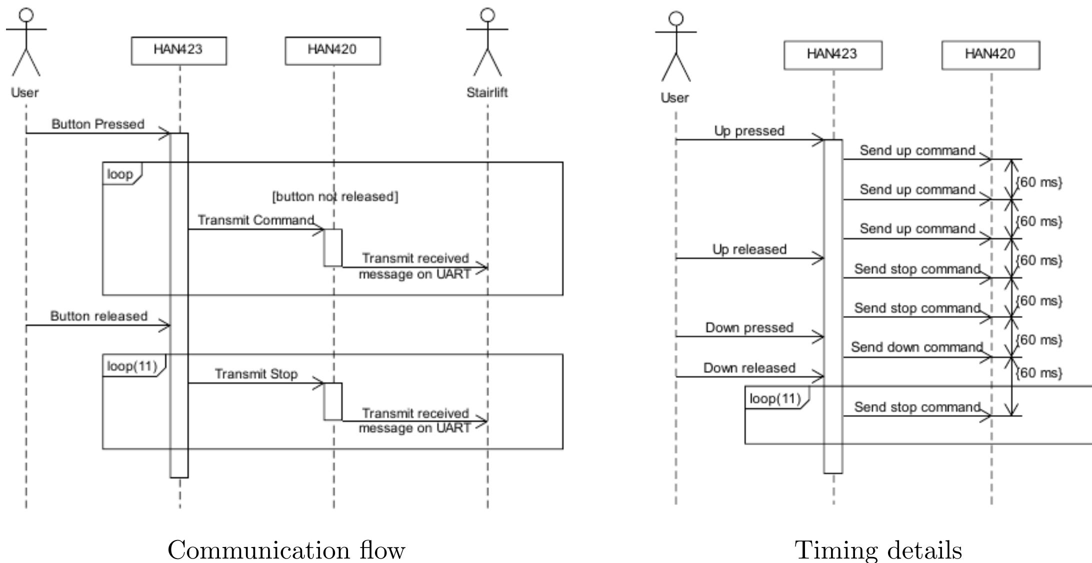
[13]https://pypi.org/project/pyserial/

7.1 Protocol

The interface protocol specification is a document created by Inspiro to explain the technical details. This document contains some sequence diagrams to illustrate the communication between remote and receiver, which can be seen in Figure 7.3. The document also explains that the RF remote can send the following commands:

- Lift command up, Hex: 0xEE
- Lift command down, Hex: 0xED
- Lift command park, Hex: 0xE7
- Lift command stop, Hex: 0xEB
- Security command (optional), Hex: 0xFB

The security command is only included with the first 10 of every 100 messages of a consecutive sequence and is only useful when infrared remotes are used. The interface protocol specification states that, when a button is pressed, the expected number of received commands will be based on the corresponding lift command and the time the button is pressed (see timing details in Figure 7.3). The specification also states that when the button is released, 11 stop commands will be sent. The exact reason why specifically 11 stop commands are sent is unknown, because the used protocol was adopted from an earlier revision of the remote, which was not made by Inspiro. However, this earlier revision of the remote used infrared to communicate, which is why it is speculated that the reason for the high stop command count was to be sure that the receiver receives at least one stop command in case signals get blocked. To give an example: if the "up" button is pressed for 1.2 seconds, we expect to receive $\frac{1.2}{0.06} = 20$ up commands, 0 down commands, 0 park commands, 11 stop commands and (optionally) 10 security commands.



Communication flow | Timing details

Figure 7.3: Interface protocol specification sequence diagrams

The initial setup tests, quickly showed that there were always fewer up, down or park commands received than initially expected. After some research and looking at the electrical

schematics, it became clear that the microcontroller on the remote is only powered when a button is pressed. This means that the microcontroller has to start up and initialize before the RF transmission can begin. Measurements showed that it took an average of 217 milliseconds to receive the first message in the receiver adapter when a button is pressed. The actual time the microcontroller needs to start up and initialize is lower with around 137 milliseconds, as the previously mentioned time of 217 milliseconds includes the transmission, processing and reading of the RF signal.

Another detected anomaly was that the number of stop commands received, varied between 11 to 14, instead of the documented 11 stop commands. Asking about this and some digging in the code revealed that the receiver sends stop commands itself when no more commands are received within a certain amount of time. Depending on the timing of the last received command, 1 to 3 extra stop commands are sent to the stairlift for safety purposes. This is why the number of stop commands received, differed from the documented 11 stop commands. The problem with these findings is that the number of commands that will be received varies non-deterministically. For instance, the microcontroller can start up more quickly than expected, resulting in more received commands. Moreover, the number of received stop commands can differ with each transmission, or commands can be lost due to the unreliable nature of RF transmissions.

7.2 Different modelling strategy due to timing constraints

At first, the idea was to create sequence diagrams that closely resemble the sequence diagrams in Figure 7.3. The reason for this is that these sequence diagrams give proper documentation of how the communication protocol should work. This yielded the sequence diagram shown in Figure 7.4.

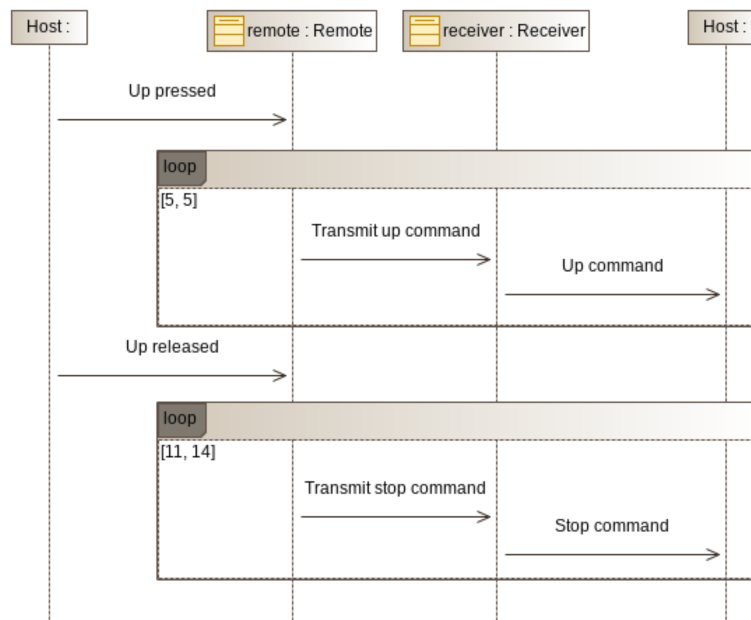


Figure 7.4: Initial sequence diagram

However, after creating the adapters and running the tests, it became clear that the testing process could not keep up with the communication speed between the RF remote and receiver. The button was sometimes not released fast enough, such that no more up commands would be received. Because the sequence diagram states that no more up commands should be received after the button is released, TorXakis gives an error that it received an up command but expected a stop command. This is unfortunate, as the sequence diagram in Figure 7.4 diagram provides a clear overview of how the protocol works, which why it would be useful to be able to use this diagram for testing to be able to test what is documented.

Measurements seem to confirm that the 60-millisecond window between transmissions is often too small to release the button before another command is received. This problem exists because of a combination of actions, which introduce extra latency between each transmission. Figure 7.5 shows the actions that have to be performed between each transmission. These actions are numbered and clarified below:

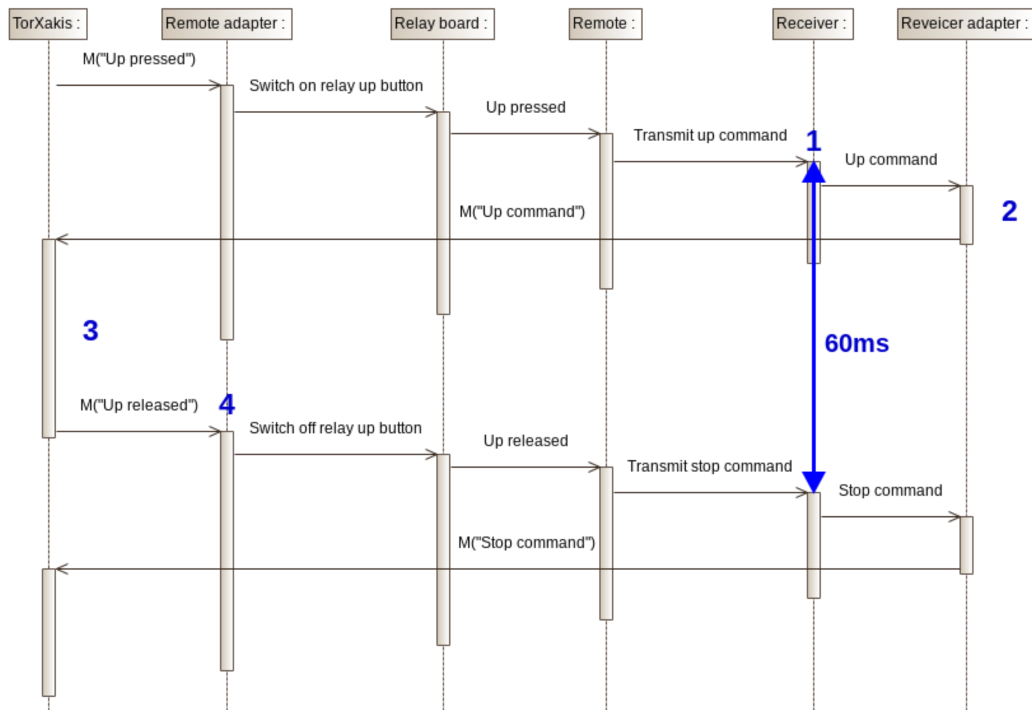


Figure 7.5: Actions between transmissions

1. Receive and process the message on the receiver and send the serial data to the receiver adapter;
2. Process the received data on the receiver adapter and send a response to TorXakis via sockets;
3. Receive and process the response in TorXakis, move to the next state in the STAUTDEF and send a release request from TorXakis to the remote adapter via sockets;
4. Receive and process the request in the remote adapter and send a command to release the button(s) to the relay board;

Step 3 seems to introduce the most latency, where measurements between 15-35ms were taken. The difference between the lowest and highest value in this range shows that the latency TorXakis and the socket communication introduces, can vary quite a bit. The other actions also introduce some latency, which is harder to measure because these steps are mostly performed on hardware. This case study shows that, when testing real-time systems, the added actions could introduce enough latency to let the test fail while it should not have failed. This means that when dealing with (embedded) software where timing constraints are below 100 milliseconds, TorXakis is often not fast enough to test a system.

7.3 Requirements

The requirements specification created by Inspiro does not mention specific requirements to test for the RF remote and receiver. As explained in the previous sections, testing with the exact number of expected commands is also not feasible. This is why we have defined our own requirements based on the interface protocol specification. The requirements, shown in Table 1, are grouped into categories and sorted by identifier. The categories are defined below:

- The SB category contains the single button press requirements, which shows the expected behaviour for when a single button is pressed and released;
- The MB category contains the multiple button press requirements, which shows the expected behaviour for when multiple buttons are pressed and released at the same time;
- The SW category contains the switching button press requirements, which shows the expected behaviour for when first a button is pressed, and then a different button is pressed after a small delay;
- The G category contains the general requirements that should always hold;

As mentioned before, the security commands only add value for communication with infrared remotes, which is why they will be ignored while testing the RF version of the remote and receiver.

ID	Requirement
SB-01	The receiver always eventually receives at least 11, and at most 14 stop commands, after a button is released and no more buttons are pressed.
SB-02	When the "up" button is pressed, the receiver does not receive down or park commands.
SB-03	When the "down" button is pressed, the receiver does not receive up or park commands.
SB-04	When the "park" button is pressed, the receiver does not receive up or down commands.
SB-05	The receiver receives the same number of up commands as time the "up" button was pressed, divided by 60. It is accepted if the number of received commands is plus or minus 1 of what was expected.
SB-06	The receiver receives the same number of down commands as time the "down" button was pressed, divided by 60. It is accepted if the number of received commands is plus or minus 1 of what was expected.

SB-07	The receiver receives the same number of park commands as time the “park” button was pressed, divided by 60. It is accepted if the number of received commands is plus or minus 1 of what was expected.
MB-01	When multiple buttons are pressed at the same time, no button commands are received.
MB-02	When multiple buttons are pressed at the same time, the number of received stop commands is at least the time pressed, divided by 60, with a maximum of time pressed, divided by 60 + 14.
SW-01	The receiver receives the same number of commands as time the first button was pressed, divided by 60. For the first button press, it is accepted if the number of received commands is plus or minus 1 of what was expected. Also, the receiver receives the same number of commands as time the second button was pressed, divided by 60. For the second button pres, it is accepted if the number of received commands is between a range of plus 1 to plus 4 of what was expected, as the microcontroller is already started up.
SW-02	The receiver receives the number of received stop commands that fit in the time between button presses, plus the regular amount of stop commands after the second button is released (see SB-01).
G-01	The last 11 received commands do not contain up, down or park commands.

Table 1: Requirements

7.4 Testable UML sequence and class diagrams

From the requirements, the following three test cases can be identified:

1. Single: a single button (up, down or park) is pressed for an arbitrary amount of time, and then released.
2. Multiple: multiple buttons (2 or 3) are pressed at the same time, for an arbitrary amount of time, and then released.
3. Switch: a button is pressed for an arbitrary amount of time, and then released. A delay between 200ms and 700ms happens where no buttons are pressed, after which a different button is pressed for an arbitrary amount of time, and then released.

For every test case, only the corresponding requirements category has to be satisfied. Except for the general requirements G, which has to hold for all test cases. The class diagram $cd_{stairlift}$ illustrated in Figure 7.6 has been created to test these test cases. This class diagram contains the remote and receiver (sub)systems under test, as well as the multiple message formats: SingleButtonPress, MultipleButtonPress, SwitchButtonPress and Result. Here, the prefix of the ButtonPress message format corresponds to the test cases enumerated above and is used for the input to the remote. The Result message format is used for the output from the receiver and is the same for all test cases and contains the number of received up, down, park and stop commands, as well as a value that depicts if the last 11 received commands were not up, down or park.

The sequence diagram $sd_{SingleButtonPress}$ in Figure 7.7 shows the up single button press variant of the Single test cases. The sequence diagrams with test cases in this category test all requirements in category SB. Which button is pressed (up, down or park) can be identified

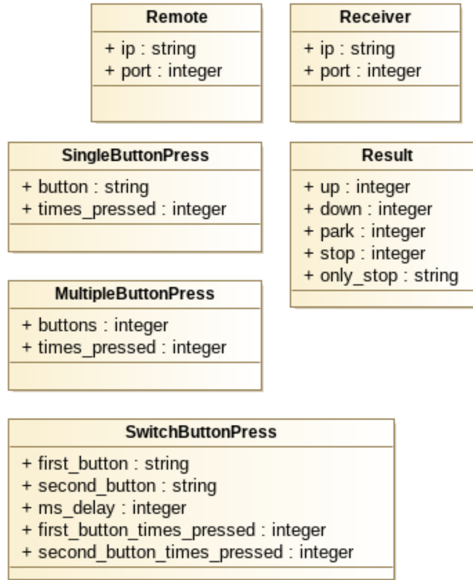
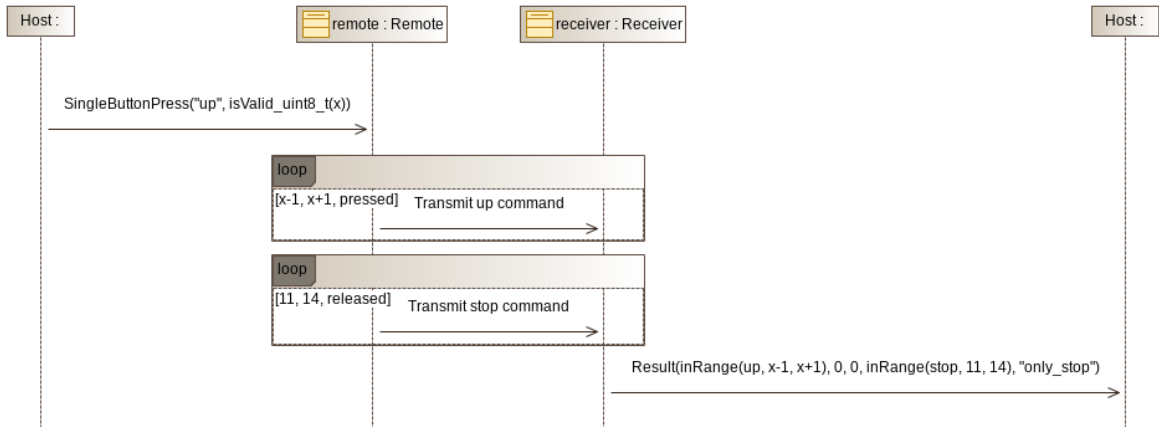


Figure 7.6: $cd_{stairlift}$

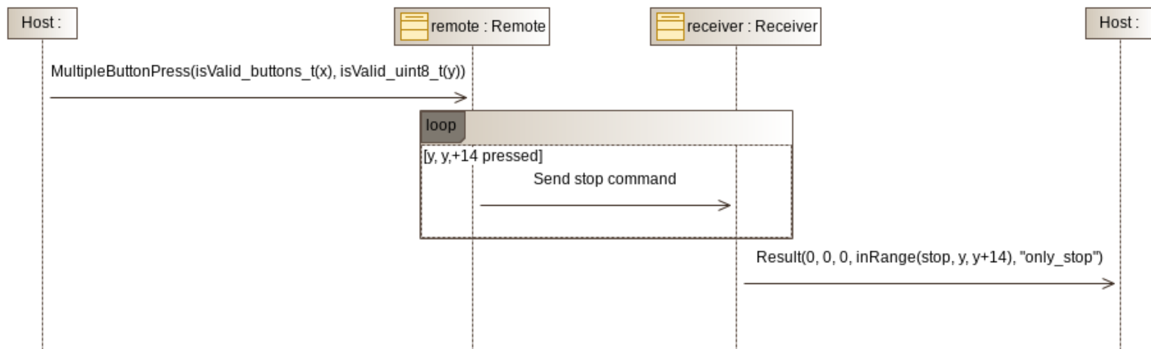
by the first argument of the SingleButtonPress message format, which specifies the button that has to be pressed. The second argument contains a function call to generate a value for time pressed. The function call (explained in Section 5.3.3) generates this value based on the range of a uint8_t, which is a value between 0 and 255. This value resembles the amount of commands we expect to receive, as it is multiplied by the time between transmissions (60 milliseconds) in the adapter to get a time in milliseconds. The sequence diagram also shows the communication between the remote and receiver. However, as explained in Section 5.1, internal communication will not be included in the generation of the symbolic transition system. The inner communication in the sequence diagram resembles the *Communication flow* sequence diagram in Figure 7.3. The main difference here is that the press and release actions are all included in the initial SingleButtonPress message. The reason for this is explained in Section 7.2.

The Result messages in the sequence diagrams contain the expected output. For example, $sd_{SingleButtonPress}$ presses the up button for a specified amount of time x. The expected result message should then contain: $x \pm 1$ up commands, 0 down commands, 0 park commands, 11 to 14 stop commands and the string "only stop". The sequence diagrams $sd_{MultipleButtonPress}$ and $sd_{SwitchButtonPress}$ in Figure 7.7 test the other requirement categories MB and SW. Category G should always hold for all test cases.

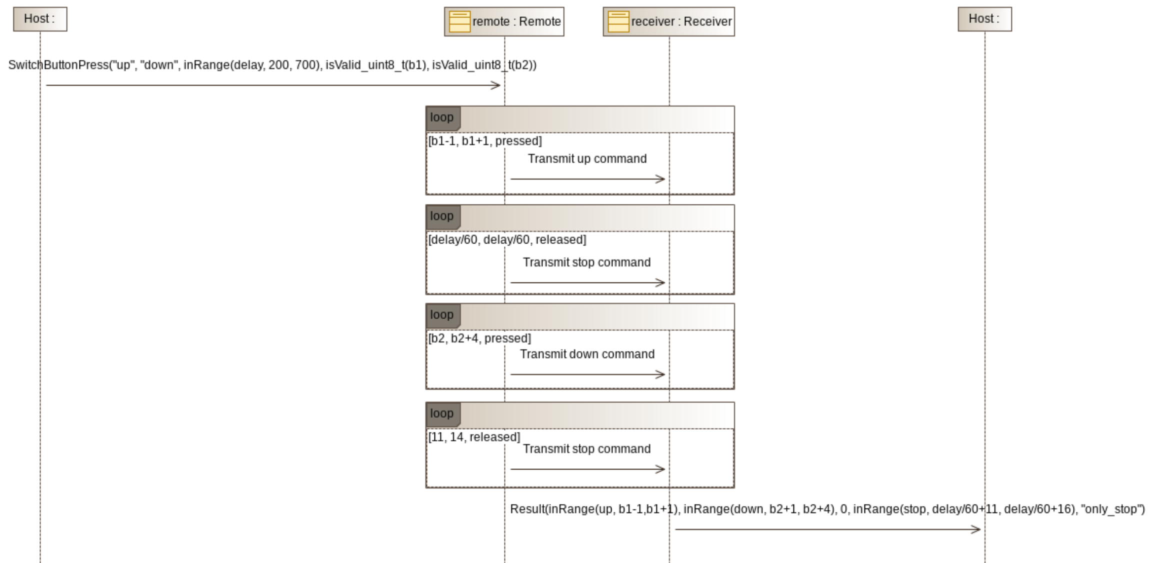
A small Python script has been written that executes the SUT and TorXakis 25 times with each time 500 test cases. If TorXakis discovers an error prematurely or finishes the 500 test cases, the performed tests are logged and the traces to replay the test are saved.



(a) $sd_{SingleButtonPressUp}$



(b) $sd_{MultipleButtonPress}$



(c) $sd_{SwitchButtonPressUpDown}$

Figure 7.7: Sequence diagrams for Single, Multiple and Switch button press test cases

8 Results and Discussion

8.1 Results

This section provides the results obtained from the case study described in Chapter 7. The outcomes from the three executed test cases (Single, Multiple and Switch) are shown in Table 2.

Category	Single	Multiple	Switch	Total
Tests executed	790	279	884	1953
Tests failed	**0**	**1**	**24**	**25**
Fail rate	0.0000%	0.0036%	0.0271%	**0.0128%**

Table 2: Test results first run

From the 1953 performed tests in total, 25 failed. From these 25 failed tests, 24 fails belong to the Switch test case and only a single fail belongs to the Multiple test case. Based on the fail message, and after studying the software and electrical schematics, we discovered why these tests failed. All of the 24 Switch test cases failed due to too few commands being received from the second button press. However, in 22 of the 24 Switch test cases, this was due to a delay that was above 620 milliseconds. The reason for this is that the microcontroller will shut itself down after all the commands have been sent, which is based on the 11 stop commands ($11 \times 60 = 660$ milliseconds) that the remote sends after the button is released. This means that the microcontroller has to start up again, which takes some time. All the failed Switch test cases showed that one fewer stop command was received than expected.

The remaining 2 Switch test cases failed because the generated value for the times the first button should be pressed was 0. This means that the button is only pressed for the average startup time of the microcontroller (around 137 milliseconds). Then the button is released before the microcontroller is able to send an up, down or park command, a delay happens, and another button is pressed. Even though 0 up, down or park commands are sent during the first button press, the microcontroller is still already started up, shown by the extra stop commands received during the delay in between button presses. This is why we would expect more commands from the second button press. However, the second button press does not result in the expected number of commands, and the reason for this is still not known.

The only Multiple test case that failed contained a park byte, while the requirements say that when multiple buttons are pressed at the same time, none of the up, down or park commands should be sent. Why this happened once out of the 279 multiple button presses, is still not known. When the test trace is replayed, it does not produce the same result. To see if other faults can be identified, the requirement SW-01 for the switch test cases was relaxed:

SW-01 - The receiver receives the same number of commands as time the first button was pressed, divided by 60. For the first button press, it is accepted if the number of received commands is plus or minus 1 of what was expected. Also, the receiver receives the same number of commands as time the second button was pressed, divided by 60. For the second button pres, it is accepted if the number of received commands is between a range of plus 0 to plus 4 of what was expected, as the microcontroller is already started up.

Also, the delay between button presses in the SW test cases was changed from a range of 200ms to 700ms, to a range of 100ms to 500ms. This was altered to avoid the microcontroller automatically shutting down after it has sent all the stop commands when the delay is too long. With these updated requirements, the tests were rerun for a full night. The obtained results for the second test run are shown in Table 3.

Category	Single	Multiple	Switch	Total
Tests executed	4440	1502	4525	10467
Tests failed	**1**	**5**	**1**	**7**
Fail rate	0.0002%	0.0033%	0.0002%	0.0007%

Table 3: Test results second run

Interestingly, the fail rate of the test cases decreased a lot but fails still appeared. The fail messages of the test cases show that the reasons for the fails differ greatly in this second test run. First of all, a Single test case has failed while this has never happened before. The fail message indicates that quiescence was the received output, which means that no output at all was received until the timeout of 20 seconds was reached. The Multiple test cases all failed for the same reason as in the previous test run, where a park command was received while this should not happen. The failed Switch test case received a stop command too many. All of the identified fails resulted in the following failed requirements:

- **SB-01**: due to quiescence, whereas stop commands were expected;
- **MB-01**: due to instances where a park command had been received while multiple buttons were pressed;
- **SW-01**: due to receiving too few commands when the first button press is just long enough to start up the microcontroller;
- **SW-02**: due to too many received stop commands in a single fail instance of the Switch test case;

8.2 Discussion

8.2.1 Test results

During testing of the remote and receiver, some faults were found, as well as inconsistencies with regard to the documentation. First of all, the communication flow sequence diagram in Figure 7.3 does not show the 1 to 3 extra stop commands that the receiver will send to the stairlift when no RF has been received for a timeout period. Secondly, the time between pressing the button and sending commands via RF is not explicitly documented, which is caused by the startup time and initialization of the microcontroller. This is especially an issue when counting the number of received commands and comparing them to the expected number of received commands.

Some unexpected behaviour was noticed when executing the test cases. For instance, some test cases failed because either too many or too few commands were received. The most exciting failures happened when the first button was only pressed for long enough to start up the microcontroller, then during the second button press, a few more commands are expected as the microcontroller does not have to start up. This is always the case when the first button is pressed long enough to send a command. However, in the case that the first button was only pressed for long enough to start up the microcontroller, these commands were not received, while the number of received stop commands was as expected.

Some actual faults were identified during the second test run, where the Multiple test case sent a park command while no up, down or park commands should have been sent. Moreover, a test case failed because of quiescence (no output). These test results showed that the remote and receiver sometimes behave unexpectedly. Due to the nature of RF being error-prone and limited code and hardware inspection, it is still unclear whether the detected faults are due to the RF protocol, hardware, software or even timing issues on the host PC. However, the faults that occurred were not entirely random. This might indicate that the cause of the faults does indeed reside in the hardware or software.

The identified faults are probably not even noticeable in regular use, which could be the reason why they were never found before. Finding what caused the faults is also hard, as executing precisely the same test sequences again, does not reproduce the same fault.

8.2.2 Sequence diagrams as specification

Sequence diagrams model scenarios and do not give a full specification of a system. State machine diagrams often give a more complete model of the system. However, sequence diagrams are easier to understand and offer documentation to stakeholders with different levels of technical background knowledge. They are also easier to create in the early stages of a project, as they can be based on use cases and requirements, which are often created in the initial stages of the design phase.

Because sequence diagrams only model scenarios, a lot of them are needed to model every scenario in a state diagram, which means that it becomes hard to test every path in a system. Features such as data, data-dependent constraints and variables give some more options to the creator of the diagram to test different a single function with generated constrained input data without having to model every possible input value. However, the user has to supply self-defined functions to perform coverage-based testing, which is often more interesting than randomized testing.

8.2.3 Verbose messages

When using regular messages or message formats with only static values or variable as arguments, the readability of sequence diagrams does not degrade. However, when adding expressions or function calls, the messages in a sequence diagram can become long and verbose, which harms the readability of the sequence diagram. Reduced readability defeats the purpose of sequence diagrams as documentation for developers or stakeholders. However, it may still be simpler to create more advanced tests by using verbose sequence diagrams than by creating test cases with a different framework. These sequence diagrams should only be created as test cases that are not used as documentation.

8.2.4 Limitations

The main limitation that we encountered was that we were unable to model and test real-time behaviour. Section 7.2 in the case study illustrated this well, where all timing behaviour had to be set up in the message, and all results also had to be fully processed before they could be sent in the output message. This is also why the sequence diagrams created look so different from the sequence diagrams in the interface protocol specification. The main reason for this is that TorXakis communicates with sockets. Sockets give the user many interfacing options, but they sometimes introduce enough latency to make it impossible to test time-critical applications. This is especially a problem in embedded systems, as these systems are often subject to real-time constraints [35].

The UML modelling tool Modelio does also does not support the notion of time in an actual unit between events in a sequence diagram. However, there are ways in Modelio to add constraints to almost every object in sequence diagrams, which means that this feature could be added with a special syntax for timing constraints. Another option could be to add a timing diagram that relates to a single or sequence of events in a sequence diagram to specify the timing constraints.

9 Conclusion and Future Work

9.1 Conclusion

In this thesis, we explored the possibility to transform UML sequence and class diagrams to models suitable for model-based testing, to reduce the effort and time needed to create and maintain these models, with the intent to make model-based testing more accessible for smaller projects and smaller companies. Based on the performed case study with the created sd2txs tool, it can be concluded that by only creating some small and straightforward sequence diagrams, we were able to create an input model for the model-based testing tool TorXakis that was able to test a remote and receiver for a stairlift thoroughly. Some weird behaviour was observed during testing, which may indicate the presence of bugs in the software or hardware.

The performed case study highlighted a shortcoming with TorXakis when it comes to testing embedded systems with timing constraints. Due to this shortcoming, a different modelling strategy had to be adopted to combat the time-critical actions modelled by the sequence diagrams present in the interface protocol specification documentation. These original sequence diagrams documented the protocol well. The sequence diagrams created with the different modelling strategy were able to test the embedded system by letting the adapter deal with the time constraints but, unfortunately, did not provide the same level of documentation. The reason for this is the decrease in readability, intuitiveness, and usefulness of the sequence diagram. Seeing that the primary goal of creating UML diagrams is to provide visual documentation for a system, the sd2txs tool is currently better suitable for (embedded) software that is not subject to timing constraints.

In this thesis, we also researched the syntax and semantics of sequence diagrams, from which it can be concluded that the syntax is well standardised and clearly defined. Available tooling mostly abides by this standard, but many tools do not offer the full set of specified syntax. This is also the case for the export to XMI functionality. Some tools do not offer this functionality, and from the tools tested, it became clear that with the same model, tools often produce different XMI output. This research also showed that the semantics of sequence diagrams are not clearly defined, difficult, and sometimes ambiguous. This is why other research often chooses to define their own interpretation of the semantics or uses predefined semantics from other research, to be able to reason formally about sequence diagrams.

Furthermore, we have shown that labelled and symbolic transition systems are most suitable as an intermediate model representation, as the TorXakis modelling language is based on these types of transition systems. This means the decision to adopt labelled transition systems as an intermediate model representation was quickly made, which later switched to symbolic transition systems when data, variables, and data-dependent constraints were added. These concepts can be added to the UML sequence diagrams by using a class diagram. With this type of diagram, users are able to define custom message formats with typed attributes, which can be populated with static values or variables that get a dynamically generated value from TorXakis. These variables can be reused by the user in the sequence diagram on other message formats attributes that have the same type. Constraints to the generation process of the typed value can be introduced by calling user-defined TorXakis library functions directly from the sequence diagrams. These extra features allow for more freedom and flexibility in the creation of the testable model but reduce the readability and documentation value of the sequence diagram.

9.2 Future work

Some ways to improve or extend this research are listed below:

- In the case study, a different modelling strategy had to be adopted due to timing constraints in the SUT. Currently, the sd2txs tool does not support extracting timing constraints from the XMI model, nor does TorXakis support real-time testing. To be able to test real-time systems, support for extracting timing constraints from the XMI model, as well as support for generating a model for a real-time model-based testing tool have to be added;
- During testing, some of the test cases failed. However, it is still not known what the cause of these failures is. Future work could include debugging the code and hardware to find out what caused the failures;
- A formal proof has to be given that when translation a sequence diagram to a symbolic transition system, the system is able to produce exactly the same set of traces and does not introduce unwanted quiescence behaviour because of the τ-transitions that were added;
- Support for asynchronous communication or enforcing a ban on non-determinism has to be added in the model generation part of sd2txs. The non-determinism examples have shown that race conditions can occur when allowing asynchronous communication in combination with non-determinism;
- During testing, it became evident that TorXakis uses a Gaussian random number generator to generate integers in a specific range. However, edge cases are often more interesting. Currently, edge cases can be generated more frequently by calling a library function that is designed to return the desired edge cases. However, it would be interesting if TorXakis would offer the option to generate edge cases to speed up the testing process;
- The intermediate model representation, deduced from the sequence diagrams, could be used in case studies with formal validation techniques such as model checking. A generator should be added to the desired file format, as the current generators include a dot file generator for visualisation and a TorXakis model file generator for model-based testing;
- Sequence diagrams model scenarios and do not give a full specification of a system. State machine diagrams often give a more complete model of the system, which is why support for these diagrams would have to be implemented in the sd2txs tool to give more freedom to the tester;
- Comparing the XMI export feature of different modelling tools showed that the XMI output is not always precisely identical when the same model is exported to XMI [30]. This means that supporting the XMI output from Modelio, does not imply that the XMI output from, e.g., Enterprise Architect is also supported. To be able to support multiple modelling tools, either the tools have to abide by the same standard, or support for each tool has to be implemented in sd2txs;
- The sequence diagram to STS transformation algorithm could be optimised, to eliminate the need for τ-transitions in the STS when combined fragments are present in the sequence diagram;

References

[1] Song Wang and Zhixia Li. Exploring the Mechanism of Crashes with Automated Vehicles using Statistical Modeling Approaches. *PLOS ONE*, 14, Mar 2019.

[2] Prof. Jacques-Louis Lions et al. ARIANE 5 Flight 501 Failure. Technical report, ESA and CNES, 1996.

[3] Muhammad Shafique and Yvan Labiche. A Systematic Review of Model Based Testing Tool Support. Technical Report SCE-10-04, Carleton University, Apr 2010.

[4] Sarbada Shashank, Praneeth Chakka, and Dr. Vijay Kumar. A Systematic Literature Survey of Integration Testing in Component-Based Software Engineering. *2010 International Conference on Computer and Communication Technology (ICCCT)*, pages 562–568, 2010.

[5] Jan Tretmans. Model Based Testing with Labelled Transition Systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008.

[6] Johan Gustavsson. A Comparative Study of Automated Test Explorers. Master's thesis, Linköping University, 2015.

[7] Monika Müllerburg, Leszek Holenderski, Olivier Maffeis, Agathe Merceron, and Matthew Morley. Systematic Testing and Formal Verification to Validate Reactive Programs. *Software Quality Journal*, 4(4):287–307, Dec 1995.

[8] Mirko Conrad, Ines Fey, and Sadegh Sadeghipour. Systematic Model-Based Testing of Embedded Automotive Software. *Electronic Notes in Theoretical Computer Science*, 111:13 – 26, 2005. Proceedings of the Workshop on Model Based Testing (MBT 2004).

[9] M. Mikucionis, K. G. Larsen, and B. Nielsen. T-uppaal: online model-based testing of real-time systems. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 396–397, Sep 2004.

[10] Jan Tretmans. On the Existence of Practical Testers. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, pages 87–106, Cham, 2017. Springer International Publishing.

[11] Axel Belinfante. *JTorX: Exploring Model-Based Testing.* PhD thesis, University of Twente, Netherlands, Sep 2014. IPA Dissertation series no. 2014-09.

[12] Kristian Karl. Graphwalker. https://graphwalker.github.io/.

[13] S. Mohacsi, M. Felderer, and A. Beer. Estimating the Cost and Benefit of Model-Based Testing: A Decision Support Procedure for the Application of Model-Based Testing in Industry. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 382–389, Aug 2015.

[14] N. Thompson and R. Platt. *The Evolution of UML*, pages 348–353. IGI Global, Jan 2015.

[15] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, June 2013.

[16] A.M. Fernández-Sáez, Michel Chaudron, and Marcela Genero. Exploring costs and benefits of using UML on maintenance: Preliminary findings of a case study in a large IT department. *CEUR Workshop Proceedings*, 1078:33–42, Jan 2013.

[17] Roy Grønmo and Birger Møller-Pedersen. From Sequence Diagrams to State Machines by Graph Transformation. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations*, pages 93–107, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[18] Debasish Kundu, Debasis Samanta, and Rajib Mall. An Approach to Convert XMI Representation of UML 2.x Interaction Diagram into Control Flow Graph. *ISRN Software Engineering*, 2012, 03 2012.

[19] Luciana Brasil, Valdivino Santiago Júnior, and Nandamudi Vijaykumar. Transformation of UML Behavioral Diagrams to Support Software Model Checking. *Electronic Proceedings in Theoretical Computer Science*, 147, 04 2014.

[20] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Test Case Generation by means of UML Sequence Diagrams and Labeled Transition Systems. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 1292–1297, Oct 2007.

[21] Muthusamy MD and Badurudeen GB. A New Approach to Derive Test Cases from Sequence Diagram. *Journal of Information Technology & Software Engineering*, 04, 01 2014.

[22] Oluwatolani Oluwagbemi and Hishammuddin Asmuni. An Approach for Automatic Generation of Test Cases from UML Diagrams. In *International Journal of Software Engineering and Its Applications*, volume 9, pages 87–106, 2015.

[23] Vikas Panthi and Durga Mohapatra. Automatic Test Case Generation Using Sequence Diagram. *0.814*, 2, May 2012.

[24] Jan Peleska. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. *Electronic Proceedings in Theoretical Computer Science*, 111, Mar 2013.

[25] Markus Dahlweid, Oliver Meyer, and Jan Peleska. Automated Testing with RT-Tester - Theoretical Issues Driven by Practical Needs. In *In Proceedings of the FM- Tools 2000, number 2000-07 in Ulmer Informatik Bericht*, 08 2000.

[26] Object Management Group. OMG Unified Modeling Language. https://www.omg.org/spec/UML/2.5.1/PDF, Dec 2017.

[27] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why Timed Sequence Diagrams Require Three-Event Semantics. In Stefan Leue and Tarja Johanna Systä, editors, *Scenarios: Models, Transformations and Tools*, pages 1–25, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

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

[29] Object Management Group. XML Metadata Interchange (XMI) Specification. https://www.omg.org/spec/XMI/2.5.1/PDF, Jun 2015.

[30] Una Zusāne, Oksana Nikiforova, and Konstantins Gusarovs. Several Issues on the Model Interchange Between Model-Driven Software Development Tools. In *ICSEA 2015 : The Tenth International Conference on Software Engineering Advances*, 11 2015.

[31] Mark Timmer, Ed Brinksma, and Mariëlle Stoelinga. Model-Based Testing. In *Software and Systems Safety - Specification and Verification*, 2011.

[32] S. Nidhra. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications*, 2:29–50, Jun 2012.

[33] Petra van den Bos and Jan Tretmans. *Coverage-Based Testing with Symbolic Transition Systems*, pages 64–82. Springer International Publishing, 09 2019.

[34] Jan Tretmans. On the existence of practical testers. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, pages 87–106. Springer International Publishing, Cham, 2017.

[35] Paula Herber and Sabine Glesner. *Verification of Embedded Real-time Systems*, pages 1–25. Springer Fachmedien Wiesbaden, Wiesbaden, 2015.