MASTER

Design and assessment of analysis techniques for UML sequence diagrams

van Amstel, M.F.

Award date:
2006

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
Design and Assessment of Analysis Techniques for UML Sequence Diagrams

By:
M.F. van Amstel

Supervisor:
Dr. M.R.V. Chaudron (TU/e)

within the EmpAnADa project

Eindhoven, July 2006
Abstract

Software systems and their designs grow larger and more complex while their quality level needs to remain high. The de facto standard for describing software designs is the Unified Modeling Language (UML). This project has been carried out in the context of the EmpAnADa project. The goal of the EmpAnADa project is to improve the quality of UML designs. This work addresses one specific diagram type contained in the UML, the sequence diagram.

A methodology has been developed and implemented to apply automated model checking to UML sequence diagrams. The tool developed as part of this project can also automatically analyse the sequence diagrams in a UML model for syntactic inconsistencies. Furthermore, the tool can automatically detect patterns in sequence diagrams.

The analysis techniques that can be performed on sequence diagrams were validated and rated by means of case studies and structured interviews. These interviews were also used to learn how sequence diagrams are used in practice.
Preface

My study of computer science started in the year 2000 at the Eindhoven University of Technology. During a course on object oriented programming and the software engineering project at the end of the bachelor phase of my study, the concepts of software architecture and the use of UML were taught. This is when I started to get interested in software architecture. In the first year of the master phase I also followed a course on software architecture, which I enjoyed very much.

It was clear long before I started on my graduation project that the ideal topic for me would be in the field of software architecture. Therefore I made an appointment with the lecturer of the master course on software architecture, Dr. Michel Chaudron. We talked about several topics and decided to develop a methodology for applying model checking techniques to UML sequence diagrams. Later during the project more analysis techniques for UML sequence diagrams were explored.

I am grateful to my supervisor Dr. Michel Chaudron for giving me the opportunity to work on a very interesting graduation project and for all his feedback throughout the project. Special thanks go to Ir. Christian Lange and Ir. Martijn Wijn for their useful assistance during the project. I would also like to thank Dr. Judi Romijn and Dr. Ruurd Kuiper for their time and their feedback on my project. Furthermore, I would like to thank Dr. Ruurd Kuiper and Dr. Kees Huizing for taking place in my examination board. And last, but not least I would like to thank the following people for providing cases to validate my analysis techniques and for their feedback:

- Jos van Maarseveen from ASML in Veldhoven
- Jelle Gerbrandy from Getronics PinkRoccade in Amsterdam
- Rob Cloutier from the Stevens Institute of Technology in Hoboken (USA)
- Xin Liu from the Eindhoven University of Technology in Eindhoven

Marcel van Amstel
System Architecture and Networking (SAN)
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACP</td>
<td>Algebra of Communicating Processes</td>
</tr>
<tr>
<td>CADP</td>
<td>Construction and Analysis of Distributed Processes</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer-Aided Software Engineering</td>
</tr>
<tr>
<td>µCRL</td>
<td>micro Common Representation Language</td>
</tr>
<tr>
<td>CTL*</td>
<td>Computation Tree Logic</td>
</tr>
<tr>
<td>EmpAnADa</td>
<td>Empirical Analysis of Architecture and Design Quality</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU’s Not UNIX</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Markup Language</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunication Union</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MDX</td>
<td>Filetype used by Rational XDE</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Chart</td>
</tr>
<tr>
<td>PNG</td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>Promela</td>
<td>Process Meta Language</td>
</tr>
<tr>
<td>SAN</td>
<td>System Architecture and Networking</td>
</tr>
<tr>
<td>SD</td>
<td>Sequence Diagram</td>
</tr>
<tr>
<td>SMV</td>
<td>Symbolic Model Verifier</td>
</tr>
<tr>
<td>SPIN</td>
<td>Simple Promela Interpreter</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SquAT</td>
<td>Sequence Diagram Analysis Tool</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XDE</td>
<td>Extended Development Environment</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSL</td>
<td>Extensible Stylesheet Language</td>
</tr>
</tbody>
</table>
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Acronyms</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Introduction
- 1.1 Motivational Background: 1
- 1.2 Project Goal: 1
- 1.3 Document Outline: 2

## 2 UML Sequence Diagrams
- 2.1 UML Version 1.5 Sequence Diagrams: 4
- 2.2 Features Considered in the Analysis: 7

## 3 Related Work
- 3.1 Software Design Quality: 9
- 3.2 Sequence Diagrams: 9
- 3.3 Positioning this Work: 10

## 4 Analysis Techniques
- 4.1 Statistical Analysis: 12
- 4.2 Syntactic Analysis: 14
- 4.3 Interaction Patterns: 16
- 4.4 Interaction Analysis: 19

## 5 Tool
- 5.1 Requirements: 35
- 5.2 Design: 39
- 5.3 Implementation Notes: 45
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Empirical Validation</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Approach</td>
<td>51</td>
</tr>
<tr>
<td>6.2</td>
<td>Results</td>
<td>51</td>
</tr>
<tr>
<td>6.3</td>
<td>Observations</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>65</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>65</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>66</td>
</tr>
<tr>
<td>8</td>
<td>Project Evaluation</td>
<td>67</td>
</tr>
<tr>
<td>8.1</td>
<td>Project Overview</td>
<td>67</td>
</tr>
<tr>
<td>8.2</td>
<td>Lessons learned</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>69</td>
</tr>
<tr>
<td>A</td>
<td>Representations</td>
<td>73</td>
</tr>
<tr>
<td>A.1</td>
<td>ITU's Textual Representation</td>
<td>73</td>
</tr>
<tr>
<td>A.2</td>
<td>Process Algebra Specification</td>
<td>74</td>
</tr>
<tr>
<td>A.3</td>
<td>Petri-Net</td>
<td>77</td>
</tr>
<tr>
<td>B</td>
<td>Tool Details</td>
<td>81</td>
</tr>
<tr>
<td>B.1</td>
<td>Requirements</td>
<td>81</td>
</tr>
<tr>
<td>B.2</td>
<td>Design</td>
<td>84</td>
</tr>
<tr>
<td>C</td>
<td>SquAT User Manual</td>
<td>99</td>
</tr>
<tr>
<td>C.1</td>
<td>Introduction</td>
<td>99</td>
</tr>
<tr>
<td>C.2</td>
<td>Window Arrangement</td>
<td>99</td>
</tr>
<tr>
<td>C.3</td>
<td>Analysis</td>
<td>101</td>
</tr>
<tr>
<td>C.4</td>
<td>Settings</td>
<td>104</td>
</tr>
<tr>
<td>C.5</td>
<td>Ontology Editor</td>
<td>105</td>
</tr>
<tr>
<td>D</td>
<td>Validation</td>
<td>107</td>
</tr>
<tr>
<td>D.1</td>
<td>Case Study</td>
<td>107</td>
</tr>
<tr>
<td>D.2</td>
<td>Interview Template</td>
<td>113</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Sequence diagram for a coffee machine ........................................ 4
2.2 Message overtaking ................................................................. 6

4.1 Entity-relationship diagram: Sequence diagram representation .......... 11
4.2 Entity-relationship diagram: Class diagram representation ................. 13
4.3 Example of an incoherent sequence diagram .................................. 15
4.4 Example model .......................................................................... 17
4.5 Entity-relationship diagram: Recurring events ................................. 19
4.6 Transition graph in which $A(G(p \rightarrow Fq))$ holds ......................... 25
4.7 Example sequence diagram ......................................................... 26
4.8 Message flow graph for example sequence diagram .......................... 26
4.9 Generalized message flow graph for example sequence diagram .......... 30
4.10 Message flow graph and generalized message flow graph for explanatory example ........................................................................... 32
4.11 Finite state machine for explanatory example ................................ 32
4.12 Finite state machine for example sequence diagram ......................... 33
4.13 Visualization of behavior deviating from visual order in example sequence diagram .................................................................................. 33

5.1 Tool environment ......................................................................... 38
5.2 Use-case diagram ......................................................................... 39
5.3 Package diagram ......................................................................... 40
5.4 Complete class diagram ................................................................ 44
5.5 Graphical User Interface ............................................................. 45
5.6 Main window ............................................................................. 47
5.7 Statistics window ........................................................................ 48
5.8 Call pattern window .................................................................... 48
5.9 Finite state machine .................................................................... 49

A.1 Example sequence diagram and its corresponding message flow graph .... 73
A.2 Process graph for the example sequence diagram ............................... 77
A.3 Petri-net: initialization and end constructs ....................................... 78
A.4 Petri-net: asynchronous and synchronous communication constructs ... 78
A.5 Petri-net for the example sequence diagram .................................... 79

B.1 Class diagram: Internal storage .................................................... 85
B.2 Entity-relationship diagram: Sequence diagram representation .......... 85
B.3 Entity-relationship diagram: Promela representation ......................... 86
B.4 Entity-relationship diagram: Finite state machine representation 86
B.5 Class diagram: Representation converter 87
B.6 Class diagram: Statistics extractor 88
B.7 Class diagram: Syntax analyzer 88
B.8 Class diagram: Patterns detector 89
B.9 Class diagram: Model checker 89
B.10 Class diagram: Behavior visualizer 90
B.11 Class diagram: HTML exporter 91
B.12 Use-case diagram 91
B.13 Sequence diagram: Load UML model 92
B.14 Sequence diagram: Give sequence diagram model statistics and perform a coverage analysis 93
B.15 Sequence diagram: Perform a syntax analysis 94
B.16 Sequence diagram: Detect patterns in the sequence diagrams 95
B.17 Sequence diagram: Check the syntax of an LTL formula 96
B.18 Sequence diagram: Verify the validity of an LTL property in a sequence diagram 97
B.19 Sequence diagram: Show all possible event sequences in a sequence diagram 98

C.1 Window arrangement 100
C.2 Tool bar 100
C.3 Settings dialog 104

D.1 Method Frequency 110
D.2 Classifier Frequency 111
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Call pattern</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Evaluation summary of representations</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of the packages</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>Case study characteristics</td>
<td>52</td>
</tr>
<tr>
<td>6.2</td>
<td>Case study details</td>
<td>53</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary of sequence diagrams in practice</td>
<td>63</td>
</tr>
<tr>
<td>6.4</td>
<td>Summary of observations</td>
<td>64</td>
</tr>
<tr>
<td>B.1</td>
<td>Priority level description</td>
<td>81</td>
</tr>
<tr>
<td>B.2</td>
<td>Constraint Requirements</td>
<td>82</td>
</tr>
<tr>
<td>B.3</td>
<td>Functional requirements</td>
<td>84</td>
</tr>
<tr>
<td>B.4</td>
<td>Mapping of methods to defects to detect</td>
<td>88</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter describes the motivational background for the project and states the project goals. The last section of this chapter contains the outline for the rest of the thesis.

1.1 Motivational Background

Today software systems grow larger and more complex, and so do their designs. This is a trend likely to continue in the near future. The only constant factor is that high quality of a software system is required. There is a relation between the quality of a design and the quality of a system, i.e. bad designs lead to bad systems. In order to raise the quality of a system there should thus be a means to analyze the quality of its design. The advantage of ensuring quality in the design phase is that removing faults in this phase is less costly.

This thesis is the result of a project which has been performed in the context of the Empirical Analysis of Architecture and Design Quality (EmpAnADa) [LC04b] project running at the Eindhoven University of Technology. The goal of the EmpAnADa project is developing practical techniques that help improve the quality of software designs described in the Unified Modeling Language (UML).

The work presented in this thesis focuses on one particular diagram type included in the UML, the sequence diagram. A sequence diagram is used to model the behavior of a system in a scenario-based fashion. The main problem with sequence diagram is that they are often incomplete [LC04a]. Because of this incompleteness the behavior modeled in a sequence diagram does not necessarily have to be in conformance with its requirements. A means to verify whether or not a sequence diagram is modeled according to its requirements is model checking [CGP99]. Incompleteness’s also entail other problems like the risk of misinterpretation [LC06]. These risks can be decreased by verifying whether a sequence diagram is modeled according to the syntactic rules of the UML. Also, modeling inconsistencies should be prevented. Patterns found in a large part of a design are usually intended for the whole system design. Deviations from patterns usually indicate modeling inconsistencies which should be avoided.

1.2 Project Goal

The main goal of the project is to enable the improvement of the quality of UML sequence diagrams by indicating possible defects in them. To achieve this goal, research will be per-
formed to find a way to enable the application of model checking to UML sequence diagrams. Furthermore, research will be conducted to find additional analysis techniques which can indicate defects and identify patterns in sequence diagrams. All these analysis techniques will be implemented in a tool in order to automate the analysis of UML sequence diagrams. The usability of the analysis techniques will be assessed by an empirical study.

To summarize, the four sub-goals for the project are:

1. Apply model checking techniques to UML sequence diagrams
2. Research to find additional analysis techniques to indicate defects and identify patterns in UML sequence diagrams
3. Implement the analysis techniques in a tool
4. Assess the usability of the analysis techniques

1.3 Document Outline

The rest of this document is structured as follows. In chapter 2 a short introduction to the UML is given and the features of sequence diagrams are described in detail. Also, a motivation will be given for why certain features are omitted in the analysis. The current state of the art concerning software design quality improvement and especially UML sequence diagrams is discussed in chapter 3. Chapter 4 contains the core of the project, the analysis techniques. The purpose of the techniques is described, as well as a methodology to perform them automatically. In chapter 5 the tool that implements the analysis techniques is addressed, the requirements are made explicit and the software design of the tool aimed at fulfilling these requirements is described. This chapter is ended by some notes about the implementation of the tool. The tool has been used to perform some case studies on research and industrial models to verify whether it accomplishes its tasks. The results of these case studies are described in chapter 6, as well as the feedback that was given on the results by the designers of the models. This chapter also discusses the results of a structured interview that was held to learn the usability of the tool and the way sequence diagrams are used in the design process in practice. In chapter 7 the conclusions of the project are stated. The last chapter, chapter 8 contains a short evaluation of the project.
Chapter 2

UML Sequence Diagrams

The Unified Modeling Language is a graphical language for visualizing, specifying, constructing, and documenting the architecture of a software system [BRJ99]. The UML was developed in the mid 1990s by unifying a large amount of object-oriented software modeling methodologies. In the meanwhile, UML has grown to become the de facto standard for describing software designs.

A system can be modeled in the UML from different perspectives. To this purpose nine diagram types are available (thirteen in UML 2.0 [OMG03a]):

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

The sequence diagram and the diagram types closely related to sequence diagrams will shortly be described to provide the necessary context.

Interaction diagrams are a type of diagram that is used to model the dynamic behavior of a system in the UML. The sequence diagram is one of the two types of interaction diagrams, the other one is the collaboration diagram which will briefly be explained later. In sequence diagrams, the focus is on the ordering of messages sent and received between objects in time. More precisely, a sequence diagram visualizes the interaction between the participants in a two-dimensional chart by displaying messages sent and received between them.

Sequence diagrams were already used before the UML even existed, in the mid 1970s the International Telecommunication Union (ITU) started using their Message Sequence Charts (MSC). It was not until 1992 that their first recommendation was issued [ITU93]. MSC’s
became very popular for its intuitive graphical notation which can easily be understood by people with different backgrounds. This makes it a good means for communicating the behavioral properties of a system among stakeholders.

The diagram types that are closely related to the sequence diagram are the class diagram, the use case diagram, and the collaboration diagram. A class diagram addresses the static design view of a system. It consists of classes and methods and their mutual relationships. In a sequence diagram the interaction between, among others, instances of classes is described. The messages in a sequence diagram represent the methods modeled in a class diagram. A use case diagram addresses the design from a users perspective by showing a set of use cases, actors and their mutual relationships. Usually a sequence diagram describes a realization of a use case. The collaboration diagram is the other type of interaction diagram, it focusses on the structural organization of sending and receiving objects. A sequence diagram can easily be converted to a collaboration diagram and vice versa, they both describe the same behavior only their focus is different.

In the next section of this chapter the features of sequence diagrams of the UML version 1.5 [OMG03b] will be discussed. The last section contains a motivation for the features that are considered in the analysis, as well as for which are not.

### 2.1 UML Version 1.5 Sequence Diagrams

A sequence diagram consists of a list of participants on the horizontal axis and time on the vertical axis. The participants are usually arranged in chronological order of activation. A participant can be any classifier role. A classifier role represents the behavior of a classifier in a certain context, in this case the sequence diagram. A classifier is a UML element that has attributes and operations. More specifically, these are classes, actors and interfaces.

Messages sent to and received from participants are denoted by arrows and arranged along the vertical axis in order of time sent or received. Messages represent the operations of the participants.

An example of a sequence diagram for a coffee machine is depicted in figure 2.1.

![Figure 2.1: Sequence diagram for a coffee machine](image-url)
2.1. UML Version 1.5 Sequence Diagrams

2.1.1 Lifeline

Every participant in a sequence diagram has a lifeline which represents its existence in time. A lifeline is depicted as a box, or a stick figure in case of an actor, with a label and a dashed vertical line attached underneath it. The syntax of the label is as follows:

\(<\text{role-name}> : \langle\text{classifier-name}\rangle\)

Usually participants exist throughout the whole interaction, these participants are aligned at the top of the diagram. Sometimes however, participants are created during the interaction, then their lifelines start with the receipt of a create message. This create message represents a constructor call. Participants can also be terminated during the interaction, this happens on the reception of a destroy message. This destroy message represents a destructor call. The termination of a lifeline is depicted by a cross at the end of it.

2.1.2 Focus of Control

The period during which a participant can perform actions is depicted in the diagram by a small rectangle on the lifeline. The top of the rectangle is aligned with the start of the action and the bottom is aligned with the completion of the action. This rectangle is called the focus of control. Nesting, for example caused by recursion or self-activation, is depicted in the diagram by another focus of control rectangle that is moved a little to the right on top of the other one.

2.1.3 Message

Lifelines communicate with each other by means of messages, in fact the whole interaction is described by messages. Three types of messages can be distinguished:

- Asynchronous messages, depicted by an arrow with an open arrowhead
- Synchronous messages, depicted by an arrow with a filled arrowhead
- Return messages, depicted by a dashed arrow with an open arrowhead

An asynchronous message consists of a separate send and receive part, which means between sending and reception other events may take place. A synchronous message is, in contrast to an asynchronous message, one single action. The send and receive part of a synchronous message coincide, so no other events can take place between sending and receiving. A return message is just an asynchronous message that represents the return from a procedure call.

Messages are ordered along the vertical axis of the diagram in order of time sent or received relative to the other messages on the same lifelines. Time progresses downward in the diagram. To indicate that messages take a long time to arrive, they can be drawn with a diagonal line. Participants can send messages to themselves. Because it is, of course, impossible to send messages back in time, the tail of the arrow has to be higher in the diagram than the incoming arrowhead. Also, it is assumed that messages sent to and received from the same lifelines cannot overtake each other like in the sequence diagram depicted in figure 2.2.
Messages should have a label for identification, the labeling must adhere to the following syntax:

```
[\('",<\text{return\ value}>[",<\text{return\ value}>]^{*}\)', ':'] <\text{message\ name}>
[\('",<\text{argument}>[",<\text{argument}>]^{*}\)']
```

The `<\text{return\ value}>` field is an optional field indicating the return value of the message. The `<\text{message\ name}>` field is the only obligatory field, it indicates the name of the message. This name should be the same as the method in the class diagram represented by the message. The `<\text{argument}>` fields are optional fields which indicate the arguments of the method, separated by commas. The parameters in the argument list must be matched either by a value or by a dash (–). The *-symbol indicates one or more.

### 2.1.4 Note

It is possible to add some comment in the form of a note to a sequence diagram, this is done by placing text in a dog-eared box. Such a note may be placed anywhere on the diagram.

### 2.1.5 Local Attributes

In a sequence diagram definitions of local attributes may occur, for example to bind the values of a certain variable. Attribute definitions are depicted near the top of the diagram or within a note anywhere on the diagram.

### 2.1.6 Timing

In sequence diagrams, it is possible to describe time observation and to specify timing constraints. The duration of the sending of a message can be measured by appending a variable to the message label. The duration will now be measured and stored in that variable.

Timing constraints can be added to a message, by adding to the label of a message between curly brackets the minimum and maximum time sending of the message may take.

The interval between sending (or receiving) of two messages can be constrained. This is done, again, by adding between curly brackets the minimum and maximum duration of this interval. For ease of reading this is sometimes depicted by adding a double-headed arrow between the two messages labeled with this interval.
2.2 Features Considered in the Analysis

Also the point in time at which a message is sent can be stored in a variable. This variable can then be used to constrain the time another message may be sent or received.

2.2 Features Considered in the Analysis

In this project, the focus is on automating the analysis of sequence diagrams. This is mainly for efficiency reasons. To enable automation it is necessary to import UML models in some standard format. The most obvious import format is the XML Metadata Interchange (XMI) format (see section 5.1.1.2), however this file format imposes some limitations.

The most important features of sequence diagram are supported by the XMI format, lifelines and messages. However, XMI does not distinguish between asynchronous and return messages. For the purpose of the analysis techniques that will be discussed in this document this will not pose any problems.

The XMI standard does not support timing information, therefore no analysis techniques regarding timing will be considered. Also, local attributes will not be considered in the analysis as they do not have a fixed syntax.
Chapter 3

Related Work

This chapter describes the current state of the art of the research related to this work. The first section of this chapter briefly discusses the research that has been going on in the field of software design quality. In the second section research on UML sequence diagrams and ITU’s message sequence charts is addressed. The last section of this chapter describes how this work relates to, as well as how it differs from the work of others.

3.1 Software Design Quality

Quite a lot of work has been done in the field software design quality. In [LC04a], rules are defined to assess the syntactical completeness of a UML model. Follow-up research has been conducted to study the effects of both intra- and inter- diagram consistency defects in a model [LC06]. In [MCB05], an approach is described to indicate inter-diagram inconsistencies by using relation partition algebra. A framework for visualizing both intra- and inter-diagram inconsistencies is described in [CCMS02]. They also use simulation and model checking to analyze UML models.

Research has also been conducted to using design metrics to identify weak spots in a software design [MCL04]. The MetricView Evolution tool developed at the Eindhoven University of Technology described in [Wij06] can visualize these design metrics and also quality attributes on top of UML diagrams. In [Gus02], another tool is described which can calculate quality metrics and perform an automated design pattern detection.

3.2 Sequence Diagrams

As messages sequence charts have already been used quite long, a lot of research has been performed in this field. In [AHP96], a tool is described that can automatically find race conditions in message sequence charts. Another tool [MHK02] has been developed to simulate the behavior that is modeled in an adapted type of message sequence charts, life sequence charts [DH01]. In [AEY03] and in [UKM01] is described how new behavior that can originate from a composition of message sequence charts can be derived. Research has also been conducted to applying model checking algorithms to MSC’s [AY99, LL96]. In [GMP03] also an approach is described to apply model checking to MSC’s, they include compositions of MSC’s. Closely related to this is the conversion of message sequence charts to automata described in [LL94, LL95], and in [LKC02].
In [Tsi01], an approach for verifying the consistency between sequence diagrams, class and statechart diagrams in a UML model is described. An analysis technique for checking sequence diagrams for timing inconsistencies is described in [LL99]. They also describe an inter-diagram timing analysis.

3.3 Positioning this Work

The tool created as part of this project implements the automatic detection of the sequence diagram related syntactical inconsistencies presented in [LC04a], as well as some other syntactic defects.

The tool can also detect some patterns, one of these patterns is a call pattern. In [vH03] the visualization of a call pattern is presented. The difference is that here the call pattern is derived from the sequence diagrams, rather than provided as input to a visualization engine.

The work of [LL96] has been adapted to enable the application of model checking for UML sequence diagrams. Performing model checking on compositions of MSC’s as described in [GMP03] is impossible for UML sequence diagrams as there is no means to specify ordering between different diagrams. In [CCMS02] model checking is also used. They also include other UML diagrams to verify safety requirements.

The ideas presented in [AHP96], and [LL94, LL95] are combined. A finite state machine is used to indicate behavior present in a sequence diagram that deviates from the visual order. Something similar has been done for (compositions of) MSC’s in [AEY03]. The work presented in [LKC02] uses a finite state machine representation for generating test cases.
Chapter 4

Analysis Techniques

This chapter describes the analysis techniques that were studied during the project. Four types of analysis techniques will be considered:

1. Statistical analysis
2. Syntactic analysis
3. Interaction patterns
4. Interaction analysis

The main project goal is to perform research to find a way to enable the application of model checking to UML sequence diagrams. Section 4.4.1 describes how this can be done. The other sections describe analysis techniques which indicate defects and patterns in sequence diagrams.

For performing the analysis it is convenient to have a data model available. The data model that is used to model sequence diagrams is depicted in figure 4.1.

A sequence diagram has a name, a unique identifier, and contains zero or more classifier roles and messages. Each classifier role has a unique identifier and a name assigned. A classifier
role should also be associated to a classifier with an identifier and a name. If the fields with the identifier and name of a classifier are empty the classifier role is not associated to a classifier. Each message also has a unique identifier and a name, but also a sender identifier, a sender name, a receiver identifier, and a receiver name to indicate the identifiers and names of respectively the sending and receiving classifier role of the message. A message also has a Boolean attribute to indicate whether it is a synchronous or an asynchronous message and a sequence number attribute to indicate the order in which the messages occurs in the sequence diagram. A message should always be related to a method in the class diagram. The method a message is related to is described by the method identifier. If this field is empty, the message is not related to any method in the class diagram.

4.1 Statistical Analysis

Statistics about sequence diagrams can provide insight into the utilization of methods and classifiers fast. The statistical analysis consists of two parts, basic statistics and coverage analysis. The use of these analysis techniques, as well as the way in which they can be derived will be discussed in this section.

4.1.1 Basic Statistics

In order to get insight in the frequency of use of certain methods and classes, basic statistical data about sequence diagrams is used. With this data it is easy to discover if some classes are (possibly) overused in interactions, and whether it might be desirable to split them in order to decrease the workload of the objects instantiating these classes. Statistical data can also be used to discover which methods are used most frequently such that they can be optimized in order to increase performance of the overall system.

Statistical data can be used to analyze whether or not there is too much synchronous communication in the system. Synchronous communication implies that two objects are needed for communication. This means that it can occur that one object has to wait for another object to become ready to receive a message, which incurs a delay. It might be possible to avoid this delay by switching from synchronous communication to asynchronous communication.

The statistical data that will be considered in the analysis is the following:

- Number of sequence diagrams considered in the analysis
- Number of synchronous and asynchronous messages modeled
- Frequency table with used messages
- Frequency table with used classifier roles
- Frequency table with used classifiers

4.1.1.1 Derivation

This statistical data can easily be derived from the representation depicted in figure 4.1 by counting.
4.1. Statistical Analysis

The number of sequence diagrams that are considered in the analysis can be derived by counting the different sequence diagram identifiers.

The number of asynchronous messages can be acquired by counting in every sequence diagram the number of messages that have the value true in the field isAsync. The number of synchronous messages can be determined similarly, in every sequence diagram the number of messages that have the value false in the field isAsync should now be counted.

To create a frequency table with used messages, the frequency of occurrence of a message name in every sequence diagram should be counted.

The frequency table with used classifier roles is derived in a similar way. The frequency of occurrence of a classifier role name in every sequence diagram should be counted.

The frequency table with used classifiers is created in almost the same way as the frequency table with used classifier roles, the only difference is that now the type field of each classifier role should be considered instead of the name field.

4.1.2 Coverage Analysis

Coverage analysis is used to indicate which methods and classes are unused in any of the sequence diagrams in the model. The results of this analysis can then be used to determine whether some classes or methods are obsolete, or to conclude that there is still some interaction that has not been modeled yet.

The coverage analysis will produce the following data:

- List of classes not used in sequence diagrams
- List of methods not used in sequence diagrams
- Percentage of classes used in sequence diagrams
- Percentage of methods used in sequence diagrams

4.1.2.1 Derivation

For the coverage analysis information from the class diagram is needed. The data provided by the class diagram that is considered is depicted in the entity-relationship diagram of figure 4.2.

The list of classes not used in sequence diagrams can be obtained by iterating over all classes and checking whether or not there is a classifier role in any sequence diagram that has that identifier as value in the field Classifier Id. The classes that are not found in this way are unused. The list of methods not used in sequence diagrams can be derived in a similar way, now the identifiers of the methods should be compared with the field Method Id of the

![Diagram](image-url)

Figure 4.2: Entity-relationship diagram: Class diagram representation
messages in all sequence diagrams. The percentages can easily be calculated by dividing the number of used classes and methods by the number present in the class diagram.

4.2 Syntactic Analysis

When modeling sequence diagrams, chances are that sometimes a message or role-name is forgotten or that an obsolete lifeline is added. This section describes six types of syntactic defects that can (and do) easily slip into a sequence diagram. Note, the collection of defects described in this section is non-exhaustive.

The threat of these inconsistencies is that they lead to a large variety of misinterpretations and miscommunication [LC06]. A result of this could be that it takes more effort to understand a model. Also, because the quality of the model is lower when there are inconsistencies, all sorts problems could arise in the implementation and maintenance phase of the development cycle.

The description of the detection of syntactic inconsistencies again uses the sequence diagram representation depicted in figure 4.1.

4.2.1 Uninstantiated Classifiers

It frequently occurs in sequence diagrams that lifelines are modeled that have a classifier name but do not have an instance name. The syntax rules for naming a lifeline require that an instance name is specified, therefore this is considered to be a defect. This defect could easily lead to communication problems when a classifier occurs more than once in a sequence diagram.

4.2.1.1 Detection

Lifelines without instance names are detected by searching all classifier roles and looking for empty Name fields.

4.2.2 Lifelines without Associated Classifiers

In a lot of the studied sequence diagrams lifelines are used that are not associated to a classifier. Typically this means that the lifeline representing a class instance is not linked to a class, i.e. the role name is present but no class name is associated with it. This would mean that behavior of a non-existing class is modeled. Again, if this occurs there is not adhered to the syntax rules for naming a lifeline, which requires the presence of a classifier name. Because the syntax rules are not obeyed and consistency of the model is at risk due to this inconsistency, this is considered to be a defect.

4.2.2.1 Detection

Lifelines without associated classifiers are detected by searching all classifier roles and looking for empty Type fields.
4.2.3 Unnamed Messages

Another syntactical inconsistency that frequently occurs is messages without a name. When the name is missing a message cannot be related to a method. As messages should have a name, the UML syntax rules are again disobeyed. This defect is not beneficial for the readability and understandability of the diagram. Moreover, this also leads to an inconsistent model, so this should not occur.

4.2.3.1 Detection

Unnamed messages are detected by searching all messages and looking for empty Name fields.

4.2.4 Messages with Unspecified Methods

Messages in a sequence diagram represent methods from another diagram, therefore there should be a one-on-one mapping from messages to methods. This is however not always the case, often messages are modeled that do not have a method associated with them. This is undesirable because this leads to an inconsistent model.

4.2.4.1 Detection

Unnamed messages are detected by searching all messages and looking for empty Method Id fields.

4.2.5 Incoherence

Incoherence in a sequence diagram means that two or more disjoint sets of lifelines only communicate mutually, but not with lifelines of other sets. For a clarifying example see the sequence diagram depicted in figure 4.3.

In this example, lifeline A only communicates with lifeline C and lifeline B only communicates with lifeline D. If this occurs the sequence diagram is incorrect, because it is not clear when lifeline B has control. This anomaly is most likely to occur when one or more message arrows are not modeled.

![SD Incoherence](image)

Figure 4.3: Example of an incoherent sequence diagram
4.2.5.1 Detection

Before incoherence can be detected in a sequence diagram an assumption has to be made. It is assumed that the interaction starts with sending the message that is depicted the highest in the sequence diagram, i.e. the first message modeled. In practice this does not always have to be the case, this will become clear in section 4.4.

A set of coherent lifelines can now be maintained. This set is initialized with the sender and receiver of the first message. The set of coherent lifelines can be expanded by iterating over all messages and considering the messages which have a sending or receiving lifeline already in the set. The lifelines sending and receiving these messages that are not yet in the set will be added to it. This process can stop when the set does not grow anymore. If the set of coherent lifelines does not include all lifelines modeled in the sequence diagram, it is incoherent.

It is possible (and allowed) that the lifelines not in the set of coherent lifelines together form one or more sets of coherent lifelines. So there may be interaction between these lifelines too.

4.2.6 Unused Lifelines

It sometimes occurs that a lifeline is depicted in a sequence diagram, but it does not send or receive any messages. There are a few explanations for this. When an alternative flow is modeled in a separate sequence diagram that does use the classifier role represented by the lifeline it often remains in the main flow to indicate that it is involved in the modeled interaction. It can also happen the other way around, the lifeline is used in the main flow but it remains in the alternative flow. It is also possible that the lifeline actually should participate in the interaction, thus indicating that some messages are not modeled. A last, simple explanation is that it is just unused and hence the lifeline is obsolete.

Note that an unused lifeline is a special case of incoherence as described in section 4.2.5.

4.2.6.1 Detection

Unused lifelines can be detected by iterating over all messages and maintaining a set of lifelines participating in the interaction. All lifelines that are sender or receiver of a message and not yet in the set of participating lifelines will be added to the set. The lifelines that are on the sequence diagram, but not in the set of participating lifelines never send or receive a message, hence they are unused.

4.3 Interaction Patterns

This section describes some patterns that can be found in UML sequence diagrams. The patterns described here were identified during a brainstorm session [vA06].

The purpose of detecting patterns in sequence diagrams is to verify whether a system is modeled in a consistent way. It is likely that patterns used in a large portion of the sequence diagrams are intended for the whole system design. Therefore, sequence diagrams that deviate from these patterns can be indicated such that a designer can decide whether this deviation is intended or not.
4.3. Interaction Patterns

4.3.1 Patterns Involving Users

Usually a user has to work with a system and therefore he (or she) has to be considered in the design of a system.

As sequence diagrams are often instantiations of use cases, and use cases are modeled from a user’s perspective, most of the time a user will initiate the interaction modeled in a sequence diagram. However, it can occur that the user is unintentionally not modeled at all, which results in interactions being initiated by another classifier role. This can lead to incorrect behavior which is, of course, unwanted.

When a user participates in an interaction usually some feedback from the system is expected. So sequence diagrams in which a user participates in the interaction, but does not receive any feedback are likely to be flawed.

Both these deviations can also occur when a message is unintentionally not modeled.

4.3.1.1 Detection

For the detection of patterns involving users the same assumption has to be made as in section 4.2.5.1, the interaction starts with sending the first message modeled.

If the classifier represented by the sender of the first message is of a user type, a user is considered to initiate the interaction. Detecting whether or not a user gets feedback from the system can be done by iterating over all messages and checking if any of them is received by a lifeline that represents an instantiation of a classifier of a user type. If there is none such message and there are classifiers of a user type present on the sequence diagram, there are users which do not receive any feedback from the system.

4.3.2 Call Pattern

Usually a classifier refers only to a relatively small set of other classifiers, simply because it only has a limited amount of interfaces. Also, some classifiers will be referred to more often than others. To find out which classifiers communicate with each other and how often, a call matrix can be created in which this is indicated [vH03]. This call pattern can then be used to identify calls that are not allowed, or calls that should have been modeled.

Consider, for example, the small model, without any detail, depicted in figure 4.4. The call pattern belonging to this model is given in table 4.1. In this table zeroes are omitted.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4: Example model
A calls B once, B calls C a total of four times, C calls A once, and C calls B once.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Call pattern

On this level it might be hard to discover classifiers that refer to each other that should not do so. When classifiers are grouped per package they belong to it is more easy to find references that should not have been modeled. Suppose classifiers from package X refer to a lot of classifiers from package Y, and there is one reference from a classifier from package X to a classifier from package Z. It is likely that this reference should not have been modeled.

4.3.2.1 Detection

The call pattern can be derived by first creating a matrix of all classifiers that occur in sequence diagrams on both the horizontal and vertical axis and fill it with zeroes. Hereafter should be iterated over all messages of all sequence diagrams, and for every message the value at coordinate (sender, receiver) in the matrix should be raised by one.

The grouping per component was dropped, because detailed information about the package diagram is needed.

4.3.3 Recurring Message Sequences

It is not uncommon that a certain sequence of messages occurs more than once in the sequence diagrams of a UML model. For example in a database system update actions can occur in different places of the system, which have to be modeled separately. It is to be expected that all of these different update actions follow a similar sequence of events. Consider for example the following sequence of events:

Retrieve Data $\rightarrow$ Update Data.

If now in another sequence diagram for example the retrieve step is found, but thereafter a ‘Modify Data’ step is found, it is likely that a mistake has been made.

The recurring message sequences detected in this way are not always modeled intentionally. This analysis technique can thus make implicit patterns explicit.

4.3.3.1 Detection

Recurring event sequences can be found by iterating over all messages in order of occurrence on the sequence diagram, the message with the lowest sequence number first. For every message its successor and its predecessor should be stored according to the data structure depicted in the entity-relationship diagram of figure 4.5.

If a message occurs as a predecessor or successor of a message for the first time, the value of the Occurrence field should be set to zero. If a message has occurred as a predecessor or successor of a message before the value of the Occurrence field should be raised by one.
If the occurrence count of the predecessor with the highest occurrence count is larger than a predefined threshold value, it is considered to be a pattern. The pattern is then $\text{Predecessor} \rightarrow \text{Message}$. If the highest occurrence count is non-unique, then all message sequences with a highest occurrence count are considered to be a pattern. All other message sequences are considered to be deviations from the pattern. In a similar way a pattern of the form $\text{Message} \rightarrow \text{Successor}$ can be found.

### 4.3.4 Design Patterns

There are a lot of architectural design patterns known [GHJV95], most of them concern the static aspects of a system’s design, but there are also some behavioral patterns. On first sight it looked that the mediator pattern could easily be automatically detected in sequence diagrams alone. After a more thorough study in literature [HHHL03] it turned out that also an interpretation of the relationships between classes in the class diagram is needed. Therefore the detection of this pattern was dropped.

### 4.4 Interaction Analysis

A sequence diagram is a means to model the behavior of a system. Because of the different types of communication, synchronous and asynchronous, interleaving may lead to unforeseen behavior. In the interaction analysis the focus is on the ordering of messages. To verify whether the behavior represented by a sequence diagram is correct, two analysis techniques will be described, model checking and visualization of behavior. These two analysis techniques are closely related. Model checking can be used to verify whether certain behavior is present in a sequence diagram. Through visualization the behavior can be seen in a picture.

When sequence diagrams are small, the visualization technique can be used to quickly check whether the execution traces that can be derived from the sequence diagrams do not pose any problems. If sequence diagrams get bigger, model checking can be used to check whether certain undesired event sequences can occur or whether all desired behavior is modeled.

### 4.4.1 Model Checking

Model checking is an automated verification method to check whether a formally specified property holds for a model of a system [CGP99]. The correct functioning of complex systems
is usually verified by testing [Het88], this means that a system is verified in a late stage of the development process when source code already exists. The disadvantage of this approach is that design errors found in such a late phase (if they are found) are expensive to repair as a lot of rework may have to be done. Therefore, it is desirable to verify whether the design of a system is modeled according to its requirements. To this purpose model checking can be used.

Model checking is performed by creating a finite state model of the system to be checked. Then an exhaustive state space search is performed by an automated model checker to determine whether a certain property holds in the model of the system. Model checking is commonly used for verifying (asynchronous) concurrent systems, the finite state systems generated from such concurrent systems can grow enormously large. As the result of a verification needs to be obtained fast, model checkers use algorithms that are optimized to deal with this state explosion problem.

For applying model checking to a design, three tasks need to be performed: [CGP99]

1. Modeling
   The system to be verified must be converted into a formalism that is accepted by an automated model checker.

2. Specification
   The properties that the system must satisfy have to be stated in some logical formalism that is accepted by an automated model checker. Usually a form of temporal logic is used for this.

3. Verification
   When the system is modeled and the properties that need to be verified are stated in a logical formalism, an automated model checker is used to verify whether the property holds in the model or not. If this is not the case, a counterexample in the form of a trace violating the property can be provided.

So model checking is used to determine whether a certain property holds in a model of a system. In the case of UML sequence diagrams this verification model is the design model of a system. Model checking will be applied to check whether the ordering of messages in sequence diagrams is according to the requirements. In this way it can easily be verified whether a sequence diagram is correctly modeled. Model checking can also be used to verify whether interleaving of messages results in some new behavior, such that it can be removed if it is undesired.

In the remainder of this section is described how the three tasks that need to be carried out in order to be apply model checking need to be performed in order to verify properties of UML sequence diagrams. In the last section, an example will be given of how a specification can be verified in a sequence diagram.

### 4.4.1.1 Modeling

The diagrammatic nature of sequence diagrams make them unsuitable for automatically proving properties about the behavior they represent. Therefore, another formalism is needed to represent sequence diagrams. The formalisms that were considered to use for performing model checking on sequence diagrams are:
4.4. Interaction Analysis

- ITU’s textual representation
- Process algebra
- Finite state machine (FSM)
- Petri-net
- Process meta language (Promela)

Another formalism that could have been considered is the UML meta model [OMG02]. This has not been done because it was immediately clear that there is no suitable automated model checker available.

The conversion of a sequence diagram to ITU’s textual representation, a process algebra specification, and a Petri-net are described in appendix A. The conversion to Promela is described in section 4.4.1.4, and the conversion to a finite state machine is described in section 4.4.2.1. These appendices and sections also contain a short introduction to each of the formalisms.

The suitability of these formalisms was determined by evaluating several criteria. This evaluation is described in the next few paragraphs.

4.4.1.1.1 Features
The features of sequence diagrams that need to be modeled are sending and receiving of synchronous and asynchronous messages. It is not necessary to explicitly introduce concepts for create and destroy messages as they are also just messages.

The textual representation is unable to handle synchronous communication and therefore not a suitable representation at all. For modeling asynchronous messages the process algebra needs an additional concept, the state operator, which is inconvenient as all other representations do not require such an additional concept.

Promela, Petri-nets and finite state machines are perfectly suitable to model all features required to represent sequence diagrams.

4.4.1.1.2 Ease of Creation
It is desirable that the creation of the representation of a sequence diagram is not too complicated, otherwise this would require devising complex algorithms to perform the conversion.

The textual representation, as well as the Promela representation are easy to create. Both are almost a direct mapping from a sequence diagram. The process algebra requires more work, as the state operator to model asynchronous communication has to be taken into account.

The creation of a finite state machine is fairly complicated, though it can be easily automated. Creating a Petri-net is also complicated, and can also be automated, though not as easily as the creation of an FSM.

4.4.1.1.3 Size
As the representation will be used on a desktop computer, the size of the representation is of concern for efficiency reasons.

The textual representation and the Promela representation are small as no large state space needs to be generated.
The finite state machine models all possible behavior which can be very much. Consider for example a system with lots of processes in which one process sends one asynchronous message to every other process. Receiving these messages by the other processes can be in any order, thus creating a lot of possible execution traces. In the FSM this is represented by a lot of states, hence it will be big.

The process algebra generates a huge state space as all interleaving is modeled, even for synchronous communication. After applying the state operator it is as big as the state space of an FSM.

The Petri-net representation of a sequence diagram is smaller than the finite state machine and the process algebra representation, because the complete state space does not have to be modeled. However, the Petri-net representation is larger than the Promela and the textual representation, because a message in a Petri-net model requires five places, two transition, and six edges. Synchronous communication requires even more space.

4.4.1.1.4 Automated Model Checking
As the goal is applying automated model checking, it is needed to evaluate whether model checking techniques can be applied automatically to the different representations.

It is possible to state the specification of a property that should hold in the model of a system in the form of a finite state machine. Because it is possible to model both the system and the specification in the same formalism, the validity of a specification can easily be verified by taking the intersection of the finite state machines [CGP99]. If the intersection is empty there is no violating behavior. This verification can be automated, therefore a finite state machine is a suitable representation to apply automated model checking to.

The Spin model checker [Hol97] is designed for automatically verifying LTL formulas. As Promela is the input language for the Spin model checker it is perfectly suitable for this purpose.

There are tools available that can automatically verify the validity of an LTL formula in a Petri-net specification [Mäk02]. However, the visual representation of a Petri-net first has to be converted to an algebraic form. This is not a very complicated process, so a Petri-net representation is suitable for automated model checking.

There are also tools available to apply automated model checking to process algebra specifications, one of them is the Construction and Analysis of Distributed Processes (CADP) toolset [FGK+96]. The disadvantage of using this toolset is that another representation is needed, because a process algebra specification cannot directly be used by the toolset. However, it can be used in combination with the micro Common Representation Language (μCRL) toolset [Wou01], which does accept process algebra specifications. A disadvantage is that two toolsets are needed before it is possible to apply model checking automatically.

4.4.1.1.5 Summary
Table 4.2 summarizes the advantages and disadvantages of the different representations that have been discussed in the previous paragraphs.
Based on this evaluation Promela has been chosen as formalism to represent sequence diagrams for the application of automated model checking. This is the reason that the conversion of a sequence diagram to Promela is described in section 4.4.1.4.

### 4.4.1.2 Specification

Temporal logics are a formalism used for specifying the properties a system must satisfy. They are designed by philosophers to study the way time is used in natural language arguments [HC68]. The strength of temporal logics for the purpose of model checking is that they can describe the ordering of events in time without explicitly introducing time. This is achieved by describing sequences of transitions between the states of a finite state system.

There are several ‘flavors’ of temporal logic, only one of them, Linear Temporal Logic (LTL) [Eme90], will be considered here because the automated model checker SPIN uses LTL. LTL is a sub logic of the Computation Tree Logic (CTL*) that can express properties of individual computations. This is done by composing an LTL formula of a path quantifier, Boolean operators, and temporal operators.

Model checking is used to verify whether a specification holds for a model of a system or not. This means when it is concluded that a specification does not hold, that either the model is faulty or the specification is wrong. It is assumed that the LTL formula is always correct. When it is concluded that an LTL formula does not hold in the model of a system, the model is considered to be erroneous.

#### 4.4.1.2.1 LTL building blocks

In LTL there are two path quantifiers and five temporal operators:

- **A**: for all computation paths
• **E**: for some computation path

• **X**: short for next time, this specifies that a property holds in the next state of the path

• **F**: short for future, this specifies that a property will eventually hold at some state on the path

• **G**: short for globally, this specifies that a property holds at every state on the path

• **U**: short for until, this operator has two variables, it specifies that if there is a state on the path where the second property holds and at every preceding state the first property holds

• **R**: short for release, this operator also has two variables and is the dual of the **U** operator. It specifies that the second property holds up to and including the first state where the first property hold. Note, the first property is not required to hold eventually

By following the rules described below, these building blocks can be used to compose an LTL formula.

A distinction between two types of formulas can be made, path formulas and state formulas. Path formulas hold along a specific computation path and state formulas hold in a specific state on a path. \( P \) is the set of Boolean propositions, which in this case are used to express which messages have been sent and received in a sequence diagram. The syntax of LTL path formulas is as follows:

- If \( p \in P \) then \( p \) is a path formula
- If \( f \) and \( g \) are path formulas, then \( \neg f, f \lor g, f \land g, \) and \( f \Rightarrow g \) are path formulas
- If \( f \) and \( g \) are path formulas, then \( X f, F f, G f, f U g, \) and \( f R g \) are path formulas

The syntax of LTL state formulas is as follows:

- if \( f \) is a path formula, then \( A f \) is a state formula

So for example \( A(FG p) \) is a valid LTL formula, which expresses that on all paths there is some state from which \( p \) will hold forever. The formula \( AG(AF p) \) is not a valid LTL formula as \( AF p \) is a path formula and \( AG \) requires a state formula as argument.

Consider the transition graph depicted in figure 4.6. In this graph the LTL property \( A(G(p \rightarrow Fq)) \) holds. This formula expresses that on all paths in all states holds that if \( p \) holds, then \( q \) must hold somewhere in a later state on that path. In the state on the bottom right \( q \) does not hold, this does not matter as \( p \) did not hold in any preceding state.
4.4.1.3 Verification

One of the reasons that Promela was chosen as a formalism to represent sequence diagrams is for the application of automated model checking is the fact that it is the input language for the automated model checker SPIN [Hol97]. The formalism SPIN uses for specifying properties is LTL. Another great advantage of using SPIN is that when it turns out that a property does not hold, traces can easily generated that shows a violation of the property.

SPIN is a verification system in which the design of asynchronous processes can be verified [Hol97]. This verification is aimed at proving the correctness of process interaction. SPIN has been successfully used for detecting design errors in a wide variety of systems. An advantage of SPIN is that it is designed in such a way that it is fast and uses a minimal amount of memory.

Apart from SPIN two other model checkers were considered, the Symbolic Model Verifier (SMV) [McM93] and the CADP toolset [FGK+96]. The reason SMV was not chosen is that in [McM93] it is advised not to use a certain construct which is necessary for the conversion. The CADP toolset was found less suitable because another toolset is needed to be able to apply model checking on a process algebra specification of a sequence diagram.

4.4.1.4 Promela Representation

The process meta language (Promela) is a high level language to specify system descriptions that is used by the software verification and simulation tool SPIN. The conversion presented here is an adaptation of the one presented in [LL96].

The conversion of a sequence diagram to a Promela representation will be explained on the basis of the example depicted in figure 4.7.

4.4.1.4.1 Message Flow Graph

For the conversion, an intermediate representation is needed in which events are made explicit. This is done by creating nodes for every event and modeling the control flow as a relation on these nodes. The resulting graph is a so-called message flow graph [LL94, LL95]. The message flow graph corresponding to the example sequence diagram from figure 4.7 is depicted in figure 4.8.

The nodes in the graph represent events. The sending of a message $x$ is denoted by $!x$ and the receiving of a message $x$ is denoted by $?x$. A message flow graph has start
nodes and end nodes, depicted by \textit{Start} and \textit{End} respectively. Furthermore, there are three types of arrows, a next event arrow, indicated by an ordinary arrow, a signal event arrow for asynchronous communication, depicted by a dashed arrow, and a signal event arrow for synchronous communication, depicted by a double dashed arrow (not in figure 4.8).

Formally, a message flow graph is defined as a tuple:

$$N = (S, R, X, ne, sig, ST, stype, Start, End)$$

Where $S$ is the set of send nodes, $R$ the set of receive nodes and $X$ the set of nodes used for start and end nodes. The ‘next event’ relation, $ne$, is defined as $(S \cup R \cup X) \times (S \cup R \cup X)$, and the ‘signal event’ relation, $sig$, is defined as $S \times R$. Then there are $ST$, the set of signal types and $stype$, the labeling function for the signal edges. Note that $ST$ and $stype$ are closely related. If the $stype$ is $x$, then the corresponding send and receive edges $!x$ and $?x$ are in the set $ST$. \textit{Start} and \textit{End} are the labels for the start and end nodes respectively.

This is all there is needed to create a Promela representation of a sequence diagram.

4.4.1.4.2 Promela Representation

The complete syntax and semantics of the Promela language is irrelevant since only a small subset of the language constructs will be used. Therefore, only the relevant language constructs are described here. For more information on Promela the reader is referred to [Hol04].
Processes
Each lifeline in a sequence diagram has to be modeled as a separate process. A process with the name $A$ is declared and instantiated using the following clause:

```proctype active A() {
...
}
```

The body of the process will be defined between the curly brackets.

Channels
In Promela, messages are send and received via channels. For sending asynchronous messages a channel with capacity one is used. A message that is on its way is represented by a message of the expected type being in the channel. A channel $SR$ with capacity one is defined as follows:

```chan SR = [1] of {byte}
```

For synchronous messages, a channel with capacity zero is used. Because the capacity is zero a message cannot be ‘on its way’, hence the send and receive event must coincide. This unbuffered communication thus simulates sending and receiving of a synchronous message.

For each message in the sequence diagram a separate channel is declared. The identifier of the message in the sequence diagram can be used to name the channel.

Message Types
Messages must be sent and received via the channels. To define the types of these messages, the following construct is used:

```mtype = {...}
```

Actually this construct is used for generating unique one-byte integer constants, but it turns out to be useful to define message types in this way. For the conversion only one message type suffices as the distinction between messages will be made by flags. Therefore only the message type $m$ is declared.

Sending and Receiving
The actual sending of a message $m$ from sender $S$ to receiver $R$ is in Promela denoted by the following clause:

```SR!m
```

Here $SR$ is the channel representing the message is sent through. Receiving a message $m$ by receiver $R$ from sender $S$ is denoted by:

```SR?m
```

Flags
So now messages can be sent and received, but this is not enough. With SPIN it is impossible to determine whether a send or receive event has occurred, because the system state does not change when messages are sent over a channel. Therefore, another construct has to be introduced. To indicate that a message has been sent or received a flag will be set. For every message arrow in the sequence diagram two flags will be introduced, one for the send event, and one for the receive event. The flags for a message with identifier $x$ will be initialized as follows:
bit $S_x = 0$
bit $R_x = 0$

When a message with identifier $x$ is sent or received this is indicated by setting the flag, which is modeled as follows:

\[
S_x = 1 \\
R_x = 1
\]

**Conversion**

The constructs that are needed for the conversion are all discussed, so now the actual conversion can be performed.

The body of each process can be derived from the message flow graph. Consider the events in the message flow graph from top to bottom and add them to the body of the appropriate process. The start and end nodes should be skipped.

Because a send or receive event and setting the corresponding flag must be considered as one single event, they may not be interleaved by other actions. To achieve this they must be executed as an atomic action, this is done as follows in Promela:

\[
\text{atomic}(...)
\]

So sending a message $m$ over channel $x$ is modeled as:

\[
\text{atomic}(x!m; S_x = 1)
\]

**Compositions of Sequence Diagrams**

To be able to apply model checking to compositions of sequence diagram it is needed that a Promela representation is created of such a composition. This is done by simply merging the Promela models of the sequence diagrams that should be considered for model checking. The structure of the Promela model must remain the same, so first the message type, then the channels, then the flags, and last the processes. This merged model allows an arbitrary interleaving of events occurring in the sequence diagrams composed together.

**Example Sequence Diagram**

Following the guidelines for the conversion just presented, the Promela representation corresponding to the sequence diagram of figure 4.7 can be derived. The result of the conversion is given below.
4.4. Interaction Analysis

\[
\text{mtype} = \{m\};
\]

\[
\text{chan } k = [1] \text{ of } \{\text{byte}\};
\]

\[
\text{chan } l = [1] \text{ of } \{\text{byte}\};
\]

\[
\text{bit } Sk = 0, Rk = 0;
\]

\[
\text{bit } Sl = 0, Rl = 0;
\]

\[
\text{active proctype } A()
\]

\[
\{\text{atomic}\{k!m; Sk = 1\}; \text{atomic}\{l!m; Sl = 1\}\}
\]

\[
\text{active proctype } B()
\]

\[
\{\text{atomic}\{k?m; Rk = 1\}\}
\]

\[
\text{active proctype } C()
\]

\[
\{\text{atomic}\{l?m; Rl = 1\}\}
\]

4.4.1.5 Example

The automatic verification of an LTL property in a sequence diagram will be demonstrated, again using the example sequence diagram from figure 4.7. Suppose one wants to verify whether the reception of message \(k\) always precedes the reception of message \(l\). This can be expressed by an LTL property as follows:

\[
A(G(Rl \rightarrow Rk))
\]

This expresses that on every path in every state must hold that if \(l\) has been received, \(k\) must have been received.

When this property is verified in the Promela model presented in the previous section Spin reports that the property does not hold. The trace violating the property that is returned is as follows:

\[
\text{Send } k \rightarrow \text{Send } l \rightarrow \text{Receive } l \rightarrow \text{Receive } k
\]

This is, of course, a trace that can be derived from the model. Because \(k\) and \(l\) are both asynchronous messages it can occur that \(l\) is received before \(k\).

4.4.2 Visualization of Behavior

In order to visualize all behavior that can be derived from a sequence diagram, a finite state machine can be used. With this finite state machine it can easily be seen whether certain behavior is undesired or not.

The advantage of using this technique to find undesired behavior over model checking is that with model checking all possible undesired behavior has to be checked against the system. This requires the creation of an LTL formula for every undesired property.
In a sequence diagram, messages are depicted in order of occurrence from top to bottom and it is expected that the system will behave according to this visual order. However, due to interleaving, it is possible that the modeled behavior deviates from the visual order, which may be undesired. Undesired behavior might be found by manually running through all possible executions in the sequence diagram itself. When a sequence diagram is really big this is unfeasible. In a finite state machine this behavior deviating from the visual order can be indicated, so the presence of it can easily be observed.

In the rest of this subsection the conversion of a sequence diagram into a finite state machine is explained.

### 4.4.2.1 Finite State Machine Representation

A finite state machine is an abstract model of a digital computer [Lin01]. It has a mechanism for reading an input string over a given alphabet and a transition function to give the next state in terms of the current state. In this section a deterministic automaton will be used, this means that given the current state and the input the next state is uniquely determined. A finite state machine can be visualized by means of a transition graph.

This section describes how a finite state machine can be used to represent a sequence diagram, the conversion presented in this section is based on the work presented in [LL94, LL95].

#### 4.4.2.1.1 Conversion

A message flow graph can be converted to a finite state machine by using a generalized version of the message flow graph in which the events are labeled differently. The generalized version of the message flow graph of figure 4.8 is depicted in figure 4.9.

![Generalized message flow graph for example sequence diagram](image)

The states in the finite state machine represent sets of edges from the message flow graph. Edge \((x, y)\) represents a next event edge leading from \(x\) to \(y\) and edge \(< x, y >\) represents a signal edge leading from \(x\) to \(y\). The transitions are contained in a transition relation \(T\). The transitions are denoted as triples \(< S, x, S' >\) in which \(S\) is the current state, \(x\) the event triggering the transition, and \(S'\) the state to which the transition leads.
4.4. Interaction Analysis

Start Node
A finite state machine requires a dedicated start node. The start state, labeled $S_1$, is defined as the set of all edges leading from the Start nodes into the rest of the graph, these are always next event edges. The start node is depicted as a double circle.

End Node
Apart from a start node a finite state machine also requires a dedicated end node. The end state is defined as the set of all edges leading from the graph into End nodes, these are also always next event edges. The end node is also depicted as a double circle, it can be distinguished from a start node because it has only incoming edges.

Asynchronous Communication
A send node $s$ is enabled in a state $S$ if there is a next event edge in $S$ that has $s$ as second coordinate. A receive node $r$ is enabled in a state $S$ if there is a next event edge in $S$ with $r$ as second coordinate, and there is a signal edge in $S$ with $r$ as second coordinate.

When a send node $s$ is enabled it can perform its send action and the next event edge with $s$ as second coordinate is removed from the current state. Also, both the signal edge starting in $s$ and the next event edge starting in $s$ are added to the current state.

When a receive node $r$ is enabled it can perform its receive action and both the next event edge with $r$ as second coordinate and the signal edge with $r$ as second coordinate are removed from the current state. Also, the next event edge starting in $r$ is added to the current state.

The transition relation $T$ is updated with $< S, !x, S' >$ for a send event and with $< S, ?x, S' >$ for a receive event, where $x$ is the message sent or received.

Synchronous Communication
Synchronous communication is handled in almost the same way as asynchronous communication. In order for the synchronous communication to occur, both the sending node $s$ and the receiving node $r$ have to be enabled in a state $S$. The condition for a receive node $r$ to be enabled is less strict for synchronous communication. There does not have to be a signal edge in $S$ with $r$ as second coordinate. In fact, it is impossible to have such a signal edge in $s$, because no send event has occurred yet.

So if in a state $S$ there are both a next event edge with send node $s$ as second coordinate and a next event edge with receive node $r$ as second coordinate, the synchronous communication can occur. Both next event edges are removed from the current state and the next event edges starting in both $s$ and $r$ are added to the current state.

The transition relation $T$ is updated with $< S, x, S' >$, where $x$ is the message sent and received.

Explanatory Example
To make this conversion more clear consider the (simple) message flow graph and its generalized version depicted in figure [4.10].
Figure 4.10: Message flow graph and generalized message flow graph for explanatory example

The start state $S_1$ consists of the edges $(a, c)$ and $(b, d)$. In state $S_1$ the only enabled node is node $c$, so it will perform its send action. The edge $(a, c)$ will be removed from the current state and both the next event edge $(c, e)$ and the signal edge $<c, d>$ are added to the current state, so $S_2$ consists of the edges $(b, d), (c, e)$, and $<c, d>$. The triple $<S_1, !x, S_2>$ is added to the transition relation $T$. Now the only enabled node is node $d$, so it will perform its receive action. The edges $(c, e)$ and $<c, d>$ are removed from the current state, so $S_3$ consists of the states $(c, e)$ and $(d, f)$. The triple $<S_2, ?x, S_3>$ is added to the transition relation $T$. The current state $(S_3)$ now consists only of edges leading into $End$ nodes so this is the end state. In summary:

$$S_1 = ((a, c), (b, d))$$
$$S_2 = ((b, d), (c, e), <c, d>)$$
$$S_3 = ((c, e), (d, f))$$
$$T = (<S_1, !x, S_2>, <S_2, ?x, S_3>)$$

This leads to the following visual representation of the finite state machine:

Figure 4.11: Finite state machine for explanatory example

**Example Sequence Diagram**

For completeness the finite state machine for the example sequence diagram of figure 4.7 constructed from the generalized message flow graph of figure 4.9 is given below:

$$S_1 = ((a, d), (b, e), (c, g))$$
$$S_2 = ((b, e), (c, g), (d, f), <d, e>)$$
$$S_3 = ((d, f), (c, g), (e, i))$$
$$S_4 = ((b, e), (c, g), (f, h), <d, e>, <f, g>)$$
$$S_5 = ((c, g), (f, h), <f, g>, (e, i))$$
$$S_6 = ((b, e), (f, h), <d, e>, (g, j))$$
$$S_7 = ((f, h), (e, i), (g, j))$$
$$T = (<S_1, !k, S_2>, <S_2, ?k, S_3>, <S_2, !l, S_4>, <S_3, !l, S_5>, <S_4, ?k, S_5>, <S_4, ?l, S_6>)$$
This leads to the following visual representation of the finite state machine:

![Finite state machine diagram](image)

Figure 4.12: Finite state machine for example sequence diagram

### 4.4.2.1.2 Behavior Deviating from the Visual Order

The behavior that deviates from the visual order is determined by first deriving the behavior that is in accordance with the visual order. This can be done by considering all messages in order of occurrence and marking the corresponding events in the finite state machine. The edges that do not have a marking represent behavior that deviates from the visual order.

An example of a finite state machine in which the behavior deviating from the visual order is indicated is depicted below. The deviating behavior is indicated by dashed arrows.

![Behavior deviating from visual order diagram](image)

Figure 4.13: Visualization of behavior deviating from visual order in example sequence diagram
Chapter 5

Tool

This chapter is about the tool that was developed as part of the project. The first section contains the requirements for the tool. In the second section the design of the tool is discussed and in the third and final section of this chapter a few words are spent on the implementation of the tool.

Not all details are given in this chapter, therefore the reader will often find references to appendix B in the text.

5.1 Requirements

The purpose of the tool is to enable the performance of the analysis techniques described in chapter 4 automatically. This section describes the requirements that are needed to achieve this goal. The requirements were formulated and assigned priorities to in collaboration with the supervisor of this project, Dr. Michel Chaudron.

The remainder of this section describes the constraint requirements and the functional requirements the tool should implement.

5.1.1 Constraint Requirements

This section gives a general description of the constraints imposed by the environment on the tool. The specific constraint requirements as well as their priorities are listed in appendix B.1.

5.1.1.1 Platform

The tool should work on the Microsoft Windows XP platform, it would be nice if it could also run on other platforms but this is not necessary. It should run responsively on a modern computer, i.e. equivalent at least to a 2 GHz Pentium 4 processor with 512 MB internal memory and a DirectX 8 compatible graphics card with 64 MB memory. The language used for the tool should be English.

5.1.1.2 Input Format

Almost all UML modeling tools can export models to the XML Metadata Interchange (XMI) format. Therefore the tool has to be able to import UML models from XMI files.
It is also highly desirable that MDX files generated by Rational XDE can be used as input. This because XMI files generated by Rational XDE do not contain diagram information.

5.1.1.3 User Characteristics

The intended user for the tool is a software architect. It can thus be assumed that the user is familiar with UML related terminology.

5.1.1.4 Extensibility

It is undoubtable that there are more analysis techniques that can be performed on sequence diagrams than the ones presented in this thesis. It would be nice if the tool is designed in such a way that extra analysis techniques can be added easily.

5.1.2 Functional Requirements

This section gives a general description of the functional requirements the tool to be developed has to implement. For a list of specific requirements and their priorities the reader is referred to appendix B.1.

For each of the analysis techniques the tool has to implement as well as for some general functional requirements a short description is given of what is needed in order to perform them.

5.1.2.1 General Requirements

In order to automate the analysis on sequence diagrams in an UML model, there must be a means to import such a model. The supported file format will be XMI, as already stated in section 5.1.1.2. The UML model will be stored in a database, such that it can be retrieved if necessary.

For the user of the analysis tool it is convenient that the sequence diagrams on which the analysis is performed can be seen. Therefore it is highly desirable that the sequence diagrams contained in the loaded model can be visualized.

For communication of the results of an analysis, it would be desirable that the results of an analysis can be exported to a human readable format. This format will be HTML.

5.1.2.2 Statistical Analysis

This analysis technique is concerned with extracting some basic statistics from the sequence diagrams in a UML model and performing a coverage analysis. The statistical and coverage data that will be considered is stated in section 4.1. For performance reasons it is desirable that the necessary data can be retrieved from an internal storage facility.

5.1.2.3 Syntactical Analysis

The syntactic defects that will be detected are stated in section 4.2. For better performance of the syntactical analysis it is, again, preferable that the sequence diagrams can be retrieved from an internal storage facility.
5.1. Requirements

5.1.2.4 Interaction Patterns

For the detection of the patterns no different representation is needed, though still it is desirable for performance reasons that sequence diagrams are stored internally. The following functionality should be available in the tool:

- Detecting sequence diagrams (not) initiated by a user
- Detecting sequence diagrams in which a user does not receive feedback from the system
- Visualizing the call pattern inferred from the sequence diagrams
- Detecting recurring message sequences
- Detecting deviation from these recurring message sequences

Again for performance reasons, retrieval of the necessary data from an internal storage facility is desirable.

5.1.2.5 Interaction Analysis

The interaction analysis consists of two parts, on the one hand there is model checking, and on the other hand visualization of behavior.

In order to apply model checking to single sequence diagrams, they must be converted to Promela representations. For performing model checking on multiple sequence diagrams, it would be desirable that any composition of sequence diagrams can be translated to a Promela representation as well. These Promela representations should not have to be created all over again each time the user wants to verify the validity of a property in a sequence diagram. Therefore, it is desirable that these representations are stored internally. To perform model checking an LTL property is required. To enter LTL formulas an editor is needed, because LTL is not an easy formalism it is highly desirable that this editor is enhanced such that language constructs can be used by clicking on them. Also, it is desirable that there is an extensible library with standard LTL formulas available. Because errors can occur easily when LTL formulas are constructed there should be a syntax checker for LTL formulas which gives the user of the tool some feedback on the nature of his (or her) mistake. Feedback is also required if a property does not hold in the Promela representation of a sequence diagram. Therefore, it is desirable that an execution trace violating the property can be generated.

To visualize the behavior contained in sequence diagrams, they must be converted to finite state machine representations. Again, for performance reasons it is desirable that these representations can be stored internally. The possible execution traces should be visualized both textual and graphical. It should also be possible to visualize the behavior that deviates from the visual order, again both textual and graphical.

5.1.3 Context

Figure 5.1 shows the environment of the tool based on the requirements. An XMI file generated by a Computer-Aided Software Engineering (CASE) tool is imported by the analysis tool, which is then parsed, and stored using three different representations. The analysis techniques can be applied to the different representations and the output of the analysis is provided to the user.
Figure 5.1: Tool environment
5.2 Design

In this section the design of the tool to be developed is described by means of UML models. The diagrams types that will be considered successively are:

- Use-case diagram
- Package diagram
- Class diagram
- Sequence diagram

Also, the layout for the user interface of the tool will be sketched.

5.2.1 Use-Cases

The main tasks the tool has to perform are captured in the seven use-cases depicted in the use-case diagram below.

![Use-case diagram]

Figure 5.2: Use-case diagram

The use-cases in the diagram are related to the analysis techniques described in chapter 4, more details on the use-cases is given in appendix B.2.2 where sequence diagrams are given for each of the use-cases.

5.2.2 Packages

In figure 5.3, the package diagram for the tool is depicted. The arrows between the packages represent a ‘uses’ relationship. In the rest of this subsection a short description of the function of the elements in each of the different packages is given. Also, the requirements that should be implemented by the packages are listed, the requirements referred to are described in appendix B.1.2. Only a description of the package diagram is considered here because not too much detail is needed.
Actually the graphical user interface is linked to all of the other packages, but for the sake of readability these links are omitted in figure 5.3.

5.2.2.1 XMI/MDX Parser

The XMI/MDX parser reads an UML model stored in an XMI or MDX file and converts it to SQL queries so the model can be stored.

The XMI/MDX parser of MetricView Evolution [Wij06] can be reused, this parser supports version 1.1 of XMI which uses version 1.3 of the UML meta model, and MDX files created by Rational XDE version 1.5.0.

This package implements requirements CR04a and CR04b.

5.2.2.2 Model Storage

The model storage is an embedded SQL database that stores UML models along with diagram information so it can be extracted when needed. It can handle the SQL queries generated by the XMI/MDX parser.

The embedded database of MetricView Evolution can be reused, tough it needs to be modified as only sequence diagrams need to be stored.

This package implements requirement FR01.

5.2.2.3 Diagram Visualizer

The diagram visualizer can read (sequence) diagram information from the model storage and draw the stored sequence diagrams.

For this the visualization component of, again, MetricView Evolution can be reused.
This package implements requirement FR02.

5.2.2.4 Internal Storage

As the way sequence diagrams are represented in the embedded database is not directly suitable for the analysis that has to be performed on them, an additional storage component is used for this. The classes in this package store the sequence diagrams using three different representations:

- Sequence diagram models
- Promela representations
- Finite state machines

This class that is used for storing sequence diagram models also provides a means to extract the sequence diagram models from the model storage.

This package implements requirements FR04a, FR04b, and FR04c.

5.2.2.5 Representation Converter

The function of the representation converter is to convert sequence diagram models stored in the internal storage to representations that are suitable to apply analysis techniques to. The conversions that need to be carried out are a conversion to the Promela representation and a conversion to a finite state machine representation.

To be able to perform the conversion access to the database that stores the sequence diagrams is needed.

This package implements requirements FR03a, FR03b, and FR03c.

5.2.2.6 Statistics Extractor

The statistics extractor is concerned with extracting statistical data from the sequence diagram model as well as performing a coverage analysis.

This package implements requirement FR05, FR06a, FR06b, and FR06c.

5.2.2.7 Syntax Analyzer

The syntax analyzer analyzes the sequence diagrams for the syntactic defects described in section 4.2.

This package implements requirements FR07a, FR07b, FR07c, FR07d, and FR07e.

5.2.2.8 Patterns Detector

The patterns detector has as function automatically detecting the patterns described in section 4.3 in the sequence diagrams.

This package implements requirements FR08a, FR08b, FR09, R10a, and FR10b.
5.2.2.9 Model Checker

The model checker has as function to verify using the Spin model checker whether an LTL property holds in a Promela representation of a sequence diagram or not. When the specified LTL formula holds in the model this will be indicated by returning true, if the formula does not hold, a violating trace will be returned.

As model checking is performed using LTL formulas, syntax checking of LTL formulas will also be part of the functionality implemented in this package.

This package implements requirements FR11a, FR11b, FR11c, FR13, and FR14.

5.2.2.10 Behavior Visualizer

The behavior visualizer uses the finite state machine representation to extract all possible event sequences from a sequence diagram and visualize them, textual and graphical. Also, all event sequences deviating from the visual order will be extracted and shown.

This package implements requirements FR15a, FR15b, FR15c, and FR15d.

5.2.2.11 HTML Exporter

The HTML exporter creates a report of the results of an analysis performed by the tool. The results of all analysis techniques, except for the interaction analysis techniques will be included in the report.

This package implements requirement FR16.

5.2.2.12 Graphical User Interface

The graphical user interface enables the user to interact with the tool, it translates user commands to method invocations. The enhanced LTL editor, as well as the extensible library of LTL formulas are part of the user interface as well.

This package implements requirements FR12a, FR12b, FR12c, and FR12d.

5.2.2.13 Overview

This section gives a summarizing overview of the function of the packages and the input and output of them.

<table>
<thead>
<tr>
<th>Package</th>
<th>Function</th>
<th>Input</th>
<th>Output</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMI/MDX Parser</td>
<td>Import UML model.</td>
<td>XMI or MDX file</td>
<td>SQL queries</td>
<td>CR04a, CR04b</td>
</tr>
<tr>
<td>Model Storage</td>
<td>Store UML model.</td>
<td>SQL queries</td>
<td>UML model</td>
<td>FR01</td>
</tr>
<tr>
<td>Diagram Visualizer</td>
<td>Visualize sequence diagrams.</td>
<td>UML model</td>
<td>Sequence diagrams</td>
<td>FR02</td>
</tr>
<tr>
<td>Package</td>
<td>Function</td>
<td>Input</td>
<td>Output</td>
<td>Requirements</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------</td>
<td>---------------------</td>
<td>--------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Internal Storage</td>
<td>Store sequence diagrams.</td>
<td>UML model</td>
<td>Sequence diagram models/ Promela/ finite state machines</td>
<td>FR04a, FR04b, FR04c</td>
</tr>
<tr>
<td>Representation Converter</td>
<td>Convert sequence diagrams to different representations.</td>
<td>Sequence diagram models</td>
<td>Promela/ finite state machine representation</td>
<td>FR03a, FR03b, FR03c</td>
</tr>
<tr>
<td>Statistics Extractor</td>
<td>Extract statistics from sequence diagrams. Perform coverage analysis.</td>
<td>Sequence diagram model</td>
<td>Statistics and coverage analysis</td>
<td>FR05, FR06a, FR06b, FR06c</td>
</tr>
<tr>
<td>Syntax Analyzer</td>
<td>Detect syntactic defects.</td>
<td>Sequence diagram model</td>
<td>List of defected diagrams</td>
<td>FR07a, FR07b, FR07c, FR07d, FR07e</td>
</tr>
<tr>
<td>Patterns Detector</td>
<td>Detect patterns and deviations.</td>
<td>Sequence diagram models</td>
<td>Patterns and deviations</td>
<td>FR08a, FR08b, FR09, R10a, FR10b</td>
</tr>
<tr>
<td>Model Checker</td>
<td>Perform model checking</td>
<td>Promela representation</td>
<td>Validity of LTL formula in Promela model, with violating trace if not Correctness of LTL formula</td>
<td>FR11a, FR11b, FR11c, FR13, FR14</td>
</tr>
<tr>
<td>Behavior Visualizer</td>
<td>Show behavior (deviating from the visual order)</td>
<td>Finite state machine representation</td>
<td>Event sequences (textual/graphical)</td>
<td>FR15a, FR15b, FR15c, FR15d</td>
</tr>
<tr>
<td>HTML Exporter</td>
<td>Generate HTML report of the analysis.</td>
<td>Sequence diagram model</td>
<td>HTML Report</td>
<td>FR16</td>
</tr>
<tr>
<td>Graphical User Interface</td>
<td>Provide access to program</td>
<td>User input</td>
<td>Program Output</td>
<td></td>
</tr>
</tbody>
</table>
5.2.3 Class Diagrams

The full class diagram for the tool to be created is depicted in figure 5.4. Some classes are grouped together, this means they belong to the same package. The classes are described in detail in appendix B.2.1.

Table 5.1: Overview of the packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Function</th>
<th>Input</th>
<th>Output</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enhanced LTL editor</td>
<td>LTL standard formula library</td>
<td>FR12a, FR12b</td>
<td>FR12c, FR12d</td>
</tr>
</tbody>
</table>

The class diagrams for the XMI/MDX parser, model storage, and diagram visualizer are omitted as they are already documented in [Wij06]. This results in some relations not being
visible. The classes `TSequenceDiagramStorage` and `TCoverageAnalyzer` use the facilities of the model storage.

5.2.4 Graphical User Interface

A sketch of the graphical user interface for the tool is depicted in figure 5.5.

![Graphical User Interface](image)

Figure 5.5: Graphical User Interface

The first thing that needs to be done when a user wants to perform an analysis on a sequence diagram model is loading the model, this is done via the ‘File’ menu. The left part of the screen is used for visualizing the sequence diagrams that are in the UML model that is loaded. On the right there are buttons and boxes that provide access to the different analysis techniques. The patterns analysis, model statistics, and the syntax analysis will be displayed in a pop-up window. The results of verifying whether an LTL formula holds in the Promela model of a sequence diagram are shown in the box at the bottom of the screen.

5.3 Implementation Notes

The tool that was created is named SquAT, short for sequence diagram analysis tool. The tool was created using the C++ programming language. For the user interface the wxWidgets GUI library was used.

This section discusses some extra features that were added to the implementation as well as some deviations from the design. Also a few words are spent on the extensibility of the parser. The section will end with some screen shots of SquAT.
5.3.1 Extra Features

During testing it turned out that some sequence diagrams contain other defects than the ones considered in the analysis. There are some models that contain sequence diagrams with messages that do not have a sender or a receiver. This has a large impact on the interaction analysis. Therefore, diagrams with this kind of defects are filtered out of the analysis. If diagrams are filtered out the user will, of course, be notified.

For the detection of the patterns involving users it is necessary to identify users. It turned out that users are not always modeled as actors. Therefore, an extensible database with synonyms for user is introduced. When user patterns are detected not only actors will be considered as users, but also all classifiers and classifier roles with a name occurring in this database.

An extra feature that was added is the possibility to export the diagrams frame. The sequence diagrams in the model that appear in this frame, can be exported to a PNG or a JPEG file. This feature might be useful because these exports can be used for printing.

5.3.2 Deviations from the Design

All requirements listed in table B.2 and B.3 were implemented, except for constraint requirement CR01b. The tool is restricted to the Windows operating system only, though it should not be too difficult to make it platform-independent.

Constraint requirement CR02 states that the tool should run responsively on a modern computer. This requirement is fulfilled for most of the analysis techniques, except for the visualization of behavior. Creating a finite state machine from a sequence diagram which represents a lot of behavior can take more than several seconds to complete.

Functional requirements FR15b and FR15d, considering the graphical visualization of behavior, are implemented differently than originally intended. At first the intent was to show the finite state machines in the diagrams frame of the tool. Later it was decided to use the *dot* layout engine [GKN02]. This allows saving the finite state machines to multiple different file formats. The main advantage of this is that certain file formats allow lossless scaling which is highly convenient when dealing with large finite state machines.

During the implementation phase it turned out that the methods modeled in the class diagram of figure 5.4 would suffice to implement all functionality. However, for easing the implementation some methods were added. As a side effect these extra methods also contributed to a more efficient implementation.

In section 4.3.2 is suggested that the classifiers in the call pattern should be grouped per package. Because information of the package diagram was unavailable the classifiers in the call pattern are alphabetized rather than grouped. The analysis technique does not lose too much of its value because of this.

The description of the analysis technique of finding message sequence patterns in section 4.3.3.1 states that in the case of a non-unique highest occurrence count, multiple message sequences are considered to be a pattern. In the implementation only one of these message sequences will be considered to be a pattern and the others are considered to be deviations from it.

Originally it was intended to use the identifiers of messages and classifier roles defined in the XMI file. It turned out that some of these identifiers contain characters that are interpreted by SPIN as reserved characters. Therefore, the identifiers are converted to another
form. This conversion allows the use of $26^4$ unique identifiers, this amount should be large enough to support even very large UML models.

### 5.3.3 Parser

The parser uses XSL transformations to convert the data stored in an XMI file to SQL queries for database manipulation. These XSL transformations are stored externally such that they can be adapted if necessary. It is also possible to add new XSL transformation files to support the XMI output of other CASE tools as well. Currently supported XMI dialects are:

- Rational Rose 2003 (using Unisys Rose UML 1.3.2 Export plug-in)
- MDX files of Rational XDE 1.5.0.
- Enterprise Architect 6.1
- Borland Together 6.0
- MagicDraw UML 7.0

### 5.3.4 Screen shots

This section shows some screen shots of SquAT. The first screen shot shows the main window of the tool. It consists of a menu bar, a tool bar, a diagrams frame, and an analysis frame. The diagrams frame shows the sequence diagrams present in the UML model currently loaded. The analysis frame provides access to the different analysis techniques.

![Figure 5.6: Main window](image-url)
Figure 5.7 depicts the window that appears when a statistical analysis is performed. The different parts of the screen show the different statistics.

![Statistics window](image)

**Figure 5.7: Statistics window**

An example of a call pattern that has been derived from the sequence diagrams of a UML model is depicted in figure 5.8. The colors also give an indication for the frequency of occurrence of calls between two classifiers.

![Call pattern window](image)

**Figure 5.8: Call pattern window**

The finite state machine created by SquAT of the example sequence diagram of figure 4.7 is depicted below. The visualization that was chosen for generating the figure does not indicate behavior deviating from the visual order.
Figure 5.9: Finite state machine
Chapter 6

Empirical Validation

The purpose of the empirical validation is to assess whether the SquAT tool fulfils its task, i.e. does it enable the improvement of the quality of UML sequence diagrams by indicating defects in them. The empirical validation will also be used to learn how sequence diagrams are used in practice.

The first section of this chapter will address the approach that was used for the empirical validation. The second section contains the results of the validation, and the final section of this chapter summarizes the insights that were acquired.

6.1 Approach

To assess whether SquAT helps improving the quality of UML sequence diagrams and how sequence diagrams are used in practice, two techniques were used. First case studies were performed on a heterogeneous collection of models from industrial and research partners to verify the correct functioning of the tool. The results of these case studies were then presented to the designers of the models and they were asked for feedback on the results. These results were used to analyze the validity of the results SquAT produced. The designers were also asked to rate the analysis techniques included in SquAT in order to value the usability of the individual techniques.

The intention was to receive this feedback during a structured interview. The structure that was adhered to during the interviews can be found in appendix D.2. A structured interview was chosen because it allows control of the flow of the interview, thus ensuring the interviews being carried out in the same way for each respondent [Lit91]. Moreover, a structured interview ensures a higher reliability of the outcomes.

6.2 Results

This section contains the results of the empirical validation. The first subsection describes the studied models and highlights the most important findings. An example of a full report of a case study can be found in appendix D.1. The second subsection contains the evaluation of the analysis techniques SquAT can perform by the respondents. The last subsection discusses how sequence diagrams are used in practice.
6.2.1 Case Studies

A total of five case studies were performed on eight different UML models. From an industrial partner two simplified models for an embedded controller were studied. These two models were created in succession, so the second model is an evolution of the first model. This is also indicated by the size of the models. The purpose of these models is to use them for demonstration during UML courses.

From another industrial partner a model of a system for processing parking fines has been studied. This model is intended to be used for analyzing the requirements of the system.

A master student provided a model of a car navigation system. The purpose of this model is to use it for evaluating a product-line system design analysis tool.

From a research partner with a lot of industrial experience, three models of a design of a wireless thermometer system were acquired. These models were used to assess the applicability of patterns in systems design [Clo06]. They were created by three different designers each using different design patterns.

The last case study was performed on the UML model of the SquAT tool itself, the class and sequence diagrams can be found in appendix B.2.

The main characteristics as well as an indication of the size of the models in terms of the number of sequence diagrams and the number of classes are tabulated in table 6.1.

<table>
<thead>
<tr>
<th>#</th>
<th>Model name</th>
<th>Supplier</th>
<th>Application domain</th>
<th>Number of sequence diagrams</th>
<th>Number of classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Embedded controller 1</td>
<td>Industrial partner</td>
<td>UML course</td>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>Embedded controller 2</td>
<td>Industrial partner</td>
<td>UML course</td>
<td>19</td>
<td>79</td>
</tr>
<tr>
<td>3</td>
<td>Parking fines</td>
<td>Industrial partner</td>
<td>Requirements analysis</td>
<td>23</td>
<td>404(^1)</td>
</tr>
<tr>
<td>4</td>
<td>Car navigation system</td>
<td>Master student</td>
<td>Test case</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Wireless thermometer 1</td>
<td>Research partner</td>
<td>Test case</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Wireless thermometer 2</td>
<td>Research partner</td>
<td>Test case</td>
<td>10</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td>Wireless thermometer 3</td>
<td>Research partner</td>
<td>Test case</td>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>SquAT</td>
<td>Self</td>
<td>Analysis tool</td>
<td>7</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 6.1: Case study characteristics

Table 6.2 summarizes the case studies. All case studies are numbered, the numbers correspond to the numbers in the first column of table 6.1. The first row in the table indicates the number of sequence diagrams that were considered in the analysis of each case. The second and third row contain the percentage of the communication that was modeled by asynchronous and synchronous messages respectively. Row four and five summarize the statistical analysis, they display the average use of classifiers and methods. Row six and seven contain the results of the coverage analysis. The eighth row gives an indication of the amount of syntactic defects in the sequence diagrams in the studied models. Row nine and ten show the number of sequence diagrams that are initiated by users and in which a user receives feedback respectively. Row

\(^1\)In this model basic datatypes and tagged values are also considered as being a class, therefore this number is very large. A tagged value is an extension of the properties of a UML element, which allows the creation of new information in the elements specification.
eleven indicates the amount of recurring message sequences that were found in the sequence diagrams. The last row gives an indication of the amount of behavior deviating from the visual order present in the model.

<table>
<thead>
<tr>
<th>Case</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzed diagrams</td>
<td>6</td>
<td>18</td>
<td>23</td>
<td>9</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>Asynchronous messages</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>3%</td>
<td>100%</td>
<td>25%</td>
<td>99%</td>
<td>100%</td>
</tr>
<tr>
<td>Synchronous messages</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>97%</td>
<td>0%</td>
<td>75%</td>
<td>1%</td>
<td>0%</td>
</tr>
<tr>
<td>Avg. usage of classifiers</td>
<td>1.5</td>
<td>2.7</td>
<td>1.4</td>
<td>4.5</td>
<td>4.6</td>
<td>2.4</td>
<td>2.9</td>
<td>2.0</td>
</tr>
<tr>
<td>Avg. usage of methods</td>
<td>1.5</td>
<td>2.7</td>
<td>1.3</td>
<td>1.3</td>
<td>2.7</td>
<td>2.0</td>
<td>1.7</td>
<td>1.3</td>
</tr>
<tr>
<td>Class coverage</td>
<td>28%</td>
<td>28%</td>
<td>10%</td>
<td>47%</td>
<td>61%</td>
<td>35%</td>
<td>78%</td>
<td>95%</td>
</tr>
<tr>
<td>Method coverage</td>
<td>36%</td>
<td>24%</td>
<td>91%</td>
<td>33%</td>
<td>56%</td>
<td>97%</td>
<td>68%</td>
<td>95%</td>
</tr>
<tr>
<td>Syntactic defects</td>
<td>Few</td>
<td>Few</td>
<td>Lots</td>
<td>Lots</td>
<td>Medium</td>
<td>Lots</td>
<td>Some</td>
<td>Some</td>
</tr>
<tr>
<td>SD’s initiated by user</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>SD’s feedback to user</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Recurring Sequences</td>
<td>1</td>
<td>Some</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>Some</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Deviating behavior</td>
<td>Little</td>
<td>Little</td>
<td>Little</td>
<td>None</td>
<td>Lots</td>
<td>Lots</td>
<td>Lots</td>
<td>Little</td>
</tr>
</tbody>
</table>

Table 6.2: Case study details

The remainder of this section describes the results of the case studies as well as the feedback that was received on the results.

### 6.2.1.1 Embedded Controller 1

This model contains a corrupted sequence diagram, this means that one or more messages have no sender or receiver assigned. This does not necessarily mean that the model is flawed, it can also be caused by the XMI export facility of the used modeling tool. Anyhow, the impact of such a corrupted sequence diagram on the interaction analysis is severe, therefore this diagram is left out of the analysis. The analysis is thus performed on the remaining six sequence diagrams in the UML model.

#### 6.2.1.1.1 Statistical Analysis

All communication is modeled by synchronous messages. Most messages are only used once or twice, with two outliers that occur four and five times respectively. The same holds for the frequency of use of the classifiers, also most classifiers are used only once or twice. There are also two outliers that occur four and five times respectively. The coverage of the sequence diagrams is low, only 28% of the methods and only 36% of the classes are used in a sequence diagram.

The coverage statistics indicate that it is likely that not all behavior has been modeled yet.

#### 6.2.1.1.2 Syntactic Analysis

Only one defect is found in the syntactic analysis, there is one classifier role without an associated classifier. This classifier role represents a user that has to work with the system. As this is the only defect found it can be concluded that from this point of view the quality of the model is high.
6.2.1.3 Interaction Patterns
There is one sequence diagram initiated by a user. However, this user does not get feedback from the system. This indicates that a return message may have been omitted.

The call pattern shows that there is a ‘controller’ class that never receives a message. Usually a controller should receive messages, this also indicates that some return messages might not have been modeled.

Only one message sequence pattern has been found.

6.2.1.4 Interaction Analysis
Even though the sequence diagrams used in this model only communicate by means of synchronous messages still some behavior that deviates from the visual order has been found. As there is little deviating behavior, the images of the finite state machine suffice to identify whether this is problematic.

6.2.1.5 Feedback from the Designer
The designer of this model has four years of industrial experience in using the UML. Most of the knowledge was acquired in a learning-by-doing fashion, but also a course on a design methodology very closely related to the UML was taken. The received training is sufficient to fulfill all UML related tasks.

The designer could not tell why there was a corrupted sequence diagram in the model, it could possibly be caused by the XMI export facility of the modeling tool.

All communication is modeled by synchronous messages for convenience reasons only. The low coverage is caused by the modeling tool, it automatically generates a large amount of classes that were unneeded.

The only actor modeled never gets feedback, a ‘controller’ type classifier never receives a message, and there is behavior possible that deviates from the visual order. All this is caused by the fact that return messages are omitted. The deviating behavior could be problematic, in order to avoid this the designer will model the return messages.

6.2.1.2 Embedded Controller 2

This model also contains a corrupted sequence diagram, which is also removed from the analysis. The analysis is performed on the remaining 18 sequence diagrams in the UML model.

6.2.1.2.1 Statistical Analysis
As this model is created by the same designer as the previous one, it is not strange that also in this model only synchronous communication is used. Most messages are used relatively little, but there is one outlier that is used 25 times. The frequency of use of classifiers shows something similar, most are used relatively little only here are two outliers which are used 14 and 15 times respectively. These classifiers represent a controller and a user interface. In the frequency table of used classifiers a ‘Java’ related classifier appears, this looks like an implementation detail that should not be present. Coverage of this model is even lower than of the previous model, only 28% of the methods and only 24% of the classes are used in a sequence diagram.

Again, the coverage statistics indicate that it is likely that not all behavior has been modeled yet.
6.2. Results

6.2.1.2.2 Syntactic Analysis
The syntactic analysis shows that there are two diagrams that contain classifier roles that do not have an associated classifier. This inconsistency is clarified by a note on the diagram. The classifier that should have been modeled is a generic class. Even though the inconsistency is noticed and explained, it should not occur.

6.2.1.2.3 Interaction Patterns
In this model no sequence diagram is present in which a user occurs. Therefore no user patterns are detected.

The call pattern shows that there are quite a lot classifiers that never send a message and that there is only one classifier that never receives a message. It can also be concluded that the user interface is the most active classifier.

Some message sequence patterns were detected.

6.2.1.2.4 Interaction Analysis
Also in this model, behavior is present that deviates from the visual order. Again the images of the finite state machines suffice for verifying the absence of undesired behavior.

6.2.1.2.5 Feedback from the Designer
This model is created by the same designer as the previous one. The reason for the corruption of a sequence diagram in this model is also unknown.

The reason that one method is used 25 times is that a method with the same name occurs in multiple classes. Therefore the 25 instances of the method are not necessarily instances of the same method. The ‘Java’ related classifier was one of the classes that was automatically generated by the modeling tool. As in the previous model, the automatic generation of classes is the reason for low coverage.

The observation that the classifier roles without an associated classifiers should actually be instances of a generic class is correct. The reason that this has not been modeled in this way is simply because the designer had not thought about it. The designer indicated however that this defect will be fixed.

Also in this model the deviating behavior does pose problems, therefore the designer will model the omitted return messages.

6.2.1.3 Parking Fines
In this model ten sequence diagrams have the same name, this makes distinguishing between them very hard.

6.2.1.3.1 Statistical Analysis
The statistical analysis shows no peculiarities. In this model also only synchronous communication is used. The 120 methods in this model are all used relatively little, at most three times. Most of the classifiers are used only once, with a few exceptions. The classifiers that represent a user are used more often. Coverage analysis shows that 91% of all methods is used, so from this point of view coverage is high. However, only 10% of the classes has been used in a sequence diagram. This low value can be explained by the fact that datatypes and tagged values are also considered as classes.
6.2.1.3.2 Syntactic Analysis
Of all classifiers 82% is not instantiated, therefore all diagrams contain classifiers that are not instantiated. Also, in 14 diagrams there are classifier roles that are not associated to a classifier. Almost all sequence diagrams contain messages that are not related to a method, there is even one sequence diagram with an unnamed message. The last syntactic defect found is a diagram with an unused lifeline.

There are quite a lot syntactic inconsistencies, so from this point of view the model has a low quality.

6.2.1.3.3 Interaction Patterns
In eleven sequence diagrams a user is present, in ten of them the user initiates the interaction. However, in only three of the sequence diagrams the user receives feedback from the system. This could indicate that one or more return messages have not been modeled.

From the call pattern can be derived that ‘Mgt’-type classifiers never send a message. Another striking thing is that the classifiers that do not receive a message are of type ‘Controller’ or ‘Form’, there are however some ‘Controller’ and ‘Form’ type classifiers that do receive a message. This again indicates that some return messages might not have been modeled, especially because it is to be expected that controllers do receive messages.

There are two message sequence patterns found.

6.2.1.3.4 Interaction Analysis
Although all communication is modeled by synchronous messages, still quite a lot of behavior that deviates from the visual order is present. There are five sequence diagrams that contain such deviating behavior. As there is not much deviating behavior, model checking does not necessarily have to be used. The images of the finite state machines suffice.

6.2.1.3.5 Feedback from the Designer
No feedback on this model had been acquired at the moment this thesis was written.

6.2.1.4 Car Navigation System

6.2.1.4.1 Statistical Analysis
Of the 66 methods used in the sequence diagrams in this model only two are asynchronous, the rest is synchronous. Most methods are used only once or twice, with one outlier that is used four times. All classifiers are used four or five times on average, there are no real outliers here. The classifier that is used the most is called ‘Core’, this classifier occurs on all sequence diagrams. Coverage is fairly low as only 47% of all classes and only 33% of all methods is used in a sequence diagram.

It is likely that the two asynchronous methods slipped into the model unintentionally and my be replaced by synchronous methods. The ‘Core’ class is likely to be fairly complex as it is used very often, the name of the class also suggests that it is important. From the low coverage can be concluded that not all behavior has been modeled yet.

6.2.1.4.2 Syntactic Analysis
Of the messages, 96% is not associated to a method. Apart from this no other syntactic inconsistencies are found.
The fact that so little messages are related to methods and the low coverage found in the statistical analysis could indicate that this model is far from complete.

6.2.1.4.3 Interaction Patterns
No user has been modeled in the sequence diagrams, so no user patterns are detected. This is strange because the behavior of a consumer product is modeled.

The call pattern shows that the classifier ‘Core’ is most active, it participates in 93% of the interaction. As already was concluded in the statistical analysis, it is to be expected that the class ‘Core’ is very complex.

Two recurring message sequences are found.

6.2.1.4.4 Interaction Analysis
No behavior that deviates from the visual order is present in any of the sequence diagrams.

6.2.1.4.5 Feedback from the Designer
The designer of this model is a master student who received UML knowledge during a university course and by literature study. The designer indicated that the received training was insufficient to fulfill all UML related tasks. Also, the knowledge of the modeling tool is insufficient to model the desired design. This is the reason for the syntactic defects and the fact that only two asynchronous messages are used. The designer will repair the detected defects.

The reason that only a small part of the classes and methods is used is that not all dynamic behavior has been modeled in a sequence diagram yet.

Users were not modeled because they are implicitly present. Interactions are initiated by methods indicating that a button has been pressed. These buttons should be pressed by users of the system.

The ‘Core’ class that is used extensive turned out to be a mediator that coordinates the interaction. The choice for applying the mediator pattern was not explicitly made, it turned out to be a convenient way of modeling.

The two recurring messages that were indicated are the consequence of a modeling convention that was used because this model is a product-line design. Replaceable devices are modeled explicitly instead of using generalization. Therefore the same messages are consecutively send to each of the replaceable devices.

6.2.1.5 Wireless Thermometer 1

6.2.1.5.1 Statistical Analysis
All communication in the sequence diagrams in this model is modeled by asynchronous messages. The frequency table of used methods shows that most methods are used relatively little with two exceptions. The frequency of use of the different classifiers shows no peculiarities. In this model, 61% of the classes and 56% of the methods are used.

From the coverage analysis can be concluded that it might be possible that not all behavior has been modeled yet.

6.2.1.5.2 Syntactic Analysis
All sequence diagrams contain uninstantiated classifiers, apart from this the sequence diagrams are syntactically correct.
6.2.1.5.3 Interaction Patterns
All but two sequence diagrams in this model are initiated by a user, however in none of these sequence diagrams the user receives any feedback. This indicates that some return messages may have been omitted. In the sequence diagrams that are not initiated by a user there is a user present that does get feedback from the system.

The most active classifier is a classifier of type ‘controller’. Apart from this the call pattern shows no oddities.

There are some message sequence patterns detected.

6.2.1.5.4 Interaction Analysis
Because all communication is modeled by asynchronous messages it is to be expected that due to interleaving behavior that deviates from the visual order is present. This is true for most of the sequence diagram. In these diagrams there is a lot of behavior deviating from the visual order. The absence of erroneous behavior should therefore be verified by means of model checking.

6.2.1.5.5 Feedback from the Designer
The experience of the designer of this model is unknown.

Only asynchronous communication is used because this was a requirement. The fact that roughly half of the methods are unused is because this is a high level design created to capture the system architecture and not the complete code to be written. The unused methods will be used later in the detailed design phase. Also 40% of the classes modeled is not used, this is partly because of the same reason as why not all methods are used. Moreover, some of the unused classes represent objects that do not have behavior, like electrical power or a user manual. These were put in the design in order for the engineers to now it is present.

The designer did not intentionally use any behavioral pattern, the detected patterns are all implicit patterns. The fact that users not always receive feedback is because the designer decided not to model return messages.

No assumptions are made to avoid behavior deviating from the visual order.

6.2.1.6 Wireless Thermometer 2
6.2.1.6.1 Statistical Analysis
Roughly 75% of the communication in the sequence diagrams in this model is modeled by synchronous messages, the rest is modeled by asynchronous messages. Most methods are used relatively little, but there are two major outliers that are used 18 times each. The frequency table of used classifiers shows one outlier that is used eight times, the rest is used at most five times. Coverage analysis showed that 97% of all methods is used, but that only 35% of all classes is used.

The coverage statistics hint that there are a lot of classes that do not have methods.

6.2.1.6.2 Syntactic Analysis
All classifiers in all sequence diagrams are uninstantiated. There are also a lot of messages present that are not related to a method in the class diagram. The model also contains one sequence diagram with an unused lifeline, which automatically means that that sequence diagram is incoherent.
6.2. Results

6.2.1.6.3 Interaction Patterns
In all but two sequence diagrams a user is present. However, in none of these diagrams a user sends the first message. Also, in none of the sequence diagrams a user gets feedback from the system. This could indicate that some return messages are omitted.

The call pattern shows one classifier never sending nor receiving a message. This is not coincidental because there was one unused lifeline present in the model. The call pattern also indicates that the most active classifier is of a ‘controller’ type.

A few message sequence patterns were found in this model. Although this model is not really big one of these message sequences occurred 18 times.

6.2.1.6.4 Interaction Analysis
Again, there is a lot of behavior present that deviates from the visual order, so much even that the images of the finite state machines are hardly readable. Therefore model checking should be used to verify the absence of faulty behavior.

6.2.1.6.5 Feedback from the Designer
The feedback that was given on the previous model also holds for this model. The reason that messages are not associated to methods is also caused by the fact that this model represents a high level design.

In none of the sequence diagrams a user initiates the interaction. Probably this should have been the case or another device should have appeared as an actor.

The fact that actors do not receive any feedback is caused by the fact that the designer decided not to model return messages.

6.2.1.7 Wireless Thermometer 3

6.2.1.7.1 Statistical Analysis
Almost all communication in this model is modeled by asynchronous messages, however there is one synchronous message used. The frequency of use of methods shows nothing interesting, all methods are used at most five times but most are used only once or twice. The frequency table with used classifiers shows that a user and a ‘core’ class are used most often. Apart from this noting interesting can be found. Roughly 78% of all classes is used in a sequence diagram and about 68% of all methods is used. What strikes is that most unused classes are likely to represent hardware components.

6.2.1.7.2 Syntactic Analysis
All sequence diagrams contain uninstantiated classes. It was noticed there was only one synchronous message modeled in the sequence diagrams, it turns out that this message is also unnamed and not related to any method. This indicates that this message might be obsolete.

6.2.1.7.3 Interaction Patterns
In all but two sequence diagrams a user is present and initiates the interaction. In most of these diagrams a user receives feedback from the system.

The call pattern shows that a ‘core’ class is most active, this is not strange as it is the most used classifier in the sequence diagrams.

Two recurring message sequences were found.
6.2.1.7.4 Interaction Analysis
In this model also such a large amount of behavior deviating from the visual order is present that it would be advisable to use model checking for verifying the absence of erroneous behavior.

6.2.1.7.5 Feedback from the Designer
The feedback that was given on the previous two models also holds for this model. The only synchronous message that was modeled is obsolete and should be removed.

6.2.1.8 SquAT
6.2.1.8.1 Statistical Analysis
In this model all communication is modeled by means of asynchronous messages. Most methods are used only once or twice, but there are two outliers they are both used six times. Also, no classifiers are used only once or twice, but there are three outliers. The classifiers ‘GUI’ and ‘User’ both occur on every diagram so they are used seven times and the classifier ‘TSequenceDiagramStorage’ is used five times. The coverage in this model is very high, 95% of the classes and also 95% of the methods is used in a sequence diagram.

6.2.1.8.2 Syntactic Analysis
Apart from a few messages that are not related to a method, no syntactical inconsistencies are found.

6.2.1.8.3 Interaction Patterns
All sequence diagrams are initiated by a user and in all but one the user gets feedback from the system.

   From the call pattern can be derived that the classifier ‘User’ only calls the class ‘GUI’, this is to be expected as the user has to work with the system. The call pattern also shows that the classifier ‘GUI’ is most active.

   Two recurring message sequences are found. A ‘RetrieveSequenceDiagram’ message is always followed by a message ‘SequenceDiagram’, and a ‘GetSequenceDiagrams’ message is followed four times by a ‘RetrieveSequenceDiagram’ message. From this last pattern are two deviations, ‘RetrieveSequenceDiagram’ is twice preceded by ‘GetSequenceDiagram’. From this can be concluded that the following pattern occurs four times:

\[
\text{GetSequenceDiagrams} \rightarrow \text{RetrieveSequenceDiagram} \rightarrow \text{SequenceDiagram}
\]

6.2.1.8.4 Interaction Analysis
Due to the amount of asynchronous communication it is to be expected that there will be diagrams that contain behavior that deviates from the visual order. This is the case in two of the sequence diagrams.

6.2.1.8.5 Feedback from the Designer
The designer of this model is the author of this thesis. UML knowledge was acquired during university courses and literature study.

   The fact that all communication was modeled by asynchronous messages was a choice, the correct ordering of the messages will be enforced by explicit modeling. The user has to work
with the system by means of the graphical user interface. This is the reason that they appear on every sequence diagram. The only class not used in an interaction is a generic class.

All messages unrelated to methods that were found all represent return messages. Either a datatype or feedback to a user is returned.

The recurring message sequences that were identified were intended. The deviations occur because sometimes only one sequence diagram needs to be retrieved.

Behavior deviating from the visual order does not pose any problems.

6.2.2 SquAT Evaluation

The purpose of the evaluation is to assess the value of the different analysis techniques included in the SquAT tool. The respondents who were asked to rate SquAT were also asked for ideas to further improve the tool.

The three people who provided the case studies presented in the previous section were also asked to rate SquAT. All three were enthusiastic about the analysis performed on their models and rated SquAT 4.5 on a scale of 1 to 5.

The two interaction analysis techniques, model checking and visualization of behavior, were considered to be most valuable. Although model checking could not be performed on the models because the intended behavior was unknown, the respondents see great potential in it. The visualization of behavior was praised because unintended behavior can be located fast in sequence diagrams that do not contain too much behavior. A sequence diagram contains not too much behavior if the finite state machine showing the behavior is small enough to be comprehensible.

The syntactic analysis is also valued highly by the respondents. This can primarily be explained by the fact that higher quality of the model can be achieved fast by fixing (often simple) syntactic defects. It also helps to indicate spots in the model which are not finished yet.

One of the respondents rated the detection of user patterns and the call pattern less than the others. The reason for this is unknown because this evaluation was acquired after the interview. The detection of recurring message sequences and deviations are considered valuable by all respondents. One of the respondents pointed out another use for the pattern identification. It could be used to analyze the differences in modeling style of different architects, which can then be used as an indication for the possible need for modeling conventions. It was also suggested that patterns should be identified between different design stages and monitor their evolution. Based on this risks can be reduced and the quality of the design can be improved.

The statistical analysis was rated lowest because some of the features lack functionality according to one of the respondents. It was suggested to enable the exclusion of classes with a certain stereotype from the coverage analysis. Some classes represent hardware features or are automatically generated by a modeling tool, which will not be considered in the dynamic design of the system. As their presence has great impact on the coverage percentages, it is desired that they can be excluded from the analysis. Also, the frequency of occurrence of methods currently does not distinguish between methods from different classes which is inconvenient according to one of the respondents.

Apart from the suggestions pointed out in the previous paragraph only two suggestions for improvement were made. The first feature that is considered to be a valuable addition is the calculation of design metrics. The suggested metrics are all complexity related metrics,
life lines per sequence diagram, messages per sequence diagram, and sequence diagrams per use case. Metrics will not be considered as this has already been implemented in MetricView Evolution [Wij06]. The other suggested improvement is a consistency analysis between class and sequence diagrams, i.e. if instances of classes send messages to each other in a sequence diagram an association between these classes should be modeled in a class diagram.

Two of the three respondents think SquAT is a valuable tool and indicated they will seriously consider using it. The other respondent would most likely not use the tool because the step from a scientific analysis method to an industry usable tool has not been taken. This means licensing issues should be taken into account and facilities for interfacing with other tooling and output generation in a computer processable format should be implemented.

6.2.3 Sequence Diagrams in Practice

It is valuable to get insight into the way sequence diagrams are used in practice, because these insights can be used to extend or adapt the collection of analysis techniques to better suit the needs of the industry. New analysis techniques can be developed based on methodologies that are already used in the industry to improve the quality of sequence diagrams. Another reason for acquiring insight in how sequence diagrams are used in practice is for validating some of the assumptions that were made during the project.

In practice, sequence diagrams are primarily used for prototyping the dynamic behavior of a system. This is not strange as sequence diagrams are designed for modeling the dynamic behavior of a system. Another purpose for sequence diagrams indicated by the respondents is validation of the software design, for instance to verify whether all necessary interfaces a class needs in interactions are modeled. The respondents were asked whether sequence diagrams are used differently as the software development process progresses. They all indicated the sequence diagrams will become more detailed and low level later in the project. Also, the focus shifts from prototyping dynamic behavior to validation of the architecture.

Two of the three respondents indicated that their intention is to always model all use cases as a sequence diagram, although this usually is utopian. Therefore use cases will be selected for modeling based on their criticality. The other respondent indicated that approximately 30% of the use cases will be modeled as a sequence diagram. The criteria for selecting the use cases that will be modeled as sequence diagram are complexity, criticality, importance, and value.

The respondents were asked if they use patterns or modeling conventions when modeling sequence diagrams. None of them uses patterns though one of them remarked that patterns should be used. Two of the three respondents indicated they use modeling conventions. One of them sometimes uses color for highlighting. The other one uses a convention for modeling product-line systems, replaceable devices are modeled explicitly instead of using generalization. Therefore the same messages are consecutively send to each of the replaceable devices. These messages are drawn on top of each other. The last respondent tries to model as uniformly as possible but has no specific modeling conventions. However, the introduction of modeling conventions is planned for the near future.

For validating whether it is justified to take a merged Promela model to represent a composition of sequence diagrams, as described in section 4.4.1.4.2, it is needed to know how concurrent execution is treated. Two of the three respondents indicated that they do not take concurrent execution of sequence diagrams into account. The other respondent indicated that concurrent execution will be explicitly modeled and that the execution of a composition of
sequence diagrams is considered to be sequential.

In section 4.2.6 it is stated that an unused lifeline can be a consequence of modeling alternative flows. To validate this assumption it is necessary to know how alternative flows are modeled, because UML version 1.5 does not provide a mechanism for this. One of the respondents uses separate sequence diagrams to model the main and alternative flows. Another respondent uses dotted lines to indicate alternative messages. The third respondent never had the need to model alternative flows.

Table 6.3 summarizes how sequence diagrams are used in practice.

<table>
<thead>
<tr>
<th>Usage</th>
<th>Prototyping behavior and validation of architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage later in development cycle</td>
<td>More detail</td>
</tr>
<tr>
<td>Percentage of use cases modeled</td>
<td>30% – 100%</td>
</tr>
<tr>
<td>Criteria for selecting use cases</td>
<td>Mostly criticality (also complexity, importance, value)</td>
</tr>
<tr>
<td>Usage of patterns</td>
<td>No, though should be considered</td>
</tr>
<tr>
<td>Usage of modeling conventions</td>
<td>Some</td>
</tr>
<tr>
<td>Meaning of composition</td>
<td>Not considered or sequential</td>
</tr>
<tr>
<td>Modeling alternative flows</td>
<td>Separate diagram or dotted message arrows</td>
</tr>
</tbody>
</table>

Table 6.3: Summary of sequence diagrams in practice

6.3 Observations

Feedback on the analysis techniques was acquired from only three people. This is unfortunate, as this is actually a too small sample to draw well-founded conclusions from. It is therefore desirable that more designers are interviewed. However, the amount of case studies performed should be enough to assess the correct functioning of the analysis techniques.

The statistical analysis was found least useful, especially the frequency table with used classifier roles. This was not used during the case studies as it did not provide any added value. The coverage analysis however was considered to be quite useful. The results of it force designers to think about why certain classes and methods are not used in interactions.

All designers liked the syntactic analysis, mainly because the defects found can (in most cases) be fixed relatively easily. All syntactic defects found during the case studies were correctly identified. Some of the interviewed designers indicated that they will fix the syntactic defects that were identified.

The user patterns were not all correctly identified. The reason for this is that not all classifiers representing users were known on beforehand. The lower rating for this analysis technique might be caused by the type of systems that were analyzed during the case studies. For designs of systems created from a users perspective this analysis technique will probably be more useful.

It turned out that complex classifiers can be identified using a combination of the statistical analysis and the call pattern. Such complex classifiers are used often and have a large amount of interfaces to other classifiers. Whether it is useful that this type of classifiers can be detected should be examined. Complex classifiers often appears on multiple sequence diagrams, this is indicated by a high value in the frequency table of used classifiers. In the call pattern a complex classifier stands out because it calls and is called by other classifiers often. Also, complex classifiers usually call and are called by a large number of different classifiers. The case studies show that classifiers with names as ‘core’ and ‘controller’ can be identified in this
way. It turned out that these classifiers were indeed complex. Apart from this application, the call pattern was found quite useful.

The technique of finding recurring message sequences was found useful by all of the respondents. Most of the detected recurring message sequences were not intended, so implicit patterns were made explicit. There were also some recurring message sequences intended, they were all positively identified. It never occurred that intended patterns were not found, so there are no false negatives.

Visualization of behavior was found a very useful analysis technique. Some of the deviations that occurred in the case studies represent erroneous behavior. None of the designers took any measures to avoid this behavior. Sometimes the deviations indicate that return messages are omitted, in which case the model can easily be fixed. The fact that the identification of behavior deviating from the visual order was found very useful also explains the potential of model checking, as erroneous behavior can be indicated with it too. Unfortunately the usability of model checking could not be verified.

A really positive point is that the conclusions drawn based on the outcomes of the analyses are for the greater part correct. The only incorrect conclusion is that low coverage in the embedded controller cases is not an indication that not all behavior has been modeled yet. Instead, low coverage was caused by automatic generation of classes by the modeling tool.

In section 4.4.1.4.2 is assumed that a composition of sequence diagrams represents their interleaving. However, one of the respondents indicated that a composition is treated as a sequential composition and some of the designers do not even consider concurrent execution. Therefore it can be concluded that this assumption was incorrect. The fact that this assumption is incorrect does not make the analysis technique less valuable. If compositions of sequence diagrams are not considered, then properties of single sequence diagrams can still be verified. If however a composition of sequence diagrams is treated as a sequential composition, it is known on beforehand that messages in one sequence diagram will always precede messages in another sequence diagram. So this can be taken into account when LTL properties are specified.

Another assumption is made in section 4.2.6, i.e. unused lifeline can be caused by modeling alternative flows. As alternative flows are modeled in separate sequence diagrams by some designers, this assumption is correct.

Table 6.4 summarizes the observations made during the empirical validation.

<table>
<thead>
<tr>
<th>Analysis technique</th>
<th>Usefulness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistical analysis</td>
<td>Not useful</td>
</tr>
<tr>
<td>Coverage analysis</td>
<td>Quite useful</td>
</tr>
<tr>
<td>Syntactic analysis</td>
<td>Useful</td>
</tr>
<tr>
<td>User patterns</td>
<td>Not useful</td>
</tr>
<tr>
<td>Call pattern</td>
<td>Quite useful</td>
</tr>
<tr>
<td>Recurring message sequences</td>
<td>Useful</td>
</tr>
<tr>
<td>Model Checking</td>
<td>Has great potential</td>
</tr>
<tr>
<td>Visualization of behavior</td>
<td>Very Useful</td>
</tr>
</tbody>
</table>

Table 6.4: Summary of observations
Chapter 7

Conclusions

This chapter starts with the conclusions of the project. In the second and last section of this chapter some directions for future work are suggested.

7.1 Conclusions

The goal of the project, as stated in section 1.2, is to enable the improvement of the quality of UML sequence diagrams by indicating possible defects in them. This has been done by developing and assessing multiple analysis techniques.

The coverage analysis, which is part of the statistical analysis, is found fairly useful. It indicates that some classes may be obsolete or that still some behavior needs to be modeled. A drawback is that the results of it are sometimes pessimistic because classes that are intended to have no behavior are also taken into account. The other statistical analysis techniques, the frequency tables, are found not so useful.

The syntactic analysis indicates minor defects, as well as inconsistencies between sequence diagrams and class diagrams. Most of the defects found by this analysis technique can be repaired relatively fast. Repairing these defects results in a model that is less prone to misinterpretations and miscommunication.

The patterns involving users are found not useful, at least not for the type of systems considered during the empirical validation. Perhaps this analysis technique is useful in more user centered systems. Unfortunately, no case study was performed on such a system, except for the really small SquAT case. This case is too small and the results are biased, so no well-founded conclusions can be drawn from it. The call pattern is a pretty useful analysis technique. It helps identifying calls that should not have been modeled, and also calls that are omitted. Usually these omitted calls should have been modeled by return messages, so the call pattern helps finding incompleteness’s in sequence diagrams. The identification of recurring message sequences is also a useful analysis technique. It can make implicit patterns explicit and forces designers to think about deviations from a pattern.

Unfortunately, it has not yet been possible to fully exploit the capabilities of model checking. Therefore its usefulness is still unknown, although it has great potential. This potential can primarily be explained by the fact that the visualization of behavior is valued very high. In case much behavior can be derived from a sequence diagram due to interleaving, the visualization is a good indicator for undesired behavior. In case the finite state machine derived from a sequence diagram is small, which means the diagram does not contain a lot of behavior,
the visualization can help localizing possible errors.

The overall conclusion is thus that the analysis techniques implemented in SquAT do indeed enable the improvement of the quality of UML sequence diagrams.

7.2 Future Work

During the empirical validation phase of the project some ideas for further developing the SquAT tool were proposed. The most obvious extension would be the support of the UML meta model version 2.0 [OMG03a]. This complicates the conversion of a sequence diagram to a representation suitable for model checking, but on the other hand it makes this analysis technique all the more interesting. Due to the use of fragments the flow of a sequence diagram can be influenced, thus making it harder to verify its correctness.

Another desired feature is the possibility to edit the sequence diagrams displayed in the diagrams frame of SquAT. As this is outside the scope of analysis, the tool has not been designed to support this feature. For editing sequence diagrams more than enough specialized tools are available.

There were also some suggestions for less radical changes of the tool. For example it would be nice if syntactic errors in the model could be located in a diagram by clicking on the specific defect. One of the designers of the models studied during the empirical validation suggested enabling the exclusion of classes with a certain stereotype from the coverage analysis. Currently a user initiates the interaction if it sends the message with the lowest sequence number, it would be more useful if interleaving is taken into account. Due to this interleaving it might be possible that a user in fact can initiate the interaction although this is not explicitly modeled. In section 4.3.2 it was suggested that the classifiers in the call pattern should be grouped per package, this would be a useful extension.

A more thorough inter-diagram consistency analysis would be worthwhile considering. When a message is exchanged between two classes in a sequence diagram these classes should be related in a class diagram. This can even be extended to a higher level. If classifiers from different packages exchange messages, there should be a relationship between the packages.

The patterns that can be detected by SquAT were identified by studying the sequence diagrams of some UML models. More of these studies could be carried out in order to increase the collection of patterns SquAT can detect.

To get a better insight in the usefulness of the tool, more case studies should be performed and discussed with designers, preferably on larger models. This would result in more reliable usability statistics.
Chapter 8

Project Evaluation

The evaluation of the project starts with a summary of the activities that were performed. Thereafter the most important lessons learned during the project are described.

8.1 Project Overview

The project was divided into four parts which were carried out more or less in succession. However, towards the end of every part there was always a little overlap with the next part.

Roughly the first two months were used to study literature and to develop the analysis techniques. Most of the time was consumed by researching the application of model checking to sequence diagrams. Several representation formalisms and a number of automated model checkers were studied for their suitability to this specific purpose. Thereafter literature was searched for patterns and pattern detection in UML sequence diagrams. It turned out that little research was conducted in this field. Therefore a brainstorm session \[vA06\] was held to identify patterns in the sequence diagrams of industrial cases. Literature was also searched for frequently occurring syntactic defects in sequence diagrams. This did not take too much time as research on this topic was already performed within the EmpAnADa project \[LC04a\]. Some other defect types were also found during the brainstorm session.

The next phase was the design of the SquAT tool which implements the analysis techniques. Formulating the requirements and designing the architecture of the tool took about two weeks.

The following two and a half month were used to implement and debug the SquAT tool. In the beginning progress was made only slowly because the programming language C++ had to be learned. Once the knowledge of C++ was at a sufficient level, the implementation progressed faster. Unit and integration testing was done on the fly.

After the implementation, roughly one and a half month was spent on the empirical validation. The case studies were performed first, but this took way more time than initially planned. It turned out that the XMI files of the cases were generated by different CASE tools. As XMI is a standard this should not pose too much problems, but it did. For the XMI export of three cases a new parser had to be written. For this, knowledge of XSL transformations is needed, which was not present at that time. So first XSL transformations had to be learned and only thereafter the parsers could be created. All this took a lot of time. When the case studies were performed, a template for the structured interview with the designers was prepared. This template could then be adapted for specific questions on
the cases. The last part of the empirical validation was interviewing the designers, this was a very enjoyable experience.

The final task was writing this thesis. This took a little more than a month.

8.2 Lessons learned

The most important lesson learned is the value of the empirical validation. Not only is it good to have the tool tested and evaluated, but also the acquired feedback was of very good use. With hindsight can be concluded that it probably would have been better to also interview designers earlier during the project such that their suggestions for improvements could have been taken into account. Using a structured interview was a good choice, in this way the feedback of the designers could easily be compared.

It was very enjoyable and instructive to work in an academic research setting. It encourages unraveling existing work and developing new ideas. Also, the feedback acquired during the intermediate presentation was useful for further developing the analysis techniques.

The implementation phase of the project was also very educational. At the beginning of it the knowledge of C++ was insufficient, but towards the end programming went reasonably well. Although it was unfortunate that performing the case studies incurred a big delay because different parsers needed to be created, it was also a good reason to learn XSL transformations.

From the beginning of the project most of the findings were documented. This turned out to be very helpful and time saving during the implementation phase, everything needed was ready to hand. Also later on, when writing this thesis, most of the documentation created earlier could be reused.
Bibliography


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.


Appendix A

Representations

This appendix describes the conversion of a sequence diagram into ITU’s textual representation, a process algebra specification, and a Petri-net. These are the representations that were found less (or not) suitable for the application of model checking to sequence diagrams.

For the first two conversions presented in this appendix, it is necessary that the events that occur in the sequence diagram are made explicit. To this purpose the message flow graph described in section 4.4.1.4.1 will be used. The explanation of the conversion in this appendix is also done on the basis of the example sequence diagram from figure 4.7. For the sake of clarity the example sequence diagram and its corresponding message flow graph will be repeated here.

![Example Sequence Diagram and Message Flow Graph](image)

Figure A.1: Example sequence diagram and its corresponding message flow graph

A.1 ITU’s Textual Representation

A message sequence chart has, apart from a visual representation, also a textual representation. This textual representation was initially intended for exchanging sequence diagrams between computers [MR94].

A.1.1 Syntax

The syntax for the textual representation is as follows:
Appendix A. Representations

\[
\begin{align*}
<s_d> & ::= s_d <s_d_i_d>; <s_d \ body> \ endsd; \\
<s_d \ body> & ::= <> | <\text{inst def}> <s_d \ body> \\
<\text{inst def}> & ::= \text{instance } <i_i_d>; <\text{inst body}> \ endinstance; \\
<\text{inst body}> & ::= <> | <\text{event}> <\text{inst body}> \\
<\text{event}> & ::= \text{in } <m_i_d> \ \text{from } <i_i_d>; \ | \ \text{out } <m_i_d> \ \text{to } <i_i_d>;
\end{align*}
\]

The terminals \( \text{out} \) and \( \text{in} \) represent send events and receive events respectively.

The non-terminals \(<\text{i}_i_d>\), \(<\text{m}_i_d>\), and \(<\text{s}_d\text{i}_d>\) represent instance identifiers, message identifiers, and sequence diagram identifiers respectively.

A.1.2 Conversion

The textual representation belonging to the example sequence diagram can easily be derived from its corresponding message flow graph. The result is as follows:

\[
\begin{align*}
s_d \ \text{example}; \\
\ \text{instance } a; \\
\ \text{out } k \ \text{to } b; \\
\ \text{out } l \ \text{to } c; \\
\ \endinstance; \\
\ \text{instance } b; \\
\ \text{in } k \ \text{from } a; \\
\ \endinstance; \\
\ \text{instance } c; \\
\ \text{in } l \ \text{from } a; \\
\ \endinstance; \\
\ endsd;
\end{align*}
\]

A.1.3 Synchronous Communication

The textual representation does not provide any constructs to include synchronous communication. Therefore, it is impossible to model synchronous communication using this representation for sequence diagrams.

A.2 Process Algebra Specification

Process algebra is an algebraic means to specify and study the behavior of parallel systems [Bae05]. The process algebra considered here is the Algebra of Communicating Processes [BBR06], with an extension to correctly represent asynchronous communication [MR94]. The axioms for the ACP will not be given, for this the reader is referred to [BBR06].

This section describes how process algebra can be used to represent sequence diagrams. The conversion presented in this section is based on [Mau96, MR94].

A.2.1 Communication

The representation of message exchange in sequence diagrams will be discussed in this section.

A.2.1.1 Asynchronous Communication

Asynchronous communication consists of a send and a receive part. Sending a message \( m \) by
sender instance \( s \) to receiver instance \( r \) is denoted by \( \text{out}(s, r, m) \), receiving a message \( m \) by
receiver instance \( r \) from sender instance \( s \) is denoted by \( \text{in}(s, r, m) \). More formally this is:

\[
E_o : \{ \text{out}(s, r, m) | s, r \in \text{iid}, m \in \text{mid} \}
\]
\[
E_i : \{ \text{in}(s, r, m) | s, r \in \text{iid}, m \in \text{mid} \}
\]

A.2.1.2 Synchronous Communication

To represent synchronous messages, the send event \( \text{out}(s, r, m) \) and its corresponding receive
\text{in}(s, r, m) \) must coincide. For this purpose the communication function \( \gamma \) is used,
along with the encapsulation operator \( \partial_H \). The synchronization of the events \( \text{out}(s, r, m) \) and
\( \text{in}(s, r, m) \) is modeled as \( \gamma(\text{out}(s, r, m), \text{in}(s, r, m)) \equiv \text{comm}(s, r, m) \). To make sure the
send and receive events cannot occur asynchronously, they both have to be added to the
encapsulation set \( H \):

\[
H = H \cup \text{out}(s, r, m) \cup \text{in}(s, r, m)
\]

A.2.2 Conversion

Now it is possible to convert a message flow graph into an equivalent process algebra specifi-
cation. The lifelines are represented by processes. The bodies of the processes can be derived
from the message flow graph. Consider the events in the message flow graph from top to
top bottom and put them in a sequential composition. Consider the example sequence diagram
and its corresponding message flow graph from figure A.1. Process \( a \) is modeled in the process
algebra as:

\[
\text{out}(a, b, k) \cdot \text{out}(a, c, l)
\]

To represent the entire sequence diagram in the ACP, all processes need to be executed in
parallel.

A.2.3 State Operator

Because a message should not be received before it is sent, the ordering between a send event
and its corresponding receive event must be maintained. This is done by extending the ACP
with a state operator \( \lambda \) [MR94]. This operator stores all send events that have occurred in a
set \( M \) and it only allows a receive event if the corresponding send event occurs in \( M \). The
axioms for the state operator are as follows:

\[
\lambda_M(\varepsilon) = \varepsilon \quad \text{if } M = \emptyset
\]
\[
\lambda_M(\varepsilon) = \delta \quad \text{if } M \neq \emptyset
\]
\[
\lambda_M(\delta) = \delta
\]
\[
\lambda_M(a \cdot x) = a \cdot \lambda_M(x) \quad \text{if } a \notin A
\]
\[
\lambda_M(\text{out}(i, j, m) \cdot x) = \text{out}(i, j, m) \cdot \lambda_M(\text{out}(i, j, m) \cup x)
\]
\[
\lambda_M(\text{in}(i, j, m) \cdot x) = \text{in}(i, j, m) \cdot \lambda_M(\text{in}(i, j, m) \cup x) \quad \text{if } \text{out}(i, j, m) \in M
\]
\[
\lambda_M(\text{in}(i, j, m) \cdot x) = \delta \quad \text{if } \text{out}(i, j, m) \notin M
\]
\[
\lambda_M(x + y) = \lambda_M(x) + \lambda_M(y)
\]
A.2.4 Example

Consider again the example sequence diagram and its corresponding message flow graph from figure [A.1]. It is easy to derive the processes they represent:

\[
\begin{align*}
P_a &= \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
P_b &= \text{in}(a, b, k) \\
P_c &= \text{in}(a, c, l)
\end{align*}
\]

When putting these processes in parallel all possible interleaving can be derived:

\[
P_a \parallel P_b \parallel P_c
\]

After some calculation this results in:

\[
\begin{align*}
\text{out}(a, b, k) \cdot \text{in}(a, b, k) \cdot (\text{out}(a, c, l) \cdot \text{in}(a, c, l) \\
+ \text{in}(a, c, l) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot (\text{in}(a, b, k) \cdot \text{in}(a, c, l) \\
+ \text{in}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, c, l) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, b, k) \cdot \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, b, k) \cdot \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, b, k) \cdot \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, b, k) \cdot \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k)
\end{align*}
\]

Traces in which \( \text{in}(a, b, k) \) precedes \( \text{out}(a, b, k) \) or \( \text{in}(a, c, l) \) precedes \( \text{out}(a, c, l) \) are undesirable so they have to be removed. This is done by applying the state operator \( \lambda \emptyset \) to the expression. This results in the more friendly specification:

\[
\begin{align*}
\text{out}(a, b, k) \cdot (\text{in}(a, b, k) \cdot \text{out}(a, c, l) \cdot \text{in}(a, c, l) \\
+ \text{out}(a, c, l) \cdot (\text{in}(a, b, k) \cdot \text{in}(a, c, l) \\
+ \text{in}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, b, k) \cdot \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, b, k) \cdot \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k) \\
+ \text{in}(a, b, k) \cdot \text{out}(a, b, k) \cdot \text{out}(a, c, l) \\
+ \text{out}(a, c, l) \cdot \text{in}(a, b, k)
\end{align*}
\]
A.2.5 Process graph

To every closed process expression (a process expression in which no variable occurs), a process graph can be associated by using the action relations. The process graph for the example derived in the previous section is depicted below:

![Process graph](image_url)

Figure A.2: Process graph for the example sequence diagram

Note that this process graph is, except for the edge labeling, exactly the same as the finite state machine from figure 4.12. This should be the case as both figures represent the same sequence diagram.

A.3 Petri-Net

A Petri-net is a graphical and mathematical modeling tool [Pet62]. It consists of places and transitions connected by arrows. Places are connected to transitions and transitions are connected to places. Places can have tokens which are needed to change the state of the system. When a transition has a token in each of its input places, it is considered enabled and can ‘fire’, i.e. perform a transition to a new system state. When a transition fires, it removes a token from each of its input places and places a token in each of its output places.

This section describes how a Petri-net can be used to represent sequence diagram, the conversion presented in this section is based on [CSB01, GRG93].

A.3.1 Conversion

The net is constructed from a few different building blocks. For instance initialization and instance destruction two constructs are needed, these are depicted in figure A.3. In the instance initialization node a token is present, this is because the execution of a sequence diagram starts in this node.

For the exchange of asynchronous and synchronous messages two different patterns are needed. Both these patterns are depicted in figure A.4.

The label $(m, s, r)$ is shorthand notation for (message id, sender id, receiver id). The difference between the asynchronous and the synchronous pattern lies in the synchronization transition, this ensures that the left process cannot continue its actions until the right process has received the message.
A.3.2 Example

The petri-net acquired from the example sequence diagram of figure A.1 is depicted in figure A.5. For every message the corresponding building block presented in the previous subsection should be used.
Figure A.5: Petri-net for the example sequence diagram
Appendix B

Tool Details

This appendix contains the details that were omitted in chapter 5. In the first section the requirements are made specific and assigned priorities to. The second section describes the details of the design of the tool.

B.1 Requirements

This appendix states the requirements for the tool. Each requirement has a unique identifier, a description, and a priority from 1 to 4 assigned. The priorities were assigned based on the importance of the requirement for the project and the estimated amount of work it would take to implement the requirement. The meaning of each of the priority levels is described in the table below.

<table>
<thead>
<tr>
<th>Priority level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The tool will implement this requirement</td>
</tr>
<tr>
<td>2</td>
<td>The tool should implement this requirement</td>
</tr>
<tr>
<td>3</td>
<td>The tool might implement this requirement</td>
</tr>
<tr>
<td>4</td>
<td>It would be nice if the tool implements this requirement</td>
</tr>
</tbody>
</table>

Table B.1: Priority level description

B.1.1 Constraint Requirements

The table below summarize the constraint requirements described in this section.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR01a</td>
<td>The tool will run on the Microsoft Windows XP platform.</td>
<td>1</td>
</tr>
<tr>
<td>CR01b</td>
<td>It would be nice if the tool could run on other platforms.</td>
<td>4</td>
</tr>
<tr>
<td>CR02</td>
<td>The tool should run responsively on a computer equivalent at least to a 2 GHz Pentium 4 processor with 512 MB internal memory and a DirectX 8 compatible graphics card with 64 MB memory.</td>
<td>2</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR03</td>
<td>The language of the tool will be English.</td>
<td>1</td>
</tr>
<tr>
<td>CR04a</td>
<td>The tool will be able to import UML models from XMI files.</td>
<td>1</td>
</tr>
<tr>
<td>CR04b</td>
<td>The tool might be able to import UML models from MDX files.</td>
<td>3</td>
</tr>
<tr>
<td>CR05</td>
<td>It would be nice if additional analysis techniques can be added easily.</td>
<td>4</td>
</tr>
</tbody>
</table>

Table B.2: Constraint Requirements

---

### B.1.2 Functional Requirements

The table below contains all the functional requirements the tool should fulfil.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR01</td>
<td>The UML diagrams will be stored in a database.</td>
<td>1</td>
</tr>
<tr>
<td>FR02</td>
<td>Sequence diagrams in the UML model will be visualized.</td>
<td>1</td>
</tr>
<tr>
<td>FR03a</td>
<td>It will be possible to generate Promela representations of single sequence diagrams.</td>
<td>1</td>
</tr>
<tr>
<td>FR03b</td>
<td>It might be possible to convert single sequence diagrams to finite state machines.</td>
<td>3</td>
</tr>
<tr>
<td>FR03c</td>
<td>It might be possible to convert any composition of sequence diagrams to a Promela representation.</td>
<td>3</td>
</tr>
<tr>
<td>FR04a</td>
<td>Sequence diagrams will be stored internally.</td>
<td>1</td>
</tr>
<tr>
<td>FR04b</td>
<td>Promela representations of sequence diagrams will be stored internally.</td>
<td>1</td>
</tr>
<tr>
<td>FR04c</td>
<td>Finite state machine representations of sequence diagrams will be stored internally.</td>
<td>1</td>
</tr>
<tr>
<td>FR05</td>
<td>It should be possible to display statistical data.</td>
<td>2</td>
</tr>
<tr>
<td>FR06a</td>
<td>It would be nice if coverage percentages can be calculated.</td>
<td>4</td>
</tr>
<tr>
<td>FR06b</td>
<td>It would be nice to create a list of classes not used in sequence diagrams.</td>
<td>4</td>
</tr>
<tr>
<td>FR06c</td>
<td>It would be nice to create a list of methods not used in sequence diagrams.</td>
<td>4</td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
<td>Priority</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>FR07a</td>
<td>It should be possible to indicate sequence diagrams with uninstantiated classifiers.</td>
<td>2</td>
</tr>
<tr>
<td>FR07b</td>
<td>It should be possible to indicate classifier roles without associated classifiers in sequence diagrams.</td>
<td>2</td>
</tr>
<tr>
<td>FR07c</td>
<td>It should be possible to indicate sequence diagrams with unnamed messages.</td>
<td>2</td>
</tr>
<tr>
<td>FR07d</td>
<td>It should be possible to indicate sequence diagrams with messages with unspecified methods.</td>
<td>2</td>
</tr>
<tr>
<td>FR07e</td>
<td>It would be nice if incoherent sequence diagrams can be indicated.</td>
<td>4</td>
</tr>
<tr>
<td>FR07f</td>
<td>It should be possible to indicate sequence diagrams with unused lifelines.</td>
<td>2</td>
</tr>
<tr>
<td>FR08a</td>
<td>There should be a means to detect sequence diagrams that are not initiated by a user.</td>
<td>2</td>
</tr>
<tr>
<td>FR08b</td>
<td>There should be a means to detect sequence diagrams in which the user does not receive feedback.</td>
<td>2</td>
</tr>
<tr>
<td>FR09</td>
<td>It might be possible that the call pattern is visualized.</td>
<td>3</td>
</tr>
<tr>
<td>FR10a</td>
<td>It might be possible that recurring event sequences are detected.</td>
<td>3</td>
</tr>
<tr>
<td>FR10b</td>
<td>It might be possible that sequence diagrams in which is deviated from the recurring event sequences are detected.</td>
<td>3</td>
</tr>
<tr>
<td>FR11a</td>
<td>It will be possible to apply LTL model checking on single sequence diagrams.</td>
<td>1</td>
</tr>
<tr>
<td>FR11b</td>
<td>It might be possible that LTL model checking can be applied on any composition of sequence diagrams.</td>
<td>3</td>
</tr>
<tr>
<td>FR11c</td>
<td>It should be possible to show a violating trace if an LTL formula does not hold.</td>
<td>2</td>
</tr>
<tr>
<td>FR12a</td>
<td>There will be an editor for entering LTL formulas.</td>
<td>1</td>
</tr>
<tr>
<td>FR12b</td>
<td>There should be an enhanced editor for entering LTL formulas.</td>
<td>2</td>
</tr>
<tr>
<td>FR12c</td>
<td>There should be a library of standard LTL formulas.</td>
<td>2</td>
</tr>
<tr>
<td>FR12d</td>
<td>The library of standard LTL formulas should be extensible.</td>
<td>2</td>
</tr>
<tr>
<td>FR13</td>
<td>It will be possible to translate LTL formulas to a representation suitable for SPIN.</td>
<td>1</td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
<td>Priority</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>FR14</td>
<td>There will be a means to check the syntax of LTL formulas.</td>
<td>1</td>
</tr>
<tr>
<td>FR15a</td>
<td>It might be possible to visualize event sequences in a textual format.</td>
<td>3</td>
</tr>
<tr>
<td>FR15b</td>
<td>It would be nice if event sequences are visualized in a graphical format.</td>
<td>4</td>
</tr>
<tr>
<td>FR15c</td>
<td>It might be possible to visualize unexpected event sequences in a textual format.</td>
<td>3</td>
</tr>
<tr>
<td>FR15d</td>
<td>It would be nice if unexpected event sequences are visualized in a graphical format.</td>
<td>4</td>
</tr>
<tr>
<td>FR16</td>
<td>It would be nice if it is possible to export the analysis to a human readable format.</td>
<td>4</td>
</tr>
</tbody>
</table>

Table B.3: Functional requirements

B.2 Design

B.2.1 Class Diagrams

This section contains a description of all the classes in the class diagram depicted in figure 5.4. The class diagrams are grouped per package, the package diagram is depicted in figure 5.3.

B.2.1.1 Internal Storage

The class diagram for the internal storage consists of three classes, one for each of the different representations it has to store. The class diagram is depicted in figure B.1.

The names of the attributes and methods are very similar and so are their functions. But because they store completely different representations they are modeled in different classes rather than as subclasses of a generic storage class.

The private attributes x-SequenceDiagrams store the representation of the sequence diagrams, for the class TSequenceStorage these are the sequence diagram models, for the class TPromelaStorage these are the Promela representations of the sequence diagrams, and for the class TFSMStorage these are the finite state machine representations of the sequence diagrams. The public Store-x-SequenceDiagram methods are used for storing the different representations. The public methods Retrieve-x-SequenceDiagram(SDid) are used to retrieve the representation of the sequence diagram with the sequence diagram identifier that is provided as a parameter to the method. Storing and retrieving is modeled in this way to avoid access to the private attribute by instantiations of other classes.
B.2.1.2 Sequence Diagram Representation Storage

The sequence diagram models are stored according the data model presented in section 4, for
the sake of completeness it is depicted again in figure B.2.

![Diagram]{SequenceDiagram}

**Figure B.2: Entity-relationship diagram: Sequence diagram representation**

A sequence diagram has a name, a unique identifier, and contains zero or more classifier roles
and messages. Each classifier role has a unique identifier and a name assigned. A classifier
role should also be associated to a classifier with an identifier and a name. If the fields
with the identifier and name of a classifier are empty the classifier role is not associated to a
classifier. Each message also has a unique identifier and a name, but also a sender identifier,
a sender name, a receiver identifier, and a receiver name to indicate the identifiers and names
of respectively the sending and receiving classifier role of the message. A message also has a
Boolean attribute to indicate whether it is a synchronous or an asynchronous message and a
sequence number attribute to indicate the order in which the messages occurs in the sequence
diagram. A message should always be related to a method in the class diagram. The method
a message is related to is described by the method identifier. If this field is empty, the message

---

**Figure B.1: Class diagram: Internal storage**

```plaintext
TSequenceDiagramStorage
  - SequenceDiagrams
    + StoreSequenceDiagrams
    + RetrieveSequenceDiagram(SDid)
TFormulaStorage
  - FormulaSequenceDiagrams
    + StoreFormulaSequenceDiagram
    + RetrieveFormulaSequenceDiagram(SDid)
TFSMStorage
  - FSMSequenceDiagrams
    + StoreFSMSequenceDiagram
    + RetrieveFSMSequenceDiagram(SDid)
```
is not related to any method in the class diagram.

B.2.1.3 Promela Representation Storage

The Promela representations of the sequence diagrams are stored according to the structure depicted in the entity-relationship diagram of figure B.3:

![Entity-relationship diagram: Promela representation](image)

A Promela representation of a sequence diagram has a unique identifier and zero or more processes and channels. Each channel has a unique identifier and an attribute to indicate whether it is synchronous or asynchronous. Each process also has a unique identifier and one or more events. An event has an order, which is its primary key, that indicates the order of occurrence of the event in a process. An event also has an identifier and an event type (send or receive), note that this identifier does not have to be unique because each send event and its corresponding receive event share the same identifier.

B.2.1.4 Finite State Machine Representation Storage

The finite state machine representations of the sequence diagrams are stored according to the structure depicted in the entity-relationship diagram of figure B.4:

![Entity-relationship diagram: Finite state machine representation](image)
A finite state machine representation of a sequence diagram has a unique identifier and zero or more states and transitions. Also, each finite state machine has a unique start and end state. All states have a unique identifier to distinguish between them. Each transition consists of a state from where the transition can occur, the event triggering the transition, and a state to which the system will transfer to.

B.2.1.5 Representation Converter

The class diagram for the representation converter consists of one superclass and two sub-classes, one for every type of conversion that has to be carried out. The class diagram for the representation converter is depicted in figure B.5.

![Class diagram: Representation converter](image)

Figure B.5: Class diagram: Representation converter

This structure is chosen because all converters need to have a Convert method. Also, all converters need to have access to the sequence diagram models and it is inconvenient to model a method for that twice.

The protected method GetSequenceDiagram(SDid) is used to retrieve the sequence diagram model with identifier SDid from the internal storage so it can be converted. For this the retrieve method from the TSequenceDiagramStorage class can be used. The public Convert methods are used to convert a sequence diagram model to another representation, depending on which converter is invoked. For storing the created representations in the internal storage the public StoreRepresentation methods are used. These store methods use the store methods from the TPromelaStorage and the TFSMStorage classes from the internal storage, depending on which representation needs to be stored.

B.2.1.6 Statistics Extractor

The class diagram for the statistics extractor, which is depicted in figure B.6, consists of two classes. One is for extracting the basic statistics and the other one is for performing the coverage analysis.

The public method ExtractStatistics is used to extract statistics from the sequence diagrams in the UML model that is retrieved using the private method GetSequenceDiagrams. The other, private, extract methods are used to extract the specific statistics from the sequence diagrams.

The coverage analyzer works in a similar way. The sequence diagrams are extracted by the private method GetSequenceDiagrams and the classes and methods in the UML model are
extracted by the private method \texttt{GetClassesAndMethods}. The other three private methods are used to extract the coverage data.

### B.2.1.7 Syntax Analyzer

The class diagram for the syntax analyzer is also one that consists of only one class, it is depicted in figure B.7.

![Class diagram: Syntax analyzer](image)

To find syntactic defects in the sequence diagrams in the UML model, which are retrieved by the private method \texttt{GetSequenceDiagrams}, the public method \texttt{FindSyntacticDefects} is used. The defects will be extracted using the various, private, extract methods. The naming of the methods differs from the syntactical defects that are considered in section 4.2, the mapping of the methods to the defects is given in table B.4.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Defect</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ExtractUnnamedLifelines}</td>
<td>Uninstantiated Classifiers</td>
</tr>
<tr>
<td>\texttt{ExtractUntypedLifelines}</td>
<td>Classifier Roles without Associated Classifiers</td>
</tr>
<tr>
<td>\texttt{ExtractUnnamedMessages}</td>
<td>Unnamed Messages</td>
</tr>
<tr>
<td>\texttt{ExtractUnspecifiedMethods}</td>
<td>Messages with Unspecified Methods</td>
</tr>
<tr>
<td>\texttt{IsIncoherent}</td>
<td>Incoherence</td>
</tr>
<tr>
<td>\texttt{ExtractUnusedLifelines}</td>
<td>Unused lifelines</td>
</tr>
</tbody>
</table>

Table B.4: Mapping of methods to defects to detect
B.2. Design

B.2.1.8 Patterns Detector

The patterns detector class diagram, which is depicted in figure B.8, is also a small one, it consists of only one class.

![Class diagram: Patterns detector](image)

The public method DetectPatterns can be invoked to detect the patterns that occur in the sequence diagrams of the loaded UML model. The sequence diagrams are retrieved by calling the private method GetSequenceDiagrams. The other methods, which are all private, detect the different patterns in the sequence diagrams.

B.2.1.9 Model Checker

The class diagram for the model checker consists of four classes, it is depicted in figure B.9.

![Class diagram: Model checker](image)

The TModelChecker class is used verify whether an LTL formula holds in the Promela model of a sequence diagram or not. The public method Verify is used to initiate the process of verifying whether the LTL formula LTLFormula holds in the composition of sequence diagrams with identifiers in the list SDids. When only one item is in the list, model checking is performed on a single sequence diagram. The first step in the verification process is the
invocation of the private method GetNeverClaim to generate a never-claim of the LTL formula by instantiating an LTL syntax checker. SPIN needs a never-claim for verification, it is nothing more than the negation of an LTL formula. Thereafter an input file for SPIN is needed. To this purpose the private method GetInputFile is used. This method will instantiate an input file generator to generate the input file. When the input file is generated a verifier needs to be generated, to this extent the private method GenerateVerifier can be used. After the verifier is generated it has to be converted to an executable verifier, the private method GenerateExecutableVerifier is used for this. When the executable is ready it will be executed by the private method RunVerifier. The results of the verification will subsequently be analyzed by an instantiation of an output analyzer.

The function of the TLTLSyntaxChecker is more than only generating a never claim that can be used for the verification, it can also be used, as its name suggests, to check whether the syntax of an LTL formula is correct. The public method GenerateNeverClaim is used for the former function and the public method CheckLTLSyntax is used for the latter function of this class. Both methods take an LTL formula as input and need all private methods of this class to perform their task. First the private method DenyLTLFormula takes an LTL Formula as input and returns the same LTL Formula in a negated form. Then the private method TryToGenerateNeverClaim tries to generate a never-claim from the denied LTL formula it gets as a parameter. Finally the private method AnalyzeOutput checks whether a never claim or an error message, indicating that the syntax of the formula was wrong, was returned.

The InputFileGenerator class generates an input file for SPIN of the Promela representation of the sequence diagrams with identifier in the list SDids. The private method GetPromelaRepresentation is used to retrieve the Promela representation, it uses the RetrievePromelaSequenceDiagram method from the internal storage. All private Generate-x methods are used to generate parts of the input file from the Promela representation.

The class TOutputAnalyzer is used to analyze the output SPIN produces when the validity of an LTL formula in a Promela model of a sequence diagram is verified. The public method AnalyzeOutput is used to perform the analysis by invoking the two private methods. The private method GetOutput is used to read the output file that SPIN has produced. The other private method, ExtractResultsFromOutput, is used to extract the necessary details from SPIN’s output file. If the formula did not hold this includes the violating trace.

### B.2.1.10 Behavior Visualizer

The behavior visualizer class diagram, which is depicted in figure B.10, consists of only one class with all public methods.

![Behavior Visualizer Class Diagram](image)

Figure B.10: Class diagram: Behavior visualizer

The methods are used for generating two different types of visualizations, textual and graphical, of all behavior and of unexpected behavior.
B.2.1.11 HTML Exporter

The class diagram for the HTML exporter consists of one class, it is depicted in figure B.11.

\[
\begin{array}{|c|}
\hline
\text{HTMLExporter} \\
\hline
\text{- ExtractStatistics} \\
\text{- PerformCoverageAnalysis} \\
\text{- FindSyntacticDefects} \\
\text{- DetectPatterns} \\
\text{- ExportToHTML} \\
\hline
\end{array}
\]

Figure B.11: Class diagram: HTML exporter

The public method `ExportToHTML` is used to perform the exportation. The other, private, methods are used to invoke the different analysis techniques that need to be included in the HTML report.

B.2.2 Sequence Diagrams

In this subsection the use-cases presented in figure B.12 will be explained in more detail and a sequence diagram is given for the realization of each of them. For the sake of completeness, the use-case diagram will be depicted here again.

![Sequence Diagram](image)

Figure B.12: Use-case diagram
B.2.2.1 Load UML model

When the user wants to perform an analysis on the sequence diagrams of a UML model, the model has to be loaded and the sequence diagrams have to be extracted from it. Before the analysis can start the sequence diagrams have to be converted and stored internally. This is all done when an UML model is loaded, the sequence diagram representing this behavior is depicted in figure B.13.

![Sequence diagram: Load UML model](image)

Figure B.13: Sequence diagram: Load UML model

The underlined messages are not documented here, instead the reader is referred to [Wij06].
B.2.2.2 Give Sequence Diagram Model Statistics and Perform a Coverage Analysis

When statistical data about a sequence diagram model is required, this model has to be retrieved from internal storage. After it has been retrieved the statistics can be extracted per sequence diagram. For the coverage analysis also the sequence diagrams of the model have to be retrieved, but also some information is extracted from the class diagrams. If the necessary data is retrieved the different parts of the coverage analysis can be performed.

Note, the extraction of the statistics and the performance of the coverage analysis may be executed simultaneously.

Figure B.14: Sequence diagram: Give sequence diagram model statistics and perform a coverage analysis
B.2.2.3 Perform a Syntax Analysis

For a syntax analysis, again, first the sequence diagram model needs to be retrieved from internal storage. When this is done, the sequence diagrams are searched one-by-one for syntactic defects.

Figure B.15: Sequence diagram: Perform a syntax analysis
B.2. Design

B.2.2.4 Detect Patterns in the Sequence Diagrams

To detect the patterns that occur in a model, first the sequence diagram model needs to be retrieved from internal storage. When this is done, the sequence diagrams are searched one-by-one for patterns.

![Sequence diagram: Detect patterns in the sequence diagrams](image)

Figure B.16: Sequence diagram: Detect patterns in the sequence diagrams
B.2.2.5 Check the Syntax of an LTL Formula

When a user wants to verify the correctness of the syntax of an LTL formula, the formula is first converted to its negated form. After that, an attempt to generate a never-claim is done. When this is done the SPIN output is analyzed, if a never-claim is returned the syntax of the LTL formula was clearly correct, if an error message is returned the syntax of the LTL formula was incorrect. If the syntax was correct, this is reported to the user, if the syntax was incorrect SPIN’s output, which gives information on what is wrong, is returned.

Figure B.17: Sequence diagram: Check the syntax of an LTL formula
B.2. Design

B.2.2.6 Verify the Validity of an LTL Property in a Sequence Diagram

When the validity of an LTL property in a sequence diagram has to be checked, first a never-claim has to be generated. This is done in the same way as described in the previous sequence diagram. If the syntax of the LTL formula is incorrect the model checking stops, if it is correct it continues with the generated never-claim. When the never-claim has been generated, an input file has to be generated. This is done by an input file generator. The first thing the input file generator does is retrieving the Promela models of the sequence diagrams in which the validity of the LTL formula needs to be checked from internal storage. Hereafter it generates an input file from this model. When the input file is ready, the actual model checking can start. This is done by first creating a verifier with SPIN. When the verifier is generated, it can be compiled, using the GNU Compiler Collection (GCC), into an executable verifier. The last step in the model checking process is running the executable verifier. When the actual model checking is done, the output file generated by the executable verifier is analyzed by an output analyzer. The results of this analysis are displayed to the user.

![Sequence diagram](image)

Figure B.18: Sequence diagram: Verify the validity of an LTL property in a sequence diagram
B.2.2.7 Show All Possible Event Sequences in a Sequence Diagram

When the user wants to see which behavior is modeled in the sequence diagram it can ask for a finite state machine representing all this behavior. To create such an FSM, first it needs to be retrieved from an FSM storage object. If it is not present there it will be generated by first retrieving a sequence diagram model from storage and converting that to a finite state system representation. When the FSM is ready it can be used to create a graphical representation of it.

![Sequence diagram: Show all possible event sequences in a sequence diagram](image)

Figure B.19: Sequence diagram: Show all possible event sequences in a sequence diagram

The other visualizations can be generated in a similar way. The only thing that changes is that the `VisualizeAllGraphical(SDid)` method is replaced by any of the three others in the `TBehaviorVisualizer` class.
Appendix C

SquAT User Manual

This appendix contains the user manual of the tool that was created as part of the project. This manual is also included in the installer of the tool.

C.1 Introduction

SquAT (Sequence Diagram Analysis Tool) is a tool that allows import of a UML model to perform analysis on the sequence diagrams contained in that model.

To use the model checking capabilities in SquAT it is needed that the Gnu Compiler Collection (GCC) is installed and available in the Windows search path. GCC can be downloaded from http://www.delorie.com/djgpp/.

To enable the use of the graphical export facilities it is needed that the Graph Visualization Software (Graphviz) toolset is installed, this can be acquired from http://www.graphviz.org.

After installation, SquAT can be started by selecting the SquAT shortcut in the Windows start menu. The first time the application is started, the internal database is created, which takes some time.

C.2 Window Arrangement

The main window, which is depicted in figure C.1, consists of a menu bar, a tool bar, a diagrams frame and an analysis frame. The size of the diagram and analysis frame can be changed by dragging the splitter that divides the two. The function of the different parts of the main window will be described in the remainder of this section.

C.2.1 Menu bar

The menu bar contains three menus:

- File,
- Analysis, and
- Settings.

The file menu contains controls for loading and unloading UML models, as well as for closing SquAT. The file menu also has a control that can be used to export the contents of the diagram
frame to a file, the supported file formats are JPEG and PNG. Note that this feature is known to have some problems and may cause the save dialog to be drawn over the diagram frame itself, this can be worked around by dragging this dialog away from the diagram frame.

The analysis menu contains controls for performing multiple analysis techniques on the sequence diagrams in the UML model that is currently loaded, these analysis techniques are described in section C.3. This menu also contains a control for exporting (parts of) this analysis to a HTML-file.

The settings menu contains a control to open the dialog that can be used for changing the settings of SquAT, the different settings will be explained in section C.4. Also a control is available to get access to the internal database (use this with care!) and a control to extend the user ontology used in one of the analysis techniques. The last control in this menu is used to clear the log file that SquAT keeps.

C.2.2 Tool bar

The tool bar, which is depicted in figure C.2, contains six controls.

![Tool bar](image)

Figure C.2: Tool bar

The first control is used for loading a UML model and the second for unloading the currently loaded model. The third icon is used for enabling the diagram frame, this is needed because it is initially disabled. The fourth icon is used to zoom the view in the diagram frame to fit all the diagrams in the loaded model. The fifth icon is used for exporting (parts of) the analysis of the UML model to a 'html' file, similar to the control in the analysis menu (see section C.2.1). The last control is used to open the dialog for changing the settings of SquAT (see section C.4).

C.2.3 Diagrams Frame

Initially the diagrams frame is shown as an empty grey rectangle. When a UML model is loaded the ‘Show diagrams’ control from the tool bar can or the ‘F2’ button can be pressed...
to visualize all sequence diagrams contained in the loaded model.

The viewpoint can be changed by holding the middle mouse button and moving the mouse pointer. When the shift button is pressed while changing the viewpoint, it will tilt.

Zooming can be done by using the scroll wheel of the mouse. Also a diagram or diagram element can be zoomed in by double clicking on it.

The ‘Defocus’ control from the tool bar or the ‘Escape’ button can be pressed at any time to zoom the view in the diagram frame to fit all the diagrams in the loaded model.

C.2.4 Analysis Frame

This part of the window is used to get access to the different analysis techniques that can be performed on a UML model, these techniques are described in section C.3.

C.3 Analysis

SquAT can perform five analysis techniques on the sequence diagrams in a UML model:

- LTL model checking,
- statistical analysis,
- syntactic analysis,
- pattern detection, and
- visualization of behavior.

These analysis techniques are shortly described in the remainder of this section.

C.3.1 LTL Model Checking

The feature SquAT was originally designed for is LTL (Linear Temporal Logic) model checking, the bottom part of the analysis frame is entirely devoted to it. LTL model checking can be used to verify whether the order of send and receive events of messages that occur in a sequence diagram is correct.

The text box on top can be used to enter an LTL formula, the controls below it can be used to assist in creating the formula. After an LTL formula has been created its syntax can be checked by pressing the ‘Check LTL syntax’ button, the result of this check will be displayed in the verification result text box on the bottom of the analysis frame. There is also a library available with commonly used LTL formulas, this library can be accessed by pressing the ‘Standard LTL formulas’ button. In the dialog that is displayed the user can select an LTL formula and he or she can add or remove LTL formulas.

After specifying an LTL formula, the constants that are used need to be defined. To this purpose the list boxes and radio buttons in the middle part of the analysis frame can be used. For every constant, the first list box can be used to select the sequence diagram in which an event must be considered and in the second list box can be used to select the message of which the send or receive event must be considered. The radio buttons are then used to select whether the send or receive event must be considered.
When the LTL formula is specified and the constants are defined the ‘Verify’ button can be pressed to pop up a dialog that can be used to specify the diagrams that have to be used in the verification. When this is done the verification can start. The result of the verification is displayed in the verification result text box on the bottom of the analysis frame. If the specified LTL formula does not hold a sequence of events will be given that violates the property specified by the LTL formula.

C.3.2 Statistics

When the ‘Extract model statistics’ button is pressed the following statistics will be displayed:

- number of sequence diagrams considered in the analysis,
- number of synchronous and asynchronous messages modeled,
- number of invocations per method modeled,
- frequency table with used classifier roles,
- frequency table with used classifiers.

The statistical analysis also contains a coverage analysis, i.e. it finds the percentages of methods and classifiers that are used in the sequence diagrams. Also the unused operations and classifiers are indicated.

C.3.3 Syntactic Analysis

The syntactic analysis is performed by pressing the ‘Syntactic analysis’ button. This analysis technique checks the sequence diagrams in the loaded UML for the following syntactic errors and inconsistencies:

- uninstantiated classifiers,
- lifelines without associated classifiers,
- unnamed messages,
- messages with unspecified methods,
- incoherence,
- unused lifelines.

Diagrams that contain such errors or inconsistencies are listed. Note that a diagram with an unused lifeline is a special case of an incoherent diagram, so diagrams with unused lifelines are also considered to be incoherent.
C.3.4 Pattern Detection

By pressing the ‘Pattern detection’ button, the pattern detection dialog will pop up, in which a pattern type can be selected. SquAT is able to detect three types of patterns in the sequence diagrams of a UML model:

- user patterns,
- call pattern,
- recurring message sequences.

When the detection of user patterns is selected, diagrams in which the user does not initiate the interaction and diagrams in which a user occurs but does not receive any feedback are listed. The actor types that are considered to be users can be expanded by adding synonyms for ‘user’ to the user ontology, this can be done by using the ontology editor (see section C.5).

The call pattern is a graphical representation of the calling behavior of the objects modeled as lifelines. In the grid the rows represent the calling objects and the columns represent the called objects. The numbers in the cells of the grid represent the number of times the reference occurs, this is also indicated by the color the cells have.

Certain sequences of messages recur in the sequences diagrams of a UML model, if this frequency of recurrence exceeds a certain threshold value it is considered to be a pattern (see section C.4). All these patterns are displayed as well as all message sequences in which one of the messages occurring in a pattern occurs but that deviates from the pattern.

C.3.5 Visualization of Behavior

To fully use all the capabilities of this feature the Graphviz toolset should be installed (see section C.1). When the ‘Visualize event sequences’ button is pressed a dialog will pop up in which the diagram of which the behavior should be visualized and the type of visualization can be selected. The visualization can be either textually or graphically and there can be chosen between visualizing all behavior and unexpected behavior\(^1\), which leads to four visualizations in total. There is also a checkbox that can be used to generate finite state machines of the unexpected behavior of all sequence diagrams in the model.

After the visualization settings are set, a finite state machine is created from the selected sequence diagram. Note that generating a finite state machine of a sequence diagram with a lot of asynchronous communication can take a very long time. When the textual visualization is selected, a text box will pop up with all possible sequences of events that can occur in the selected sequence diagram displayed. When the graphical visualization is selected a save dialog pops up, use this to specify the name, location and file type to save the graphical representation of the finite state machine that will be created. When unexpected behavior is visualized, the expected behavior is indicated in the graphical representation of the finite state machine by dotted arrows.

It is also possible to check the checkbox labeled ‘Visualize unexpected behavior of all diagrams’. If this is done a file type and a location will be asked whereafter finite state machines indicating unexpected behavior for all diagrams in the model are generated in the chosen file type and saved to the chosen location.

---

\(^1\)Unexpected behavior is defined as all behavior deviating from the visual order.
The layout engine that is used for creating the graphical representation can be selected in the settings menu (see section C.4).

C.4 Settings

The settings dialog, which is depicted in figure [C.3] allows for changing the main settings needed by SquAT.

![Settings dialog](image)

Figure C.3: Settings dialog

When the first checkbox is checked notes that are placed on the sequence diagrams are displayed in the diagrams frame. If unchecked, they are not.

The next block of two settings are related to the XMI parser SquAT uses. In the top list box one of the parser types that is available can be selected. Currently available are parsers for:

- Rational Rose (Unisys Rose UML 1.3.2 Export, XMI 1.1),
- MDX file of Rational XDE 1.5.0,
- Enterprise Architect 6.1 (Unisys export format),
- Borland Together 6.0,
- MagicDraw UML 7.0,

Parsers can be added manually later.

The check box ‘Use predecessor relation defined in XMI’ was used for debugging purposes, it should always remain checked. If unchecked SquAT does not use the predecessor relation that is defined in the XMI file, instead it will consider the messages to be ordered in the order they are stored in the XMI file which may be different.

The second list box allows the user to select the layout engine that will be used for generating a finite state machine of the event sequences contained in a sequence diagrams, the options are ‘dot’ and ‘neato’. For more information on this layout engines the reader is referred to [http://www.graphviz.org](http://www.graphviz.org).

The text box can be used to enter the threshold value for the message sequence pattern detection. It represents the number of times a message sequence must occur at least in a model to be considered to be a pattern. This should be a positive integer value larger than zero.
C.5 Ontology Editor

The ontology editor can be used for adding or removing synonyms for ‘user’ that will be used in the pattern analysis regarding the role of a user in sequence diagrams. When users are modeled as actors in the UML model they will be automatically recognized as users, tough it occurs that they are not modeled as users and in that case the ontology database is consulted.

The ontology editor works in an intuitive way, so no extra explanation will be given.
Appendix D

Validation

This appendix contains one of the case studies conducted and the template used during the structured interview.

D.1 Case Study

In this section the case study of the SquAT tool is presented to give an indication of what the results of a case study look like. Although this case study is rather small, it contains enough detail to give a good idea of the contents of a case study.

D.1.1 Introduction

This document contains the results of the case study performed on the XMI file of the model of the SquAT tool with the SquAT tool. This model was created with Rational Rose.

The purpose of this case study is to indicate syntactical inconsistencies and patterns in the sequence diagrams that are present in the model. Also, potentially unexpected behavior contained in the sequence diagrams is indicated.

Section D.1.2 will describe the contents of the analysis and section D.1.3 contains the results of the case study performed on the models.

D.1.2 Contents of the Analysis

This section shortly describes the analysis techniques that were used to perform the analysis on the sequence diagrams in the UML models.

D.1.2.1 Statistical Analysis

This analysis technique is used to extract basic statistics of the sequence diagrams in the UML model. These basic statistics are:

- Number of sequence diagrams considered in the analysis
- Number of (a)synchronous messages modeled
- Number of invocations per method modeled
- Number of used classifier roles
• Number of used classifiers

These statistics can be used to find classes and/or methods that are used very often.

The statistical analysis also contains a coverage analysis, i.e. it finds the percentages of methods and classifiers that are used in the sequence diagrams. Also the unused operations and classifiers are indicated.

### D.1.2.2 Syntactical Analysis

When modeling sequence diagrams, chances are that sometimes a message or role-name is forgotten or that an obsolete lifeline is added. This section describes six types of syntactic inconsistencies that can (and do) easily slip into a sequence diagram.

#### D.1.2.2.1 Uninstantiated Classifiers

In a lot of sequence diagrams there are lifelines that do not have an instance name. This is not beneficial for the readability and understandability of the diagram, which can lead to misinterpretation and miscommunication.

#### D.1.2.2.2 Lifelines Without Associated Classifiers

It also occurs that lifelines are used in a diagram that are not associated to a classifier. Typically this means that the lifeline representing a class instance is not linked to a class, i.e. the instance name is present but no class name is associated with it. This would mean that behavior of a non-existing class is modeled. As this leads to an inconsistent model this is undesirable.

#### D.1.2.2.3 Unnamed Messages

It also frequently occurs that messages do not have a name, when the name is missing the message cannot be related to a method. This again is not beneficial for the readability and understandability of the diagram.

#### D.1.2.2.4 Messages With Unspecified Methods

Messages in a sequence diagram represent methods in a class diagram, therefore there should be a one-to-one mapping from messages to methods. This is however not always the case, often messages are modeled that do not have a method associated with them. This is not wanted because this leads to an inconsistent model.

#### D.1.2.2.5 Incoherence

Incoherence in a sequence diagram means that two or more disjoint sets of objects only communicate mutually but not with objects of another set. Consider for example a sequence diagram with four lifelines \(A\), \(B\), \(C\), and \(D\). If now lifeline \(A\) and \(B\) communicate only with each other and lifeline \(C\) and \(D\) communicate only with each other this sequence diagram is considered to be incoherent. This anomaly is most likely to occur when one or more message arrows are not modeled.
D.1.2.2.6 Unused Lifelines
It sometimes occurs that an object’s lifeline is depicted in a sequence diagram but is does not send or receive any messages. There are a few explanations for this. When an alternative flow is modeled in a separate sequence diagram that does use the object it often remains in the main flow to indicate that it is involved in the modeled interaction. It is also possible that the object should in fact participate in the interaction, thus indicating that some messages are not modeled. A last, simple explanation is that it is just unused and hence the object is obsolete.

Note that an unused lifeline is a special case of incoherence as described in section D.1.2.2.5.

D.1.2.3 Patterns
The following three types of patterns will be considered:

1. User patterns
2. Call pattern
3. Recurring message sequences

D.1.2.3.1 User Patterns
Usually a user has to work with a system and therefore he (or she) has to be considered in the design of the system.

As sequence diagrams are often realizations of use cases and use cases are modeled from the user’s perspective, most of the time a user will initiate the interaction modeled in a sequence diagram. However, it can occur that the user is unintentionally not modeled at all which results in interactions being initiated by another object. This can lead to incorrect behavior which is, of course, unwanted.

When a user initiates an interaction usually some feedback from the system is required. So it is likely that sequence diagrams in which the user initiates the interaction, but does not receive any feedback are flawed.

The sequence diagrams in which the user does not initiate the interaction and the sequence diagrams in which the user does not receive any feedback from the system are indicated.

D.1.2.3.2 Call Pattern
Usually an object calls only a relatively small set of other objects, simply because it only has a limited amount of interfaces. Also, some objects will be called more often than others. To find out which objects communicate with each other and how often a call matrix can be created in which this is indicated. Note that sending messages that are not related to a methods are considered as being a call.

The most recurring things found in the call matrix are indicated.

D.1.2.3.3 Message Sequences
It is not uncommon that a certain sequence of messages recur in the sequences diagrams of a UML model, if this frequency of recurrence exceeds a certain threshold value it is considered to be a pattern. The most striking patterns found are described as well as all message sequences that deviate from this pattern.
D.1.2.4 Unexpected Behavior

Usually a sequence diagram is read from top to bottom and it is assumed that messages are send and received in order of appearance in the diagram. However, interleaving of send and receive events of messages can result in behavior that deviates from this visual order. This behavior might be unexpected or even unwanted, therefore diagrams that have this unexpected behavior are discussed here.

D.1.3 Analysis

The analysis is performed on the seven sequence diagrams that are present in the UML model.

D.1.3.1 Statistical Analysis

The first thing that stands out is the fact that all communication in the sequence diagrams is modeled by asynchronous messages.

Most of the methods are used relatively little in the sequence diagrams, but the methods ‘RetrieveSequenceDiagram(string)’ and ‘SequenceDiagram’ are used more often, both six times. This is depicted in the bar chart of figure D.1.

![Figure D.1: Method Frequency](image)

Also the number of times a classifier appears on a sequence diagram is almost the same for every classifier, although there are three outliers: ‘GUI’, ‘User’, and ‘TSequenceDiagramStorage’. This is depicted in the bar chart of figure D.2.

The coverage analysis shows that 95% of all classifiers is used and that 95.3% of all methods is used, coverage is thus very high. The only classifier not used is ‘TConversion’. There are three unused methods, these are ‘VisualizeAllTextual’, ‘VisualizeUnexpectedTextual’, and ‘VisualizeUnexpectedGraphical’, these methods all occur in the class ‘TBehaviorVisualizer’. 
D.1.3.2 Syntactical Analysis

There are some messages that have no method associated to them, the list of these methods can be found in appendix D.1.4. Apart from this there are no syntactical inconsistencies found.

D.1.3.3 Patterns

D.1.3.3.1 User Patterns

All sequence diagrams are initiated by a user. Only in sequence diagram ‘Load UML Model’ the user does not receive any feedback.

D.1.3.3.2 Call Pattern

From the call pattern can be derived that the classifier ‘User’ only calls the classifier ‘GUI’. This is to be expected as the user has to work with the system.

The classifier ‘GUI’ is the most active one, it calls 11 times other classifiers and it is called 9 times.

D.1.3.3.3 Message Sequences

Two message sequence patterns can be found in the model:

- ‘RetrieveSequenceDiagram(String)’ → ‘SequenceDiagram’ (occurs 6 times)
- ‘GetSequenceDiagrams( )’ → ‘RetrieveSequenceDiagram(String)’ (occurs 4 times)

From the second message sequence pattern there are two deviations, both are

‘GetSequenceDiagram(String)’ → ‘RetrieveSequenceDiagram(String)’

From the two patterns can be concluded that the following pattern occurs four times:

‘GetSequenceDiagrams’ → ‘RetrieveSequenceDiagram’ → ‘SequenceDiagram’
D.1.3.4 Unexpected behavior

There are two sequence diagrams that contain unexpected behavior, ‘Load UML Model’ and ‘Give Statistics’. The pictures of the finite state machines representing this are too large to include here.

D.1.3.5 Conclusions

The model has a high coverage and is, apart from a few messages not related to methods, complete. So from this point of view the model quality is fairly high.

Although the model is quite small still some patterns have been found. It can be concluded that the sequence diagrams are created from a users perspective, this also explains why the classifiers ‘GUI’ and ‘User’ can be found on all sequence diagrams.

What also strikes is the fact that although only asynchronous communication is used there is very little unexpected behavior.

D.1.4 Messages without Associated Methods (Appendix)

This appendix contains the messages used in the sequence diagrams that are not related to methods in a class diagram.

<table>
<thead>
<tr>
<th>Diagram name</th>
<th>Message name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect Patterns</td>
<td>Detect Patterns</td>
</tr>
<tr>
<td></td>
<td>Patterns</td>
</tr>
<tr>
<td></td>
<td>SequenceDiagram</td>
</tr>
<tr>
<td>Give Statistics</td>
<td>Classes and Method Information</td>
</tr>
<tr>
<td></td>
<td>Coverage</td>
</tr>
<tr>
<td></td>
<td>Give Statistics</td>
</tr>
<tr>
<td></td>
<td>SequenceDiagram</td>
</tr>
<tr>
<td></td>
<td>SequenceDiagram</td>
</tr>
<tr>
<td></td>
<td>Statistics</td>
</tr>
<tr>
<td>LTL Syntax Check</td>
<td>Check LTL Syntax</td>
</tr>
<tr>
<td></td>
<td>Correct</td>
</tr>
<tr>
<td>Load UML Model</td>
<td>Diagram Information</td>
</tr>
<tr>
<td></td>
<td>Diagrams</td>
</tr>
<tr>
<td></td>
<td>Load UML Model</td>
</tr>
<tr>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td></td>
<td>SequenceDiagram</td>
</tr>
<tr>
<td>Model Check</td>
<td>InputFile</td>
</tr>
<tr>
<td></td>
<td>LTL Formula is correct</td>
</tr>
<tr>
<td></td>
<td>PromelaSequenceDiagram</td>
</tr>
<tr>
<td></td>
<td>Result</td>
</tr>
<tr>
<td></td>
<td>Verify LTL claim</td>
</tr>
<tr>
<td>Perform Syntax Analysis</td>
<td>Analyze Syntax</td>
</tr>
<tr>
<td></td>
<td>SequenceDiagram</td>
</tr>
<tr>
<td></td>
<td>Syntactic Defects</td>
</tr>
<tr>
<td>Show All Event Sequences</td>
<td>Event Sequences</td>
</tr>
<tr>
<td></td>
<td>FSM</td>
</tr>
</tbody>
</table>
D.2 Interview Template

This section contains the template that was used as guideline during the interviews.

D.2.1 Background

1. What is your role (job description)?
   - Architect
   - Developer
   - Researcher
   - Project manager
   - Consultant
   - Student
   - Other:

2. What is the main task of your responsibility?
   - Requirements
   - Analysis
   - Architecture
   - Design
   - Implementation
   - Testing
   - Maintenance
   - Project Management
   - Other:

3. How many years of experience do you have using the UML?

4. What kind of UML-training did you receive (check all applicable)?
   - University
   - UML-course
   - Self-teaching
   - Literature study
   - No training
   - Other:

5. How good is the UML-training you received to fulfill your UML-related tasks on a scale of 1 to 5?
D.2.2 UML Sequence Diagrams

1. How are sequence diagrams used in the development process?
   - Finding out which interfaces are needed between classes
   - Finding out what logic should be designed in order to provide the desired behavior
   - Prototyping dynamic behavior
   - Communication of system aspects between designers
   - Validation of architecture
   - Other:

2. Are sequence diagrams used differently as the project evolves? If so, how?

3. What percentage of the use cases/system scenarios is usually documented as a sequence diagram?

4. What criteria are used to determine which scenarios are modeled as sequence diagrams?
   - Complexity
   - Criticality/risk
   - Importance
   - Value
   - Other:

5. Do you take into account concurrent execution of sequence diagrams? If so, how?
   - Default interpretation
   - Explicit modeling

6. What does a composition of sequence diagrams mean?
   - Arbitrary interleaving (different object instances)
   - Arbitrary interleaving (same object instances)
   - Sequential
   - Other:

7. How are alternative flows modeled?
   - In a separate sequence diagram
   - Only the differences in another sequence diagram
   - With a note on the diagram
   - Other:

8. Are design patterns used for modeling sequence diagrams? If so, which?

9. Are there any modeling conventions regarding sequence diagrams?
   - Use only synchronous or only asynchronous messages
D.2. Interview Template

- Do (not) draw return messages
- Always/never instantiate classes
- Conventions for features not supported by the modeling tool
- Other:

D.2.3 Case Study

1. What was the purpose of the model?
   - Actual product
   - Research model
   - Other:

2. Specific questions related to the case study performed...

D.2.4 Tool Evaluation

1. How would you rate SquAT’s analysis techniques on a scale of 1 to 5?
   - Model checking
   - Statistic analysis
   - Syntactic analysis
   - User patterns
   - Call pattern
   - Message sequences
   - Visualization of (unexpected) events

2. How would you rate the SquAT tool in general on a scale of 1 to 5?

3. What other analysis techniques would you like to see for UML sequence diagrams?

4. Would you consider using SquAT?