TU WIEN Informatics

Using Model-Based Testing for Creating Behaviour-Driven Tests

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Simon Schneider, BSc
Matrikelnummer 01226825

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Dipl.-Ing. Dr. Tanja Mayerhofer, BSc

Wien, 26. Oktober 2020

Simon Schneider	Gerti Kappel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Using Model-Based Testing for Creating Behaviour-Driven Tests

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Simon Schneider, BSc
Registration Number 01226825

to the Faculty of Informatics

at the TU Wien

Advisor: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Dipl.-Ing. Dr. Tanja Mayerhofer, BSc

Vienna, 26th October, 2020

Simon Schneider	Gerti Kappel

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Simon Schneider, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. Oktober 2020

Simon Schneider

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Our dependency on reliable software keeps growing but so does the size and complexity of the code that powers it. Automated software testing has become a necessity for professional software projects. Behaviour-driven development (BDD) is a popular strategy in modern agile development teams and strives to bring business stakeholders, developers and testers together to jointly describe a system specification in natural language. Using the business domain language and describing the expected behaviour in a "*Given* a precondition; *When* some action is performed; *Then* an outcome is achieved" (GWT) pattern improves communication between the involved parties and lowers the risk of costly misunderstandings. At the same time, it allows the description to be leveraged to drive automated tests verifying the specification is implemented as designed. While this approach certainly works well in practice it suffers from some disadvantages: It is informal and verbose, thus bearing the risk of failing to specify all parts of a system, and it is repetitive and ill-suited for combinatorial testing.

Model-based testing (MBT), on the other hand, is a structured approach for automatically generating test cases out of models describing the system under test following well-defined coverage criteria. It is well-suited for describing complex interactions and cross-linked code paths using models. Even basic graphical state machines can define what takes many pages to write down in natural language. At the same time, models are flexible as they describe a system on a higher level of abstraction and allow to quickly recreate test cases in the event that the behaviour of the system changes or gets extended.

This thesis presents a testing approach that combines BDD with MBT based on state machine models to automate the process of writing BDD test cases following the GWT structure. The intent is not to replace the fundamental principle of BDD—bringing stakeholders together to write the specification—but rather to use MBT where it shines and to avoid tedious and incomplete manual specifications of complex parts of a system. A prototype that generates BDD tests from graphical state machine models has been developed and evaluated in a case study on the example of a voice over IP (VoIP) gateway. The results are promising: The generated test cases covered the functionality of the tested system, and the effort to create them was comparable or even lower to writing similar test cases by hand. A survey among practitioners showed that while users were able to identify automatically generated BDD tests, in some instances, they preferred them over manually written ones.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Unsere Abhängigkeit von zuverlässiger Software wächst und mit ihr die Größe und Komplexität des zugrundeliegenden Quelltextes. Automatisierte Softwaretests sind in modernen Softwareprojekten unabkömmlich. Behaviour Driven Development (BDD) ist eine beliebte Strategie in agilen Entwicklerteams um die involvierten Akteure - Kunde, Entwickler, Tester - zusammenzubringen um gemeinsam eine Systembeschreibung in natürlicher Sprache zu verfassen. Das Verhalten wird dabei in einer domänenspezifischen Sprache mit einer vorgegebenen Struktur beschrieben: "*Gegeben* ist eine Vorbedingung; *Wenn* eine Aktion stattfindet; *Dann* wird eine Ausgabe produziert". Die so verschriftlichte Spezifikation fördert die Kommunikation zwischen den Akteuren und beugt teuren Missverständnissen vor. Gleichzeitig kann diese Beschreibung zur Steuerung automatisierter Tests verwendet werden um zu zeigen, dass die Spezifikation wie geplant implementiert wurde. Der Ansatz funktioniert in der Praxis gut, leidet aber unter einigen Nachteilen: Er ist informell und kann zu langatmigen Beschreibungen führen, die das Risiko bergen, Teile des Systems nicht vollständig zu spezifizieren und eignet sich schlecht für kombinatorisches Testen.

Modellbasiertes Testen (MBT) ist hingegen ein strukturierter Ansatz zur automatischen Generierung von Testfällen. Durch den Einsatz von Modellen eignet er sich hervorragend zur Beschreibung komplexer Strukturen. Selbst einfache grafische Zustandsdiagramme können intuitiv ausdrücken, was viele Seiten textuelle Beschreibung nicht vermögen. Modelle sind durch ihren Abstraktionsgrad flexibel und ermöglichen das einfache Neugenerieren der Testfälle, wenn sich die Spezifikation verändert.

Diese Diplomarbeit präsentiert einen Ansatz, der BDD und MBT mittels Zustandsdiagrammen kombiniert und so eine automatische Generierung von BDD-Testfällen ermöglicht. Der Grundgedanke von BDD, einer kollaborativen Spezifikation, soll damit nicht infrage gestellt werden. Vielmehr soll die Integration mit MBT mühsame und unvollständige Beschreibungen komplexer Systemteile vermeiden. Mit einem Prototypen wird gezeigt, wie die BDD-Spezifikation aus Modellen generiert werden kann. Eine Fallstudie, die einen kommerziellen Voice over IP (VoIP) Gateway untersucht, zeigt vielversprechende Resultate: Die generierten Testfälle konnten die Funktionalität des Systems gut abdecken und der benötigte Aufwand war vergleichbar oder geringer als die Erstellung ähnlicher manueller Tests. Eine Umfrage unter BDD-Experten ergab, dass die Benutzer zwar sehr wohl automatisch generierte BDD-Testfälle zu erkennen vermochten, in einigen Fällen diese aber gegenüber manuell geschriebenen Tests bevorzugten.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

1.1 Motivation and Problem Definition

With the growing dependency on reliable software, modern software development projects have embraced automated software testing as a necessity that can aid development instead of seeing it as a hurdle [14]. Behaviour-driven development (BDD) is one strategy that has found widespread use in agile software development teams [48]. Instead of focusing on writing tests, it encourages practitioners to come up with a specification of the behaviour of a system in such a way that makes it possible to use it as automated test cases directly. By writing the specification using the business domain language in a "*Given* a precondition; *When* some action is performed; *Then* an outcome is achieved" style it simplifies the communication about the system behaviour within the development team. However, at the same time, these so called BDD stories written in a Given/When/Then (GWT) syntax can be overly verbose, which makes it hard to keep track of which behaviour has already been described and to maintain test specifications in case the behaviour of the system under test (SUT) changes.

When looking at how software engineers have found ways to handle the complexity of their work, model-driven engineering (MDE) comes to mind [43]. Nowadays, software development cannot happen without models, they are omnipresent. Ranging from simple drawings made up by developers to convey behaviour, states or relationships to entire systems like workflow engines running on models. Automata like finite-state machines form the basis of computer science itself, so it is no wonder that models are used in so many stages of the software development process, including testing.

One testing approach building heavily on top of models is model-based testing (MBT). It is a very structured approach to testing and facilitates good test coverage by systematically generating test cases for the system's behaviours based on a model description. Whilst

different MBT approaches exist, the most complete and useful approach is the automatic generation of test cases from behavioural state models [56]. The involved models need to be detailed enough to allow the test generation process to predict the next state of the SUT after each invoked action. While (mental) models themselves are used in software testing all the time, actual MBT itself has never caught huge popularity in practice [37]. Reasons for the slow adoption include the necessity for practitioners to learn new modelling skills and incompatibilities between tools [28, 33]. So far, a widespread adoption of MBT in a large amount of software projects has not happened and it is used only in a few industries that have a specific focus on reliability, even though the benefits have been shown both in theory and in practice [35, 6].

1.2 Aim of the Work

This thesis aims to investigate the combination of MBT and BDD to develop a testing approach that has both of their main advantages: a good test coverage through the automatic generation of test cases and a comprehensible test description language that allows the specification of tests in the business domain language. Specifically, an automatic mapping method to generate BDD stories from models shall be designed and evaluated in a case study using a prototypical implementation of the new mapping method. The prototype shall be able to generate test cases from state machine models and transform them into BDD stories, which can then be executed using existing BDD test automation frameworks. To be able to demonstrate the utility of the developed prototype, a case study shall be conducted using a voice over IP (VoIP) gateway as the test subject for which BDD stories already exist. VoIP has been chosen because it is a representative for a complex technology and because the session initiation protocol (SIP) standard, one of the main VoIP protocols, already defines various models that can serve as the basis for test models.

The expected outcome of this thesis is an approach for translating graphical state machine models describing the behaviour of a software system into BDD stories that can be used to test the implementation of that software system. In particular, a mapping shall be elaborated that specifies how to translate a unified modeling language (UML) state machine into an executable BDD story. In contrast to other methods, the mapping shall be able to make use of existing human-readable BDD steps (reusable test steps) and thus be able to generate human-readable BDD stories. It should preserve the main advantage of MBT enabling practitioners to generate an unlimited amount of test cases with a systematic coverage of the model and the SUT by employing different traversal and coverage mechanisms and combine them with the advantages of human-readable tests, such as maintainability.

An evaluation of the developed testing method shall analyse the results of this thesis to answer the following research questions:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1. Is it feasible to generate meaningful BDD stories that adequately cover the functionality of the SUT?

2. Are generated BDD stories comprehensible enough so that they can be understood by the main stakeholders, in particular developers and testers?

3. How high is the effort required to create a model and enrich existing BDD steps for their usage in generated tests? Is the effort required for specification changes in the SUT lower when using this method than adapting the manually developed BDD stories?

1.3 Methodology and Approach

The methodological approach follows the design science methodology [22], which offers instructions for creating successful information system artefacts. As such it requires the design of an innovative and purposeful artefact which must be evaluated to ensure its practicality and benefit. Finally, the research result should be communicated from a technological and management standpoint. Based on these guidelines, the following overall methodology for the conduction of this thesis is chosen:

Analysis will be performed based on a literature review to form a theoretical background on MBT and BDD. This includes highlighting the advantages and potential shortcomings of both methods and designing an approach to combine them. As the models will be based on UML, a good understanding of this modelling notation is required in order to identify potential extension points that will be necessary to perform an effective test generation.

Prototype development will be based on the outcome of the analysis stage. The prototype will be capable of reading graphical UML state machine models, finding suitable test cases in the model and generating executable, human-readable BDD stories all while reusing already developed test steps that perform the necessary setup, action execution, validation and tear-down of a SUT. The prototype will be based on existing open source software. The necessary tasks will include developing a suitable model description language (meta-model), which can then be translated to BDD stories that can be executed using a BDD framework.

Evaluation will be performed by means of a case study, demonstrating the utility of the developed approach. It will consist of adapting a set of existing models that define the operations of a VoIP gateway to be able to use them to generate BDD stories that test the implementation. Additionally, a survey among BDD practitioners will be conducted to understand whether the generated BDD stories are comprehensible in comparison to manually written ones. Specifically, the approach will be evaluated with regards to (i) the resulting test coverage of the SUT, (ii) the utility and comprehensibility of the generated BDD stories and (iii) the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

effort required to create and maintain the resulting test cases thereby answering the research questions posed above.

1.4 Structure of the Work

The remainder of this thesis is organised into six chapters as follows:

Chapter 2 - Background: This chapter introduces how BDD can help the stakeholders of modern software development projects to collaborate better and design executable system specifications and briefly introduces existing frameworks that aid in the implementation of BDD. Furthermore, the basic principles of MBT are explained, e.g. how models can be used to generate abstract test case specifications.

Chapter 3 - Mapping State Machine Models to Executable BDD Stories: In this chapter, an approach is presented that allows the mapping of graphical UML state machine models into BDD test cases, which make use of existing test steps.

Chapter 4 - Implementation: This chapter describes how the concept from Chapter 3 can be implemented in practice. It summarises the critical parts required to implement a prototype based on existing open source MBT and BDD frameworks.

Chapter 5 - Evaluation: In this chapter, the prototype is evaluated in a case study on the example of a commercial VoIP gateway. It focuses on the research questions underlying this thesis and summarises the results of the conducted survey among BDD practitioners.

Chapter 6 - State of the Art: This chapter summarises related works that has been published. This comprises (i) existing MBT approaches with a focus on those based on state machine models (ii) existing work trying to combine MBT and BDD (iii) existing work trying to combine MBT and keyword-driven testing (KDT) which to a certain extent relates to BDD and (iv) natural language processing.

Chapter 7 - Conclusion and Outlook: The last chapter recaps the contributions of this thesis and discusses possible improvements and future work.

In addition to these chapters, Appendix A contains a running example, which is used throughout this thesis. It comprises a state machine model, the required mappings to reusable BDD steps and the BDD stories generated from the model and the mappings. Additionally, Appendix B contains the raw data from the questionnaire that was conducted as part of the evaluation and is described in details in Chapter 5.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

This chapter introduces the key technologies and design aspects this work is based on. It consists of two larger sections introducing at first in more detail what behaviour-driven development (BDD) means and how good BDD system specifications can improve the communication between the stakeholders involved in a software development project while at the same time serving as a description for automated test cases. The second part explains how model-based testing (MBT) works, how test cases can be selected from the model and what ways exist to transform abstract test cases generated from models into executable test code.

2.1 Behaviour-Driven Development

BDD was first introduced in 2003 by Dan North [53] as an evolution of test-driven development (TDD) [2]. In his article, he describes TDD as too generic and to leave many question, like where to start testing, the scope of what should be tested and how to name tests, unanswered. Initially, he proposes naming test methods using sentences where each sentence starts with the word "should" to encourage developers to narrow down tests to what the class at hand should do. Many of North's ideas are aiming to change how developers and testers think about their work. As such, he argues that by using the word 'should' instead of 'test' it immediately allows people to challenge a failing test by asking 'Should it?'. He claims that it is often difficult for developers to remove tests in fear of reducing their code quality even though the test is no longer valid. Another example is using 'behaviour' instead of 'test', which allowed North to find many answers to the questions developers had about TDD. What should a test be named? A sentence describing the behaviour exposed by the system. What should be tested? As much as is described by that behaviour, which when sticking to a single sentence is not that much [53].

Following his guidelines allows to identify when classes should be split up, and expressive sentences help to understand the meaning of a test in the case it fails for which he describes three typical reasons and necessary actions to take:

- A bug was introduced - Fix the bug
- The described behaviour is still relevant but has moved elsewhere - Move the test to the class it belongs to
- The behaviour is no longer correct - Remove the test

Inspired by the work of Eric Evans on domain-driven design [17], North later proposed a template [53] for writing users stories that focus on the behaviour of a component. Each BDD story should provide a business value which forces reflection on what is important for the customer and what is not. By focusing on what is missing from a system, stakeholders should be able to identify which behaviour should be implemented next, which was another question raised by many employing a TDD workflow [53]. Evans suggests a "ubiquitous language" focusing on the actual domain and leaving out technical details making it possible for all stakeholders, regardless of their technical background (from customers to developers), to understand each other unequivocally [17, 48].

To prove the concept, North began work on the JBehave[1] framework and later refined his ideas of user stories in RSpec[2]. JBehave's initial goal was to replace JUnit and focused mainly on using sentences starting with 'should' instead of the word 'test', which was mandatory in earlier versions of JUnit. Over time and with input from other BDD inspired developers like Aslak Hellesøy and David Chelimksy [49] RSpec received support that allowed writing plain text, natural language stories which could be executed to verify the behaviour of a system against its specifications. This feature was eventually also ported back to JBehave 2.0, which is introduced in more details in Section 2.1.2.

The idea of driving development by executable specifications (also known as acceptance test-driven development (ATDD)) had previously been introduced by Beck [2], but dismissed as impracticable on a unit-test level. BDD over the years became much more than just a variation of TDD and is especially popular in agile development [18, 48].

2.1.1 BDD Stories

As mentioned before, BDD test cases are written in terms of plain text user stories. Each story not only defines the business value of the component (e.g. *As a* user *I want to* execute a specific action *so that* I can reach my goal) but also a set of scenarios that should be automatable and characterising the behaviour of the system. Each scenario consists of a variable number of natural language steps, which should follow the structure of "*Given* a precondition; *When* some action is performed; *Then* an outcome is achieved"

[1] JBehave: https://jbehave.org/
[2] RSpec: https://rspec.info

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

(GWT). This template has been proposed by North [55] and can be seen in Listing 2.1. While this structure is generally supported by all tools implementing the BDD approach, there are some variations. One widespread implementation is the Gherkin language [18] found in the Cucumber[3] framework (which originated from the aforementioned RSpec framework). As the proposed prototype in this thesis will be based on the JBehave framework, the following explanation is based on the JBehave variant of the Gherkin language [25], which closely resembles what was initially described by North [55].

```
Title (one line describing the story)

Narrative:
As a [role]
I want [feature]
So that [benefit]

Scenario: Title summarising the scenario
Given [context]
And   [some more context]
When  [event]
And   [another event]
Then  [outcome]
And   [another outcome]

Scenario: ...
```

Listing 2.1: Syntax for a typical JBehave story resembling Dan North's original design [55]

Steps can be written in a natural language and still consist of fixed and variable parts allowing parametrisation of steps and thus reusability. In case stories are targeting a non-English audience, keywords like Given/When/Then can be localised in the native language for better integration in the text.

Listing 2.2 shows a sample story file that illustrates how a BDD story defines the business value from the perspective of a caller of an emergency number and how the system behaves when a new emergency call is placed under various scenarios. The narrative section usually contains a small user story itself, highlighting what the entire story is about. The example also shows how parameters ("no free" and "a free") can be integrated seamlessly into the natural language steps.

The next section introduces JBehave's implementation of BDD in more detail.

2.1.2 JBehave

JBehave's original goal was to be a replacement for unit test tools such as JUnit. It was designed with the new BDD vocabulary in mind (e.g. using 'behaviour' over 'test'), and

[3] Cucumber: https://cucumber.io

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

test methods started with 'should' [53]. It was mostly rewritten later on to incorporate the new idea of having plain text, natural language user stories, which already existed in other BDD frameworks such as RSpec [54]. Over time JBehave has received support for an extended format for BDD stories mainly due to the need for more extensive test automation support. The general structure is still the same starting with an optional story title and the optional narrative section (indicated by the `Narrative:` keyword) explaining the business value followed by a list of executable scenarios. In-depth information can be found in the JBehave documentation [25], so only a brief overview is given in the following section.

```
Announcement playback

Narrative:
As a caller of the emergency number
I want to receive feedback on my call
So that I know that I am connected and an operator will take my call

Scenario:  The control room is busy
Given there is no free operator available
When a new call is received
Then an announcement is played back

Scenario:  The control room is not busy
Given there is a free operator available
When a new call is received
Then the operator is notified about the new call
And the caller is immediately connected to the operator
```

Listing 2.2: Example of a BDD story in JBehave.

One of the benefits the story structure enforced by BDD provides over other plain text descriptions of system features is the ability to derive executable acceptance tests which verify the behaviour of the system [48]. The process of linking a step to executable code is dependant on the used BDD framework. As an example, JBehave maps steps to Java methods using annotations as illustrated in Listing 2.3, which matches both "Given" steps from the initial story example in Listing 2.2.

```
@Given("there is $status operator available")
public void givenOperatorAvailability(String status) {
   //TODO implement the actual logic
}
```

Listing 2.3: Java step method annotated with configurable step text in JBehave

Several features were added to the story syntax for managing the state of the SUT to address the needs of test automation engineers. In particular, a list of `GivenStories` can be provided for each story or scenario which will be executed as a precondition for the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

actual story. Furthermore, steps allow for parameters which are converted into Java values by a set of converters transforming the plain text input to the appropriate Java object. Special support exists for tabular parameters which can be useful in steps where listing parameters in a sentence is counter-productive. Entire scenarios can be parametrised using example tables where for each row in the table, the scenario is repeated using the provided test data. This feature is especially useful to iterate over multiple cases of the same scenario without having to repeat it many times. Listing 2.4 shows a story that references a `SUTReady.story` file in the `GivenStories` section and contains a scenario with an example table.

```
Advanced feature showcase with GivenStories and table of examples

GivenStories: SUTReady.story

Scenario:  Route call to destination based on phone number extension
Given the default call routing setup
When an incoming call to <extension> is received
Then the call is routed to <destination>

Examples:
| extension | destination   |
| 11        | Main Office   |
| 13        | Second Office |
```

Listing 2.4: Story showcasing advanced JBehave features

Not all of these features are really in the spirit of BDD (a topic which is further addressed in Section 2.1.3) as they can lead to fine-grained stories putting too much focus on the technical implementation. Having fine-grained steps might be required in some cases but can lead to lengthy and over-complicated stories in other cases. Consider as an example a scenario describing the login procedure. On its own, it is probably expected to walk through the login process in detail, but having these details in each scenario that requires a logged-in user is drawing away attention from the actual behaviour. To remedy this, so-called composite steps have been introduced. They group a number of steps together to form a new step and can either be defined as Java code or more quickly as plain text as shown in Listing 2.5.

It should be noted that many of these features are strictly speaking unnecessary from a technical standpoint and other BDD frameworks may have explicitly decided against implementing them. As an example, preconditions required for each scenario could be integrated into the scenarios' actual `Given` step instead of using `GivenStories`. Similarly, composite steps are simply an easier way to group steps together instead of creating a new step whose implementation could delegate to the existing steps. Their primary purpose is to simplify the development of automated test cases.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Composite: Given user $username is logged in
Given no user is logged in
When the user <username> enters the correct username and password
And the user clicks the login button
Then the user <username> is logged in
```

Listing 2.5: Composite step in JBehave

2.1.3 BDD as a Development Process vs. BDT for Test Automation

According to Keogh [27], while BDD originally may have been envisaged to be a "small, simple change from existing practices - replacing the word 'test' with the word 'should' " it has since then evolved into a lot more than just a testing technique. Faragó, Friske, and Sokenou [18] claim that people referring to BDD with the often interchangeably used term behaviour-driven testing (BDT) suffer from the misconception that BDD is mainly a test automation tool. They ground this claim by citing [1]: "There isn't much point in going through examples that illustrate existing cases; that doesn't improve understanding. When illustrating using examples, look for examples that move the discussion forward and improve understanding". This is a direct critique on features such as the aforementioned example tables which are targeted towards combinatorial testing. Instead, they suggest other methods such as model-based testing (MBT) [56] or fuzzing [52] as better suited alternatives [18].

The view of BDD being misunderstood as test automation is also shared by Hellesøy [21], the inventor of the Gherkin and language who says BDD is "the world's most misunderstood collaboration tool". According to him, BDD should be done in two stages. In the first stage, business analysts in charge of defining requirements develop BDD stories together with programmers and testers as they discuss features to be implemented. In the second so-called "outside-in" development stage, programmers repeatedly run those stories which tell them what needs to be implemented next, starting with functionality closest to the user working towards the innards such as business logic (thus the name "outside-in"). However, if this approach is replaced with dedicated testers developing BDD stories after the system has already been implemented, the effort required to do BDD is, according to Hellesøy [21], not justified.

A similar "iterative decomposition process" is suggested by Solís and Wang [48] in their study of characteristics of BDD confirming that BDD should be done continuously from the planning phase to analysis into the implementation.

2.1.4 Best Practices for BDD Stories

Regardless of the workflow that led to a BDD story, several criteria should be considered when judging the quality of a story [18, 47, 48].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Language. A domain-based (common) language should be adopted. This applies to the narrative, naming of a story and scenario titles as well as individual steps.

Narrative. The description of the business value should be found in the narrative section. It is good practice to follow a template (such as the one provided in Listing 2.1), ensuring it is clear who the user is, what the feature is and how this feature benefits the user.

Scenario. Similar to unit test methods, scenarios should be self-contained and independent from each other. The goal is that scenarios specify the detailed behaviour as part of a larger feature description. A certain amount of abstraction is required to keep scenarios small and concise.

Step. The step starting word should indicate the type of step, e.g. a 'Given' step should put the system in a known state. Each scenario should only have a single 'When' step indicating the user action. Finally, 'Then' steps should describe the expected outcome.

2.2 Model-Based Testing

As models are omnipresent in software development, it is no surprise that models have been used in testing extensively. Model-based testing (MBT) is all about how models can not only aid in developing test cases but can be deeply integrated into various parts of the testing lifecycle itself in an automated fashion:

> "MBT encompasses the processes and techniques for the automatic derivation of abstract test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases." (Utting, Pretschner, and Legeard [57])

In contrast to BDD, testing approaches that promote the usage of models like a finite-state machine (FSM) have been around since the 1970s [12]. Utting and Legeard [56] have identified four main approaches to the general term of MBT. Broadly speaking the idea is to generate elements of a test, or the entire test using models:

Test input based on domain model. In this case, the model holds the knowledge about the valid inputs for a particular domain and can be used to derive input data. A test case based solely on 'input models' could implement only basic verification such as verifying whether the system crashes or produces an exception as it does not know how the system should react to any given input [57]. These types of models are also commonly used in other testing techniques, such as fuzzing [50].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Test case generation from environmental models. This approach makes use of environmental models to generate test cases. An example would be a usage model denoted as a Markov chain. In this model notation nodes represent usage states and are connected with arcs that represent transitions or input stimuli. Each transition is also assigned a probability indicating the likeliness of this transition to occur. Such a statistical approach can give insight into which parts of the SUT should be tested first, e.g., based on the probability of a particular state. Such models are also easier to define than standard behavioural models as they do not require modelling internal states [36, 51]. However, this means that statistical models can only generate input data as they only know how a user interacts with the system. As with the first approach, additional test oracles are needed to give a proper test verdict apart from confirming that the SUT did not crash [56].

Test case generation based on behavioural models. In this case, an executable test case is generated using models describing the expected behaviour of the SUT. The model must be able to predict the outcome of each interaction in order to verify the behaviour. According to Utting and Legeard [56], this is the only approach that covers the entire test process from input values, to executable test cases, including proper pass/fail verdict information.

Test script generation from abstract tests. For this approach, abstract descriptions of a test using e.g. a UML sequence diagrams are transformed into lower-level executable test scripts.

Most interesting with regards to this thesis are MBT approaches that help generate test cases from behavioural models. According to a 2014 survey with 100 MBT practitioners this is also the most often used type [6]. In such approaches, executable test cases are generated using models describing the behaviour of the SUT. The fact that the produced tests not only drive the SUT but also provide a verdict makes them more challenging to implement than, e.g., using a model to deduce input data only.

2.2.1 Model-Based Testing Process

The main advantage of MBT is that it allows for a high degree of automation in the test process [56]. Instead of having to write automated test cases by hand, an abstract model of the SUT is used in conjunction with a suitable tool to generate the test cases from the model. Depending on the test selection strategy, different test cases can be generated like in-depth tests for certain features of the SUT or quick smoke tests to cover basic functionality. MBT approaches can potentially even provide an automatic requirements traceability matrix and other coverage reports [56].

The MBT process is illustrated in Figure 2.1 and consists of three specific steps [56]:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1. Create a model of the SUT based on its requirements and the test plan. At this stage, requirements can be linked to the model, which allows the generation of a traceability matrix. For this, an abstraction level has to be chosen which consists of deliberately omitting some details while at the same time encapsulating other details in other components, such as the test automation adapter.

2. Generate a set of abstract test cases using a tool that parses the model and applies a suitable test selection criteria. Test selection criteria are explained in more detail in Section 2.2.4. The resulting abstract test cases are only a high-level description of the concrete test cases. For example, using an FSM as the model with a random walk test selection criteria, an abstract test case would be a single path through the FSM [57].

3. Transform the abstract test cases into executable tests scripts, which is discussed in more detail in Section 2.2.5.

Steps 4 and 5 from Figure 2.1 are not specific to MBT and consist of executing the generated test scripts and analysing the results.

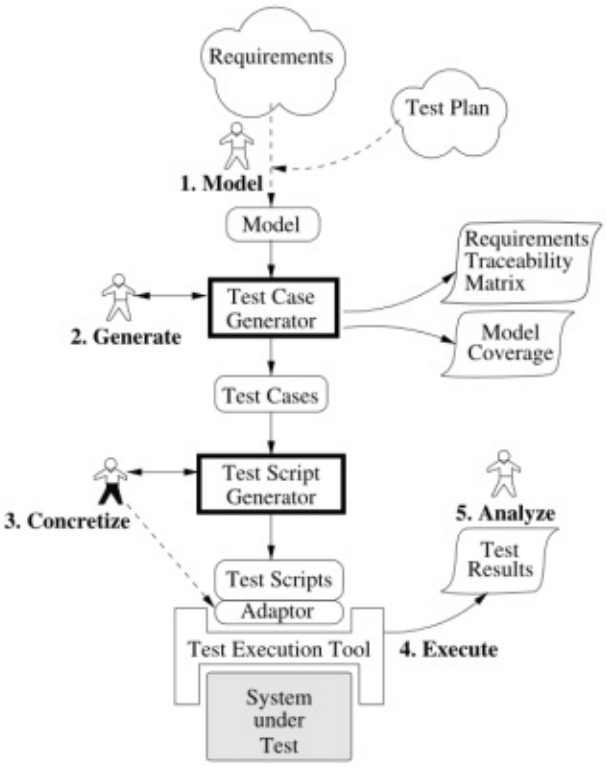

Figure 2.1: Overview of the model-based testing process [56]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Some processes combine steps 2 through 4 in what is called online model-based testing, whereas the individual stages are required for offline MBT. Offline model-based testing produces a set of test cases or just test stubs that need to be implemented to interface with the SUT but can then be executed as usual at any point. In contrast, the online MBT technique generates test cases ad hoc during the test execution. Instead of generating a set of test cases and executing it at a later point, each test step is immediately executed and interfaces with the SUT. This requires more sophisticated MBT tools and models that allow linking abstract test steps to some test interface driving the SUT. Online approaches are required whenever the SUT is highly dynamic and can make non-deterministic/autonomous decisions. Depending on the outcome of a test step, the test generation may choose a different path according to the model, which allows testing highly flexible systems where it is impossible to predict the next state in advance. In contrast, offline approaches make it easy to use existing tools for running tests and allow for creating inspectable test cases which can be reviewed, manually adapted and repeatedly run, e.g. for regression tests [56].

2.2.2 Model Development

When implementing a test plan based on models, a decision has to be made how models are developed and what they are based on. Usually, it is not possible to reuse development models without making changes as they are not designed on the right abstraction level, even though from an economic standpoint it might sound very desirable [56]. Development models often focus on structure (like class diagrams) and not on behaviour and when they do focus on behaviour, they often do not portray the right amount of details. In cases where suitable models are available (sometimes referred to as *test-ready* models [5], which is often the case if code is generated from models during the development), it might nevertheless be a bad idea conceptually, as a test will likely exhibit the same faults as the developed software [56].

Utting and Legeard [56] argue that the opposite case, where models are developed solely for the purpose of testing without reuse, is actually common, which is also confirmed by Neto et al. [33]. This way, the independence between the test model and the development model allows finding as many differences between the implementation and the expected behaviour as possible. However, it is still a good strategy to develop test models which take certain aspects from development models by formalising them for consumption by test case generation tools [56].

2.2.3 Finite-State Machines and UML

While there exist many different formal methods fitting the requirements for MBT, such as model-based languages (including Z, VDM or B), process algebra languages (like CSP, CCS or LOTOS) or algebraic languages (like OBJ or the Common Algebraic Specification Language - CASL), the focus of this thesis will be on FSM languages and more specifically on graphical representations which are the most commonly used format for designing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

test cases according to Binder, Legeard, and Kramer [6]. An in-depth overview of the other mentioned modelling languages can be found in Hierons et al. [23].

FSMs are essential to software engineering and provide a useful formalism for describing software behaviour [3]. UML provides a dedicated diagram, the state machine diagram, for modelling the behaviour of a system depending on its states. Other graphical notions exist, such as statecharts (Harel [19]) which UML took inspiration from. In the following, a brief overview of what finite state machines are is provided and their basic syntax is introduced in UML notation.

State machines consist of two main building blocks:

State. Represents the current set of variable assignments which was reached through past inputs in the system and dictates what behaviour the system will expose with future inputs [5]. In UML states are depicted as nodes using rounded rectangles containing a label describing the state. Besides, states can also indicate internal activities that are executed e.g. while the state is active, on entering the state or when leaving the state [44].

Transition. Connects two states together to form a sequence indicating that in case of a certain event being triggered in the *source state*, a determined *target state* is reached. The event or trigger is any input to the system or a time period being passed (e.g. a timeout). The event is used as a label for the transition. In a UML state machine diagram, the transition is represented as a directed edge (arrow) and additional attributes can be defined for the transition such as a guard (written in square brackets) that can be checked before allowing the transition to happen. UML also offers the possibility to formalise guards using the object constraint language (OCL). Outputs of a transition (also called activities or effects) are written after a slash [5, 44].

Figure 2.2 showcases a sample UML state machine diagram that roughly correlates to the BDD story shown in Listing 2.2. The initial pseudostate state is depicted as a black filled circle and represents the start of the state machine. Vertices represent states that, in case of this model being used in the context of MBT, should be verified after events (represented by edges) occur. The `[operator available]` and `[not operator available]` statements represent guards that guide through the model execution. UML additionally defines three internal activities in the lower compartment of the state rectangle: (i) the *entry* activity must be executed as soon as the system enters a state, (ii) the *do* activity is executed as long as the system remains in the state and finally (iii) the *exit* activity is executed once the system leaves a state.

The model shown in Figure 2.2 starts with the system in the state “idle”. Once a new call emerges, the “announcement playing” state is entered if the guard “no operator

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

available" evaluates to true, or else the state "connected to operator" is entered. While the "announcement playing" state is active, the "playback annoucements" activity is also active, whereas for the "connected to operator" state, the "connect to operator" activity is only activated when the state is entered. Finally, when the call ends the "call ends" transition connects both states back to the "idle" state.

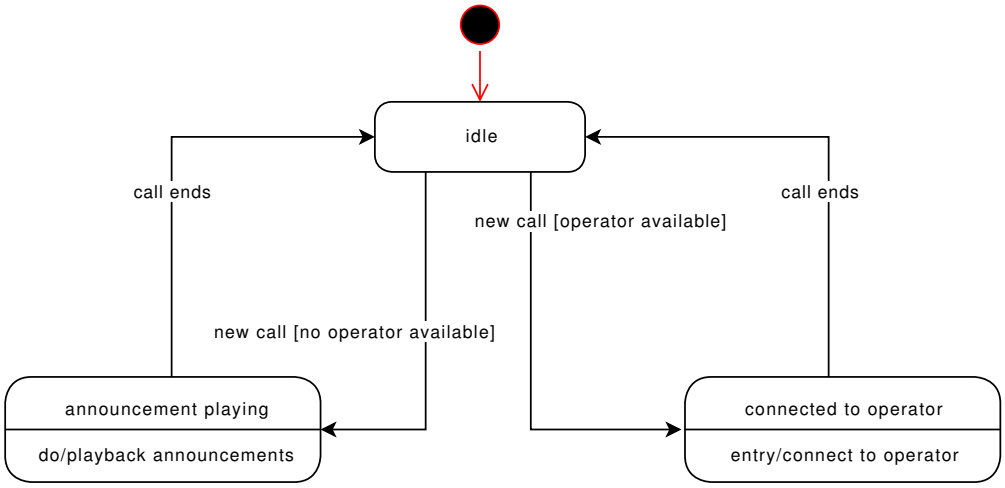

Figure 2.2: Example UML state machine diagram describing similar behaviour as the BDD story in Listing 2.2

2.2.4 Test Selection Criteria

Test selection is an important aspect of any MBT approach. It is performed by the MBT tool based on parameters provided by the testers and can have a significant impact on the fault detection rate the generated tests can achieve. A test selection criterion consists of two parameters, a stop condition indicating how many test cases should be generated and an algorithm that should be used to select test cases from the input model [56].

When considering test case selection in the context of MBT two different types of test coverage have to be distinguished: code coverage and model coverage. Typically developers and testers are more familiar with code coverage, which is often used as a measurement of the test quality and overall code quality. In its most basic form it represents the number of lines of code executed while running a test. It is a sub-optimal metric for judging the quality of tests because, trivially, just running the software without checking any output can produce a large code coverage but only certifies that the software has not crashed, not that it behaved as expected. Model coverage is specific to MBT and defines how well the generated test suite covers the test model [56].

Utting and Legeard [56] differentiate between six different test selection criteria families that include statistical criteria, requirements-based criteria, criteria deriving directly from the structure of the model, data coverage criteria, fault-model criteria that try to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

prove the absence of predefined faults and explicit test case specifications written by test engineers to investigate specific areas of a model. Most of these test selection criteria require additional data to be provided in the model, which is not the focus of this thesis. Structural model coverage criteria, as the name implies, mostly derive the test selection from the structure of the model itself and can further be divided into subcategories:

Control flow criteria: These criteria are mostly based on existing code coverage criteria such as statement coverage or decision coverage. They are especially useful when e.g. OCL expressions are used or the model is written in some other coding language (see Section 2.2.3 for examples).

Data flow criteria: These criteria require extra information on the data that is being used in a model and make it necessary to track the data flow throughout the model paths.

Transition criteria: These criteria are especially interesting in the context of this thesis, as they are designed specifically with state machines in mind. Trivial examples include all-states coverage that require each state to be visited at least once and all-transition coverage that requires each transition to be used at least once. For state machines that include parallelism there exists e.g. an all-configuration coverage where a snapshot of the states that can be active in parallel is counted as a configuration. More complex coverage criteria, such as all-transition-pairs or all-loop-free-paths, can be used to detect software faults that occur if states are traversed in a specific order. Obviously, these coverage criteria can quickly lead to an explosion in the number test cases, and for non-trivial state models that include complex guards and actions, it might be impossible for an MBT tool to find all paths efficiently. Search algorithms for graphs are often used to reduce the number of generated test cases (in comparison to a random walk). Utting and Legeard [56] provide an excellent entry point for more information including case studies that compare random walks against sophisticated algorithms, which are, however, out-of-scope for this thesis.

UML-based criteria: These criteria are specific to UML and make use of other UML diagrams such as class, object and sequence diagrams. Examples include coverage criteria that try to generate test cases that produce all specified multiplicities in a class diagram or testing generalisation by running tests on every subclass of a specific superclass.

2.2.5 Transforming Abstract Tests to Executables

While an MBT approach can be used to simply define abstract test cases which are then executed manually, it is generally more desirable to be able to automatically transform these into executable tests. Unfortunately, the test model is often not detailed enough to generate test cases that can directly interface with the SUT. An executable test case

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

not only requires a sequence of test steps driving the SUT, it also must account for the initialisation of the SUT, the data that is used by the test steps and the return values received by the SUT test application programming interface (API) [56].

According to Utting and Legeard [56], there are two main approaches for bridging the gap between abstract test cases and the interfaces offered by the SUT (see also Figure 2.3):

Adaptation. This approach requires a manually developed adapter that is able to read the abstract test case and translate it into sequences of low-level calls to the SUT. It must take care of the setup of the SUT, translate test steps originating from the model into calls to the SUT's API, translate the output of these API calls back into the abstract language used by the test case definition and finally perform an orderly shutdown of the SUT.

Transformation. For this approach, abstract test cases are transformed into test scripts, which interact with the SUT and take care of the setup, teardown as well as the transformation of input and output data. The test scripts can be based on manually developed templates that can be populated with data and assembled together to form the concrete test case during the transformation approach. In contrast to the adaptation approach, the test scripts are actual code files that can be investigated, manipulated and stored in a version control system (VCS) and can run independently of the MBT tool.

Typically, MBT approaches use some combination of the two approaches. Adaption is more prominent in online testing where a tight integration between the SUT and the MBT tool is required. The transformation approach has the advantage of generating test scripts that fit into the existing test environment. It can reuse existing test code and data and the resulting test reports can be the same as with manually written test scripts. In this scheme, MBT is only replacing the design of test cases and the way test scripts are created, but the rest of the process can stay the same [56].

2.2.6 Benefits and Drawbacks of Model-Based Testing

Utting and Legeard [56] report that MBT has been shown to often find at least as many faults as manually developed test cases with certain case studies reporting more than ten times the amount of faults being found. However, they also note that it very much depends on the skills and experience of the practitioners. This includes knowledge about the modelling language, how test coverage criteria can be used as well as selecting a suitable tool, which makes the usage of MBT difficult and might explain the reason why many studies around MBT have not been transferred from an academic to an industrial context [33]. Another reason for the slow adoption in the industry could be the overwhelming amount of different ideas and proposals as well as academic tools that did not yet lead to the development of a commonly accepted MBT approach [57].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

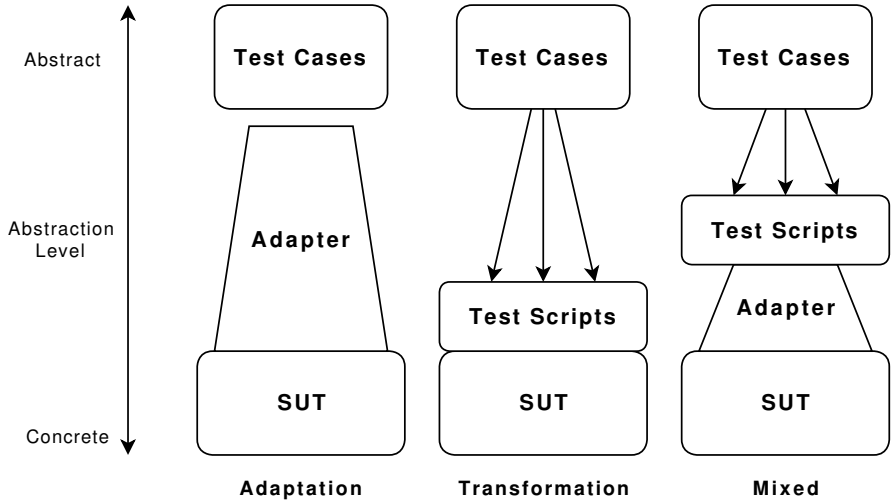

Figure 2.3: Approaches for transforming abstract tests into concrete executable tests [56]

A benefit of MBT is the necessity to transform informal system specifications to formal models, which can help finding issues in these informal descriptions. The need to design a precise model can lead to questions that cannot be answered by the usually natural language based system specifications and thus expose requirements issues [56].

Some MBT approaches also allow to trace test cases back to their requirements. The basic approach is to record the relation between the model and the generated test case, but it can be extended to also trace the informal system requirements specifications to the model and thus relate a concrete test case to a specific part of the informal requirements description. This link makes it possible to see which requirements have not yet been incorporated in the model and lets stakeholders understand why a certain transition, state or behaviour exists in the model or even in the concrete test case. In case requirements change, it is easy to adapt the model and regenerate and execute only the test cases that are affected by the changes [56].

MBT is also not suited for all areas of testing. It is most often applied for system and integration level tests [6] and usually cannot test non-functional requirements, such as usability, security and reliability [33]. MBT test cases can also be more difficult to analyse in case they fail. Generated test cases are often more complex and unstructured, making it more difficult to even understand if a test case implements a valid behaviour and thus has detected a fault in the SUT, or if it is simply wrong, e.g. because it was based on an outdated specification [56].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Mapping State Machine Models to Executable BDD Stories

Model-based testing has advantages over manually defining test cases [56], and at the same time, behaviour-driven development offers promising solutions for notating test cases using its story format [48]. The combination of the two strategies should allow for a systematic way to generate test cases in a format that is already known by many practitioners. This chapter illustrates how graphical UML state machine models and a mapping relation to BDD steps can be used to generate fully executable BDD stories. The generated stories should be readable by developers, tester as well as other involved stakeholders and consist of individual scenarios that highlight various state combinations of the SUT. By using a model-based approach, it should be possible to generate these stories quickly and to efficiently cover possible (modelled) paths through the system.

This chapter describes the conceptual framework required to implement such an approach. The ideas presented in this chapter have been implemented in a prototype which is described in more detail in Chapter 4.

As the target audience for the approach described in this thesis is testers already familiar with BDD and writing stories, the focus will be put on an easy to use system that does not require a deep understanding of advanced UML concepts such as e.g. OCL. Instead, a reduced state machine formalism will be used that provides enough flexibility for generating model-based tests that are readable and writable by anybody with a basic understanding of the domain.

3.1 Overview

The process of generating BDD stories from a graphical state machine model consists of the following components, which are also depicted in Figure 3.1:

1. **Input Model:** The input for the process is a state machine model describing the behaviour of the SUT as an FSM. It has to be in a suitable format for processing by an MBT tool.
2. **Path generation:** Based on the model, an MBT tool generates an execution path that traverses the model. Depending on the generation strategy and model context, the path can be completely random or try to achieve a defined coverage goal.
3. **Scenario generation:** For each generated path, the individual elements that make up that path are mapped to a BDD step using a mapping description. The mapping description can contain optional meta-data used for the scenario generation such as a fragment that eventually forms the scenario title.
4. **Execution:** Once a scenario has been generated, it can be directly executed by a BDD test framework (online MBT) or be recorded in a story file for later execution (offline MBT). If the scenarios are executed online, it is possible to provide feedback to the MBT framework for the further path generation
5. **Reporting:** Lastly, the execution results in either successful or failed step executions which make up the final test execution report listing each of the executed BDD steps grouped into scenarios and their execution result

These process steps are in the following explained in detail.

3.2 Input Model

As stated above, a reduced version of the UML state machine formalism is used in the developed mapping approach. This is to ensure that the formalism is well suited to be used by anybody with experience in BDD and some experience in reading state machine models. Figure 3.2 depicts an example state machine that models a basic call flow. The behaviour defined in the model can be summarised as follows:

- Start the SUT.
- Verify the SUT is started.
- Send a SIP `INVITE` message either via TCP or UDP. Store the used transport method in a context variable called “transport”.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

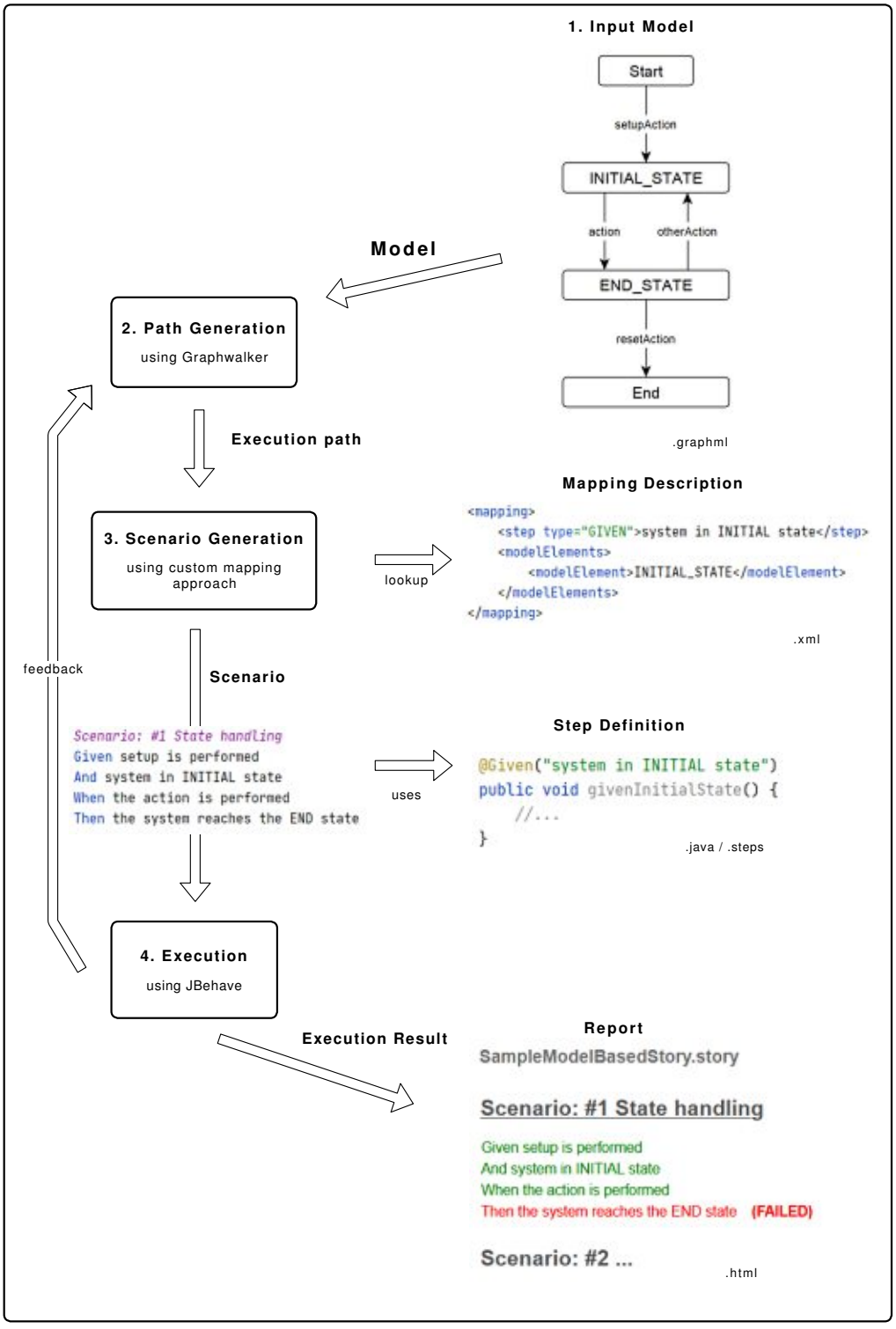

Figure 3.1: Overview of state machine to BDD story mapping process

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- Verify that there is a new call in the state "TRYING" or "RINGING".
- Wait until the SUT receives an `OK` response. Note that the response needs to be generated by the test adapter.
- Verify the SUT now has an established call.
- Perform any of the following transitions:
  - Send a SIP `MESSAGE` with a large payload. This transition is only possible if the model context variable "transport" is set to the value "TCP" as UDP does not support large payloads.
  - Send a SIP `BYE` message.
  - Wait for a SIP `BYE` message. Note that the message needs to be generated by the test adapter.
  - Hold the call. This is done by sending another SIP `INVITE` message with a specific payload.
- Eventually, if in the state "TERMINATED" or "ON_HOLD", the "reset" edge can be selected by the MBT tool in which case some cleanup tasks are performed. Reaching the "End" state marks the completion of one test case.

Figure 3.2 also highlights the state machine concepts that are used by the developed mapping approach. The following concepts are used:

1. **Mandatory start vertex:** Each model starts with a single mandatory "Start" vertex that indicates where the MBT framework should start laying out paths through the model. The start vertex is not linked to a BDD step.
2. **Setup edge:** There must be exactly one edge leaving the "Start" vertex. This setup edge is only executed once for each model. It is responsible for bringing the SUT as well as the test adapter into a known configuration.
3. **Nodes representing states:** Nodes (or vertices) represent some form of state or configuration for verification. Each node needs to be labelled with a model element identifier which is used to map it to a BDD step in the scenario generation.
4. **Edges representing transitions:** Edges represented by arrows represent a transition or stimulus for the SUT. Similarly to nodes, they are labelled with a model element identifier serving as a link to a BDD step. Guards can be placed after the event label in square brackets and must contain an expression that evaluates to true or false. The expression is evaluated against the model context. If the expression evaluates to false, the edge can not be followed as the model context currently prevents it. After the (optional) guard a slash can indicate additional

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

actions. These actions can modify the model context, e.g. by setting variables which can later be used in guards or even provide input data that can be accessed during the test execution.

5. **Reset edge:** The purpose of the reset edge is to terminate the test case and dispose of any open resources. Note that the reset edge is also mapped to a step and can have an arbitrary model element identifier linking to the BDD step.

6. **End nodes:** The model must contain one or more "End" nodes which indicates the completion of a test case. These nodes are not mapped to a BDD step and must always be labelled "End".

3.3 Path Generation

Based on the input model, a test selection strategy and a stop condition (outlined in Section 2.2.4), paths through the model can be generated by an MBT tool. During the path generation, the model is also associated with a context consisting of variables defined in the model, which is a common concept found in extended FSMs [56]. The model context can be modified using actions and checks can be implemented using guards. The model element identifiers (i.e., labels of states and transitions) encountered when traversing the model define the resulting path. Each path represents a single BDD scenario and multiple scenarios form a BDD story. The model element identifiers making up the path can can be mapped into a sequence of BDD steps using a mapping description as done in the next step (see Section 3.4). An example path based on the model shown in Figure 3.2 is shown in Listing 3.1.

```
Start => startSUT => SUT_STARTED => sendInviteUDP => TRYING_RINGING
 => awaitOK => ESTABLISHED => holdCall => ON_HOLD => unholdCall
 => ESTABLISHED => awaitBye => TERMINATED => reset => End
```

Listing 3.1: Example path through the model given in Figure 3.2

One issue that needs to be considered when generating a path that should be mapped to a BDD scenario is where the path should start and where it should stop. While FSMs do have a defined start and end state, the model needs to be designed in such a way that the resulting paths are of reasonable length, as otherwise the resulting BDD scenarios become too long and thus less readable. Careful model design and additional abstractions where required can mitigate this issue.

Standard UML state machine models terminate when reaching the end state. If the scope of the state machine is rather small, e.g., limited to a single class of a UML class diagram, terminating the SUT after each completed state machine run and thus after each test case execution might be reasonable. However, BDD tests are often used for acceptance testing from a user-perspective [18] and thus operate on larger, integrated

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

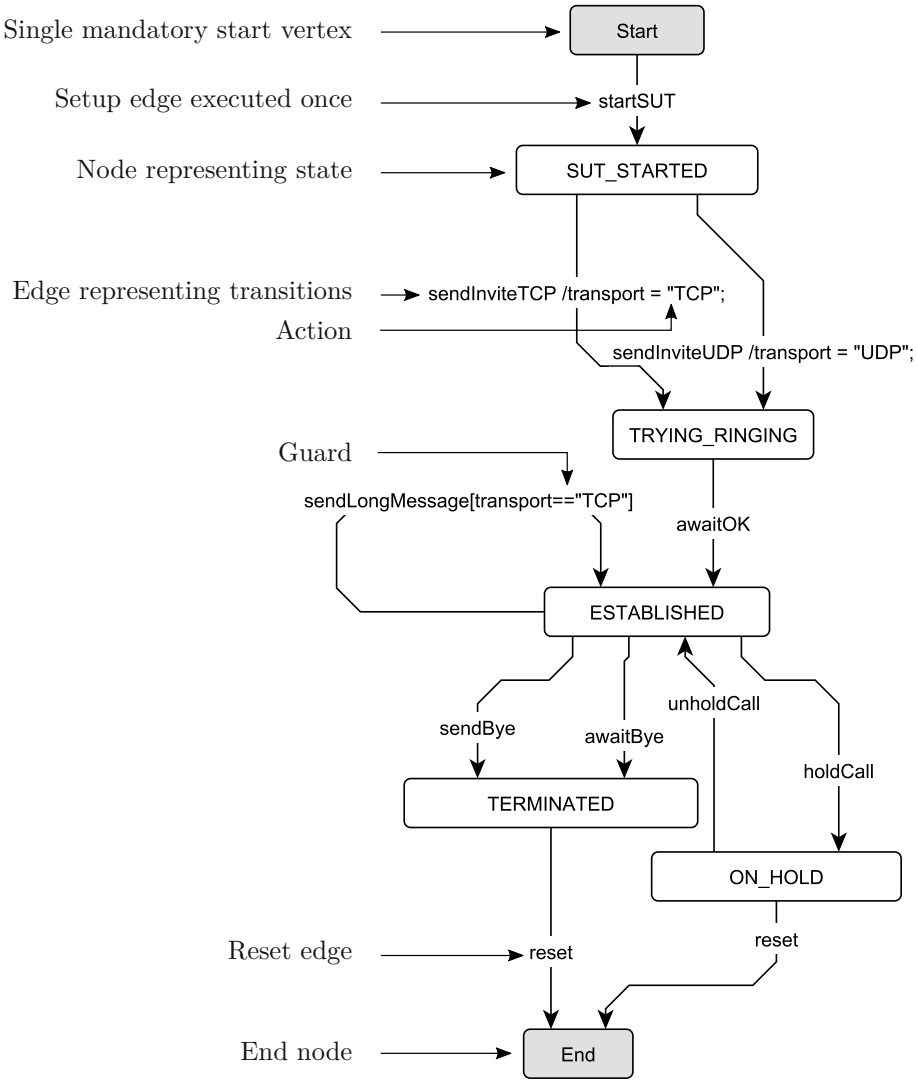

Figure 3.2: Example input model annotated with modelling concept descriptions

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

systems. These systems often cannot be restarted for each test case as this would neither represent their typical usage pattern, nor be feasible in terms of time efficiency. A simple solution would be to avoid reaching the end state too often and instead design the model in such a way that it naturally leads to longer paths through the system, and thus fewer restarts. To not require the explicit modelling of such reset edges, the mapping approach proposed in this thesis implicitly reroutes the previously introduced reset edges to the beginning of the model instead of the end state, specifically to the first state after the start node, which is the state `SUT_STARTED` in the example model. This way, the single outgoing edge from the start node can be used to start and setup the SUT, while the reset edge takes care of restoring the state of the SUT as it is expected by the model.

3.4 Scenario Generation

BDD scenarios consist of a title summarising the scenario and a series of GWT steps. A mapping is required to map the generated model path to a BDD scenario consisting of steps. Any number of suitable formats can be used for the mapping description: A minimal mapping for generating scenarios must contain the model element identifier and the BDD step it maps to including any parameter values that are required for this step. Listing 3.2 illustrates a potential XML-based data structure for defining a mapping from the `sendInviteTCP` and `sendInviteUDP` edges to the BDD step "When the SUT sends an INVITE using transport $transport". Parameters such as the used `transport` method, are a common way of making BDD steps reusable and can be assigned a value using a `data` element in the mapping, such as the values "TCP" and "UDP" in the example.

As discussed in Section 2.1.4, ideally, each BDD scenario should be runnable in isolation

```
<mapping>
    <step type="WHEN">the SUT sends an INVITE using transport $transport</step>
    <modelElements>
        <modelElement>
            <id>sendInviteTCP</id>
            <data key="transport">TCP</data>
            <scenarioInfo>Outgoing TCP call</scenarioInfo>
        </modelElement>
        <modelElement>
            <id>sendInviteUDP</id>
            <data key="transport">UDP</data>
            <scenarioInfo>Outgoing UDP call</scenarioInfo>
        </modelElement>
    </modelElements>
</mapping>
```

Listing 3.2: Example XML-based mapping of model elements to BDD steps

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

from other scenarios and only consist of a few steps so that a story reader is still aware of the preconditions given at the beginning of the scenario while reading the end. The structure of the scenario should consist of steps establishing a given precondition (`Given` statements), define one action that takes place in the SUT (`When` statement) and finally performing assertions on the outcome (`Then` statements). Whilst this scenario setup is desirable, real-world examples (as shown in the Evaluation Chapter 5) often do not follow it, primarily when more extensive integration tests are implemented that consist of many preparation and action steps.

Some limitations need to be accepted for the model-based generation of BDD scenarios: In the context of FSMs, each state is the precondition of its outgoing transitions and postcondition of its incoming transitions. Consequently, there is no structure that groups a set of precondition and postcondition states as anticipated by BDD scenarios. With these limitation in mind, the desired characteristics of a generated scenario can be defined as follows:

- Each scenario should have a title summarising the content of the scenario.
- Scenarios should be runnable in isolation and not depend on each other.
- The structure of a scenario should start with defining preconditions, followed by actions and assertions.

While the approach developed in this thesis has limitations in the fulfillment of these requirements, it tries to mitigate them using the strategies described in the following.

3.4.1 Scenario Title

There are multiple ways to generate a scenario title, starting with the easiest of just enumerating the model element identifiers defining the path through the model exercised by the scenario. Based on the model shown in Figure 3.2, a potential scenario title would look like: “Start - SUT_STARTED - sendInviteTCP - TRYING_RINGING - awaitOK - ESTABLISHED - sendBye - TERMINATED - reset - End”.

The scenario title satisfies the basic requirement of summarising the content of the scenario, but it lacks readability and contains parts which are likely superfluous and repetitive. As an alternative, the mapping description can be enriched with optional scenario information. This allows the creation of fluent scenario titles such as “Outgoing TCP call and send BYE” (which is based on the path given above) or “Outgoing UDP call is held then released from hold and receives BYE” (see path in Listing 3.1). The two examples are using the scenario information outlined in Table 3.1. This additional scenario information is stored together with the step mappings as shown in Listing 3.2 using the `<scenarioInfo>` tag.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Model element identifier	Scenario information
startSUT	
SUT_STARTED	
sendInviteTcp	Outgoing TCP call
sendInviteUdp	Outgoing UDP call
TRYING_RINGING	
awaitOK	
ESTABLISHED	
sendLongMessage	- long message -
sendBye	and send BYE
awaitBye	and receives BYE
TERMINATED	
unholdCall	then released from hold
holdCall	is held
ON_HOLD	
reset	

Table 3.1: Example of scenario information mapped to model elements

3.4.2 Executable in Isolation

Scenarios should ideally form individual test cases which can be run in isolation. Failures in previous scenarios should not affect subsequent scenarios. Thus, each scenario has to specify its preconditions in a form that allows the test adapter to reset the SUT to a known state. Traditionally, UML state machines are terminated whenever the end state is reached, which aligns with the requirement of being able to execute BDD scenarios in isolation. However, as discussed in Section 3.3, BDD often addresses system level tests and it is less than ideal to restart a large SUT after each test scenario. The solution of implicitly linking each end state with the beginning of the model (in the example Figure 3.2 labelled “SUT_STARTED”) and using this state as anchor vertex for all scenarios helps to address this issue. In practice, this means that (i) the first scenario starts with the setup edge followed by the start anchor vertex and (ii) all further scenarios start with the reset edge followed again by the start anchor vertex. By mapping the setup edge, the anchor vertex and the reset edge to suitable “Given” steps, each scenario starts with a description of the preconditions and how to achieve them, which can be seen in the sample story shown in Listing 3.3. In the end, to fulfill the requirement that scenarios are run in isolation the reset step needs to ensure that the SUT is reset in such a way that there is no interference between scenario executions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.4.3 Scenario Structure

The structure of a BDD scenario should ideally have "Given" steps before a single "When" step followed by "Then" steps, and steps within the same category should be linked using the "And" keyword. Real-world examples illustrated in Chapter 5 show that even manually written stories do not always follow this recommendation. Nevertheless, the mapping process can use certain hints to select the best step type depending on the context and provided that the preferred step type is available (as not all steps are available with all keywords). The following rules are implemented:

- Use "Given" steps for the setup and reset edge as well as for the start anchor vertex which describes the preconditions for each scenario.
- Use "When" steps for edges as they represent transitions/actions in an FSM.
- Use "Then" steps for vertices as they represent states of the system.

These rules lead to alternations between "When" and "Then" steps in the main part of the scenario, as can be seen in the example shown in Listing 3.3. Still, in case the preferred step type is not available for a given mapping, an alternative step type can be used.

```
Scenario:  #1 Outgoing UDP call and sends BYE
Given SUT is started and the test adapter is ready
And the SIP application is running
When the SUT sends an INVITE using transport UDP
Then CALLEE CallState is TRYING or RINGING within 500 ms
When CALLEE answers incoming calls
Then CALLEE CallState is ESTABLISHED within 500 ms
When the SUT ends the call
Then CALLEE CallState is DISCONNECTED within 500 ms

Scenario:  #2 Outgoing TCP call and receives BYE
Given calls are ended and message stores are cleared
And the SIP application is running
When the SUT sends an INVITE using transport TCP
...
```

Listing 3.3: Example of a generated BDD story

Note that the full example story and its mapping description can be found in Appendix A.

3.5 Execution

The scenario execution is performed by the BDD framework and supports two modes of operations:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Offline. In the offline mode, scenarios are generated as outlined in the previous sections and then combined in a final executable BDD story which is persisted as a BDD story file. The story execution can be performed as usual which has the advantage that the BDD framework is oblivious to how the test story has been created and thus does not have to be modified for the approach to work. The generated stories can also be modified by testers before the execution, e.g. by adding a "Narrative" section. Also, the resulting stories can be added to a VCS.

Online. In the online mode, each generated scenario is immediately executed by the BDD framework. For this to work, the BDD framework needs to be aware that scenarios are generated on-the-fly, which likely means modifying the tool implementation to accommodate this unusual flow. Benefits of this method include the option to continue generating scenarios that run for a pre-defined amount of time (e.g. a nightly run of the model) and the option to tightly integrate the model context with the state of the SUT and test adapter. For example, it is possible to use the runtime state of the SUT or test adapter in model guards instead of only pre-defined variables populated in the action part of a transition.

Regardless of the operation mode, the execution drives the test adapter, which in turn interacts with the SUT and eventually generates a report.

3.6 Reporting

The reporting of the test results is the last part of the process and unaware of how a story has been created. If desired, the report could be extended to include additional information like:

- the model that provided the behaviour for the story execution
- model coverage information in either a numerical form (like edge coverage and vertex coverage) or visually in the form of an execution heatmap that colours areas of the model according to their execution frequency.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Implementation

This chapter describes the implementation of a prototype for the model-based BDD story generation approach introduced in Chapter 3.

4.1 Requirements for the Prototype

Given that the previously elaborated mapping is targeted towards testers already familiar with BDD and story writing, it is important that the implementation is easy to use for the target audience. This means that no programming skills should be required and no extensive prior knowledge of UML state machine models should be necessary. The usage of the JBehave framework was also a requirement, mainly because the preexisting sample BDD stories used throughout the case study conducted as part of the evaluation (see Chapter 5) were based on JBehave as well.

4.2 Processing the Input Model

For reading and processing the model, the Graphwalker framework was selected. This open source framework supports a simple graphical model notation and provides a Java API to pre-process the model and generate paths through the model using various test generators and stop conditions. It is also actively maintained by its community. In contrast, many other open source MBT tools have either been abandoned (such as ModelJUnit[1]), are largely academic (like MoMuT[2]) or do not support modelling using a graphical notation (e.g. OSMO[3]).

[1] ModelJUnit: https://sourceforge.net/projects/modeljunit/
[2] MoMuT: https://momut.org/
[3] OSMO: https://github.com/mukatee/osmo

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Graphwalker supports various input formats but until very recently (with the introduction of version 4.x), the recommended format was *graphml*, which can be viewed and edited using the yEd Graph Editor[4]. The supported notation is very similar to what has been described in Section 3.2 but lacks the support for reset edges and end nodes. Support for this can be implemented by transforming the model before the execution. During the transformation, the model also needs to be annotated with meta-information, so that it is later possible to identify certain edges and nodes, such as the setup edge, the start anchor vertex and the reset edges. Listing 4.1 implements this transformation by:

- Adding a `SETUP_EDGE` property to the single outgoing edge of the "Start" vertex.
- Adding an `INITIAL_VERTEX` property to the first vertex after the "Start" vertex.
- Connecting all reset edges pointing towards the “End” vertices to the first vertex after the “Start” vertex and adding a `RESET_EDGE` property to these edges for later identification.
- Removing all “End” vertices as those can never be reached (because their incoming edge has been redirected to the vertex after the “Start” vertex).

Figure 4.1 shows the original and the transformed example model according to the rules outlined above.

4.3 Path Generation with Graphwalker

Once the model has been prepared for its processing, Graphwalker can be used to generate execution paths by providing the model as well as a test generation strategy and stop conditions as inputs. The tool supports, among others, the following test generation strategies [13]:

random. Navigates through the model randomly while honouring guards preventing certain transitions.

quick_random. Randomly walks through the model by choosing an unvisited edge and then following the shortest path to this edge using Dijkstra’s algorithm. However, the algorithm does not honour guards, making it unsuitable for models that use them.

a_star. Implements the A-star algorithm [20] but only works for stop conditions specifying the name of a vertex or edge as a target. The A-star algorithm searches for the shortest path to a target node by using a heuristic function that estimates the length of the remaining path and is thus in many cases faster than Dijkstra’s algorithm.

[4] yEd Graph Editor: https://www.yworks.com/products/yed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

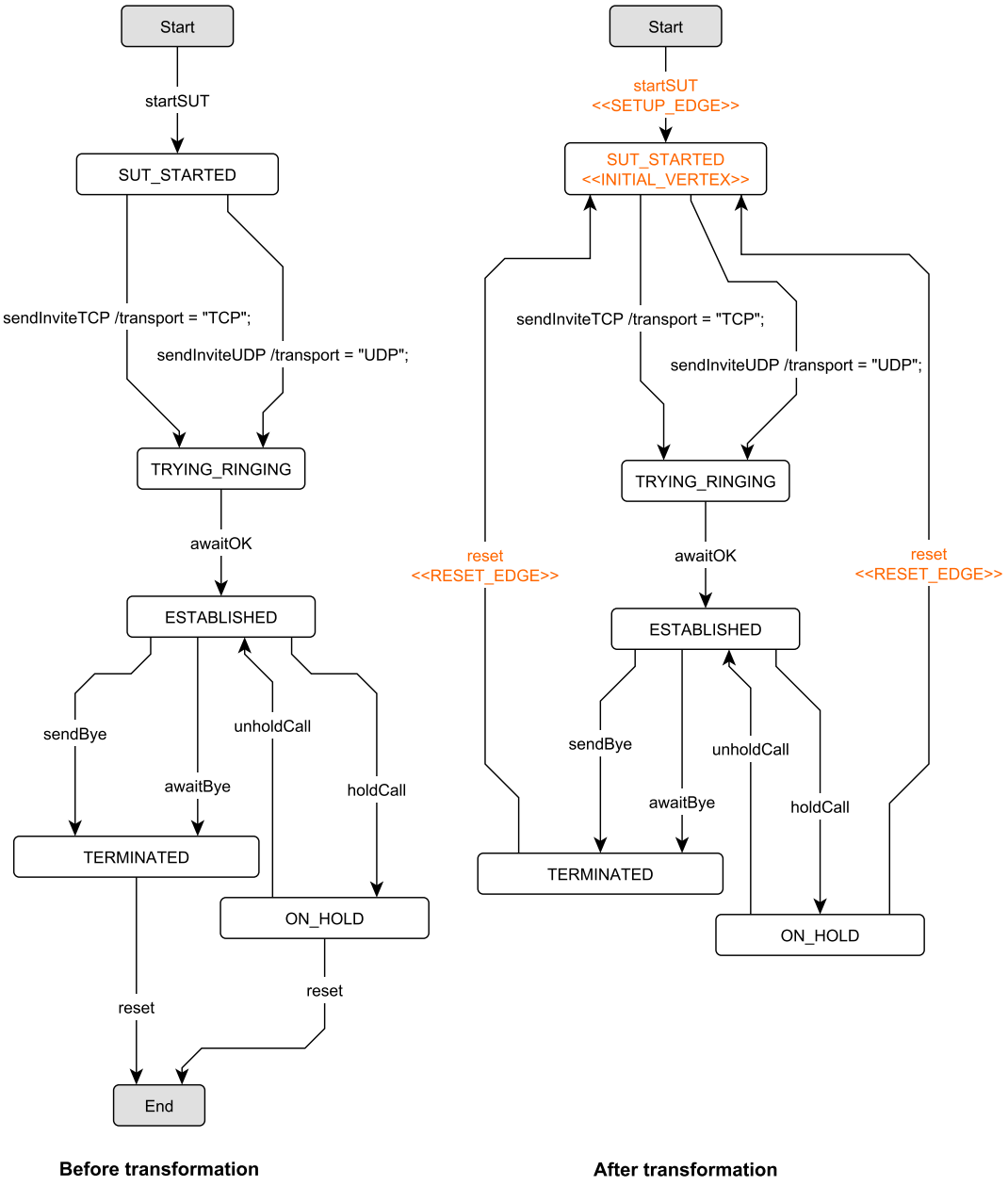

Figure 4.1: Example input model before and after the model transformation performed prior to the path generation with Graphwalker

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
void transformModel(final Context context) {
  Model model = new Model(context.getModel());
  Edge.RuntimeEdge setupAction = context.getNextElement();
  if (setupAction == null)
      throw new IllegalStateException("Missing setup edge from Start");

  //The setup edge from the old model can be found by ID
  //in the new model and receives the SETUP_EDGE property
  final Edge setupEdge = model.getEdges().stream()
    .filter(edge -> edge.getId().equals(setupAction.getId()))
    .collect(MoreCollectors.onlyElement());
  setupEdge.setProperty(SETUP_EDGE, true);

  //Find the first vertex that we execute, this is usually a
  //precondition state and we mark it so we can assign it a GIVEN step
  String initialVertexId = setupAction.getTargetVertex().getId();
  final Vertex startVertex = model.getVertices().stream()
    .filter(vertex -> vertex.getId().equals(initialVertexId))
    .collect(MoreCollectors.onlyElement());
  startVertex.setProperty(INITIAL_VERTEX, true);

  //Find all the reset edges; modify their target vertex to point to
  //the initial vertex (making loops); add the RESET_EDGE property to
  //detect when we have to finish a scenario
  model.getEdges().stream()
    .filter(e -> e.getTargetVertex()
      .getName().equalsIgnoreCase(END_VERTEX_NAME))
    .forEach(edge -> {
        edge.setTargetVertex(startVertex);
        edge.setProperty(RESET_EDGE, true);
    });

  //Delete the unused end vertices
  model.getVertices().removeIf(vertex ->
    vertex.getName().equalsIgnoreCase(END_VERTEX_NAME));

  //Lastly build the new model and modify the context to make use of it.
  Model.RuntimeModel transformedModel = model.build();
  context.setModel(transformedModel);
  context.setNextElement(setupEdge);
}
```

Listing 4.1: Transformation of input models before path generation with Graphwalker

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The test generation strategy needs to be combined with a stop condition such as by providing a percentage for the desired edge coverage, vertex coverage or by defining a limit in terms of execution time or path length.

4.4 Scenario Generation

Graphwalker's API provides the necessary methods to generate a path through the model. In particular, it offers the methods `hasNextStep()` and `getNextStep()`, which allow the stepwise iteration through the generated path. With these, an algorithm as outlined in Listing 4.2 can be implemented. It creates a single scenario by generating model element identifiers representing the path through the model and resolves them to the actual BDD step and the scenario information which is provided with the model mapping description. If it finds the `RESET_EDGE` property while traversing the model, it knows that the scenario has ended and stores this reset edge so that it can be added to the next scenario (see Section 3.4.2 for the rationale behind this).

The step type is selected based on a simple algorithm that is presented in Section 3.4.3 and is formalised in Listing 4.3. Note that the algorithm simply selects a preference, but in case a mapping for the preferred step type has not been specified, the resulting step will use whatever is specified in the mapping. Also note that the step keyword is altered to "And" if the previous step is of the same type, which makes the scenario text more fluent to read.

4.5 Story Execution with JBehave

Once the model has been processed and converted into test scenarios, the final part is to execute them as part of a BDD story. Two modes of operations have been implemented. The generated scenarios can either be aggregated and written into a story file for later execution (offline execution), or they can be immediately executed if the model generation takes place inside a BDD story execution run (online execution).

4.5.1 Offline Execution

For offline execution, the generated scenarios are transformed to their textual representation and then aggregated to form a BDD story. The resulting story can be transparently executed by JBehave as it is syntactically indistinguishable from a manually written story. A simple command-line interface application has been developed to control the story generation process.

4.5.2 Online Execution

This execution mode is more demanding, as it requires the generation of the scenarios to be integrated with the BDD framework. Figure 4.2 depicts a simplified version of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Step lastResetStep;
int numScenario = 0;
Machine machine = newMachine(...);

ModelScenario generateScenario() {
  boolean scenarioFinished = false;
  List<ResolvedStep> steps = new ArrayList<>();
  HashSet<String> scenarioInfo = new LinkedHashSet<>();

  if (lastResetStep != null) steps.add(lastResetStep);
  while (machine.hasNextStep() && !scenarioFinished) {
    Context nextStep = machine.getNextStep();
    scenarioFinished = nextStep.hasProperty(RESET_EDGE);
    StepType preferredStepType = getPreferredStepType(nextElement);
    ResolvedStep step = resolveStep(nextStep.getName(), preferredStepType)

    step.getScenarioInfo().ifPresent(scenarioInfo::add);
    if (scenarioFinished) {
      lastResetStep = step;
    } else {
      steps.add(step);
    }
  }

  String scenarioTxt = "#" + ++numScenario +" "+ join(" ", scenarioInfo);
  return new ModelScenario(scenarioTxt, steps);
}

Machine newMachine(Path graphmlFile, String strategyStopCondition) {
    //Parse generation strategy/stop condition e.g. "random(length(10))"
    PathGenerator gen = GeneratorFactory.parse(strategyAndStopCondition)

    //Load the *.graphml file containing the model
    List<Context> contexts = new YEdContextFactory().create(graphmlFile);
    Context context = contexts.get(0);
    context.setPathGenerator(gen);

    //Transform the model and create the Graphwalker FSM (SimpleMachine)
    transformModel(context);
    return new SimpleMachine(Collections.singletonList(context));
}
```

Listing 4.2: Scenario generation algorithm

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
StepType getPreferredStepType(final RuntimeBase element) {
    if (element.hasProperty(RESET_EDGE)
        || element.hasProperty(INITIAL_VERTEX)
        || element.hasProperty(SETUP_EDGE)) {
        return StepType.GIVEN;
    } else if (element instanceof Vertex.RuntimeVertex) {
        //represents all other states
        return StepType.THEN;
    } else {
        //represents all other transitions
        return StepType.WHEN;
    }
}
```

Listing 4.3: Step keyword preference selection algorithm

the resulting class diagram that includes support for executing .graphml model files in addition to standard JBehave .story files and required changing some of the original JBehave classes as they were not built with such extensions in mind.

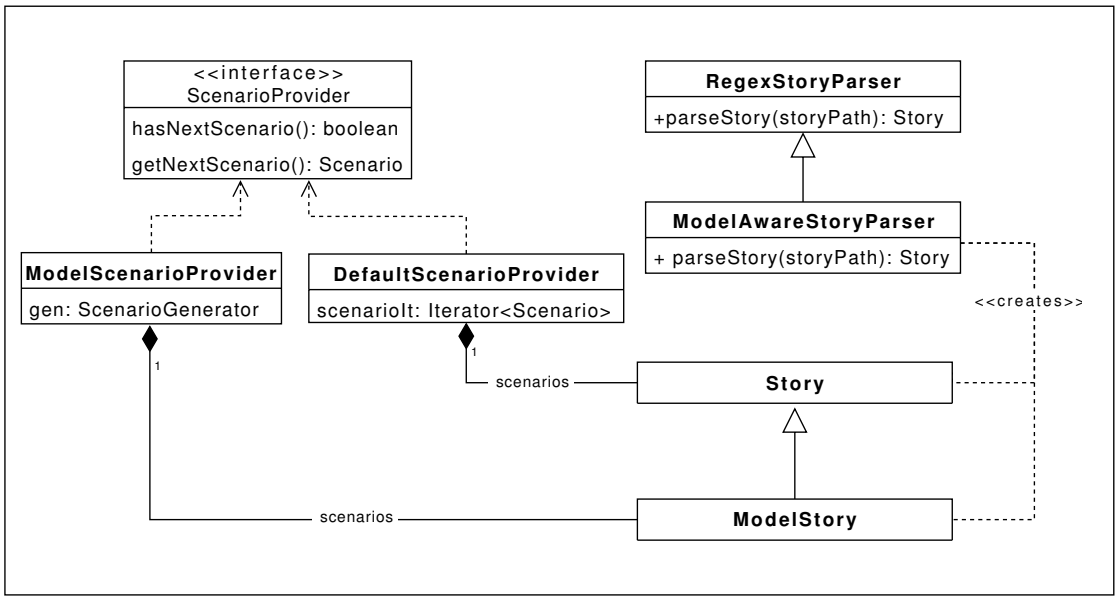

Figure 4.2: Class diagram of the online MBT execution mode implementation for JBehave

A `ModelAwareStoryParser` extends the JBehave default (`RegexStoryParser`) to distinguish between creating plain `Story` objects and new `ModelStory` objects based on the file extension. In the original implementation, the `Story` class contained all its scenarios and the story execution iterated over them during the execution. To allow for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

dynamically generating the scenarios on-the-fly and accommodating for the fact that the number of scenarios is not known in advanced, a `ScenarioProvider` interface was introduced to replace the default list of scenarios. The default implementation used for standard .story files (`DefaultScenarioProvider`) simply serves each scenario as it is requested, whereas the `ModelScenarioProvider` makes use of the previously described algorithm to create new scenarios as they are being generated by Graphwalker.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 5

Evaluation

In this chapter, the evaluation of the developed testing approach is presented. First, Section 5.1 summarises the research questions and the methodology used to answer them. Section 5.2 introduces the SUT that was used throughout the evaluation as a test subject. Afterwards, the following sections elaborate the conducted evaluation in detail.

5.1 Evaluation Design

The evaluation of this thesis is targeted to answer the following research questions:

1. Is it feasible to generate meaningful BDD stories that adequately cover the functionality of the SUT?
2. Are generated BDD stories comprehensible enough so that they can be understood by the main stakeholders, in particular developers and testers?
3. How high is the effort required to create a model and enrich existing BDD steps for their usage in generated tests? Is the effort required for specification changes in the SUT lower when using this method than adapting the manually developed BDD stories?

A case study was conducted in order to investigate these questions. It used a VoIP gateway that translates between the SIP and WebSocket protocol as the test subject. To answer the first question regarding feasibility, code coverage measurements were performed. The measurements were conducted for an existing set of test cases, which were developed by testers of the SUT, and for test cases generated using the MBT approach outlined in this thesis. The results were then compared to see whether it is possible to generate useful test cases with the help of the developed approach.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

A survey was conducted among developers, testers and test managers to answer the second research question and evaluated whether the generated BDD stories are comprehensible. Participants had to rate and differentiate manually written and automatically generated test cases in a blind comparison.

Lastly, to answer the final research question with regards to the required effort for the developed method, editing steps required to introduce new and change existing tests manually were compared to editing steps required to model the SUT and adapt the model in case of changes. Two different scenarios were investigated: introducing a completely new test and extending an existing test to incorporate changes in the behaviour.

5.2 Case Study

The case study was built around a proprietary VoIP gateway that translates SIP messages into a WebSocket-based protocol. As the gateway keeps track of the state of each SIP session, it can perform certain logic without intervention, e.g. it can monitor a SIP endpoint on its own. Figure 5.1 showcases the messages sent to and from the gateway when a new call is established.

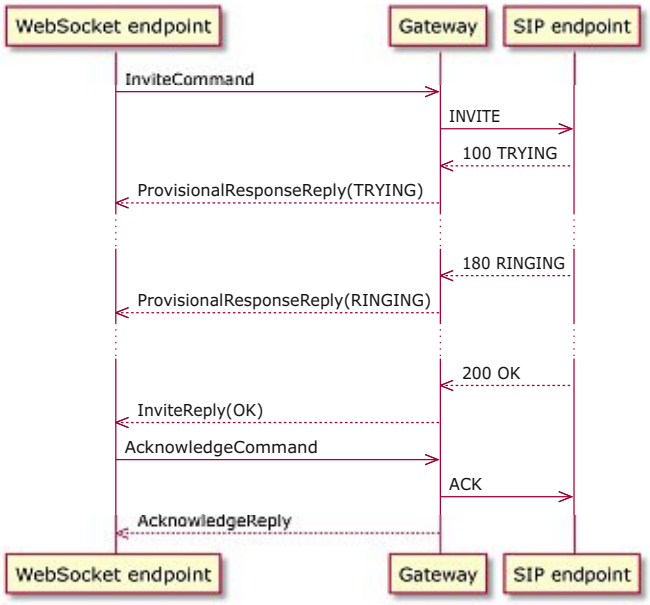

Figure 5.1: Example messages processed by the VoIP gateway

To test the VoIP gateway, BDD stories need to interact with the system using both the SIP and the WebSocket protocol. This way, steps can trigger a behaviour via the WebSocket protocol and verify the response on the SIP side and vice versa.

The existing BDD stories for the gateway consist mostly of high-level integration and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

load tests. They were developed after the actual implementation of the feature was already completed. In total, the test project is comprised of 33 stories with a total of 171 scenarios consisting of 321 steps. The stories cover the following features of the gateway:

- Incoming / outgoing call establishment via UDP and TCP
- Incoming / outgoing calls with background load (e.g. out-of-dialog messages)
- Out-of-dialog messaging with UDP and TCP as well as with and without proxies
- SIP forking
- Error handling, e.g. in case the target is unavailable or in-dialog messages are received after the dialog has ended
- Session monitoring
- `OPTIONS` monitoring

Examples for the case study were chosen out of the already existing test cases. Load tests were omitted as they had not been implemented in a manner that was particularly interesting for answering the research questions. They consist only of very few steps and handle most of the logic within the Java implementation linked to each step.

5.3 Code Coverage

Code coverage measurements were performed for both the manually written stories that were developed by the testers of the VoIP gateway as well as for the stories generated with the developed MBT approach. Eventually, those results were compared in order to answer the first research question with the focus on confirming that the described method can adequately cover the code base of a SUT. For the comparison to be meaningful care had to be taken that the model and the manually written story examine the same features of the SUT. As one of the requirements was to reuse existing steps, differences in the measured code coverage should only occur in case the generated stories „discovered" a new path in the SUT that a tester missed when writing the stories manually. Three different features of the SUT were looked at in this evaluation:

Basic call handling: This feature simulates a standard call flow for both incoming as well as outgoing calls established using SIP over TCP or UDP. Additionally, it includes sending `UPDATE`, `INFO` and `MESSAGE` requests in an existing SIP dialog.

This feature has been chosen as it represents the basic use case of SIP — establishing a phone call. It resembles the state machine implemented by SIP RFC 4235 [39], although the concrete model is rather high level and does not provide insight into the more fine-grained transaction handling state machine.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The state machine model developed for this feature is depicted in Figure 5.2. Figure 5.6 shows sample stories illustrating test cases for this feature set. In particular A) shows a manually written story whereas B) has been generated from the model.

Out-of-dialog messaging with proxy support: This feature covers sending of SIP `MESSAGE` requests with varying responses (OK, decline, timeout) out-of-dialog, either directly to another SIP endpoint or by passing through several proxies. Out-of-dialog in the context of SIP means not within a dialog (i.e. without an established call), which is typically the case for e.g. heart beats, automated background services, etc. Please refer to Figure 5.3 for the graphical model and Figure 5.8 for tests of this feature.

OPTIONS monitoring: This feature starts and stops the automatic monitoring of a SIP endpoint using an `OPTIONS` request based heartbeat (see Figure 5.4). Whenever the monitoring is active, the gateway sends periodic `OPTIONS` messages. In case the monitored endpoint changes its response, a message containing the new response is transmitted to the WebSocket controller. This model is relatively small since not many pre-existing stories and steps were available for testing this feature.

The process of conducting the measurements consisted of (i) designing the model based on the SUT documentation and the existing test cases, (ii) generating the test cases using a random walk through the model with and edge coverage of 100% as stop condition and (iii) executing the generated stories using the offline execution mode and collecting the coverage data. Table 5.1 shows the number of generated BDD stories, scenarios and steps in comparison to the number of manually written BDD stories, scenarios and steps for each of the three examples. Due to the usage of random walks through the model, the number of generated scenarios and steps is a lot higher compared to the manually written ones and can also vary greatly for each regeneration. The random walk test generation strategy also generates duplicate sequences of test steps which is another reason for the higher number. Manually written test cases are grouping feature into multiple BDD stories (e.g. distinguishing tests involving TCP from ones that are using UDP), which has not been done for the generated stories leading to just a single BDD story for each feature.

Using a more sophisticated generation strategy than random walks might lead to smaller test scenarios or improved model coverage which could result in higher code coverage, however the chosen MBT framework Graphwalker does not provide these strategies at the moment. As a remedy, an additional test case generation run (in the result Table 5.3 labelled as “MBT 2000”) was conducted for the out-of-dialog messaging with proxy support feature. The 2000 refers to the stop condition of 2000 edge-vertex pairs (path length of 2000 edges plus vertices) which is intended to simulate a higher model coverage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

	Manual			MBT		
	Stories	Scenarios	Steps	Stories	Scenarios	Steps
Basic call handling	7	43	97	1	36	283
Out-of-dialog messaging with proxy support	5	20	42	1	28	218
OPTIONS monitoring	2	13	30	1	3	30

Table 5.1: Number of BDD stories, scenarios and steps by example

Coverage data was collected using the JaCoCo[1] code coverage library. The SUT was started with the JaCoCo Java agent attached and before each test was started, the SUT was allowed to settle down. After running a set of stories, the system was again allowed to settle down for 90 seconds to finish any cleanup tasks. After that, the gateway was stopped so that JaCoCo finished collecting coverage data.

5.3.1 Code Coverage Results

Table 5.2 shows that manually written test cases for the "basic call handling" feature reach an instruction coverage of 39.37% whereas generated test cases reach 39.31% which is 0.06% less. In absolute numbers this represents a difference of 45 instructions. Similarly, the branch and line coverage were also slightly reduced by 0.3% and 0.08% respectively. Detailed coverage metrics for the other two features can be found in Table 5.3 and 5.4. They show a slightly increased coverage for the generated test cases.

As stated earlier, for the out-of-dialog messaging with proxy support feature, an additional story was generated consisting of 2000 edge-vertex-pairs, which was intended to simulate a higher model coverage, which was not possible to achieve otherwise with the limited algorithms provided by the used MBT framework. To verify whether the higher coverage from this run was achieved due to an increased model coverage and not just by repeatedly executing the same test case, a repetitive run of the existing manual tests was performed as well. The measurement result shown in Table 5.3 confirms the suspicion that the increased coverage was a result of more repetitive tests as both the longer MBT run as well as the longer manual story-based run achieved the same higher coverage.

As an experiment, an additional run was conducted combining the basic call handling model from Figure 5.2 with the OPTIONS monitoring model shown in Figure 5.4 using additional edges as links to create the model in Figure 5.5. The intent was to show that the MBT approach is capable of generating scenarios that have not been written manually by linking existing models together and thus achieving a higher coverage. The results can be seen in Table 5.5 and show that indeed a coverage increase can be observed in comparison to simply running the manually written tests in succession (instead of intertwining them).

[1] JaCoCo: `https://www.eclemma.org/jacoco/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

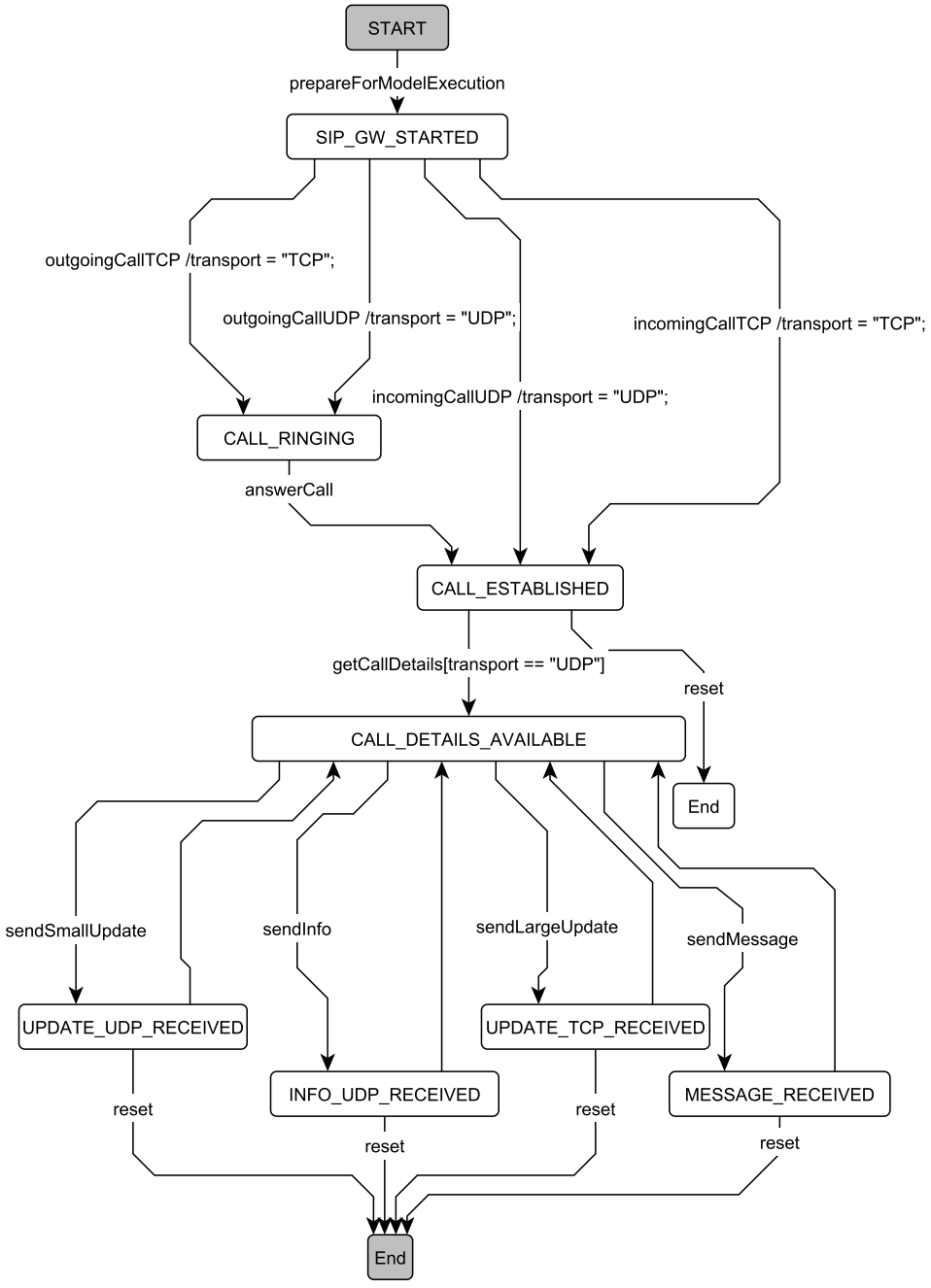

Figure 5.2: Model for the “basic call handling” feature of the VoIP gateway

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

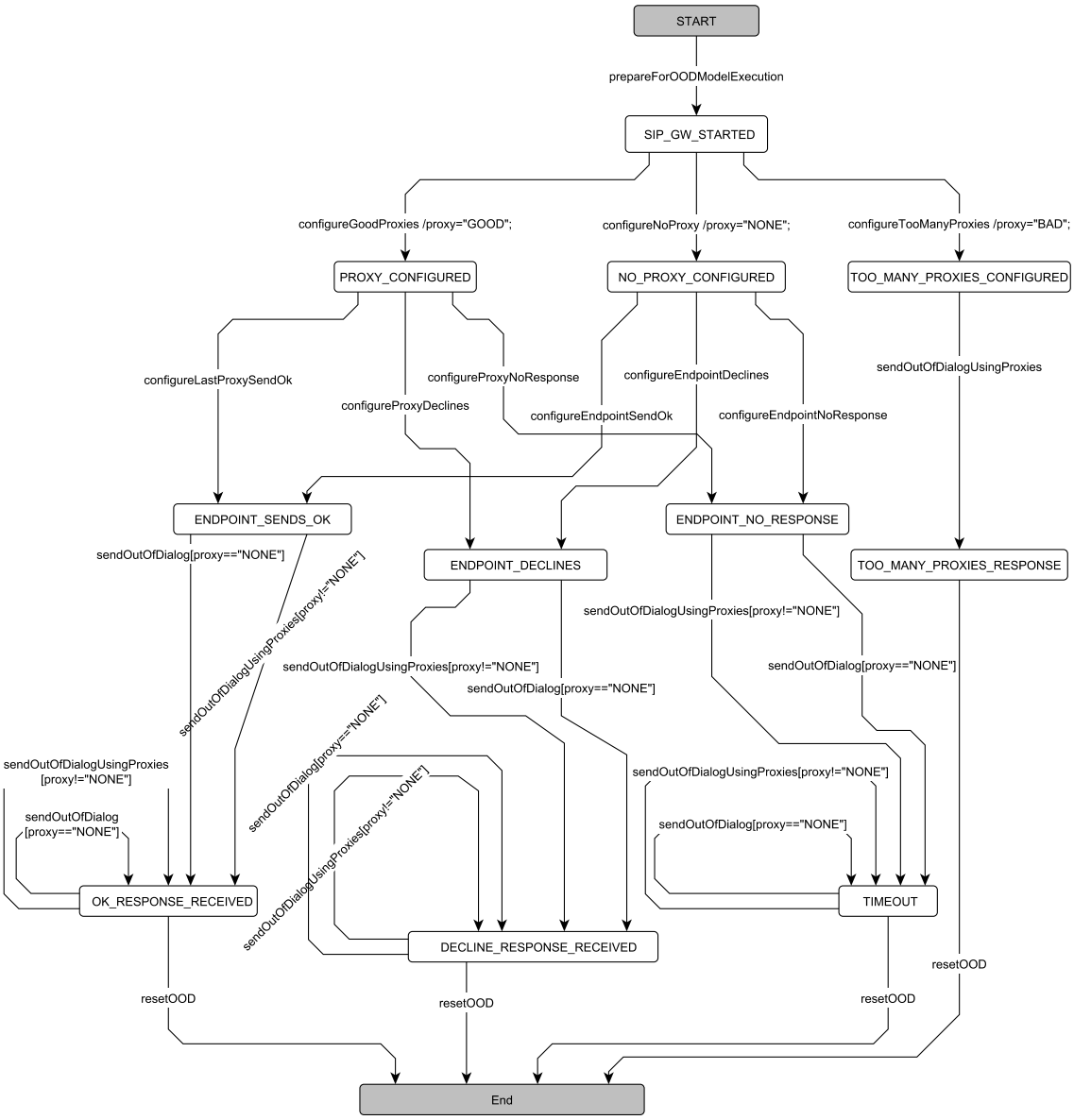

Figure 5.3: Model for the “out-of-dialog messaging with proxy support” feature of the VoIP gateway

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

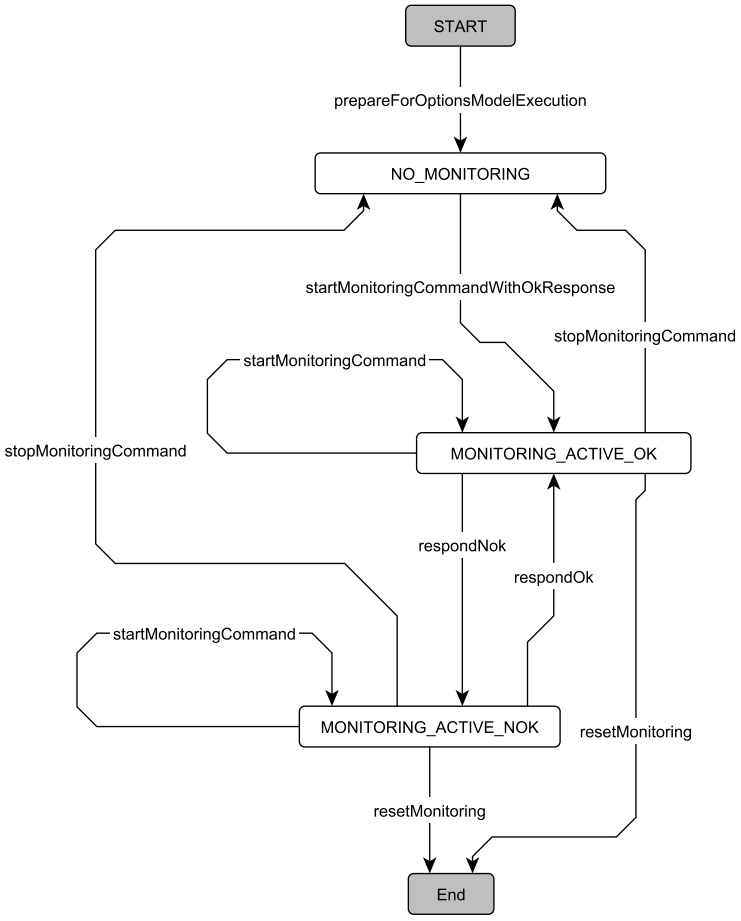

Figure 5.4: Model for the "OPTIONS monitoring" feature of the VoIP gateway

	Instructions		Branches		Lines	
	Percentage	Abs.	Percentage	Abs.	Percentage	Abs.
Manual (Baseline)	39.37	27238	20.92	711	41.70	6939
MBT	39.31 -0.06	27193	20.62 -0.3	701	41.62 -0.08	6926
Total		69178		3399		16642

Table 5.2: Coverage data for the "basic call handling" feature of the VoIP gateway

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

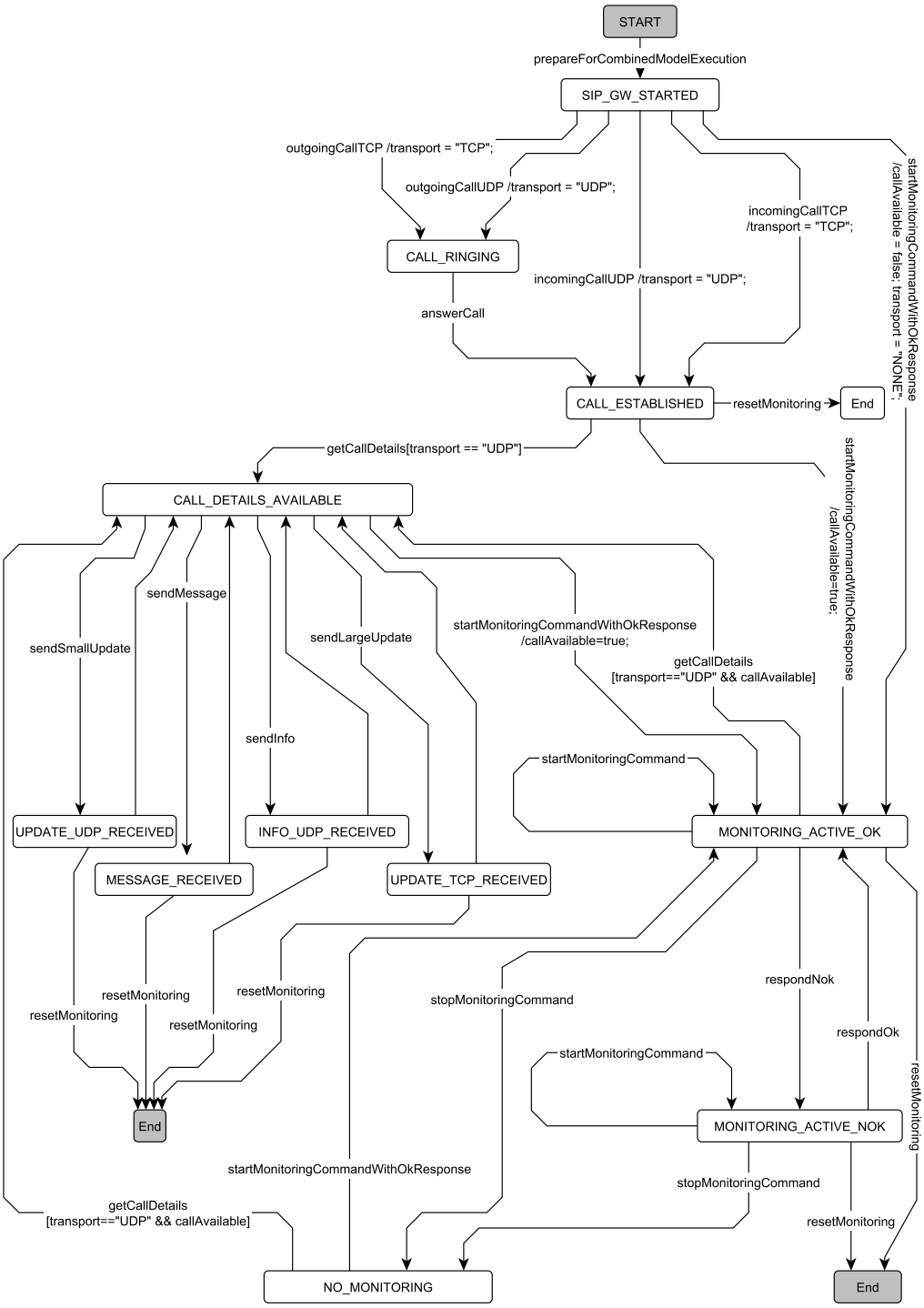

Figure 5.5: Model combining the “basic call handling” feature with the “OPTIONS monitoring” feature of the VoIP gateway

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

	Instructions		Branches		Lines	
	Percentage	Abs.	Percentage	Abs.	Percentage	Abs.
Manual (Baseline)	17.51	12116	6.88	234	18.15	3020
MBT	17.52 +0.01	12118	6.88	234	18.15	3020
MBT 2000	17.54 +0.03	12133	6.94 +0.06	236	18.18 +0.03	3025
Manual Repetitive	17.54 +0.03	12133	6.91 +0.03	235	18.17 +0.02	3024
Total		69178		3399		16642

Table 5.3: Coverage data for the "out-of-dialog messaging with proxy support" feature of the VoIP gateway

	Instructions		Branches		Lines	
	Percentage	Abs.	Percentage	Abs.	Percentage	Abs.
Manual (Baseline)	16.91	11700	5.77	196	17.50	2913
MBT	16.92 +0.01	11706	5.80 +0.03	197	17.52 +0.02	2915
Total		69178		3399		16642

Table 5.4: Coverage data for the "OPTIONS monitoring" feature of the VoIP gateway

	Instructions		Branches		Lines	
	Percentage	Abs.	Percentage	Abs.	Percentage	Abs.
Manual (Baseline)	41.16	28472	21.68	737	43.43	7228
MBT	41.81 +0.65	28923	22.18 +0.5	754	44.04 +0.61	7329
Total		69178		3399		16642

Table 5.5: Coverage data for the combination of the "basic call handling" and the "OPTIONS monitoring" feature of the VoIP gateway

5.3.2 Code Coverage Result Interpretation

The collected data indicates that MBT based stories can reach a coverage that is on par with manually written stories and are even able to reach a higher code coverage. The observation that in the best cases investigated in the conduced case study only a few additional lines could be covered by MBT based stories is not surprising because the models used for the test generation were developed based on the existing manually written test cases and testing additional features was intentionally avoided. Even though the same BDD steps were used to cover the same features, the MBT approach managed to execute a few more lines in two out of three cases. Detailed analysis of these cases revealed the following reasons:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- The generated scenarios did sometimes “discover” new paths, e.g. in OPTIONS monitoring tests an allowed request is to start the monitoring again even though it is already running, which was never tested manually.

- MBT scenarios were more extensive and due to the random generation strategy contained duplicate test cases (see Table 5.1). As an example, the out-of-dialog messaging with proxy support feature was tested manually using five stories with a total of 20 scenarios resulting in 42 executed steps whereas the MBT version achieving a 0.01% increased coverage consisted of 28 scenarios and 218 steps. The increased number of steps also quadrupled the execution time from 4.4 seconds to 16.9 seconds. This observation suggests that the increased coverage was partly due to running the same tests repeatedly, which can trigger alternative code paths.

For the basic call handling test, the MBT based stories achieved a lower coverage most likely because the manually written tests inconsistently used two different sets of steps to handle calls. The older set of steps was already deprecated but still used in some stories and exposed slightly different behaviours e.g. with respect to which SIP headers were set. The MBT version replicated all test scenarios using the newer version of the steps exclusively.

The model-based testing method showed particular benefits when it was not restricted to replicating existing tests. This was observed when combining the “basic call handling” and “OPTIONS monitoring” feature sets. In comparison to executing the manually written test cases covering the same features in succession, the tests generated from the combined model achieved a 0.61% higher line coverage which amounts to 101 additionally covered lines of code.

5.4 Comprehensibility

To answer the second research question concerning the understandability and readability of the generated BDD stories, a survey was conducted in the form of a questionnaire. It consisted of two parts with the first part containing general questions and the second part comprised of three comparisons between manually written and automatically generated stories covering roughly the same features of the investigated VoIP gateway. Due to the different structure and scope of generated and manually written stories, the chosen stories did not entirely correspond to each other regarding covered functionality.

The general questions outlined in Table 5.6 were intended to gain some general understanding for the knowledge and experience of the respondents regarding testing and modelling. The main part focused on the comparison between model-generated stories and manually written stories. Three side-by-side examples were given labelled as “A)” / “B)” (see Figures 5.6, 5.7 and 5.8). To avoid any bias towards a method, no information

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Question	Answer options
Professional background	Software Engineer Test Engineer Test Manager Other
How many years of experience do you have in your field?	free form
Which of these testing tools have you used in the past year?	Unit test frameworks (JUnit, NUnit, TestNG,...); BDD test frameworks (JBehave, Cucumber, Jasmine,...); MBT testing frameworks (Spec Explorer, TestWeaver, Graphwalker,...)
How much experience do you have with behaviour-driven development/testing?	1 None ... 5 Very much
How much experience do you have with model-based testing?	1 None ... 5 Very much
How much experience do you have with VoIP?	1 None ... 5 Very much
Which of these diagrams are you familiar with?	Class diagram State machine diagram Activity diagram Sequence diagram
During development/testing how often do you use any kind of models (mental models, sketches, tool supported modeling,...)?	Daily Few times per week Once per week Once per month A few times per year Less frequent Never
In case you are using models for testing already, for which purposes are you using them?	Automatic test case generation Input data generation Document your tests Plan your tests Model the system that you need to test
Do you use a modeling tool for creating your models?	Yes/No

Table 5.6: Survey: General questions

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

as to whether a story was manually written or generated was provided. For each of the three examples, the following groups of questions had to be answered:

Comparison questions. For the following questions, participants had to decide between the story labelled "A)" and "B)" or select "Undecided":

- I prefer this version
- I think this story is better readable
- I think this story is better suited for discussion with co-workers
- I think this story is easier to debug in case an error occurs
- I think this story is more scalable (in terms of writing many tests)
- I think this story has a better structure
- I think this story was manually written

Detailed perception questions. The following questions had to be answered using a 5-point Likert scale labelled as 1 = Strongly agree, 2 = Agree, 3 = Undecided, 4 = Disagree, 5 = Strongly disagree. Participants had to rate both examples "A" and "B" individually:

- Scenario titles were helpful to understand the test case
- The scenarios were short and concise
- The structure of the story was good
- I would accept this story in my own test project

Free text comment. A last optional free form question allowed the participants to provide other insights and comments that were not captured by the questions so far: "Room for comments - e.g. are there other reasons why you preferred one story?"

5.4.1 Survey Results

The questionnaire was sent to 40 practitioners already familiar with BDD, most of them having a decent understanding of VoIP, which was needed as the sample stories covered VoIP related use cases. The raw data from the questionnaire can be found in Appendix B.

In total, 25 practitioners replied with an average of 5.8 years of experience in their field (see the distribution chart in Figure B.2). Most of the participants were test engineers (14) followed by software engineers (7) managers (3) and a single system engineer (see Figure B.1). Participants had much experience with BDD scoring an average of 3.8/5 on the scale where 1 equals no experience and 5 represents very much experience. Similarly, participants had at least some experience with VoIP (on average 3.04 - see Figure B.5). In contrast, participants had little experience with MBT scoring only 1.52 (refer to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Meta: @AfterStory: setup\CleanMessageStore.story                                A)

Scenario: Booking
Given booked profiles:
| profile                 | group         | host      | identifier |
| cats-agent-b2b-websocket | b2bua-adapter | <anyhost> | b2bAdapter |

Scenario: Setup B2BUA instance
Given b2bua instance is started
Then declare presence request is received on cats-agent-b2b-websocket b2bua-adapter

Scenario: Preparing the default external party
Given booked profiles:
| profile            | group  | host          | identifier      |
| cats-agent-b2b-voip | Callee | <callee.host> | externalCallers |
And named SIP contacts:
| key          | profile         | display-name | user-entity  | sip-uri                        |
| simpleCaller | externalCallers | SimpleCaller | simpleCaller | sip:simpleCaller@1.1.1.1:1234 |

Scenario: Outgoing Phone Call
Given an client want to call a simpleCaller
When the client sends the request to call on the cats-agent-b2b-websocket b2bua-adapter
Then expected a call to be established for contact with key simpleCaller within 500 milliseconds

Scenario: Terminate the call
Then Terminate the call simpleCaller
Then all contacts are removed from simpleCaller                          Manual
-------------------------------------------------------------------------------
Scenario: #1 Incoming call (UDP) info request                                   B)
Given the b2bua is ready for basic call handling tests
And b2bua instance is started
When making a call from CALLEE using UDP to sip:b2bua@$1.1.1.1:1234
Then CALLEE CallState is ESTABLISHED within 5000 ms
When VoIP CALLEE.Contact1 call ID is stored as :=> CALL_ID
Then report ${CALL_ID}
When cats-agent-b2b-websocket 1 send a small sip info for call-id ${CALL_ID}
Then b2b receives Info reply with transport UDP and with 200 reason
     code to cats-agent-b2b-websocket 1
And report ${CALL_ID}

Scenario: #2 Outgoing call (UDP) answered
Given calls are ended and message stores are cleared
And b2bua instance is started
When the client wants to call a sip contact CALLEE.Contact1
     using cats-agent-b2b-websocket 1 using transport UDP
Then CALLEE CallState is RINGING within 5000 ms
When CALLEE answers incoming calls
Then CALLEE CallState is ESTABLISHED within 5000 ms

Scenario: #3 Incoming call (TCP)
Given calls are ended and message stores are cleared
And b2bua instance is started
When making a call from CALLEE using TCP to sip:b2bua@${env.b2bua.host}
Then CALLEE CallState is ESTABLISHED within 5000 ms
When VoIP CALLEE.Contact1 call ID is stored as :=> CALL_ID
Then report ${CALL_ID}                                                      MBT
```

Figure 5.6: Survey story example 1: Basic call handling

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

A)

```
GivenStories: demo/shared/EstablishCall.story

Scenario: Verify that the call is not on hold
Then VoIP PHONES.CALLER is in hold state NONE within 1 sec
Then VoIP PHONES.CALLEE is in hold state NONE within 1 sec

Scenario: setting call on hold
When VoIP PHONES.CALLER puts call on hold
Then VoIP PHONES.CALLER is in hold state HOLDING within 2 sec
Then VoIP PHONES.CALLEE is in hold state HELD within 2 sec
When VoIP PHONES.CALLEE puts call on hold
Then VoIP PHONES.CALLER is in hold state DOUBLEHOLD within 2 sec
Then VoIP PHONES.CALLEE is in hold state DOUBLEHOLD within 2 sec
When VoIP PHONES.CALLEE releases call on hold
Then VoIP PHONES.CALLER is in hold state HOLDING within 2 sec
Then VoIP PHONES.CALLEE is in hold state HELD within 2 sec
When VoIP PHONES.CALLER releases call on hold
Then VoIP PHONES.CALLER is in hold state NONE within 1 sec
Then VoIP PHONES.CALLEE is in hold state NONE within 1 sec

Scenario: Cleanup
Given VoIP PHONES are terminated
Then PHONES CallState is INITIAL within 1000 ms
And PHONES are removed
```

Manual

B)

```
Scenario: #1 Remote holds local => HELD; Remote releases local from hold
          Local holds remote => HOLDING; => DOUBLEHOLD; Local releases remote from hold
Given a call is established
Then VoIP PHONES.LOCAL is in hold state NONE within 2 sec
When VoIP PHONES.REMOTE put call on hold
Then VoIP PHONES.LOCAL is in hold state HELD within 2 sec
When VoIP PHONES.REMOTE releases call on hold
Then VoIP PHONES.LOCAL is in hold state NONE within 2 sec
When VoIP PHONES.LOCAL put call on hold
Then VoIP PHONES.LOCAL is in hold state HOLDING within 2 sec
When VoIP PHONES.REMOTE put call on hold
Then VoIP PHONES.LOCAL is in hold state DOUBLEHOLD within 2 sec
When VoIP PHONES.LOCAL releases call on hold
Then VoIP PHONES.LOCAL is in hold state HELD within 2 sec
When VoIP PHONES.LOCAL put call on hold
Then VoIP PHONES.LOCAL is in hold state DOUBLEHOLD within 2 sec
When VoIP PHONES.REMOTE releases call on hold
Then VoIP PHONES.LOCAL is in hold state HOLDING within 2 sec

Scenario: #2 Remote holds local => HELD; Local holds remote => DOUBLEHOLD;
Given terminate calls and reset phones
And a call is established
Then VoIP PHONES.LOCAL is in hold state NONE within 2 sec
When VoIP PHONES.REMOTE put call on hold
Then VoIP PHONES.LOCAL is in hold state HELD within 2 sec
When VoIP PHONES.REMOTE put call on hold
Then VoIP PHONES.LOCAL is in hold state HELD within 2 sec
When VoIP PHONES.LOCAL put call on hold
Then VoIP PHONES.LOCAL is in hold state DOUBLEHOLD within 2 sec
When VoIP PHONES.LOCAL put call on hold
Then VoIP PHONES.LOCAL is in hold state DOUBLEHOLD within 2 sec
```

MBT

Figure 5.7: Survey story example 2: Hold states

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

A)

```
Scenario: #1 Last proxy accepts - OK response
Given the b2bua is ready for basic call handling tests
And b2bua instance is started
And named SIP proxies:
| key    | profile         | display-name | user-entity | sip-uri                |
| proxy1 | externalCallers | proxy1       | proxy1      | sip:proxy1@1.1.1.1:1234 |
| proxy2 | externalCallers | proxy2       | proxy2      | sip:proxy2@1.1.1.1:1234 |
| proxy3 | externalCallers | proxy3       | proxy3      | sip:proxy3@1.1.1.1:1234 |
| proxy4 | externalCallers | proxy4       | proxy4      | sip:proxy4@1.1.1.1:1234 |
| proxy5 | externalCallers | proxy5       | proxy5      | sip:proxy5@1.1.1.1:1234 |
| proxy6 | externalCallers | proxy6       | proxy6      | sip:proxy6@1.1.1.1:1234 |
| proxy7 | externalCallers | proxy7       | proxy7      | sip:proxy7@1.1.1.1:1234 |
| proxy8 | externalCallers | proxy8       | proxy8      | sip:proxy8@1.1.1.1:1234 |
| proxy9 | externalCallers | proxy9       | proxy9      | sip:proxy9@1.1.1.1:1234 |

Given an client want to send out dialog messages to a simpleCaller with last proxy sends OK
When cats-agent-b2b-websocket b2bua-adapter sends 1 out of dialog messages with strict routing
Then b2b sends a MessageReply with 200 reason code to cats-agent-b2b-websocket b2bua-adapter
When cats-agent-b2b-websocket b2bua-adapter sends 1 out of dialog messages with strict routing
Then b2b sends a MessageReply with 200 reason code to cats-agent-b2b-websocket b2bua-adapter
```

MBT

B)

```
Meta: @AfterStory: setup\CleanMessageStore.story

Scenario: Booking
Given booked profiles:
| profile                 | group         | host      | identifier |
| cats-agent-b2b-websocket | b2bua-adapter | <anyhost> | b2bAdapter |

Scenario: Preparing the default external party
Given booked profiles:
| profile             | group  | host          | identifier      |
| cats-agent-b2b-voip | Callee | <callee.host> | externalCallers |
And named SIP contacts:
| key          | profile         | display-name | user-entity  | sip-uri                        |
| simpleCaller | externalCallers | SimpleCaller | simpleCaller | sip:simpleCaller@1.1.1.1:1234 |
And named SIP proxies:
| key    | profile         | display-name | user-entity | sip-uri                |
| proxy1 | externalCallers | proxy1       | proxy1      | sip:proxy1@1.1.1.1:1234 |
| proxy2 | externalCallers | proxy2       | proxy2      | sip:proxy2@1.1.1.1:1234 |
| proxy3 | externalCallers | proxy3       | proxy3      | sip:proxy3@1.1.1.1:1234 |
| proxy4 | externalCallers | proxy4       | proxy4      | sip:proxy4@1.1.1.1:1234 |
| proxy5 | externalCallers | proxy5       | proxy5      | sip:proxy5@1.1.1.1:1234 |
| proxy6 | externalCallers | proxy6       | proxy6      | sip:proxy6@1.1.1.1:1234 |
| proxy7 | externalCallers | proxy7       | proxy7      | sip:proxy7@1.1.1.1:1234 |
| proxy8 | externalCallers | proxy8       | proxy8      | sip:proxy8@1.1.1.1:1234 |
| proxy9 | externalCallers | proxy9       | proxy9      | sip:proxy9@1.1.1.1:1234 |

Scenario: Sending out of dialog messages
Given an client want to send out dialog messages to a simpleCaller with last proxy sends OK
When cats-agent-b2b-websocket b2bua-adapter sends 1 out of dialog messages with strict routing
Then b2b sends a MessageReply with 200 reason code to cats-agent-b2b-websocket b2bua-adapter
```

Manual

Figure 5.8: Survey story example 3: Out-of-dialog messaging

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Figure B.4). Familiarity with UML diagrams was high (see Figure B.6), but tool-assisted model creation was rather low, with only 16% making use of tools for creating models (see Figure B.9) even though 48% said they were using at least some kind of model once per week or even more often (see Figure B.7).

The remainder of this section summarises the results of the survey for the comparison questions and detailed perception questions in the categories "preference", "readability", "communication and debugging" as well as "scalability" and finally indicates how well participants were able to identify the type of the story (manually written or automatically generated).

The analysis of the detailed perception questions based on a 5-point Likert scale was done using an interpolated median [40] to find the most common category. The answers were mapped from "Strongly agree" = 1 to "Strongly disagree" = 5. Using the `agreement` function [41] the median can be put into context. It ranges from −1 to 1:

Agreement = 1 all respondents agreed on a single category

Agreement = -1 perfect polarisation, half of the respondents strongly agreed and half strongly disagreed

Agreement = 0 even distribution of responses in all categories

Preference

The first question for each comparison was regarding preference. The manually written stories were preferred twice. Only in Example 3, the generated story was preferred as outlined in more detail in Figure 5.9. Participants noted in the comment section that manually written stories had many setup steps that should be moved, e.g. into `GivenStories`. Regarding the model-generated stories, a participant noted that it was less technical but did not have a clear "Given", "When", "Then" structure, which was however, true also for some manually written story, e.g. Example 1 A).

In addition to the direct question about preference, participants were also asked whether they would accept the given stories story in their test project. The analysis showed that manual stories had a better acceptance rate with an interpolated median of 2.13 in contrast to a median of 2.71 for generated stories, meaning that participants tended to agree to accept a manual story as part of their own project. In contrast, they tended to vote "Undecided" for model-based stories. The agreement on this question was almost identical at around 0.5 for both manual as well as generated stories.

Readability and Structure

Readability was measured by directly asking participants to select which story of each of the example story pairs they regarded more readable and by allowing them to rate each

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

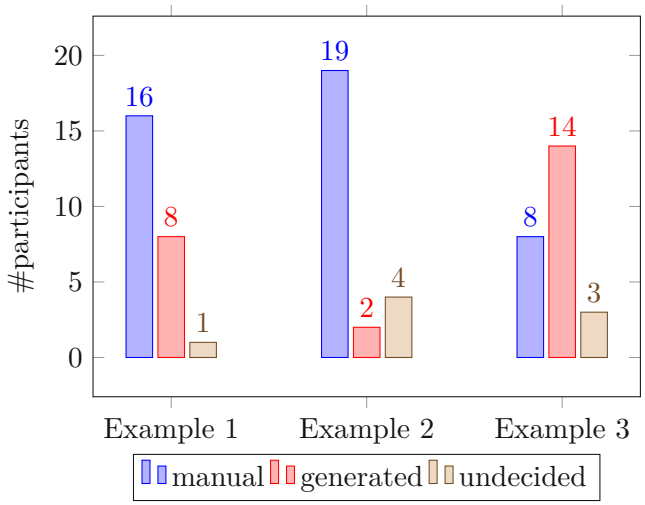

Figure 5.9: Survey results: Preference by example

story in terms of the helpfulness of the scenario title, the conciseness of scenarios and the overall structure of the story.

The manually written story was preferred in two out of the three examples for the question "I think this story is better readable" as outlined in Figure 5.10. Coincidentally Example 3 was the only one where only a single generated scenario was shown covering a single test case wheres in the other examples multiple scenarios were shown in order for the story pair to be roughly the same size.

The results for the question "I think this story has a better structure" were similar and can be seen in Figure 5.11. The structure of the generated story in Example 2 was rated very poorly, but at the same time, participants also didn't like the structure of the manually written story for the same example leaving many participants undecided.

Looking at the results of the detailed perception questions regarding understandability and readability shown in Table 5.7, the picture is not as clear. Participants did not strictly dislike generated stories but on average scores were lower. For example for the question "Scenario titles were helpful to understand the test case" the interpolated median for all manually written stories was 1.89 compared to 2.64 for all generated stories.

Communication and Debugging

Two questions directly focused on how well a story is suited for supporting other development tasks namely as discussion basis with a co-worker (see Figure 5.12) and for debugging (see Figure 5.13). As with all other classifications, the model-based story was the preferred one for Example 3. It remains unclear why participants rated the MBT story as preferred choice in terms of debugging also for Example 1 as the responses for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

	Interp. median		Agreement	
	Manual	MBT	Manual	MBT
Scenario titles were helpful to understand the test	1.89	2.64	0.46	0.32
The scenarios were short and concise	2.04	3.11	0.52	0.63
The structure of the story was good	2.29	2.76	0.47	0.35

Table 5.7: Survey results: Readability and structure measures

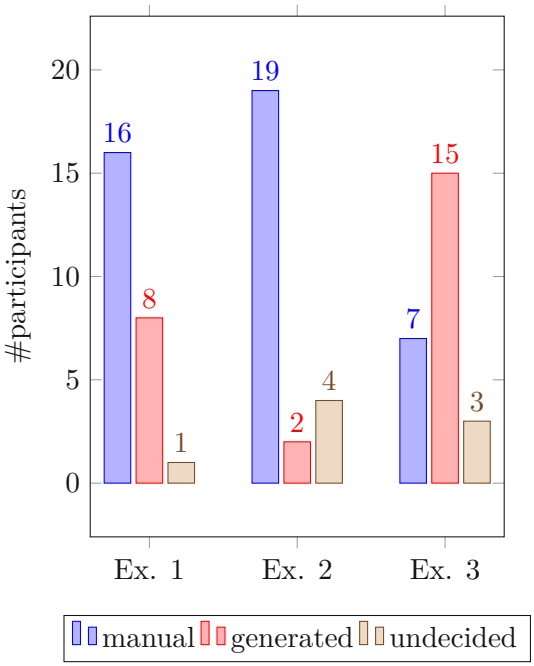

Figure 5.10: Survey results: Readability by example

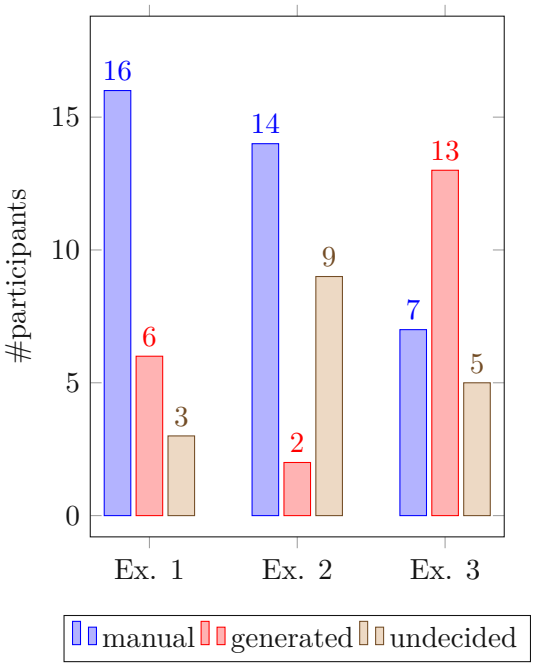

Figure 5.11: Survey results: Preferred structure by example

the detailed perception questions were not in favour of this particular story.

Scalability

The question "I think this story is more scalable (in terms of writing many tests)" was answered very similarly to the results in other categories (see Figure 5.14). As with the other categories, Example 3 was the only one where participants preferred the model-based story over the manually written.

Identification

Participants were mostly able to distinguish between the manually written and the generated stories. Most likely because the structure of the generated stories was quite

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

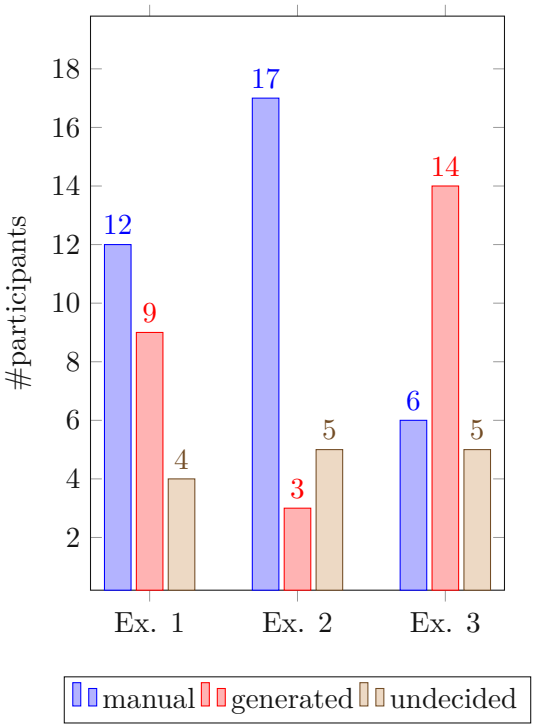

Figure 5.12: Survey results: Suitability for discussion by example

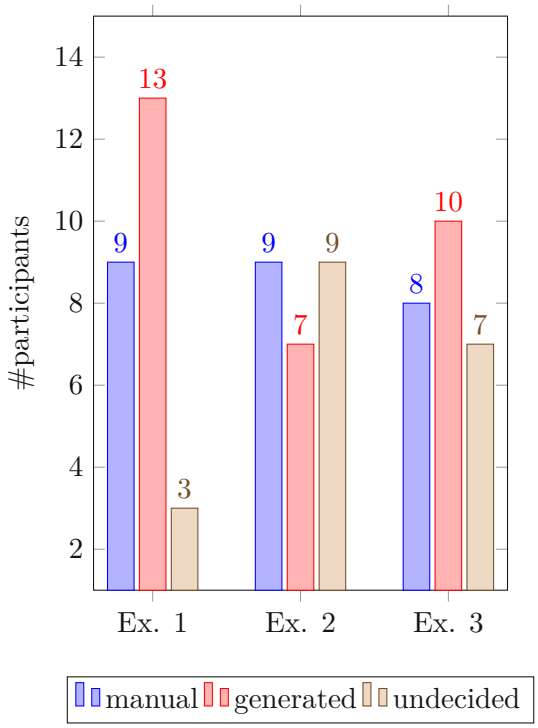

Figure 5.13: Survey results: Suitability for debugging by example

different from the structure of the manually written stories and the participants were able to recognise the structure of the stories they were most familiar. Figure 5.15 depicts the number of participants correctly identifying the manually written story.

5.4.2 Survey Result Interpretation

While the survey participants rated the manually written stories generally higher in two out of three examples, the results still indicate that stories generated with the presented approach are comprehensible and usable by BDD stakeholders. In particular, the generated story for Example 3 was preferred, rated as better readable, better suited for discussion and also better suitable for debugging. Still it was identified as being generated from a model by a vast majority of participants suggesting that generated stories can be at least as good as manually written ones.

A potential reason why users favoured the remaining two manually written example stories could be the familiarity with this style of story writing. The model-based stories (except for Example 3) were more compact and contained more test cases compared to their manually written counterpart. Some participants noted this alternative structure of the generated stories in the free form section as favourable as it leads to less technical stories and a better structure where one scenario forms one independent test case.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

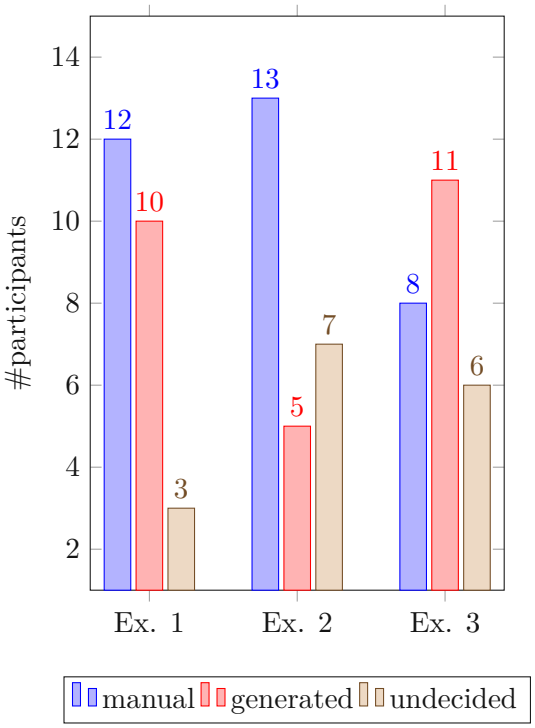

Figure 5.14: Survey results: Scalability by example

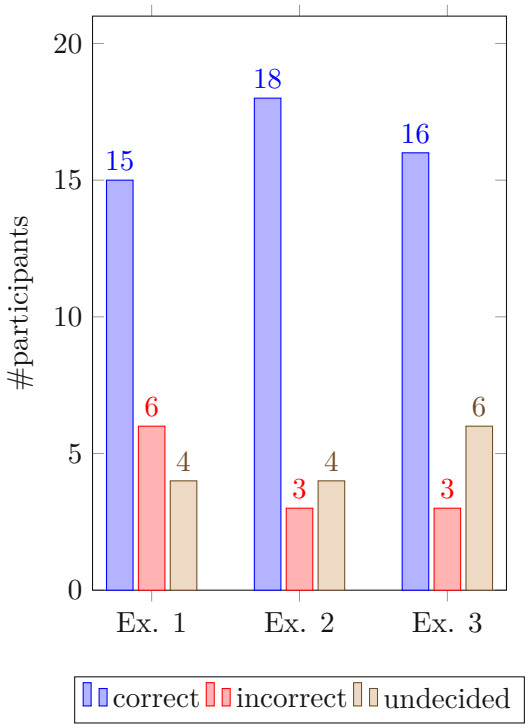

Figure 5.15: Survey results: Identification of model-based stories by example

5.5 Effort

To answer the final research question about the effort required to use the described MBT approach, an effort analysis based on editing steps was conducted. Two extensions to the VoIP gateway were analysed in terms of how many changes are needed to implement the desired tests for the changed behaviour either by manually adapting a BDD story, or by using models to regenerate the story automatically. Each action, such as line edits, adding, removing or changing nodes and edges in the model were counted as a single editing step. An effort analysis based on actual time spent would have been an interesting alternative, but would have required a number of participants tracking how much time they needed to complete a task in order to generate comparable results that are not skewed e.g. by previous knowledge. However, such an effort analysis was not planned in the scope of this thesis and is hence subject to future work.

In this effort analysis, two changes were analysed in particular:

Introduce tests for OPTIONS monitoring: The functionality of the OPTIONS monitoring feature has already been discussed in Section 5.3. The effort to implement the initial set of test cases was analysed. The tests for this feature were

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

introduced in a single commit consisting of the newly written story, the step methods and their implementation written in Java and Groovy code. As both the manually written story as well as the generated story are dependent on the steps implementation it was omitted from the further analysis.

Extend behaviour of call handling: This extension of the VoIP gateway concerns the basic call handling feature also described in Section 5.3. The new behaviour includes declining a ringing call, which requires extending the model from Figure 5.2 with new transitions from the `CALL_RINGING` state as shown in Figure 5.16. This feature is documented in [38, Section 21.6.4]. Unfortunately, even though the required steps to implement the tests for this feature already existed in the code base of the SUT test project, they were not yet used, thus the number of editing steps required to implement the tests manually had to be estimated and could not be taken from the VCS.

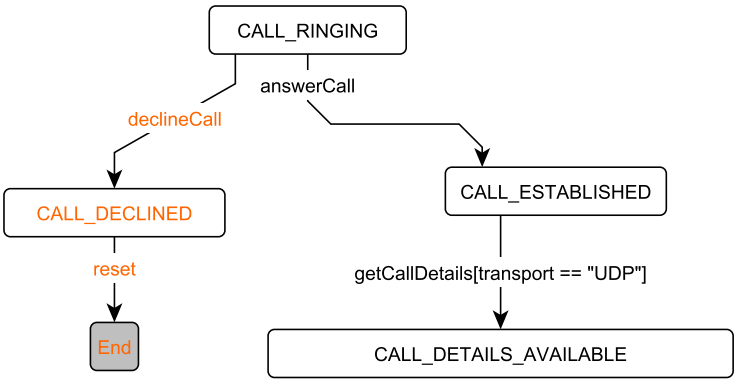

Figure 5.16: Extension of the model for the "basic call handling" feature of the VoIP gateway

During the analysis of the required editing steps, it quickly turned out that the used XML-based mapping description in combination with counting each changed line as one editing step drastically skewed the results. As described in Section 3.4 other description formats can be used that are less verbose but equally capable of describing the required mapping structure. For this reason, a more compact mapping domain specific language (DSL) was used for calculating the editing steps (see Listing 5.1).

5.5.1 Effort Comparison Results

Introduction of the OPTIONS Monitoring

The existing manual story for this feature consists of 37 lines and adds a single test scenario to the project exercising the 'Manual path' shown in Listing 5.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
<mapping>
    <step type="WHEN">caller answers OPTIONS request with OK</step>
    <modelElements>
        <modelElement>
            <id>respondOk</id>
            <scenarioInfo>OK response</scenarioInfo>
        </modelElement>
    </modelElements>
</mapping>
```

```
respondOk ("OK response") ==> WHEN caller answers OPTIONS request with OK
```

Listing 5.1: XML-based mapping of model elements to BDD steps vs. DSL-based mapping

The following editing steps are required to introduce the generation of similar stories via the model shown in Figure 5.4:

1. Create XML mapping for 10 new steps: +10 lines (or +81 lines using XML-based mapping)
2. Create setup composite steps: +10 lines
3. Create reset composite step: +3 lines
4. Create model: +5 nodes, +10 edges

These changes result in 38 editing using the modified mapping description DSL, which is almost identical to the 37 line edits that were required to manually implement the story. In contrast, using the XML-based mapping description would have required 109 editing steps. In addition, it should be noted that using composite steps for the setup and reset is optional but recommended as it reduces the number of nodes and edges needed in the model.

The main advantage of the model-based approach is the increased number of test scenarios that can be generated from a single model. In addition to the model path covered by the manually written story, the model-based variant covers many more paths as the examples in Listing 5.2 show. Due to structure of the model containing loops, an infinite number of different scenarios can be generated.

Extension of the Call Handling Behaviour

The second example concerns the effort for changing the behaviour of an existing feature by adding new states and transitions. There were no real examples for such an extension

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Manual path:
START => prepareForOptionsModelExecution => NO_MONITORING
=> startMonitoringCommandWithOkResponse  => MONITORING_ACTIVE_OK
=> respondNok => MONITORING_ACTIVE_NOK => resetMonitoring => End

Potential generated path 1:
START => prepareForOptionsModelExecution => NO_MONITORING
=> startMonitoringCommandWithOkResponse => MONITORING_ACTIVE_OK
=> resetMonitoring => End

Potential generated path 2:
START => prepareForOptionsModelExecution => NO_MONITORING
=> startMonitoringCommandWithOkResponse => MONITORING_ACTIVE_OK
=> startMonitoringCommand => MONITORING_ACTIVE_OK
=> resetMonitoring => End

Potential generated path 3:
START => prepareForOptionsModelExecution => NO_MONITORING
=> startMonitoringCommandWithOkResponse => MONITORING_ACTIVE_OK
=> stopMonitoringCommand => NO_MONITORING
=> startMonitoringCommandWithOkResponse => MONITORING_ACTIVE_OK
=> resetMonitoring => End
```

Listing 5.2: Example paths through the model in Figure 5.4

in the project's VCS as BDD tests were primarily developed after the implementation of a feature was finished. Hence, an example was invented that extends the call handling behaviour. This example was chosen because BDD steps and their implementation already existed in the test project's code base, however they were not yet used in any stories.

The existing story for outgoing calls consists of 31 lines. A copy of this story was created and extended in six further editing steps that implement the desired behaviour. These include modifying the test adapter to not automatically answer a call so that it stays in the "ringing" state, declining the call and verifying it has been declined, as well as removing steps verifying the call has been established. To handle both TCP and UDP two version of the story are required resulting in a total of 56 additional lines of code. The resulting manual story can be seen in Listing 5.3.

For the model-based approach, the existing basic call handling model (refer to Figure 5.2) can be extended to include a `declineCall` transition from the `CALL_RINGING` state into a new `CALL_DECLINED` state and a reset transition to the End state. The changed parts of the model are shown in Figure 5.16. A total of two edges and a single node need to be created in addition to two more mappings for the new transition `declineCall` and the new node `CALL_DECLINED`. As with the previous example using a mapping

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Meta: @AfterStory: setup\CleanMessageStore.story

Narrative:
As a websocket client of the SIP gateway
I want to make an outgoing call
So that I can talk with an external party

GivenStories: setup/b2bStart.story

Scenario:  VoIP Prerequisites
Given SipContacts group CALLER:
| key      | profile               | user-entity |
| Contact1 | cats-agent-b2b-voip 1 | 11111       |
And SipContacts group CALLEE:
| key      | profile               | user-entity |
| Contact1 | cats-agent-b2b-voip 1 | 22222       |

Given phone CALLER, CALLEE is created

Scenario:  Outgoing Phone Call
When the client wants to call a sip contact CALLEE using
cats-agent-b2b-websocket b2bua-adapter using transport UDP
Then CALLEE CallState is RINGING within 1000 ms
When CALLEE decline calls
Then CALLEE CallState is DECLINED within 1000 ms

Scenario:  Terminate the call
When CALLEE terminate calls
When CALLEE, CALLER is removed
```

Listing 5.3: Manually written BDD story for testing the declining of a call in the VoIP gateway

DSL amounts to just two additional editing steps totalling five editing steps.

The manual implementation required two stories each 28 lines long, whereas the model-based variant manages to reduce the effort to just five editing steps. It also offers the benefit that different scenarios including the `declineCall` transition can be automatically generated: One for the TCP and one for the UDP case. If the model gets extended in the future with further transitions and states before the decline branching point, these additions can also be tested in conjunction with the decline feature.

5.5.2 Effort Comparison Result Interpretation

The effort evaluation for this case study was rather limited in scope but confirmed that the described approach is capable of maintaining the benefits in effort savings that have

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

been highlighted in many previous and more comprehensive case studies [56]. By being able to simply regenerate test cases based on a model, changes in the behaviour of the SUT can be tested without effort, once the corresponding model is adapted, which is usually less effort than adapting a large test base. The automatic generation of different test scenarios from one model leads to a a wide variety of test cases which is hard to achieve when manually implementing tests.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 6

State of the Art

Many approaches to MBT are described in the literature, but very few resources have proposed generating BDD stories from graphical state machine models. This chapter aims to give an outline of the existing literature and describe the distinguishing features of this work in comparison to related literature. The first section, Section 6.1, summarises how the diverse research available on MBT can be categorised and what literature exists that summarises the different approaches and tools. The following Section, 6.2, highlights related literature that shows how previous authors have combined MBT and BDD in similar or completely different ways than presented in this work. Finally, an outline of the literature on MBT for keyword-driven testing (KDT) is presented in Section 6.3 and eventually, some ideas on how natural language processing (NLP) can be used in the context of this thesis are presented in Section 6.4.

6.1 Model-Based Testing

Neto et al. [33] provide a systematic overview of the MBT techniques described in the literature. The examined approaches varied in their testing level (e.g. system testing, integration testing, regression testing), the effort required to implement the approach, the notation used to describe the model (e.g. UML based) and many more. Of the 78 reviewed papers, they identified 27 different MBT approaches based on state machine models, similar to this work. The authors also emphasise that MBT adoption depends on a good tool support and a well known and established output format which, by using BDD stories as target language, this thesis tries to provide.

Utting, Pretschner, and Legeard [57] have proposed a more in-depth taxonomy of MBT approaches by characterising any MBT approach in six dimensions:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1. **Model scope.** A binary scale defining whether a model only specifies the inputs to the SUT or also includes the expected output.

2. **Model characteristics.** This dimension includes whether the MBT approach is deterministic or non-deterministic, addresses issues related to timing, which are relevant for real-time systems, and whether the approach focuses on event-discrete, continuous or hybrid systems.

3. **Model paradigm.** This dimension represents the notation used for modelling the system. Identified notations include pre/post notations, transition-based notations such as FSMs and UML statecharts, history-based notations e.g. message-sequence charts, functional notations, which describe a system in terms of mathematical functions, and many more.

4. **Test selection criteria.** Represents the available selection criteria or stop conditions offered by an MBT approach and includes strategies such as requirements coverage, structural model coverage, fault-based criteria and more.

5. **Test generation technology.** This dimension describes the approach's ability to select test cases based on the test model and the selection criteria. Possible strategies are highly dependent on the modelling paradigm. Examples are random generations, search-based algorithms, model-checking algorithms, constraint solving and more.

6. **Test execution.** This binary scale groups MBT approaches in online and offline approaches whereby some tools support both modes of operations.

By applying this taxonomy, current and future MBT approaches can be classified and compared. The authors showcase this for three MBT tools. For the MBT approach shown in this thesis a classification could look as follows: The *model scope* of this approach is an input-output model and its *characteristics* include an untimed and discrete model which does not model non-deterministic applications. In terms of *model paradigm*, the presented approach is based on graphical UML state machine models, which support conditions using a simple Javascript syntax, and each model maintains its internal state. The approach supports a number of *test selection criteria*, such as edge and node coverage criteria, reaching a target state or walking a model for a certain time period. From a *test generation technology* standpoint, the approach supports random walks as well as more sophisticated algorithms, such as Dijkstra's algorithms or A-star. Finally, in terms of *test execution*, both online and offline testing are supported.

Bernardino et al. [4] conducted a more recent systematic mapping study on the topic addressing the tools and literature on MBT from 2006 to 2016 for which they evaluated 1197 papers and analysed 87 primary studies. They identify 70 MBT tools and found that most use UML and FSM as modelling notation, which is also used in this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Their results also show that MBT is universally applicable from testing desktop applications, automotive and other safety-critical systems, embedded system, games, protocols, telecommunication applications to web applications and many others. The reviewed papers also show a good diversity in targeted testing levels. Notably, there was no study on unit testing and the majority of papers focused on system-level testing. They conclude that "MBT is a broad and alive research field" and that due to the variety of models and tools available, choosing the right approach is challenging, but that certain features are likely to be present in any viable tool. These include support for an input model that describes the SUT (as examples they list UML and FSM) and the option to use some test generation technique (their example is a random generation), all of which is supported by the testing approach outlined in this work.

6.2 Combining Model-Based Testing and Behaviour-Driven Development

The following section describes related works that present similar ideas to the presented approach in this thesis of combining MBT and BDD. The references have been grouped by their used modelling language, starting in Section 6.2.1 with state-based models including UML state machines, followed by Section 6.2.2 describing approaches based on business process modeling (BPM). Finally, Section 6.2.3 presents some related works that also present the idea of combining MBT and BDD but do not present a specific modelling language.

6.2.1 State-Based Models

Martin [30] discovered already in 2008 that there is a tight relationship between the GWT structure used in BDD stories and FSMs, which represent the basic concept of input-output processing. He showed that a scenario consisting of GWT steps can easily be transformed into a state transition table that models the current state of a system in relation to incoming events and resulting target states and vice versa. In this way, BDD stories can be understood as a way of describing FSMs, which offers the possibility to use well established graph theories to e.g. calculate the number of paths through a system given the number of states and events are known. Using this information, one can prove that all possible combinations have been described and thus that a specification is complete. The state transition table illustrated by Martin can also be interpreted in the context of MBT, e.g. the list of start and end states can be interpreted as nodes in a state machine model which are connected by the actions indicated in such a table. Martin only provides a concept and the idea of using the similarities of these approaches to create tools that support practitioners in finding missing test cases using FSMs, but in contrast to this thesis does not provide any guidance of how such a tool could look like.

Li, Escalona, and Kamal [28] describe an approach to combine MBT and BDD very similar to what is outlined in this thesis. Their tool *skyfire* uses behavioural UML

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

models to generate BDD stories for the Cucumber framework. The model processing and translation into abstract test cases is based on their earlier work in which they describe the foundation of an MBT framework that can turn UML state machines into executable Java code [29]. The conversion from UML to BDD stories is done in multiple stages: First, the UML state machine is transformed into a graph. In this stage, more advanced UML features, such as composite states, choice, fork or junction, need to be handled and transformed to nodes and edges. In the second stage, abstract test cases are derived using various graph coverage criteria, such as node coverage, edge coverage, edge-pair coverage and others. Finally, BDD stories are created where each abstract test is converted to a scenario. Rules are employed to decide the step type, e.g. the first transition always uses the `Given` keyword, whereas state invariants use `Then` steps. The names of the transitions and constraints are directly used as the BDD step and scenario title, which leads to somewhat cryptic BDD stories as the example in Listing 6.1 illustrates.

In addition to describing the *skyfire* framework, Li, Escalona, and Kamal also conducted a case study in which they found that their test approach was helpful to develop more effective tests. They describe that using an MBT approach leads to longer scenarios which they argue better simulates the users' behaviours.

```
Scenario:  initializeWithValidKeys startEmr emrCreationSuccess
addValidStep terminateEmr emrTerminationSuccess
Given: initializeWithValidKeys
When: startEmr
Then emrCreationSuccess
When: addValidSteps
And: terminateEmr
Then: emrTerminationSuccess
```

Listing 6.1: Example of a skyfire story [28]

The MBT approach used in the *skyfire* tool is very sophisticated and offers many features, mainly due to the authors' effort of integrating almost all available UML state machine concepts. By providing more sophisticated algorithms for generating test cases, they are able to minimise the number of steps while maintaining high coverage. However, the resulting BDD stories are not in the form of readable sentences in contrast to the examples outlined in this thesis. This is true for both the scenario titles and the actual steps as they only consist of identifiers used in the model. While the authors argue that this is beneficial as it reduces the likeliness of misreading a lengthy sentence, it is not aligned with the intentions of BDD and likely diminishes the potential for non-technical readers to understand the scenarios. In some ways, the resulting test stories are closer to KDT (see Section 6.3) than to BDD. Unlike the approach outlined in this thesis, the BDD steps were explicitly developed for the generated scenarios, and no pre-existing steps were used.

Sivanandan and Yogeesha [45] illustrate a similar approach of combining MBT with

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

BDD. They show how the open source MBT tool Graphwalker can be used to generate test keywords which can be translated into BDD steps. The generated test keywords directly correlate with the names assigned to vertices and edges in the test model. While their results appear to reflect the idea proposed by this thesis—plain text readable BDD stories generated on the fly based on a behavioural model—it is not possible to reproduce their work as key steps are not documented. While they explain in detail the production of keywords using Graphwalker as MBT engine (which is very similar to what this thesis describes), their paper lacks documentation on how these keywords are mapped to BDD steps and no evaluation of their approach has been conducted.

Entin et al. [16] have shown how BDD stories can be used to define usage models in an automated fashion. Usage models in this case are describing how a software is used and are notated in terms of simplified state machine models. The rationale for this was to reduce the effort required to create these usage models and allow the entire development team to partake in creating them. For this to work, BDD scenarios are assigned a unique ID, and `Given` steps can reference other scenarios using this ID. Paths are generated based on the logical succession of scenarios which is constructed from the `Given` steps. `When` steps are interpreted as transitions and the `Then` step finally links to the SUT's target state. The generated usage models can then be executed using a custom MBT framework based on a capture and replay approach. As of now, their approach has not been further tested outside of a single development team.

6.2.2 Business Process Models

Carvalho, Manhães, and Carvalho e Silva [11] propose to substitute textual scenario descriptions with state machine models commonly found in BPM. As such, users should have the option to use graphical representations of business processes or textual descriptions using a GWT syntax. They also suggest that other notations, such as UML activity diagrams and even Petri nets, could be written in a GWT syntax. This proposal is further elaborated in [9] by showing how constructs from UML state machine models can be presented using the GWT syntax. As an example, the parallel split pattern shown in Figure 6.1 can be rewritten in the GWT syntax as shown in Listing 6.2. They recognise that some constructs involving parallelism are not available in BDD and might have to be implemented in the BDD mapping code directly. Their work does not outline any approach on how to automatically transform models into BDD stories and instead focuses on the formalisation of various UML patterns in GWT syntax.

In a later work, Carvalho, Carvalho e Silva, and Manhães [10] show how the concept of translating BPM to BDD can be applied to Petri nets in a token game. A user interactively walks through a BPM notated as a Petri net and decides which path to take. The selected path elements are translated into the GWT syntax and can then be executed to simulate an actual user working with the system. This process is different from what is outlined in this thesis as it focuses on Petri nets and involves a user making the decisions on how the model is traversed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

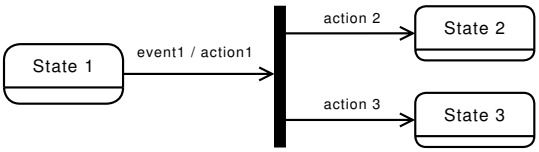

Figure 6.1: Parallel split pattern in UML state machines

```
Given State 1
When Event 1
Then Action 1
And Action 2
And Action 3
```

Listing 6.2: Parallel split pattern in state charts translated into GWT according to [9]

6.2.3 Other Approaches Combining MBT and BDD

The general idea of combining MBT using state machines and BDD is also described by Narváez [32]. However, the focus is not on being able to generate BDD test cases from models but instead on showing where these testing techniques excel and how it is possible to use both of them effectively in a single project and even in a single test case. According to the author, the lengthy setup that is often required to get the SUT into a starting state can be abstracted in a single BDD step whereas it would take many vertices and edges to represent this properly in a state machine model. On the other hand, BDD steps can lead to the impression that actions are atomic and no alternative paths for achieving the same result exists whereas state machine can intuitively model distinct paths that can be taken in a SUT to achieve the same end result.

Interesting research questions on the topic of MBT in combination with BDD have also been discussed by Eberhardinger et al. [15], for example which tools are already available to use MBT in conjunction with BDD and whether adding more testing tools can reduce costs or will increase them due to additional requirements for developers and testers. While they agree with the premise of this thesis that BDD stories can be devised automatically from models, they also propose validating a model through BDD scenarios. Lastly, they pose the question of whether it is also possible to synthesise a model from existing BDD stories.

6.3 Relating Model-Based Testing with Keyword-Driven Development

Keyword-driven testing (KDT), as outlined in [42, 58], is in many ways very similar to BDD. It revolves around a defined set of keywords that are linked to test adapter procedures. The combination of a list of keywords and their parameters form a test case which can be executed automatically by resolving each keyword to the test procedure. As the approaches are so similar, some frameworks, such as the Robot Framework[1], which are primarily designed for KDT also support BDD. From a technical implementation perspective, the syntax and the execution workflow are very similar. As such, it is

[1]Robot Frameworkhttps://robotframework.org

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

reasonable to assume that any KDT test case can be transformed into a BDD story given a mapping from keywords and their parameters to BDD steps. Following this idea, it makes sense to investigate literature on relations between MBT and KDT. The result of this investigation is presented in the following.

TEMA[2] is a test automation approach that uses domain specific state machine models to generate test keywords used in KDT [24]. The labels used to describe the edges and nodes in the model are the same that are used as the keywords that are then picked up by the KDT tool and mapped to executable actions. Its model design includes support for parallelism as well as a notion of an expected error. Pajunen, Takala, and Katara [34] evaluate how this MBT approach can be integrated with existing KDT frameworks in order to profit from the rich library of low-level keywords provided, e.g. by the Robot Framework. In terms of test case generation, different strategies are supported, such as a random walk through the model for smoke tests or a bug hunting test that employs algorithms and heuristics to walk through the model. Similarly to the implementation of the online approach shown in this thesis (see Section 4.5.2) the Robot Framework had to be extended to support on-the-fly execution of keywords. Using the KDT paradigm instead of BDD steps removes the need for finding a natural language scenario title and also does not require test steps to be grouped into preconditions, actions and verifications.

6.4 Natural Language Processing

At its core, the problems stated in this thesis revolve around transforming formal models into natural language, which is then linked to executable code. While this thesis describes a systematic manual approach to establish the connections between formal and informal descriptions, the field of natural language processing (NLP) provides means of using statistical methods and neural networks to work with natural language descriptions. In the following, related work on how NLP could be used in the area of testing using BDD and MBT is presented.

Kamalakar, Edwards, and Dao [26] focus on eliminating the glue code required to link textual BDD steps to a test adapter, which performs the desired action in the SUT. They developed a prototype nicknamed *Kerby* that uses NLP as well as code inspection from the SUT to find potential classes and actions referred to by BDD steps. Using a probabilistic matching algorithm, JUnit code can be generated where each JUnit test implements a single BDD scenario. Using this approach allows developers to iterate between working on the actual application and writing the specification in natural language using BDD and have *Kerby* generate the entire test code automatically. The authors note that their approach is still a prototype and will need more refinement to improve the accuracy of the NLP while maintaining performance which suffers when using more sophisticated algorithms and dictionaries.

[2]TEMA: `http://tema.cs.tut.fi`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Soeken, Wille, and Drechsler [46] propose another approach using NLP wherein the implementation of BDD steps is done semi-automatically. The NLP system uses BDD stories as input and tries to identify actors and artefacts, which it then uses to build a class diagram modelling the structure of the SUT as well as a sequence diagram representing the behaviour that is being tested. This information can be used to generate large parts of the implementation of the BDD story. In case NLP cannot identify a sentence correctly, the user is asked for further hints which are again stored in natural language in the background/narrative part of the BDD story. The system is able to generate many code skeletons from the synthesised class and sequence diagram by parsing BDD stories. These can then need to be linked to a suitable test adapter that drives the interactions with the SUT.

An interesting concept is the idea of synthesising the natural language specification of a formal model written in a notation like UML. Such an approach might be suitable for generating BDD stories based on the formal model-based specification of the target system, which often already exists in the form of class diagrams or state machines. Meziane, Athanasakis, and Ananiadou [31] propose such a transformation for UML class diagrams with the intent of finding errors that have been introduced in the common process of transforming the initial natural language specification into a model. By allowing the automatic reversal of the process, users that are not familiar with formal model notations can understand the system more easily. Class diagrams, however, are not useful for describing the behaviour of a system and as such cannot be used as the only source for generating a natural language system specification. The authors recognise that more research is required in this area, even though natural language generation for other formal specification languages, such as OCL, has also been investigated [8].

Brosch and Randak [7] describe a related approach of transforming formal UML class diagrams into natural language by using sentence templates combined with a traversal strategy to iterate through the class diagram. Their approach does not target software development or testing but teaching. Students are frequently asked to translate a natural language description of a system into a class diagram. Having a tool that can generate textual description relieves teachers of this tedious tasks and students get consistent natural language descriptions that they can then reverse engineer. According to the authors, most class diagram descriptions use recurring phrases, which can be parametrised to fit any class diagram model. It would be interesting to see if their approach could be applied to UML state machine models as well, but given that state machine models on their own do not offer a lot of syntax elements compared to class diagrams (like class, attribute, generalisation, association, multiplicity,...), it might be much more difficult and result in a description that is not really fluent to read.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 7

Conclusion and Outlook

7.1 Summary

This thesis presented a testing approach that uses graphical UML state machine models and test case generation techniques commonly found in the realm of MBT to facilitate the automated generation of BDD system specifications in the GWT syntax. The goal was not to diminish the ideas and workflows employed in BDD software projects, which focus on bringing the customer, developers and testers together to write down a common understanding of the behaviour of a system. Instead, the idea was to provide a structured alternative for providing tests of parts of a system that are exceedingly complex and hard to accurately describe in a GWT syntax by hand. To evaluate the novel testing approach in practice, a prototype was implemented and assessed in a case study. The results of this study look promising and give ideas for future improvements and applications of this technique.

The developed testing approach and its prototypical implementation use UML state machine models that specify not only the test inputs for the SUT but also the expected outcome. Furthermore, it offers various test selection criteria, and new ones can easily be implemented as the prototype is based on existing open source tools. By supporting both an offline as well as an online test execution mode, it is well equipped for all kinds of test scenarios and with further extensions it could even be extended to support non-deterministic applications. By using an independent mapping structure from identifiers used in the model to existing BDD steps it can (in the offline mode) make use of any number of existing BDD frameworks and is not limited to a specific framework.

The conducted case study served to answer the research questions which were outlined in Section 1.2. Not only did the study show that the automatically created tests could reach the same test coverage as comparable manually written ones, it also showed that

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

this could be done with an adequate effort. Assuming that BDD test steps already exist for the targeted SUT, the effort (measured in editing steps) to create models and link them to BDD steps was about the same as writing the tests manually. The method really shines when the behaviour of the SUT changes as it allows to regenerate test cases from the abstract model instead of having to adapt all existing test cases manually.

A large part of the evaluation was devoted to a survey conducted among experienced BDD practitioners. The survey had participants rate and judge examples of manually written and automatically generated BDD stories in a blind comparison to find out whether generated stories were useful and understandable. The outcome was mostly as expected: Practitioners preferred the manually written stories in two out of three cases and the majority was able to differentiate between them. However, at least one automatically generated example was preferred over its manually written counterpart suggesting that automatically generated stories that make use of existing test steps can be just as readable as manually written ones.

7.2 Future Work

The following are ideas for extending and improving the work presented in this thesis. They emerged during the prototype implementation and evaluation phases.

Large scale case study. To further test whether a testing approach as the one outlined in this thesis is viable, it needs to be evaluated in a more extensive case study. In hindsight, several aspects of the case study could be improved. One issue in the performed questionnaire was the limited information as to why participants preferred one example over another. Participants did not have to provide a reason for their decision and the free form comment field was hardly filled out. To improve this, a reason catalogue could be developed that provides several options that participants have to choose from to justify their rating. Performing structured interviews would be another alternative.

In terms of the effort analysis, it would be interesting to gather quantitative data in terms of hours of work needed to write and maintain test cases instead of just comparing editing steps as done for this thesis. This would require a larger user group to utilise models to generate BDD test stories and track the required effort.

Advanced test case generation strategies. The available test case generation strategies provided by the used MBT tool Graphwalker are rather limited. If guards are used in the model, the random walk strategy is essentially the only viable option as other strategies that would minimise the number of required steps to reach a coverage goal are either unsupported or still in development. Graphwalker does have an active community, so further contributions to this open source project will likely improve the situation. Advances made in this area should directly be usable with the presented testing technique.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Support for nested state machines. Graphical UML state machine models can provide a quick and easily understandable overview of the behaviour of a system as long as the number of states and transitions is kept reasonably low. As the diagram grows, it needs to be split either into various independent models or further abstraction layers need to be introduced, e.g. by allowing nested state machines. UML already supports this in the form of composite states, also known as substates. The interesting question with regards to the proposed approach of this thesis is how composite states should be mapped to BDD stories. A simple naive approach would be to map each substate to a BDD step and include it into the scenario. However, this would quickly make scenarios too long and less readable. Thus, a more refined approach is required that hides the decomposition in the BDD story. A potential solution would be to use BDD composite steps which represent a similar concept. Dynamically generated composite steps would require a name to identify the group of nested BDD steps. Potentially a similar solution as used for the generation of the scenario titles could be used.

Providing data in the form of example tables. One mechanism provided by BDD is the usage of example tables for running a scenario repeatedly using different test data sets. This is especially important for data-centric test cases as it avoids repeating the same steps over and over again. Generally, when using MBT, repetitively generating the same test case with different data is not an issue. However, if the goal is to produce human-readable test specification, it becomes more important to avoid repetitive structures. As such, it would be interesting to extend the current approach with support for data-centric test cases. A solution must consider two problems: How can the data be integrated into a state machine diagram and how can the test generation strategy and the mapping process recognise the need to generate a BDD examples table instead of individual test cases?

Natural language scenario and step naming. The current approach to constructing scenario titles is based on concatenating the scenario information provided by each step mapping. This solution can produce great results but requires some fine-tuning when designing the step mapping. With larger models (and more step mappings) this becomes increasingly difficult. A process that produces natural language could be used instead. One idea would be to use NLP to extract a summary from the generated BDD story, which could then be used as the scenario title. NLP could also be an interesting technique to define new steps if no mapping exists. Such a technique would be helpful if there are no existing test steps that are being reused. Instead, the model could provide enough information to synthesise a step definition entirely. Researchers have already shown how class diagrams can be converted into natural language specifications [31]. By combining the natural language with behavioural information from a state machine diagram it could be feasible to synthesise a textual description of the SUT behaviour in GWT syntax.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX A

Running Example

The example model for the following mapping (Listing A.1) and generated example story (Listing A.2) are based on the model shown in Figure 3.2.

```
<mapping>
  <step type="GIVEN">SUT is started and the test adapter is ready</step>
  <modelElements>
    <modelElement>
      <id>startSUT</id>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="GIVEN">the SIP application is running</step>
  <modelElements>
    <modelElement>
      <id>SUT_STARTED</id>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="WHEN">the SUT sends an INVITE using transport $transport</step>
  <modelElements>
    <modelElement>
      <id>sendInviteTCP</id>
      <scenarioInfo>Outgoing TCP call</scenarioInfo>
      <data key="transport">TCP</data>
    </modelElement>
    <modelElement>
      <id>sendInviteUDP</id>
      <scenarioInfo>Outgoing UDP call</scenarioInfo>
      <data key="transport">UDP</data>
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="THEN">$endpoint CallState is $callState within 500 ms</step>
  <modelElements>
    <modelElement>
      <id>ESTABLISHED</id>
      <data key="endpoint">CALLEE</data>
      <data key="callState">ESTABLISHED</data>
    </modelElement>
    <modelElement>
      <id>TRYING_RINGING</id>
      <data key="endpoint">CALLEE</data>
      <data key="callState">TRYING or RINGING</data>
    </modelElement>
    <modelElement>
      <id>TERMINATED</id>
      <data key="endpoint">CALLEE</data>
      <data key="callState">DISCONNECTED</data>
    </modelElement>
    <modelElement>
      <id>ON_HOLD</id>
      <data key="endpoint">CALLEE</data>
      <data key="callState">ON_HOLD</data>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="WHEN">$endpoints answers incoming calls</step>
  <modelElements>
    <modelElement>
      <id>awaitOK</id>
      <data key="endpoints">CALLEE</data>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="WHEN">the SUT sends a $size SIP MESSAGE</step>
  <modelElements>
    <modelElement>
      <id>sendLongMessage</id>
      <scenarioInfo>– long message —</scenarioInfo>
      <data key="size">large</data>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
  <step type="WHEN">the SUT ends the call</step>
  <modelElements>
    <modelElement>
      <id>sendBye</id>
      <scenarioInfo>and sends BYE</scenarioInfo>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="WHEN">VoIP $endpoint get terminated</step>
  <modelElements>
    <modelElement>
      <id>awaitBye</id>
      <scenarioInfo>and receives BYE</scenarioInfo>
      <data key="endpoint">CALLEE</data>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="WHEN">SUT send SIP RE-INVITE to hold the call</step>
  <modelElements>
    <modelElement>
      <id>holdCall</id>
      <scenarioInfo>is held</scenarioInfo>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="WHEN">SUT sends SIP RE-INVITE to release the call from
      ↪ hold</step>
  <modelElements>
    <modelElement>
      <id>unholdCall</id>
      <scenarioInfo>then released from hold</scenarioInfo>
    </modelElement>
  </modelElements>
</mapping>
<mapping>
  <step type="GIVEN">calls are ended and message stores are cleared</step>
  <modelElements>
    <modelElement>
      <id>reset</id>
    </modelElement>
  </modelElements>
</mapping>
```

Listing A.1: Complete XML-based mapping of the model elements given in Figure 3.2 to BDD steps

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Scenario:  #1 Outgoing UDP call and sends BYE
Given SUT is started and the test adapter is ready
And the SIP application is running
When the SUT sends an INVITE using transport UDP
Then CALLEE CallState is TRYING or RINGING within 500 ms
When CALLEE answers incoming calls
Then CALLEE CallState is ESTABLISHED within 500 ms
When the SUT ends the call
Then CALLEE CallState is DISCONNECTED within 500 ms

Scenario:  #2 Outgoing TCP call and receives BYE
Given calls are ended and message stores are cleared
And the SIP application is running
When the SUT sends an INVITE using transport TCP
Then CALLEE CallState is TRYING or RINGING within500 ms
When CALLEE answers incoming calls
Then CALLEE CallState is ESTABLISHED within 500 ms
When VoIP CALLEE get terminated
Then CALLEE CallState is DISCONNECTED within 500 ms

Scenario:  #3 Outgoing UDP call is held
Given calls are ended and message stores are cleared
And the SIP application is running
When the SUT sends an INVITE using transport UDP
Then CALLEE CallState is TRYING or RINGING within 500 ms
When CALLEE answers incoming calls
Then CALLEE CallState is ESTABLISHED within 500 ms
When SUT send SIP RE–INVITE to hold the call
Then CALLEE CallState is ON_HOLD within 500 ms
```

Listing A.2: Example of generated BDD story targeting 100% vertex coverage for the model given in Figure 3.2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX B

Survey Data

B.1 General Questions

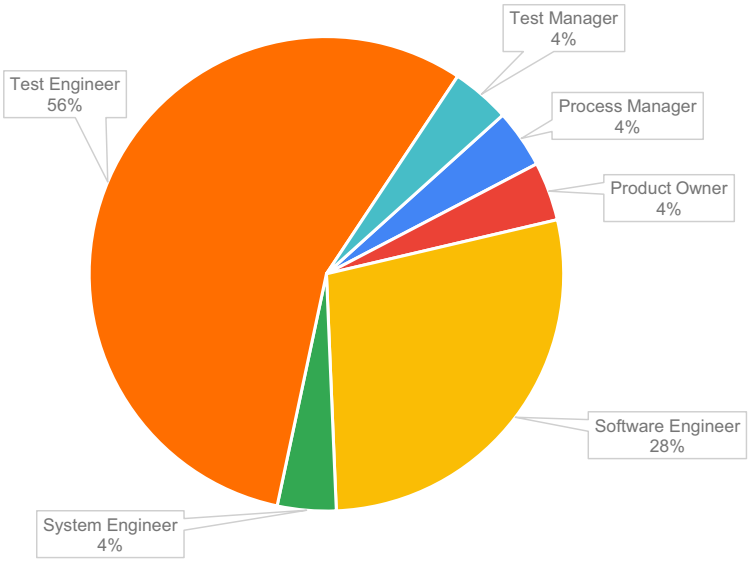

Figure B.1: Survey results: Professional background

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

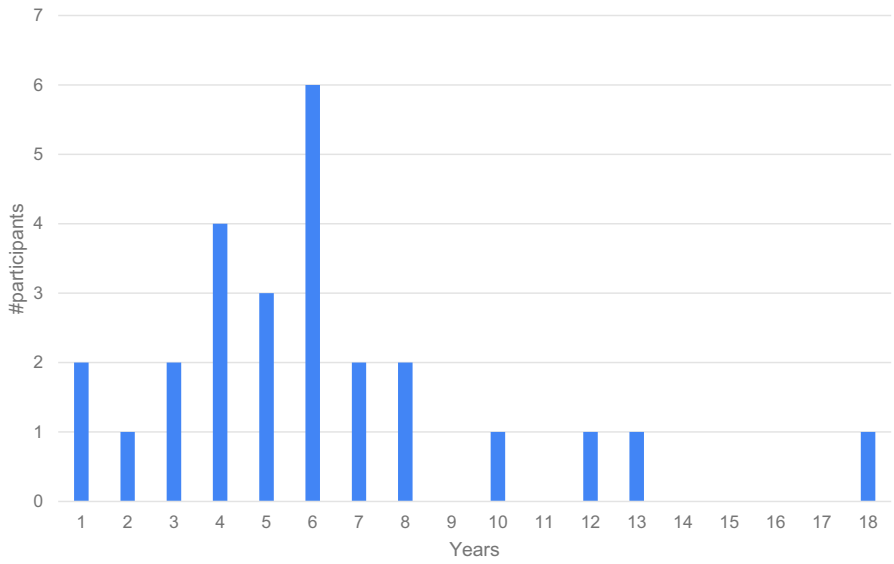

Figure B.2: Survey results: How many years of experience do you have in your field?

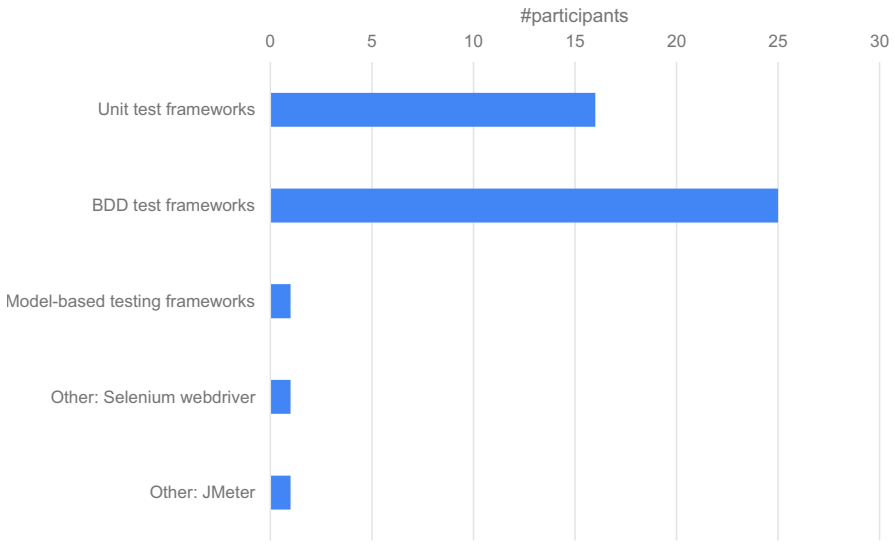

Figure B.3: Survey results: Which of these testing tools have you used in the past year?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

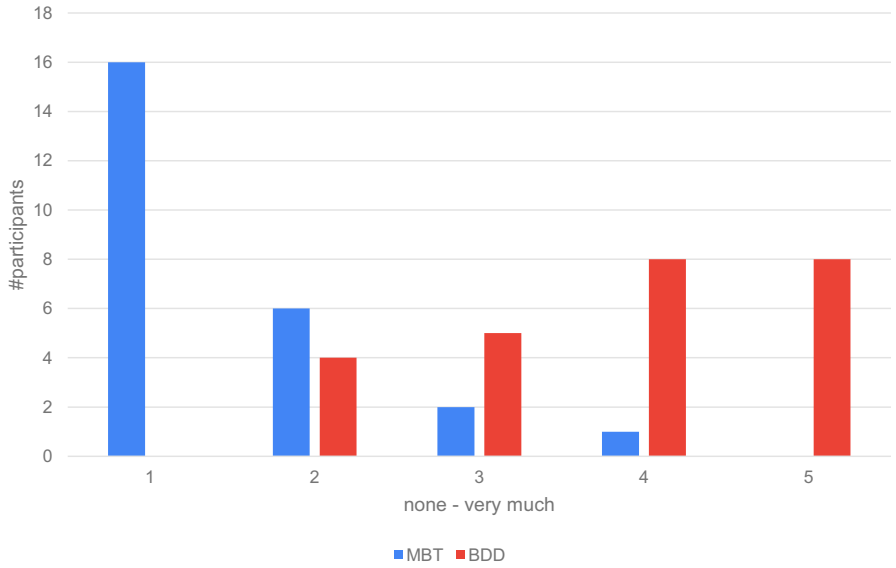

Figure B.4: Survey results: How much experience do you have with MBT/BDD?

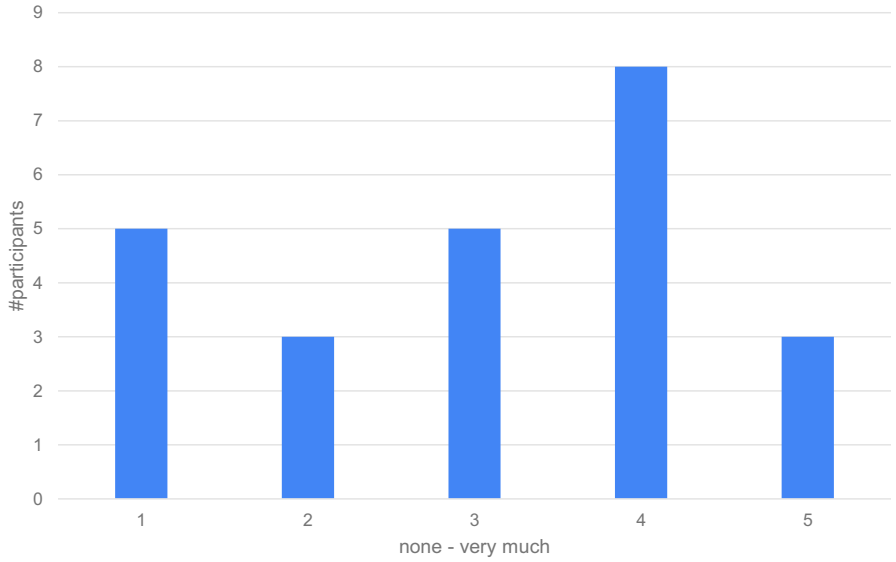

Figure B.5: Survey results: How much experience do you have with VoIP?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

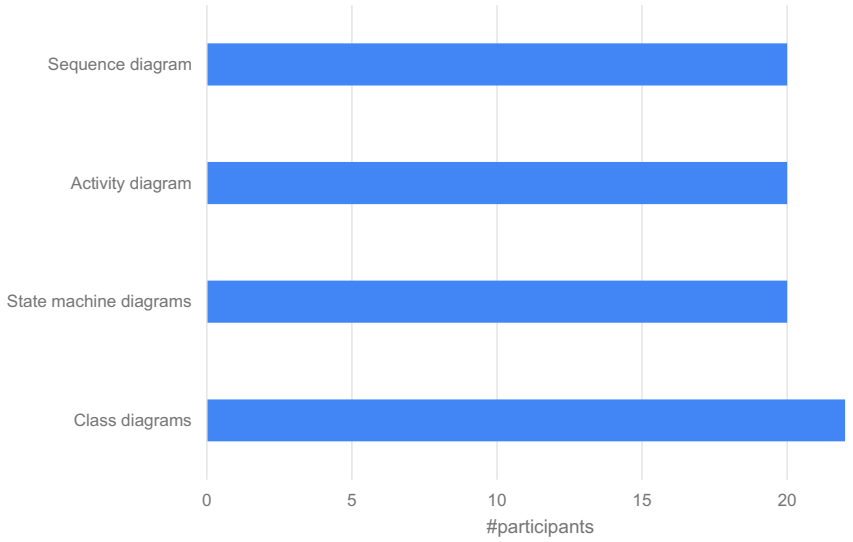

Figure B.6: Survey results: Which of these diagrams are you familiar with?

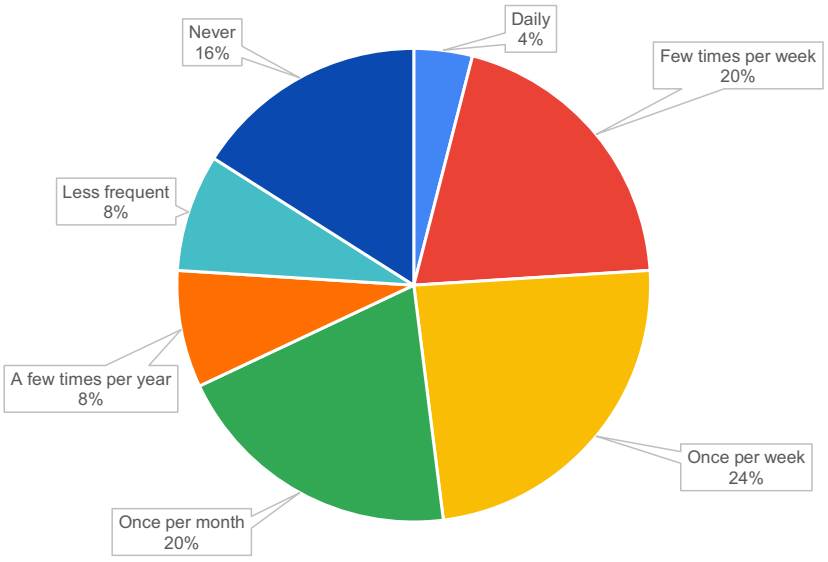

Figure B.7: Survey results: During development/testing how often do you use any kind of models (mental models, sketches, tool supported modeling,...)?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

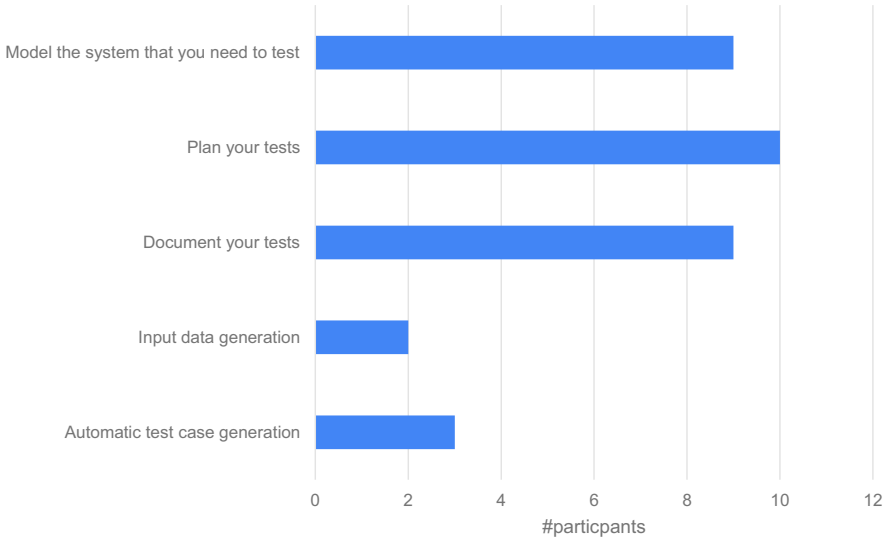

Figure B.8: Survey results: In case you are using models for testing already, for which purposes are you using them?

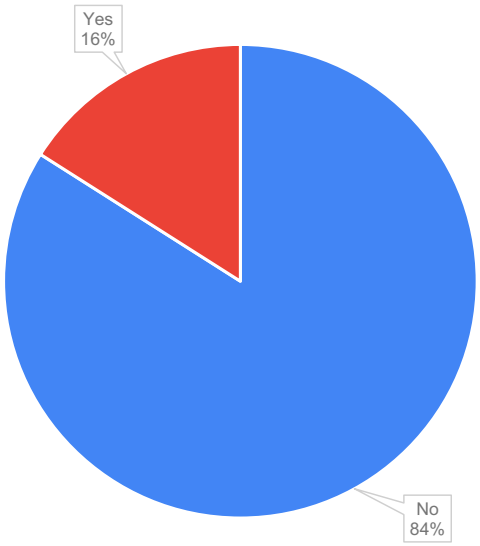

Figure B.9: Survey results: Do you use a modeling tool for creating your models?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

B.2 Comparison Questions and Detailed Perception Questions

All the tables using a 5-point Likert scale in this section use the following mappings: 1) Strongly agree, 2) Agree, 3) Undecided, 4) Disagree, 5) Strongly disagree.

B.2.1 Survey results: Example 1 - SIP Call Establishment

In this example, the story labelled "A)" was manually written and the story labelled "B)" was generated using a model.

	A	B	Undec.
Preferred version	16	8	1
This story is better readable	16	8	1
This story is better suited for discussion with co-workers	12	9	4
This story is easier to debug in case an error occurs	9	13	3
This story is more scalable (in case of writing many tests)	12	10	3
This story has a better structure	16	6	3
This story was manually written	15	6	4

Table B.1: Survey results: Example 1 - Comparison questions

	1		2		3		4		5	
	A	B	A	B	A	B	A	B	A	B
Scenario titles helpful to understand test	9	6	12	11	2	6	1	2	0	0
Scenarios were short and concise	9	0	11	9	2	6	2	8	0	2
Structure of the story was good	5	1	12	9	2	7	5	8	1	0
Accept this story in own test project	5	3	15	6	1	12	3	4	1	0

Table B.2: Survey results: Example 1 - Detailed perception questions

B.2.2 Example 2 - Hold Handling

In this example, the story labelled "A)" was manually written and the story labelled "B)" was generated using a model.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

	A	B	Undec.
Preferred version	19	2	4
This story is better readable	19	2	4
This story is better suited for discussion with co-workers	17	3	5
This story is easier to debug in case an error occurs	9	7	9
This story is more scalable (in case of writing many tests)	13	5	7
This story has a better structure	14	2	9
This story was manually written	18	3	4

Table B.3: Survey results: Example 2 - Hold handling

	1		2		3		4		5	
	A	B	A	B	A	B	A	B	A	B
Scenario titles helpful to understand test	11	3	7	6	3	5	4	10	0	1
Scenarios were short and concise	5	0	14	5	3	3	3	16	0	1
Structure of the story was good	4	0	12	7	8	10	1	8	0	0
Accept this story in own test project	8	1	10	7	6	10	1	6	0	1

Table B.4: Survey results: Example 2 - Detailed perception questions

B.2.3 Example 3 - Proxy Messaging

In this example, the story labelled “A)” was generated using a model and the story labelled “B)” was manually written.

	A	B	Undec.
Preferred version	14	8	3
This story is better readable	15	7	3
This story is better suited for discussion with co-workers	14	6	5
This story is easier to debug in case an error occurs	10	8	7
This story is more scalable (in case of writing many tests)	11	8	6
This story has a better structure	13	7	5
This story was manually written	3	16	6

Table B.5: Survey results: Example 3 - Proxy Messaging

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

	1		2		3		4		5	
	A	B	A	B	A	B	A	B	A	B
Scenario titles helpful to understand test	2	6	7	8	7	6	8	4	1	1
Scenarios were short and concise	1	1	14	14	5	3	5	6	0	1
Structure of the story was good	1	0	14	11	4	7	6	4	0	3
Accept this story in own test project	2	0	12	13	8	8	3	4	0	0

Table B.6: Survey results: Example 3 - Detailed perception questions

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

List of Listings

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Acronyms

API application programming interface. 18, 33, 37

ATDD acceptance test-driven development. 6

BDD behaviour-driven development. 1–11, 15, 21, 22, 24, 25, 27–31, 33, 37, 41, 42, 44, 45, 50, 51, 53, 60, 61, 64, 67, 69–77, 93

BDT behaviour-driven testing. 10

BPM business process modeling. 69, 71

DSL domain specific language. 62, 63, 65

FSM finite-state machine. 11, 13–15, 22, 25, 28, 30, 68, 69

GWT Given/When/Then. 1, 27, 69, 71, 72, 75, 77, 94

KDT keyword-driven testing. 4, 67, 70, 72, 73

MBT model-based testing. 1, 2, 4, 5, 10–19, 22, 24, 25, 33, 39, 41, 43–45, 50, 51, 53, 58, 61, 67–73, 75–77, 91

MDE model-driven engineering. 1

NLP natural language processing. 67, 73, 74, 77

OCL object constraint language. 15, 17, 21, 74

SIP session initiation protocol. 2, 22, 24, 41–44, 51

SUT system under test. 1–3, 8, 12–14, 17–19, 21, 22, 24, 25, 27–29, 31, 41–45, 62, 66, 68, 69, 71–77

TDD test-driven development. 5, 6

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

UML unified modeling language. 2–4, 12, 15, 17, 21, 22, 25, 29, 33, 57, 67–71, 74, 75, 77

VCS version control system. 18, 31, 62, 64

VoIP voice over IP. 2–4, 41–43, 51, 53, 61, 62, 91

XML extensible markup language. 27, 62, 63

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] Gojko Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications, 2011. ISBN: 9781617290084.

[2] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002. ISBN: 9780321146533.

[3] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990. ISBN: 9780442206727.

[4] Maicon Bernardino, Elder M. Rodrigues, Avelino F. Zorzo, and Luciano Marchezan. "Systematic mapping study on MBT: tools and models". In: *IET Software* 11.4 (2017), pp. 141–155. DOI: 10.1049/iet-sen.2015.0154.

[5] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 1999. ISBN: 0201809389.

[6] Robert Binder, Bruno Legeard, and Anne Kramer. "Model-based Testing: Where Does It Stand?" In: *Communications of the ACM* 58.2 (Jan. 2015), pp. 52–56. DOI: 10.1145/2697399.

[7] Petra Brosch and Andrea Randak. "Position Paper: m2n - A Tool for Translating Models to Natural Language Descriptions". In: *Electronic Communications of the EASST* 34 (2010). DOI: 10.14279/tuj.eceasst.34.593. URL: https://doi.org/10.14279/tuj.eceasst.34.593.

[8] David A. Burke and Kristofer Johannisson. "Translating Formal Software Specifications to Natural Language". In: *Proceedings of the 5th International Conference on Logical Aspects of Computational Linguistics, LACL 2005*. Vol. 3492. Lecture Notes in Computer Science. Springer, 2005, pp. 51–66. DOI: 10.1007/11422532_4.

[9] Rogerio Carvalho, Fernando Luiz Carvalho e Silva, and Rodrigo Soares Manhães. "Mapping Business Process Modeling constructs to Behavior Driven Development Ubiquitous Language". In: *CoRR* abs/1006.4892 (2010). arXiv: 1006.4892. URL: http://arxiv.org/abs/1006.4892.

[10] Rogerio Carvalho, Fernando Luiz Carvalho e Silva, and Rodrigo Soares Manhães. "Business Language Driven Development: Joining Business Process Models to Automated Tests". In: *Advances in Enterprise Information Systems II* (Jan. 2012), pp. 167–177. DOI: 10.1201/b12295.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[11] Rogerio Carvalho, Rodrigo Soares Manhães, and Fernando Luiz Carvalho e Silva. "Filling the Gap between Business Process Modeling and Behavior Driven Development". In: *CoRR* abs/1005.4975 (2010). arXiv: 1005.4975.

[12] Tsun S. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Transactions on Software Engineering* SE-4.3 (May 1978), pp. 178–187. ISSN: 2326-3881. DOI: 10.1109/TSE.1978.231496.

[13] Robert Dezmerean and Karl Kristian. *Graphwalker documentation: Generators and stop conditions*. https://github.com/GraphWalker/graphwalker-project/wiki/Generators-and-stop-conditions. Accessed: 2020-07-10.

[14] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison-Wesley Professional, 2009. ISBN: 978-0-321-58051-1.

[15] Benedikt Eberhardinger, David Faragó, Mario Friske, and Dehla Sokenou. "Aktuelle Fragestellungen zum Zusammenspiel von BDD, MBT und KDT". In: *Softwaretechnik-Trends* 36.3 (2016). URL: http://pi.informatik.uni-siegen.de/stt/36_3/01_Fachgruppenberichte/TAV/TAV392016_TOOPEberhardingerFaragoFriskeSokenou.pdf.

[16] Vladimir Entin, Mathias Winder, Bo Zhang, and Andreas Claus. "A Process to Increase the Model Quality in the Context of Model-Based Testing". In: *Proceedings of the 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*. IEEE Computer Society, 2015, pp. 1–7. DOI: 10.1109/ICSTW.2015.7107471.

[17] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003. ISBN: 0321125215.

[18] David Faragó, Mario Friske, and Dehla Sokenou. "Towards a Taxonomy for Applying Behavior-Driven Development (BDD)". In: *Proceedings of "Software Qualität in der Ausbildung, 43. Treffen der Fachgruppe 2.1.7 Test, Analyse und Verifikation von Software (TAV) der Gesellschaft für Informatik (GI), Bremerhaven"*. 2019. URL: https://dehla.sokenou.de/papers/tav-bremerhaven0219.pdf.

[19] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9.

[20] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions of Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[21] Aslak Hellesøy. *The world's most misunderstood collaboration tool*. https://cucumber.io/blog/collaboration/the-worlds-most-misunderstood-collaboration-tool/. Accessed: 2020-06-06. Mar. 2014.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[22] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. "Design Science in Information Systems Research". In: *Management Information Systems Quarterly* 28.1 (Mar. 2004), pp. 75–105. DOI: 10.2307/25148625.

[23] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. "Using Formal Specifications to Support Testing". In: *ACM Computing Surveys* 41.2 (2009). DOI: 10.1145/1459352.1459354.

[24] Antti Jääskeläinen, Mika Katara, Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, Tommi Takala, and Heikki Virtanen. "Automatic GUI test generation for smartphone applications - an evaluation". In: *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009 - Companion Volume*. IEEE Computer Society, 2009, pp. 112–122. DOI: 10.1109/ICSE-COMPANION.2009.5070969.

[25] *JBehave documentation*. https://jbehave.org/reference/stable/. Accessed: 2020-06-06.

[26] Sunil Kamalakar, Stephen H. Edwards, and Tung M. Dao. "Automatically Generating Tests from Natural Language Descriptions of Software Behavior". In: *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2013*. SciTePress, 2013, pp. 238–245. ISBN: 978-989-8565-62-4. DOI: 10.5220/0004566002380245.

[27] Elizabeth Keogh. "BDD: A Lean Toolkit". In: *Proceedings of the Lean Software & Systems Conference*. 2010, pp. 15–22. URL: https://silo.tips/download/proceedings-of-lean-software-systems-conference.

[28] Nan Li, Anthony Escalona, and Tariq Kamal. "Skyfire: Model-Based Testing with Cucumber". In: *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*. IEEE Computer Society, 2016, pp. 393–400. ISBN: 9781509018260. DOI: 10.1109/ICST.2016.41.

[29] Nan Li and Jeff Offutt. "A test automation language framework for behavioral models". In: *Proceedings of the 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*. IEEE Computer Society, 2015, pp. 1–10. ISBN: 978-1-4799-1885-0. DOI: 10.1109/ICSTW.2015.7107402.

[30] Robert Cecil Martin. *The Truth about BDD*. https://sites.google.com/site/unclebobconsultingllc/the-truth-about-bdd. 2008. Accessed: 2020-09-20.

[31] Farid Meziane, Nikos Athanasakis, and Sophia Ananiadou. "Generating Natural Language specifications from UML class diagrams". In: *Requirements Engineering* 13.1 (2008), pp. 1–18. DOI: 10.1007/s00766-007-0054-0.

[32] Ramón López Narváez. "Entwurf und Entwicklung einer Testautomatisierungsplattform zur Kombination von Model-based Testing und Behavior Driven Testing im Finanzsektor". Master's thesis. Technische Universität Wien, Mar. 2016. https://permalink.catalogplus.tuwien.at/AC13090107.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[33] Arilo Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme Travassos. "A Survey on Model-Based Testing Approaches: A Systematic Review". In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies. Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2007*. ACM, 2007, pp. 31–36. ISBN: 9781595938800. DOI: 10.1145/1353673.1353681.

[34] Tuomas Pajunen, Tommi Takala, and Mika Katara. "Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework". In: *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012*. IEEE Computer Society, 2011, pp. 242–251. DOI: 10.1109/ICSTW.2011.39.

[35] Jan Peleska, Jörg Brauer, and Weng-ling Huang. "Model-Based Testing for Avionic Systems Proven Benefits and Further Challenges". In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2018*. Lecture Notes in Computer Science. Springer, 2018, pp. 82–103. ISBN: 9783030034269. DOI: 10.1007/978-3-030-03427-6_11.

[36] Stacy J. Prowell. "JUMBL: a tool for model-based statistical testing". In: *Proceedings of the 36th Annual Hawaii International Conference on System Sciences HICSS 2003*. IEEE Computer Society, 2003, p. 337. DOI: 10.1109/HICSS.2003.1174916.

[37] Harry Robinson. *Obstacles and opportunities for model-based testing in an industrial software environment*. http://testoptimal.com/ref/Obstaclesandopportunitiesformodel-basedtesting.pdf. 2004. Accessed: 2020-08-10.

[38] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, and Jon Peterson. *RFC 3261 - SIP: Session Initiation Protocol*. https://tools.ietf.org/html/rfc3261. Accessed: 2020-02-10.

[39] Jonathan Rosenberg, Henning Schulzrinne, and Rohan Mahy. *RFC 4235 - An INVITE-Initiated Dialog Event Package for the Session Initiation Protocol (SIP)*. https://tools.ietf.org/html/rfc4235. Accessed: 2020-08-19.

[40] Didier Ruedin. *Interpolated Median in R*. https://druedin.com/2012/09/21/interpolated-median-in-r/. 2012. Accessed: 2020-08-09.

[41] Didier Ruedin. *An Introduction to the R Package Agrmt*. https://cran.r-project.org/web/packages/agrmt/vignettes/agrmt.pdf. 2020. Accessed: 2020-09-21.

[42] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. "On the Evolution of Keyword-Driven Test Suites". In: *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019*. IEEE Computer Society, 2019, pp. 335–345. DOI: 10.1109/ICST.2019.00040.

[43] Douglas C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *IEEE Computer Society* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[44] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer International Publishing, 2015. ISBN: 9783319127415. DOI: 10.1007/978-3-319-12742-2_1.

[45] Sandeep Sivanandan and C. B Yogeesha. "Agile Development Cycle: Approach to Design an Effective Model Based Testing with Behaviour Driven Automation Framework". In: *Proceedings of the 20th Annual International Conference on Advanced Computing and Communications, ADCOM 2014*. IEEE Computer Society, 2014, pp. 22–25. ISBN: 9781467365093. DOI: 10.1109/ADCOM.2014.7103243.

[46] Mathias Soeken, Robert Wille, and Rolf Drechsler. "Assisted Behavior Driven Development Using Natural Language Processing". In: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns, TOOLS Europe 2012*. Vol. 7304. Lecture Notes in Computer Science. Springer, 2012, pp. 269–287. DOI: 10.1007/978-3-642-30561-0_19.

[47] SmartBear Software. *Gherkin Reference*. https://cucumber.io/docs/gherkin/reference/. Accessed: 2020-06-06.

[48] Carlos Solís and Xiaofeng Wang. "A Study of the Characteristics of Behaviour Driven Development". In: *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*. IEEE Computer Society, 2011, pp. 383–387. ISBN: 9781457710278. DOI: 10.1109/SEAA.2011.76.

[49] Jan Stenberg. *BDD Tool Cucumber is 10 Years Old: Q&A with its Founder Aslak Hellesøy*. https://www.infoq.com/news/2018/04/cucumber-bdd-ten-years/. Accessed: 2020-06-06.

[50] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN: 9780321446114.

[51] Tom Swain. *Model-Based Statistical Testing*. http://jumbl.sourceforge.net/MBSTtutorialSF.pdf. Accessed: 2020-10-06. 2013.

[52] Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. 2nd. Artech House, Inc., 2018. ISBN: 978-1-60807-850-9.

[53] Daniel Terhorst-North. *Introducing BDD*. https://dannorth.net/introducing-bdd/. Accessed: 2020-02-09.

[54] Daniel Terhorst-North. *JBehave 2.0 is live*. https://dannorth.net/2008/09/08/jbehave-20-is-live/. Accessed: 2020-06-06.

[55] Daniel Terhorst-North. *What's in a story?* https://dannorth.net/whats-in-a-story/. Accessed: 2020-06-06.

[56] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007. ISBN: 0123725011.

[57] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (2011), pp. 297–312. DOI: 10.1002/stvr.456.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[58] Ayal Zylberman and Aviram Shotten. *Test Language-Introduction to Keyword Driven Testing*. https://www.softwaretestinghelp.com/wp-content/qa/uploads/2010/01/keyword-driven-testing.pdf. 2009. Accessed: 2020-09-18.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.