MODEL-BASED TESTING OF FLEXIBLE SYSTEMS

January 2015
Vignir Örn Guðmundsson
Master of Science in Computer Science
MODEL-BASED TESTING OF FLEXIBLE SYSTEMS
– AN EMPIRICAL STUDY ON NASA GMSEC AND QUIZUP ANDROID APP

Vignir Örn Guðmundsson
Master of Science
Computer Science
January 2015
School of Computer Science
Reykjavík University

M.Sc. PROJECT REPORT

ISSN 1670-8539
Model-based testing of flexible systems

by

Vignir Örn Guðmundsson

Project report submitted to the School of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

January 2015

Project Report Committee:

Dr. Luca Aceto, Supervisor
Professor, Reykjavik University

Dr. Mikael Lindvall, Co-Supervisor
Division Director, Fraunhofer CESE

Dr. Marjan Sirjani
Professor, Reykjavik University

Dr. Mohammad Reza Mousavi
Professor, Halmstad University
The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this project report entitled **Model-based testing of flexible systems** submitted by **Vignir Örn Guðmundsson** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.
The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this project report entitled **Model-based testing of flexible systems** and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the project report, and except as herein before provided, neither the project report nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author’s prior written permission.

________________________________________

Date

________________________________________

Vignir Örn Guðmundsson
Master of Science
Abstract

We present an empirical study in which model-based testing (MBT) was applied to two different flexible systems. Namely, the software bus of NASA’s Goddard Mission Service Evolution Center (GMSEC), a reusable software framework used in several NASA missions, and the Android client of QuizUp, currently the largest mobile trivia game in the world. The main goal of the study was to examine whether MBT could be used, in an effective and efficient way, to test flexible systems using the same MBT approach. Our hypothesis was that although GMSEC and QuizUp are indeed different (i.e. QuizUp has a graphical user interface, GMSEC does not) they share certain similarities that would allow us to use the same MBT approach for testing both systems. In addition, we address the question of whether we could apply our MBT approach with reasonable effort. The effort would be viewed as reasonable if we would be able to successfully apply MBT within 6 months (i.e. the typical time limit for an internship) for each study. A comparison of the cost and benefit of using finite state machines (FSM) vs. extended FSMs (EFSM) as model representations in MBT is also provided.

The study shows that we were able to use the same MBT approach to test two different flexible systems, thus, supporting our hypothesis. Non-trivial defects were detected, especially using EFSMs, on systems that were already well-tested. We found that maintaining a single behavioral model for each system was key in order to test these flexible systems in an efficient way.

The study also demonstrates that it was feasible, even for a person without previous MBT background, to successfully apply MBT to these two flexible systems. Our comparison of costs and benefits showed that MBT started paying off as soon as we had finished our first iteration of MBT in the GMSEC study.
Hugbúnaðarprófanir á sveigjanlegum kerfum med notkun stærðfræðilegra módelu

Vignir Örn Guðmundsson

Janúar 2015

Útdráttur

Þessi ritgerð kynnir rannsókn byggða á tilraunum þar sem framkvæmdar voru hugbúnaðarprófanir með notkun módelu (e. model-based testing) á tvö mismunandi sveigjanleg kerfi. Þ.e., NASA Goddard Mission Evolution Center (GMSEC), sem er endurnýtanlegt kerfi fyrir ýmis verkefni og leiðangra á vegum NASA, og Android viðmót QuizUp, sem er stærsti spurningaleikur fyrir snjalltæki í heimi. Markmiðið með rannsókninni var að athuga hvort hvert væri að nota sömu aðferð byggða á hugbúnaðarprófunum með notkun módelu, á skilvirkan og árangursríkan hátt, til að athuga áræðanleika sveigjanlega kerfa. Tilgáta okkar var sú að þrátt fyrir að GMSEC og QuizUp væru sannarlega ólík kerfi (þ.e. QuizUp hefur grafísk viðmót, GMSEC ekki), þá eiga þau margt sameiginlegt sem mun gera okkur kleyft að nota sömu aðferð til að athuga áræðanleika beggja kerfa. Að auki, þá svörum við spurningunni hvort hægt sé að framkvæma slíkar athugarir án of mikillar fyrirhafnar í tíma og erföðleika. Við ályktuðum að ef heppnuð framkvæmd tæki innan við 6 mánuði, fyrir hvora rannsókn fyrir sig, þá mætti líta á fyrirhöfnina sem viðráðanlega.

Rannsóknin sýnir að við gátum notað sömu aðferð, sem stuđist við stærðfræðileg módel, til þess að athuga áræðanleika sveigjanlega kerfa. Þar af leiðandi gátum við ekki hafnað tilgátu okkar, en teljanlegir gallar fundust í kerfum sem voru vel prófuð fyrir rannsóknina. Við teljum að lykillinn að því að prófa sveigjanleg kerfi á árangursríkan hátt, líkt og GMSEC og QuizUp, sé að viðhalda einungis einu stærðfræðilegu módeli.

Rannsóknin sýnir einnig að það var framkvæmanlegt að beita fyrrenefndum prófunum með göðum árangri án of mikillar fyrirhafnar. Samanburður okkar á kostum og göllum leiddi það í ljós að prófanirnar föru að borga sig um leið og við höfðum lokið við fyrstu ítrun í GMSEC rannsókninni.
This thesis is dedicated to my wonderful mother.
Acknowledgements

I thank Mikael Lindvall for being a great mentor and a friend. I learned a lot during my internship at Fraunhofer CESE in 2012, where Mikael helped me grow as a researcher and sharpened the importance of critical thinking through our work. I would also like to thank him for all the great moments outside work, including trips and events that were always a lot of fun.

I thank Luca Aceto for his tremendous mentorship during my time in the Masters program at Reykjavik University, which included insightful guidance during my thesis research.

I thank the other thesis examiners, Marjan Sirjani and Mohammad Reza Mousavi, for valuable questions and comments during and after the MSc. defense.

I thank Dharmalingam Ganesan, Christoph Schulze and the rest of the Fraunhofer CESE team for a great collaboration during and after my internship in College Park.

Finally, I thank Jóhann Þorvaldur Bergþórsson and the rest of the Plain Vanilla team for their open-minded approach towards the research and their warm hospitality during the summer of 2014.
Publications

This thesis is partially based on a two-page fast abstract presented at ISSRE 2013, [23], and a full paper submitted to the NASA Journal [24].
Contents

List of Figures xiii
List of Tables xiv

1 Introduction 1

2 Model-based testing with state machines 7
   2.1 Model representations 7
   2.2 Overview of MBT using state machines 8
      2.2.1 From models to test execution 9
   2.3 Hierarchical models 10

3 An Out-Of-Space Case study: NASA GMSEC 13
   3.1 About NASA GMSEC 13
   3.2 Core features of a software bus 15
      3.2.1 Testing questions 17
   3.3 Applying MBT on the GMSEC API 18
   3.4 Lessons learned 30

4 Kelevra: The MBT Project Tool 33
   4.1 About Kelevra 33
   4.2 Features 34
   4.3 How to use Kelevra 36

5 A Mobile Case Study Using Kelevra: QuizUp 41
   5.1 About QuizUp 41
   5.2 Core features of the QuizUp application 44
      5.2.1 Testing questions 49
   5.3 Applying MBT on QuizUp 50
   5.4 Lessons learned 61
6 Discussion
   6.1 Common lessons learned ........................................ 65

7 Related Work ...................................................... 69

8 Conclusions ........................................................ 73
# List of Figures

2.1 From models to test execution. ........................................... 10
2.2 First and second layer of a sample SUT hierarchical model. ............. 11
2.3 Third layer of a sample SUT hierarchical model. .......................... 11

3.1 Interaction between two application via the bus. .......................... 16
3.2 Highest layer of the GMSEC model. ........................................ 19
3.3 The second layer of the GMSEC model. ..................................... 20
3.4 Simplified version of the FSM sub-model for unsubscribe, in the third layer. The only difference as compared to the complete sub-model is that it contains 7 subjects as transitions, not 3 as the simplified picture shows. . 21
3.5 Complete EFSM sub-model for feature subscribe. An EFSM guard is denoted by “[ guard = true | false ]”. The helper function getSubject() returns a random subject from the list of subjects. ....................... 26

5.1 The access scenes of QuizUp. ............................................... 44
5.2 In-game scenes of QuizUp (3 of 9). ....................................... 45
5.3 In-game scenes of QuizUp (6 of 9). ..................................... 47
5.4 In-game scenes of QuizUp (9 of 9). ..................................... 48
5.5 Highest layer of the QuizUp model. ....................................... 53
5.6 Sub-model for the Email Log-in scene in the second layer of the QuizUp model. ................................................................. 54
5.7 Sub-model for the In-game scenes in the second layer of the QuizUp model. 55
5.8 Sub-models for the Topics scene and the Game-play scene. .......... 56
List of Tables

3.1 Mapping labels to API code for some sample transitions. . . . . . . . . . 22
3.2 Summary of detected issues for each testing question: FSMs. . . . . . . . 23
3.3 Summary on FSM model. . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.4 Setup effort in the GMSEC study. . . . . . . . . . . . . . . . . . . . . . 24
3.5 FSM effort in the GMSEC study. . . . . . . . . . . . . . . . . . . . . . 24
3.6 Summary of detected issues for each testing question: EFSMs. . . . . . 28
3.7 Summary on EFSM model. . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.8 EFSM effort in the GMSEC study. . . . . . . . . . . . . . . . . . . . . . 29
5.1 Summary of detected issues in QuizUp for each testing question. . . . . 59
5.2 Summary on the QuizUp EFSM model. . . . . . . . . . . . . . . . . . . 60
5.3 Setup effort in the QuizUp study. . . . . . . . . . . . . . . . . . . . . . 60
5.4 EFSM effort in the QuizUp study. . . . . . . . . . . . . . . . . . . . . . 60
Chapter 1

Introduction

Modern software systems are often designed with the goal of maximizing flexibility in one way or another. Built-in flexibility can be achieved by providing loose coupling, for example through service oriented architectures and other architectural styles that bring flexibility, and by indirectly using middlewares, databases, operating systems, as well as by offering the same application programming interface (API) for different programming languages etc. These design strategies make it possible to build, modify, use, and replace components and services as well as to change platform technology with limited effort.

Another example of flexibility is business flexibility, that is, flexibility from a business standpoint. In the modern world, customers of software businesses (or companies) demand that systems and applications are available on multiple platforms, operating systems, languages, etc. It therefore becomes a business goal to ensure that a system, or application, runs conceptually in the same way on multiple operating systems, for example, which can be a hard task to implement and maintain.

In both cases, increased flexibility makes it difficult to predict the system’s behavior because the number of possible configurations can increase drastically. It follows that ensuring that a flexible system behaves as expected is much more difficult than for a system without such flexibility.

Various approaches have been introduced in research to achieve reliability, testability and usability of software products [37]. Formal verification (for example, model checking), system simulation and testing are all examples of approaches for system validation [14, 40].
Model-based testing of flexible systems

Testing, which is a widely used methodology in system validation, is a key method of assessing the functionality of systems. A successful test case is one which furthers the progress of ensuring that a system meets its quality standards. This purpose is best achieved by a diligent exploration for errors [33]. However, applying testing to flexible systems requires in addition a well-organized testing strategy in order to systematically cover a reasonable number of combinations of components, services, APIs, middlewares, platforms, etc.

Traditional testing often relies on manual creation and execution of test cases that mimic realistic usage of the system. Manual testing is tedious, especially when a large number of combinations of usage scenarios for various configurations must be covered. For each configuration, the tester has to manually enter data, manually issue commands, and manually compare the actual system behavior with the expected behavior. Differences between actual and expected behavior are manually documented and reported as issues. Whenever a new version of the system under test has been developed, the manual tester must go through the same process again. Since new versions are typically released several times per year, the manual testing effort can be significant.

The problems related to manual testing have been observed by many software organizations, which therefore strive to automate the testing process. This has led to the successful use of test case execution frameworks (e.g. JUnit [11]). Such frameworks are very helpful because they can automatically run the test cases once they have been encoded as executable test cases (a.k.a. concrete test cases, test scripts, and test programs). Thus, this type of automation frees up the manual tester because the executable test case automatically enters data, issues commands, compares actual and expected values, and reports detected issues.

However, such test programs must still be created and maintained manually. Manually creating the large numbers of executable test cases that are required to test a flexible system can be difficult. One reason is that it can be difficult to imagine the many different ways in which the system can be used or misused. In addition, since a flexible system can be configured in many different ways and can run on many different platforms, the creation of executable test cases can be a very complex activity.

To overcome this limitation, model-based testing (MBT) has gained popularity in the research community and recently also in industry [26]. MBT addresses the problems of manual burden by using a model (e.g. state machines) to represent the expected behavior of a system under test (SUT). By using a tool for traversing the model, we automatically generate a set of non-executable abstract test cases, where each test case is the sequence of the traversed states and transitions. The label of all states and transitions in the model are
extracted programatically and then manually mapped to executable code fragments. The abstract test cases are then instantiated to executable test cases using the code fragments. The SUT is eventually executed automatically using the generated tests, providing results which are then analyzed.

While MBT is a promising technology that theoretically has the power to address the issues with manual testing discussed above, it is still unclear if it does so in practice, in particular for systems designed to be flexible. This motivates a major research question we address in this thesis:

Can MBT be used, in an effective and efficient way, to test flexible systems using the same approach?

In order to answer this question, and substantiate the main goal of the thesis, we examined two different flexible systems, which are the NASA Goddard Mission Service Evolution Center (GMSEC) and QuizUp.

NASA’s GMSEC is an example of a system with built-in flexibility [19]. GMSEC is built around a software bus that allows applications (which is the GMSEC term for "components") to be plugged in and out without requiring any software changes, except for changes to the configuration settings. GMSEC is especially designed to support the satellite control domain. Since many command and control systems are now GMSEC compatible, software developers can build their own applications as well as choose among a number of existing applications [20]. The GMSEC software bus, which is the system under test (SUT), is accessible through GMSEC’s API. The API is based on the publish-subscribe architectural style (pub-sub) [16].

QuizUp is an example of an application with business flexibility. QuizUp is a mobile trivia game that supports both the Android and iOS mobile operating systems [29]. Currently, QuizUp has over 20 million users combined (both Android and iOS), resulting in the application being the largest mobile trivia game in the world [4]. The game has over 400 topics that include over 220000 questions. QuizUp is a social game where users can challenge other users, friends or strangers, on a particular topic, as well as send messages, post threads to discussions, etc. The graphical user interface (GUI) of QuizUp’s Android client is the second SUT in this study.

Although NASA GMSEC and QuizUp are two very different systems in general, they have certain similarities; for example, they are both based on user interaction, which can be described as sequences of events. Therefore, our hypothesis, that is embedded in the major research question above, was that we could apply the same technology-independent MBT approach on both systems.
Another question that we wanted to address in this thesis is whether MBT is 'doable at all' for flexible systems. In the context of this research, doable means 'with reasonable effort' such as an internship where the duration is limited to 6 months during which the person has to understand the system under test, build all testing infrastructure, build the model, generate test cases, test the system, and analyze results. Our tenet is if all of these activities could be achieved in such limited time, then it would be reasonable to conclude that the technology is applicable to a wide range of systems even though we, for practical reasons, limited the study to two system. Thus, another goal of this work was to study the overall costs and benefits of using MBT in general. We also studied the costs and benefits of using finite state machines (FSMs) as compared to using extended FSMs (EFSMs) as model representations.

The results presented in this thesis show that MBT is feasible because it could be applied within reasonable effort and that it detected several, previously unknown, software bugs that, despite extensive effort, were not found by the traditional testing approach used at NASA and QuizUp. For NASA GMSEC, in particular EFSMs detected more bugs than the basic FSMs, but at a higher cost to develop the model. Based on that experience, we chose to solely use EFSMs for the subsequent QuizUp study.

This thesis, which is partially based on our two-page fast abstract presented at ISSRE 2013 and a full paper submitted to the NASA Journal [23, 24], contributes the following:

- Two case studies where MBT was successfully applied on NASA GMSEC’S software bus and QuizUp’s Android client;
- A description of Kelevra, an MBT project management tool, created as a part of the thesis work after the NASA GMSEC study and used in the QuizUp study;
- A comparison of costs and benefits from using finite state machines (FSM) vs. extended FSMs (EFSM) in MBT on a real-world system;
- Lessons learned related to applying MBT in general, but also specific to the two case studies.

**Structure of the thesis**

In Chapter 2, we present generalities of the MBT approach that was used in the two case studies reported in this thesis. In Chapter 3, we present the first case study where MBT was applied to NASA GMSEC, a reusable software framework. In Chapter 4, we introduce an MBT project tool, called Kelevra, that was motivated by our lessons learned after the GMSEC study. The tool can be used to maintain MBT projects as well as to
trigger the various phases of MBT. In Chapter 5, we present a mobile case study where MBT and Kelevra were applied to QuizUp’s Android client. In Chapter 6, we compare and discuss the similarities and differences of applying MBT on two different flexible systems. Finally, in Chapter 7, we present our conclusions based on the work done and some directions for future research.
Chapter 2

Model-based testing with state machines

2.1 Model representations

For this study, we chose to use two styles of modeling representations. Namely, FSMs and EFSMs. They were chosen because state machines are easily understood and well-studied theoretically [45]. Also, easy-to-use open source tools that generate test cases from them are readily available [27, 35].

Informally, an FSM consists of states and transitions (see for instance, [45] for a formal definition). It contains a specific state, called the start state, where the machine starts computing. In the context of MBT, a transition represents the system stimulus, or action, that moves the user from one state of the system to another. Such actions could be, for example, calling an API method or clicking a button on a GUI. The states are used to embed testing assertions to check that the user is in fact in the expected state of the system. These assertions are boolean expressions based on the return code that is expected from a particular method call when a system is in a certain state.

The EFSM model is a generalization of the traditional Finite-State Machine model with guards or helper functions associated with transitions [28]. Guards are boolean expressions or functions that are evaluated during model-traversal time. State variables allow us to store the history of traversal and can be used to embed test oracles in the model. Based on the evaluation of a guard, a model-traversing tool or algorithm randomly chooses among the transitions whose guards are satisfied, which is not possible for FSMs. Using EFSMs we are also able to encode data parameters as variables as opposed to embedding
the data at the model level as transitions or states. Such encoding can, for example, be done to parameterize, or initialize, the model at the start of each traversal, storing data as variables or other data structures. This allows us to modify the data to be tested without changing the model and thus it does not affect the number of states and transitions in the model using EFSMs. Hence, the model is easy to review, comprehend, and evolve. In addition, guards and helper functions can be used to return data from actions based on the current traversal history.

2.2 Overview of MBT using state machines

Automated testing approaches have already been successfully used by the software industry. Frameworks such as JUnit have been especially successful because they allow automated execution of executable test cases. However, as the SUT changes, new test cases have to be manually created and the existing ones have to be manually updated. Since executable test cases often “reuse” many code instructions in a copy and paste manner, changes to the SUT often require many test cases to be updated. The maintenance of the executable test cases thus creates an extra burden that limits the usefulness of this manual approach [38].

MBT addresses these problems by using a model to represent the expected behavior of the SUT and generating test cases from the model. Each test case is essentially a random path through the model. Thus a certain amount of randomness is automatically built in the testing process. In this way, MBT addresses the problem with test cases being static and not covering enough corner cases, which are often exercised by taking an unusual path through the system or using certain functionality repeatedly and in unexpected ways.

The model is created from the perspective of a user focusing on the functionality of the SUT’s interface, where a user can be either a human being or another program/system. Thus, testing using this approach is driven by the specifications of the interface of the SUT and does not use the internal structure of the software under test as the driver [47]. The assumed benefit is that even though the SUT is large, the model and sub-models typically remain manageable in size. Thus, instead of depending on the complexity of the actual system, the models are as complex as the model of the interface under test.

Below, we describe how we used MBT in an end-to-end fashion, from models to concrete test cases. It should be noted that the approach presented in what follows can be applied to any system that has an API. However, the API must provide functions that return values
that reflect the current state of the system and/or whether the function call succeeded or not. If a function under test does not return such values then it must offer other functions that provide insights into the state of the system. Otherwise, MBT and any other type of similar testing approach will be unsuccessful.

### 2.2.1 From models to test execution

Figure 2.1 shows the different steps of MBT.

- **Step 1:** We build a model based on the requirements, existing test cases, as well as by exploring the SUT. It is worth noting that there are approaches that can be used to learn models [1], as well as approaches that apply techniques to crawl or extract a model automatically from a SUT [2].

- **Step 2:** By using a tool for traversing the model, we generate a set of abstract test cases, where each test case is the sequence of the traversed states and transitions. These test cases are not executable. Each element in that sequence constitutes a state or transition label, which is a descriptive name of the expected state of the system, or the action associated with that transition.

- **Step 3:** First the labels of all states and transitions in the model are extracted programmatically. Then, using a mapping table, the labels of all states and transitions in the model are manually mapped to executable code fragments. Transition labels represent actions and are typically mapped to methods of the API of the SUT. State labels are typically mapped to code that checks the state of the system (using assertions) after the action has been carried out. The first column in the mapping table contains all labels in the model and the second column contains the corresponding code fragment.

- **Step 4:** Using an instantiator program, we then automatically create executable test cases by replacing each label in the abstract test case with the corresponding code from the mapping table. A header for the concrete executable test case (e.g. a JUnit test program) is then inserted. The header consists of all necessary definitions, declarations, objects, etc. for the test case to be compiled.

- **Step 5:** The SUT is executed automatically using the generated tests.

- **Step 6:** The test results, especially the failed test cases, are analyzed.

Note that it is outside the scope of the testing process to analyze in detail the defect that caused the test case to fail, or to remove the defect.
For this study, we chose to model the SUTs as collections of state machines that are structured in a hierarchical fashion. We refer to such collections as a hierarchical model (also known as statecharts [25]). The difference between an FSM and a hierarchical one is that a state in a hierarchical model can stand for a complex subsystem that is itself described by a (possibly hierarchical) FSM. That way, we are able to manage the complexity of the modeling effort by dissassembling the SUTs into multiple layers, rather than having a flat model. A flat model is hard to maintain when the model has reached a certain size. Since the models are visual models (or graphs), it becomes hard to read models with 15 states or more, for example. Such models are therefore not scalable for most real systems that have some, or any, built-in complexity since the state space is normally larger for such systems. A major benefit of the hierarchical models is the fact that each layer represents a certain abstraction of the SUT. In the highest layers of the model we only demonstrate the most important details of the SUT. However, in lower layers of the model one can see actions performed, embedded assertions, etc.

Let us use Figure 2.2 to demonstrate the benefits of these hierarchical models for a simple sample SUT. Please note that the color of a state denotes the type of that state. A white state is a regular state, whereas a black state is an entry state from the 1st to the 2nd layer.
and a gray state is an entry state from the 2nd to the 3rd layer. When an exit state is reached, the traversal should return to the immediate higher layer.

![Diagram of layers](image)

(a) 1st layer
(b) 2nd layer

Figure 2.2: First and second layer of a sample SUT hierarchical model.

In Figure 2.2a, we can see the highest layer of the sample SUT model. This particular sample SUT requires the user to log in prior to performing any sort of action within the SUT, and the model is designed to reflect this requirement. Note that we are not doing evil testing here, that is, attempting to perform actions without being logged in. Such design is perfectly valid, but out of scope for this example. The second layer, given in Figure 2.2b, models two sample actions that one could supposedly perform in the sample
SUT. The states for those two sample actions serve as entry states to a sub-model for each action, respectively. Therefore, the layer is designed as an intermediate layer between being logged in and performing actions.

In the third and lowest layer of the model, depicted in Figure 2.3, the sub-models for the two actions are modeled. At this layer, the actual triggering of actions, e.g., transition "PERFORM ACTION #1", and assertions as states for corresponding actions (e.g. "ACTION #1 ATTEMPTED") are modeled.

Therefore, the sample SUT model contains three layers. In general, there can be either fewer or more layers depending on the SUT and the level of detail in the modeling effort.

A model traversal starts at the start state in the highest layer of the model and stops when a given stopping criterion is met. We used yEd, [51], to create the models in this study and used Graphwalker to traverse the model [27]. Graphwalker is an open-source Java library that generates test sequences from FSMs and EFSMs that are, e.g., created using yEd models and stored in Graphml format. Graphwalker offers stopping criteria such as state coverage and transition coverage. For example, using state coverage, the model would be traversed until the specified percentage of states in the model has been covered. During such traversal, states in any of the model’s layers can be visited until the stopping criterion is met.

In the following chapters, we will provide real examples of hierarchical models for both NASA GMSEC and QuizUp and describe how we used them in applying MBT to those systems.
Chapter 3

An Out-Of-Space Case study: NASA GMSEC

3.1 About NASA GMSEC

GMSEC is a flexible and reusable framework that provides a scalable, extensible ground system for current and future missions both inside and outside NASA. The built-in flexibility has allowed many space missions (e.g., the Global Precipitation Measurement (GPM) mission, the Magnetospheric Multiscale (MMS) mission, and the Lunar Reconnaissance Orbiter (LRO)) to build their systems based on GMSEC. A GMSEC application is a large component that accomplishes a certain set of functionality. For example, the GMSEC System Display (GEDAT) application provides a visual representation of the GMSEC environment and can display data system activity at any level within the network.

The software bus which GMSEC implements allows GMSEC-compatible applications to be integrated with the system in a plug-and-play manner. There are many GMSEC-compatible applications ("application" for short) on the market already, thus one can build the basis for a new ground system in a few hours. In case the necessary application is not already available, it can be implemented in one of the supported programming languages. By adding the GMSEC API for the applicable programming language, the application becomes GMSEC-compatible and can be plugged into any system that is based on the GMSEC bus. In addition to the API that is supported in different programming languages, there are several different middleware products available on the market that are supported by GMSEC. The applications that use the GMSEC API do not access the middleware directly but through a middleware wrapper. The software bus implementation redirects
all method calls to appropriate middleware wrappers at runtime. There is one wrapper for each middleware product. While middleware products typically focus on the transportation of data, the software bus adds messaging based on the publish-subscribe style [34] in a consistent way across all supported middlewares.

Testing of the bus is a complex endeavor because of the flexibility it offers. The bus provides a set of commands related to the pub-sub style, where the user has great flexibility in creating valid command sequences making it a challenge to test them all. In addition, the system supports a flexible number of subjects (or topics) to which applications can publish, where subscriptions can be formulated using regular expressions. Since the API is implemented in several different programming languages, each such API has to be tested. The fact that there are wrappers for many middlewares, which all must be tested, adds to the testing burden. Ideally, there is a suite of test cases that cover "all" possible combinations of valid pub-sub commands. The test suite would then be automatically translated to each of the supported programming languages, and executed on each of the supported middlewares, thus indirectly testing each wrapper.

Over the years, the GMSEC team has developed a set of executable test cases that achieve some of these goals. However the test suite is manually written and limited to a couple of common pub-sub scenarios. In addition, the test suites for the different programming languages and middleware wrappers are not "identical" and thus the results cannot easily be compared across configurations. Naturally, the GMSEC team conducts large amounts of testing. They also continuously review the artifacts in order to detect potential issues. In addition to their rigorous reviews and testing, the GMSEC team prefers an independent organization such as Fraunhofer CESE to review the implementation quality [17], as well as to evaluate new technologies for potential infusion into the GMSEC project.

Since the GMSEC team is interested in improving their testing processes through automation, the primary goal of this study was to study the feasibility of using MBT on the GMSEC’s software bus. The GMSEC team was also interested in understanding the learning curve related to MBT for someone with educational background similar to most practitioners, who typically do not have a PhD and do not have specific MBT background, but are skilled in programming. Thus, it was appropriate that a programmer who did not have any previous knowledge of MBT, FSMs, EFSMs or of GMSEC or software buses applied MBT on GMSEC and recorded the effort.

A major question for the study was: Can the GMSEC software bus be modeled in such a way that test cases can be automatically generated to test the software bus and its pub-sub messaging features as well as all possible combinations of programming languages and middleware wrappers without modifying the model? Since GMSEC was already tested by
the GMSEC team, the MBT approach would be viewed as successful if it were able to detect issues that had not been uncovered by those earlier testing efforts. The work reported in this chapter covers the testing of the GMSEC software bus and its abilities to perform messaging according to the pub-sub style, thus no GMSEC applications were modeled or tested in this study. Instead, the test cases are generated from dummy applications that use the software bus’s APIs in different but realistic ways.

The scope of the study

The GMSEC supports several programming languages (C, C++, Java, Perl). For this study we decided to test the C, C++, and Java APIs because they are the most frequently used ones. GMSEC also provides different middleware wrappers for commercial and open-source middlewares, such as WebLogic, WebSphere, Tibco Smart Sockets, and ActiveMQ. In addition, GMSEC’s own middleware implementations Bolt and MB (a.k.a. Magic Bus) are provided as part of the GMSEC deliverable. We decided to test the wrappers for WebLogic, Bolt, and MB in this study. Because of commercial licensing issues, other middleware wrappers were excluded. It should be noted that while the commercial middlewares have already been tested by their vendors, the wrappers were developed as part of the GMSEC project and therefore must be tested by the GMSEC team. The study was carried out independently of the GMSEC team, but questions were asked to the GMSEC team during the process when the documentation was not specific enough. The findings were reported to the GMSEC team at the end of the study.

3.2 Core features of a software bus

In the pub-sub style, there are four key concepts in addition to the software bus (bus for short): publishers, subscribers, messages, and subjects (a.k.a. topics). Subscribers subscribe to one or more subjects, and publishers publish messages to one or more subjects. A subscription can specify the subject as a pattern using regular expressions. When a message belonging to a certain subject has been published, it is the responsibility of the software bus to deliver the message to the subscribers whose subscriptions match the subject. Thus, the subscribers and publishers are loosely coupled and unaware of each other; they are only aware of subjects and messages. Each application can be a publisher or a subscriber, or both, allowing an application to receive messages from other applications (as well as from itself) without knowing who sent them. In addition to a subject, each message has content, i.e. a structure containing a number of fields and values that carry
the data to be sent [16, 17]. The bus typically has an internal buffer where it keeps messages before they are delivered to the subscribers. Each subscriber periodically checks if there is a message waiting in the buffer. If so, the bus delivers the next available message when the subscribing application so requests.

Example 1  Figure 3.1 illustrates how two applications, App1 and App2, communicate indirectly through the bus. App1 publishes to the bus a message M that belongs to subject S. App2 receives message M because it subscribes to subject S. In the context of GMSEC, we can imagine that App1 (implemented in e.g. Java) monitors a sensor that measures the temperature in a missions operation center (MOC). Every hour, App1 publishes a message belonging to the temperature subject that contains information about the current temperature. App2 (implemented in e.g. C#) is an application that sends out alarms if the temperature exceeds a certain threshold. App2 subscribes to the temperature subject and thus will receive messages with temperature data from App1. App1 will not know from where the messages were sent, unless information about the sender is embedded into the message itself. When App3 (implemented in e.g. C++), which also sends out text messages, becomes available, the system owner can replace App2 for App3 within minutes and without any changes other than a few configuration settings. This replacement is possible since App3 is GMSEC compatible, i.e. it uses the GMSEC API (in this case the C++ API) as the interface to the bus.

![Figure 3.1: Interaction between two application via the bus.](image)

There are also a number of other behaviors related to the sequence of actions that are important to describe as rules because they dictate the type of testing that needs to be performed. See Figure 3.1.
• Establishing a connection to the bus: An application must be connected to the bus in order to interact with it. An attempt to connect if the application is not already connected should succeed, otherwise it should fail.

• Subscribing to a subject: An attempt to subscribe to a subject that the application does not already subscribe to should succeed, otherwise it should fail.

• Unsubscribing to a subject: An attempt to unsubscribe to a subject that the application already subscribes to should succeed, otherwise it should fail. A successful unsubscription to a subject removes the subscription for that subject, thus the application is allowed to subscribe again to the same subject.

• Delivery of messages: In order for a message to be delivered, an application must subscribe to the subject before messages of the same subject are published. Only if this is the case, the published message is delivered to the subscriber's buffer. Once the message is delivered to the buffer, the subscriber can retrieve that message at any time.

• Reliability: The bus should deliver one copy of each message to each subscriber, once and only once.

• Clean up upon disconnect: When an application disconnects from the bus, the bus should remove all of the application’s subscriptions as well as all of the application’s messages in the buffer that have not yet been delivered. Thus, if the application connects again to the bus, it has no subscriptions and no messages.

3.2.1 Testing questions

From the above mentioned rules, we derive the following testing questions, which together could be used to drive the modeling and the testing of any bus that is based on the pub-sub style.

• Q1. Are there issues related to the connect/disconnect functionality?

• Q2. Are there issues related to the disconnect cleanup functionality?

• Q3. Are there issues related to an application subscribing to one or more different subjects?

• Q4. Can a component subscribe to the same subject multiple times without unsubscribing in between?
Q5. Can a component unsubscribe to the same subject multiple times without subscribing in between?

Q6. Can a component unsubscribe to a subject it is currently not subscribed to?

Q7. Are there any missing messages?

Q8. Are there any extra messages?

Q9. Are there issues related to the validation of message subjects?

In the remainder of this chapter we will describe how we used MBT to answer the above questions for the GMSEC software bus.

### 3.3 Applying MBT on the GMSEC API

Modeling of the GMSEC bus was driven by the specifications for the core features of a software bus, the associated rules, and the derived testing questions. The model was then used to generate abstract test cases, which were automatically translated into three sets of concrete test cases, one for each programming language. Each set was then executed on each of the three middlewares. Thus, the abstract test suite was used to test nine different concrete configurations. We first conducted MBT using FSMs. Then the FSM models were turned into EFSM models. Detected defects and effort was recorded for both FSMs and EFSMs. The process and the collected data are discussed in more detail below.

It should be noted that GMSEC has a naming convention for subjects in order to standardize messages [31]. For both FSMs and EFSMs, we created two sets of message subjects to test that published messages were correctly matched to subscription specifications including text strings and regular expressions.

The first set includes 7 message subjects that are used when subscribing and unsubscribing to subjects. The second set includes 7 message subjects that are used for publishing messages. The subjects in the two sets were chosen so that a large number of subscribe-publish situations can be tested. They include subscription subjects specified as text strings, regular expressions, and invalid subjects as well as publishing subjects that match zero, one, two, and three subscription subjects.
The modeling goal

The modeling goal was to design the model in such way that the derived test cases would answer the testing questions and determine whether or not GMSEC is behaviorally consistent with the requirements of the pub-sub style. The goal was also for the test cases to represent realistic, but possibly unusual, interaction sequences. It is also important to note that the goal was not to model attempts to use subscribe, unsubscribe, publish or retrieve message without having an established connection to the bus (a.k.a. evil testing). The reason is that it was already known that the system would crash if an application attempted to e.g. subscribe without being connected. Thus, there was no need to further test this behavior.

The GMSEC model as hierarchical FSMs

We modeled the GMSEC as a collection of FSMs that were structured in a hierarchical fashion using three layers to manage complexity. The highest layer (see Figure 3.2) of the FSM models features related to connecting to and disconnecting from the bus and serves as an entry state to the second layer. The design explicitly ensures that the "dummy" application (the test case) has an established connection before performing any action. Unusual but valid sequences, such as repeatedly connecting and disconnecting are possible outcomes from random traversals of this model. The states are used to embed testing assertions to check that creating and destroying a connection as well as connecting to and disconnecting from the bus was successful.

![Figure 3.2: Highest layer of the GMSEC model.](image)

The second layer of the FSM model (see Figure 3.3) models the core features subscribe (two variants), unsubscribe (two variants), publish and retrieve message, where each state
serves as an entry state to a sub-model. Therefore, the second layer is designed as an intermediate layer between having a connection to the bus and using the core features in the third layer.

The GMSEC API provides two types of subscriptions and unsubscriptions: regular ones and ones with callback. The difference is that *subscribe with callback* automatically invokes the specified callback function each time a new message is retrieved. *Unsubscribe with callback* removes the subscription for the specified callback. For example, the following are treated as three different subscriptions: 1) a subscription to subject SA, 2) a subscription with callback to subject SA, function FA, 3) a subscription with callback to subject SA, function FB. Thus, unsubscribe with callback to subject SA, function FA would only remove one out of the three subscriptions. However, unsubscribe to subject SA (without callback) would remove all three subscriptions. The design of the second layer permits evil testing (testing of invalid sequences) in relation to the core features in the third layer. Invalid sequences can be generated since the model does not dictate that a certain command must precede another command. i.e. there are transitions in the model from the state "perform actions" to all six commands. An example of evil testing would be an attempt to retrieve the same message several times. Since the FSM model cannot keep track of whether or not there is a message waiting to be retrieved, this type of testing would reveal if the system crashes due to such attempts. Since FSMs are limited, what can be tested using models based on FSM is limited. For example, we could not embed assertions in any of the sub-models for the core features. The reason is that FSMs cannot keep track of previous events and therefore it is difficult to provide appropriate test oracles. Another style of modeling using FSMs is to explicitly include a finite number of fixed sequences of core features. However, since GMSEC’s original test cases already included several such fixed sequences we decided to focus the modeling effort on a complementary approach that would generate a large number of arbitrary sequences.
For example, since FMS models cannot keep track of arbitrary sequences of previous events/history, it follows that they cannot determine if and how many times a transition was taken during the traversal. Thus, it becomes impossible to determine the correct behavior in response to a subscribe command. The reason is that the model does not know if this is an attempt to subscribe to a subject for the first time (which is expected to be successful) or a second time (which is expected to be unsuccessful). One way to address this problem is to add a large number of states just to remember the history of traversal. Even if it would be feasible to create an FSM that tests the above sequence for one subject, this becomes a major problem as soon as one tries to model the handling of two or more subjects. This is one example of the well-known *state-explosion problem* [13].

Deciding the expected result for subscribing also requires keeping track of, e.g., whether an unsubscription occurred in between subscriptions to determine whether or not the subscription is an invalid duplication. In order to test unsubscription (see Figure 3.4), for example, we needed to pass the subject as a parameter since FSMs do not have the power to provide data unless the data is explicitly modeled. Thus each parameter value, modeled as transitions labels (for example, “Unsubscribe To Subject1” in Figure 3.4), is modeled as one transition. In this case, we used sets of 7 subjects for testing, which are modeled as 7 transitions in each sub-model for the appropriate features. Thus, the model becomes more complex for every value that is added. This is shown in Figure 3.4, a simplified version of the sub-model for unsubscribe, where the value for each subject parameter is modeled separately. Again, since FSMs modeled this way cannot keep track of previous subscriptions and unsubscriptions, it would be possible to detect only defects that crash the SUT.

![Figure 3.4: Simplified version of the FSM sub-model for unsubscribe, in the third layer. The only difference as compared to the complete sub-model is that it contains 7 subjects as transitions, not 3 as the simplified picture shows.](image)

**Generate abstract test cases**

Using Graphwalker, we generated 100 abstract test cases to achieve 100% state and transition coverage. These abstract test cases only contain the labels from the model and thus are independent of any programming language and any test execution technology.
Mapping labels to the GMSEC API

In order to transform the abstract test cases into executable test cases, we mapped the abstract states and transitions into concrete function calls to the GMSEC API for each of the three programming languages we consider. The state and transition labels were extracted from the model and stored in the first column of each table. The corresponding code for each label was manually inserted into the second column in each of the three mapping tables (e.g., as shown in Table 3.1). When the user presses the button to create concrete test cases, the instantiator program reads the abstract test cases and the table corresponding to the chosen programming language, replaces labels with the corresponding function calls, and writes concrete (i.e., executable) test cases (see Figure 2.1). Thus, one set of abstract test cases was instantiated into three sets of executable test cases, one set for each programming language.

Table 3.1: Mapping labels to API code for some sample transitions.

<table>
<thead>
<tr>
<th>State/Transition Label</th>
<th>API Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Unsubscribe To Subject1”</td>
<td><code>connection-&gt;Unsubscribe(“Subject1”);</code></td>
</tr>
<tr>
<td>“Unsubscribe To Subject2”</td>
<td><code>connection-&gt;Unsubscribe(“Subject2”);</code></td>
</tr>
<tr>
<td>“Unsubscribe To Subject3”</td>
<td><code>connection-&gt;Unsubscribe(“Subject3”);</code></td>
</tr>
</tbody>
</table>

Executing the concrete test cases

We executed each of the three sets of concrete test cases on the bus using the three different middlewares. The generated test cases, which did not require any manual modification, matched the test case format that was already developed by the GMSEC team. Thus, the generated test cases have exactly the same structure as GMSEC’s manually written concrete test cases. The common format facilitated integrating the generated test cases with the existing test execution and test result reporting framework.

Analyzing the results and effort

The results from executing the 100 test cases are shown in Table 3.2. In Column 3, an answer to each testing question is provided. In Column 4, the issues that can be traced back to middleware wrappers (MWW) are provided, whereas, in Column 5, issues that can be traced back to different implementations of the GMSEC API in Java, C, or C++ are provided. The value in a particular cell indicates whether or not we detected issues related to a particular testing question.
Using FSM, we detected issues for testing questions 1 and 2, which are related to connecting to and disconnecting from the GMSEC software bus. We were able to explicitly address these testing questions since we had built-in assertions in the highest layer of the model. However, for both Q1 and Q2, the issues were detected because of middleware crashes. That is, the system crashed before the assertion could catch the defect. Since all defects were unknown, it was impossible to know beforehand how they would manifest themselves, if they existed. We were not able to explicitly address testing questions 3-9 since the model lacked built-in assertions in the third layer. Therefore, we did not detect any issues related to core features subscribe, unsubscribe, publish and retrieve message due to the limitations of formulating appropriate test oracles using FSMs. The argument for modeling the core features (from the third layer) despite not being able to use asserts to detect potential issues is that if the test case execution would crash while calling any of those features, it would have given us a strong indication that the issue is related to that particular feature. However, the test case execution for FSMs did not cause the system to crash in relation to testing questions 3-9. Thus no defects were detected here.

Table 3.3: Summary on FSM model.

| 1st layer | 6 | 8 | 2 | 3 |
| 2nd layer | 8 | 13 | 0 | 0 |
| 3rd layer | 18 | 48 | 6 | 0 |
| Total | 32 | 69 | 8 | 3 |

As shown in Table 3.3, the FSM model consists of 32 states, 69 transitions, 8 requests (method calls), and 3 assertions. The reason for the high number of transitions in contrast to the number of states is that the 7 subjects we chose to test were each modeled as
one transition. Also, the low number of assertions is due to the previously mentioned limitations of FSMs that negatively impact the ability to formulate test oracles.

The setting-up effort before modeling and testing the GMSEC is shown in Table 3.4. It should again be noted that the programmer did not know MBT when this study started. The first two weeks were spent on getting to know MBT and the *Graphwalker* tool, where the programmer got assistance and material from CESE scientists. During the next two weeks, the programmer implemented instantiator algorithms and scripts for automation. With the assistance of CESE scientists, during the next three weeks, the programmer set up GMSEC’s test execution framework, middleware wrappers and configuration files. By that time, the programmer had set everything up in order to apply MBT on GMSEC.

<table>
<thead>
<tr>
<th>Task</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting to know MBT and the <em>Graphwalker</em> tool</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Designing instantiator algorithms and scripts for automation</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Setting up GMSEC’s test execution framework, middleware wrappers and configuration files</td>
<td>3 weeks</td>
</tr>
</tbody>
</table>

Please note that three of the seven weeks of set-up would have been the same if the goal was not MBT but simply test automation.

The programmer then spent time understanding GMSEC, incrementally and iteratively created the GMSEC model, and maintained the mapping tables as states or transitions were added. After the programmer finished the model and mapping tables, which took two weeks, one week was spent on executing the generated test cases and analyzing the failing test cases.

<table>
<thead>
<tr>
<th>Task</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designing the FSM model and mapping tables for GMSEC supported API languages</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Executing test cases and analyzing the results</td>
<td>1 week</td>
</tr>
</tbody>
</table>

Thus, it took 10 weeks for a programmer who was not familiar with MBT to create an FSM model, generate test cases, instantiate them for three different programming languages, execute the three test suites on three different middlewares, and analyze the detected defects.
Please note that one of the three weeks would be a part of any test automation effort and not specifically MBT. The reason is that independently of whether the executable test cases are manually written or automatically generated, they have to be executed and analyzed by a human being or a program. Additionally for test automation efforts with manually written test cases, the effort for writing them would be added.

The GMSEC model as hierarchical EFSMs

The next step was to study how the use of Extended Finite-State Machines (EFSM) would increase the possibilities to improve the testing by providing more elaborate test oracles. The goal was to implement test oracles that would allow us to embed more meaningful assertions to test the core features of GMSEC. We also wanted to study what the added effort for doing this would be.

The first and second layer of the FSM model were reused verbatim without any changes for the EFSM model. Using EFSMs we were able to encode the data parameters as helper functions as opposed to modeling each subject to be tested as a transition using FSMs. The helper functions enable us to parameterize the model at the start of each traversal, storing the subjects as variables. This allows us to modify the subjects to be tested without changing the model and thus it does not affect the number of states and transitions in the model using EFSMs. We also used helper functions to return data for actions based on the current traversal history. For example, when an application wants to subscribe, it needs a subject. For this, we implemented a helper function that returns one random subject out of the set of seven predefined subjects for subscribing and unsubscribing.

Note that we only need one transition to represent this action, not seven as in the FSM model. Such a transition thus represent many actions, since the subjects are abstracted from the model level and put into helper functions. When Graphwalker traverses the model, the helper functions checks whether or not the provided subject was used before. Depending on the situation, either the valid or the invalid path is taken. Thus, it is guaranteed that the correct oracle is provided for each subscription attempt as the model is traversed.

We used these features of EFSMs, which are described generally in Section 2.1, to embed test oracles into the model and thus all generated test cases have built-in assertions to test features such as subscribe, unsubscribe, publish and retrieve message in the third layer of the model. For example, as demonstrated in Figure 3.5, the model can automatically determine if a subject is used for the first time and thus can use an appropriate oracle for that situation, and similarly for a duplicate or invalid subscription. In the same way, the
evil testing that was implemented using FSMs, but where defects could only be detected if they crashed the system, was improved by using assertions. The reason is that using EFSMs, the model can keep track of whether a certain sequence is illegal or not. If it is illegal, the built-in assertion checks that the result is fail as expected.

Figure 3.5: Complete EFSM sub-model for feature subscribe. An EFSM guard is denoted by “[ guard = true | false ]”. The helper function getSubject() returns a random subject from the list of subjects.

Generating a new set of test cases

We used Graphwalker to generate 100 new abstract test cases with built-in assertions for the core features, which was lacking when using FSMs. As for the FSMs, we used one of Graphwalker’s generation algorithms, which randomly selects paths through the GMSEC model. The generated 100 new abstract test cases achieved state and transition coverage.
Mapping new abstract labels to concrete code fragments

Since new states for embedding assertions as well as transitions with guards were created after shifting from FSMs to EFSMs, we programmatically extracted those states from the model and added the corresponding code to each of the three mapping tables. The instantiator program was then used to create concrete test cases from the abstract test cases.

All other states and transitions in the model, which were not changed when shifting from FSMs to EFSMs, also remained unchanged in the table.

Executing the new set of test cases

We used the new set of 100 abstract test cases to automatically create concrete test cases for each of the three different programming languages. We then executed each of the three test suites on each of the three different middlewares through their wrappers.

Analyzing the results and effort

A summary of the results from executing the three suites of 100 test cases each is shown in Table 3.6. In addition to the detected issues related to questions 1 and 2 (exactly the same defects found earlier using the FSM models were detected since this layer of the model did not change), the table shows that many more issues were detected using EFSMs compared to using FSMs.

Sample issue related to middleware wrappers We detected an issue in the wrappers of middlewares Bolt and WebLogic in relation to Testing Question 8 (multiple messages). The issue was that an application retrieved several copies of the same message when in fact there was only one. When the test case was executed on MB, the message could correctly only be retrieved once, but when it was executed on either of the other two middlewares, the application was able to incorrectly retrieve extra copies of that same message whenever it checked for new messages. This issue is subtle and difficult to detect since, by design, applications are unaware of the number of messages that are sent to them at any given time. Thus, this issue could cause problems if the retrieving application does not check for duplicate messages. An application that was developed and tested using MB would, for example, not have problems with duplicate messages and therefore most likely would not
Table 3.6: Summary of detected issues for each testing question: EFSMs.

<table>
<thead>
<tr>
<th>ID</th>
<th>Testing Question</th>
<th>Answer</th>
<th>MWW</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Are there issues related to the connect/disconnect functionality?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Q2</td>
<td>Are there issues related to the disconnect cleanup functionality?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Q3</td>
<td>Are there issues related to an application subscribing to one or more different subjects?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Q4</td>
<td>Can a component subscribe to the same subject multiple times without unsubscribing in between?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q5</td>
<td>Can a component unsubscribe to the same subject multiple times without subscribing in between?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Q6</td>
<td>Can a component unsubscribe to a subject it was never subscribed to?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q7</td>
<td>Are there any missing messages?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Q8</td>
<td>Are there any extra messages?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Q9</td>
<td>Are there issues related to the validation of message subjects?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

check for duplicate message. However, if MB is switched for Bolt or WebLogic, which is effortless due to the built-in flexibility, problems with duplicate messages would suddenly occur without the programmer necessarily understanding why since the application passed all tests and worked without problems in the field. It should also be noted that this issue would not have been detected using FSMs, since it did not crash the system and since detecting this issue requires knowledge about previous actions performed during the model traversal. Thus this is an example of the usability and expressiveness of EFSMs in MBT.

**Sample issue related to GMSEC Java API**  We detected an issue in the GMSEC Java API in relation to Testing Question 3 (subscribing several times). The Java implementation of the API does not handle the differences between a regular subscription and a callback subscription correctly. As mentioned before, a regular subscription and a subscription with a callback to the same subject should be treated as two different subscriptions. The problem occurred in situations where a regular subscription was followed by a subscription with
a callback to the same subject. Both operations should return success. How-
however, the Java implementation incorrectly returns an error for the subscription
with callback. This issue would not have been found using FSMs based on
the same argument as for the sample issue related to middleware wrappers.

As shown in Table 3.7, the EFSM model consists of 49 states, 62 transitions, 8 requests
(method calls), and 15 assertions. Since we are able to keep track of visited states and tran-
sitions through state variables, assertions could be used to check the state of the system.
Assertions were only added to test features in the third layer. Subjects were abstracted
from the model level and helper functions were used instead.

During three weeks, the programmer worked incrementally on designing the EFSM model,
reusing the existing FSM model, as well as adding new states and transitions to the map-
ing tables. During those three weeks he also implemented the test oracles for the EFSM
model. There were 25 guards and helper functions (414 source lines of java-like code that
is executed during model traversal) implemented for the GMSEC model. After the pro-
grammer finished the upgraded model and mapping tables, a week was spent on executing
the new set of generated test cases and analyzing the results.

Table 3.7: Summary on EFSM model.

<table>
<thead>
<tr>
<th>Layer</th>
<th>States</th>
<th>Transitions</th>
<th>Requests</th>
<th>Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st layer</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2nd layer</td>
<td>8</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3rd layer</td>
<td>35</td>
<td>41</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>49</td>
<td>62</td>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 3.8: EFSM effort in the GMSEC study.

<table>
<thead>
<tr>
<th>Task</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turning FSM into EFSM models, and adding new states and transitions to the mapping tables</td>
<td>1 week</td>
</tr>
<tr>
<td>Implementing the test oracles for the EFSM model (Java-like code executed during model traversal)</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Analyzing the results from running the new set of test cases</td>
<td>1 week</td>
</tr>
</tbody>
</table>

Thus, it took the programmer another 4 weeks (14 weeks in total) to turn the FSM into
an EFSM model, generate new test cases, instantiate the new test cases for three different
programming languages, execute the new three suites of test cases on the three different
middlewares, and analyze the newly detected defects.
3.4 Lessons learned

Here we present our lessons learned from the case study of applying MBT on GM-SEC.

Some of these lessons are not necessarily unique to our work although we report them from our own experience of applying MBT, thus, supporting the empirical evidence reported in the existing literature. The lessons learned are written using "us" and "we" because they are based on our discussions with the programmer. However, the solutions as well as all other work related to this study (unless otherwise indicated) was performed solely by the programmer.

Ensure that the features to be modeled are indeed supported by the SUT

Problem: Precise API specifications are often difficult to obtain. For example, for GM-SEC, the API specification for the unsubscribe function gave the impression that the user can unsubscribe in very sophisticated ways. If one solely creates a model based on such documentation, the risk is that one spends significant effort building a model that does not match the actual capabilities of the SUT. The result would be reported issues that are false positives.

Solution: Since we based the initial model on existing test cases, we were sure that the functions that were covered by these test cases were actually implemented. For other functions, we checked our assumptions with a developer who acknowledged that the user can only unsubscribe to exactly the same pattern that was subscribed to earlier. Without this knowledge, we would have modeled a capability that was not supported and would have generated many test failures of no practical value. Thus, we recommend understanding, as early as possible, exactly what capabilities of the SUT are really supported.

EFSMs find much more defects, but at a price

The study revealed that using EFSMs results in many more detected defects than using FSMs. FSMs are limited but are lightweight and good for crash testing. The power that the EFSMs bring is a great benefit for modeling a system like GMESC. However, EFSMs require programming skills to implement the guards and helper functions. To automatically run the abstract test cases, we needed to resolve abstractions by mapping the actions of models to actions at the API level. This step requires programming skills as well (also for FSMs). There is also a certain downside in relation to model coverage when using
EFSMs. Indeed a transition in an EFSM model actually can represent many concrete actions. Taking the transitions once, for one of the actions the transition represents, will satisfy the criterion of covering all of the actions that the transition represents (e.g. for each of the subjects to be tested). This means that we can not guarantee that using a tool like Graphwalker we will cover all data parameters even if 100% transition coverage is achieved.

Thus, it does not need to take the transition again. The reason is that the transition is already covered once. This means that taking the transition once is equivalent to taking the transition multiple times (e.g. for each of the subjects to be tested) with the respect to covering the model using a MBT tool like Graphwalker.

**Non-executable mapping tables can be messy**

As described previously, we had a mapping table for each API language of GMSEC that was under test (C, C++, Java). Those mapping tables contained the code stubs for labels of the model. Although the code stubs were executable in their respective environment, the mapping tables as files were not. We used certain conventions for code stubs so that the instantiator program would be able to fetch the correct code stub for a given label. The code stub was then copy/pasted to a particular executable test case. Problem: Not being able to compile the mapping tables is not optimal, since you would like to catch code stub implementation mistakes as early as possible. Also, using the copy/paste method is not very clean since each time a label is traversed another copy of the corresponding code stub for the label is pasted.

Solution: The suggested solution, that was not adopted for this study, is to set up classes (or structs) in a given language for each mapping table. Those classes would contain a function for each model label. This approach will allow us to 1) compile the mapping tables as needed, and 2) call corresponding functions for labels instead of copy/pasting.

**A single mapping table for a large hierarchical model does not scale well**

Problem: As the model grew, it turned out to be a difficult task to maintain the mapping table for the model. Eventually, each mapping table was roughly 400 lines of code, and navigating between code stubs became confusing and took more time.

Solution: The suggested solution, that was not adopted for this study, is to have a mapping table for each sub-model of the hierarchical model. In relation to the solution suggested
above for non-executable mapping tables, that would mean having a separate class for each sub-model. This approach is much more modular, since adding or updating a sub-model would not change the classes of other sub-models. Thus, navigation between code stubs becomes much easier.

**A project management tool for MBT is desirable**

We realized during this study that having a tool to maintain and manage MBT projects would be desirable. The motivation for such a tool is twofold. Firstly, in the process of MBT, there can be many tools, programs and frameworks that are in use from models to test execution: A tool for modeling (e.g. yEd, [51]), a tool for deriving abstract test cases (e.g. Graphwalker, [27]), a program for instantiating test cases (e.g. our in-house instantiator program), as well as a test execution framework for driving the executable test cases (e.g. GMSEC test framework), etc. It is an appealing idea to have a single interface to trigger the different phases of MBT, instead of calling each tool separately. Moreover, such an interface can be designed to be agnostic from a particular test execution framework, so that it can be used for multiple MBT purposes. Secondly, it is often not clear how to maintain the different MBT artifacts, such as models, abstract test suites, executable test suites, results, etc for a project; this holds true especially for individuals who are new to MBT. Thus, having a systematic directory structure for maintaining and creating these different artifacts would be preferable.
Chapter 4

Kelevra: The MBT Project Tool

4.1 About Kelevra

Based on the motivation and identified need for a MBT project tool, presented in Section 3.4 devoted to the NASA GMSEC case study, we decided to implement such a tool. The tool, which is called Kelevra, is a simple command-line interface written in Python. The goal for the tool is that both inexperienced and experienced individuals in MBT can use it as a unified interface to trigger the various phases of MBT. Maintaining dozens of models or hundreds of test cases can be a hard task without a well-defined project structure. Thus, another goal was to manage the files and folders of MBT projects, instead of the user having to do so manually.

The tool is based on a couple of assumptions. We assume that users use yEd for modeling and Graphwalker for model traversing. However, the tool is agnostic from any particular test execution framework. Hence, it is designed in such way that anyone can write a new test driver for a test execution framework and use it as a plugin to Kelevra. As an example, we could write a test driver for the NASA GMSEC test execution framework and plug it into Kelevra. Currently, four test drivers have been implemented and come with the tool. Those test drivers support the following test execution frameworks; UIAutomator by Android [3], Py-UIAutomator by Xiao Cong [50], Appium by Saucelabs [41] and the well known Selenium [42]. These drivers are for mobile and browser test automation, and it is worth noting that all of them are open-source. These particular drivers were implemented since the next case study in this research is a case study where the system under test is a mobile system, namely, applying MBT on QuizUp’s Android client.
4.2 Features

The set of features can be divided into three sets; Project, Driver and MBT. The project and driver features are mostly used for significant changes of MBT projects, while the MBT features are used to trigger various phases of MBT.

Project

List projects Lists all Kelevra projects (MBT projects that are attached to Kelevra).

Fork Copies a given Kelevra project to a target project (references and files copied).

Create Creates an MBT project under path directory and attaches project to Kelevra. The project contains a demo model and a configuration file.

Attach Attaches an MBT project to Kelevra. A reference is created for the project in Kelevra’s config.

Delete Deletes a given MBT project completely. Files and folders are deleted and the project is detached from Kelevra.

Detach Detaches a given MBT project from Kelevra. The reference is removed from the project in Kelevra’s configuration file.

Driver

Create-driver A given driver is added to a particular project and a reference for the driver is created in Kelevra’s configuration file. Kelevra copies a demo setup for the chosen driver to the project executable directory.

Attach-driver Attaches a given driver to an MBT project. A reference for the driver is created for the project in Kelevra’s configuration file.

Delete-driver Deletes a given driver from a project and the reference for the driver is removed from Kelevra’s configuration file.

Detach-driver Detaches a given driver from an MBT project. A reference for the driver is removed from the project in Kelevra’s configuration file.
**MBT**

**Merge**  
Merges the graphml files (i.e. the yEd models) in models folder to a single graphml file and writes to a target file (stored in target folder).

**Update**  
Updates the mapping table for a particular test driver. Checks for all states and edges whether a corresponding function exists in the drivers mapping table. Otherwise, a code stub for that state or edge is created.

**Generate**  
Generates offline test sequences from a given model (using Graphwalker).

**Instantiate**  
Maps an abstract test suite to a test suite that can be executed using an implemented test driver of choice.

**Build**  
Builds, or compiles, an executable test suite. Some drivers (e.g. languages with compiled implementation) require this step. Other drivers, such as drivers implemented in Python (e.g. languages with interpreted implementation) do not require this step.

**Run**  
Runs a chosen test suite on a system-under-test using an implemented test driver of choice.
4.3 How to use Kelevra

In this section we will explain how one can use Kelevra. We will go through the key steps of MBT for a sample project. Since some of the commands (provided by Kelevra) generate artifacts (e.g. test cases), we also demonstrate how the project is constructed and where particular artifacts are stored.

Let us create a mobile MBT project called "TestProject" at a specified path.

```
kelevra create TestProject -p /home/User/MBT/ -t mobile
```

By defining the project as a mobile project, a sample model for the Android Clock application, which is installed on all Android devices, is created. Defining the project as a browser project will create a sample model for the Amazon website. The idea is that users can either apply MBT on the sample SUTs, or use the models to get started for their own testing purposes. Additionally, a configuration file for the project is generated where users can manipulate basic data, such as paths and sub-paths for identifying artifacts in the project (e.g. model suite).

Let us assume that we wish to build a hierarchical model for a SUT other than the sample ones. We construct a model with two layers, Layer1 and Layer2. The tool provides a command for merging the layers of the model to a flat model. The flat model will be used later on by Graphwalker to derive abstract test cases. Note that we don’t use the flat model for any other purpose.

```
kelevra merge TestProject -w MainModel.graphml
```

Now that we have a model that conceptually represents a particular SUT, we would like add a test driver that will allow us to execute test cases for the SUT. For this example, we will use Appium as a test driver, which can execute test cases for mobile applications.
The tool generates an Appium sub-folder where a sample mapping table for the Clock Android application is provided ("MappingTable.py"). To semi-automate the effort of modifying the mapping table (since we are not testing the Clock application) we can use the update command of Kelevra. The command creates code fragment stubs for each state and transition of the model, using Graphwalker to retrieve the labels. We can then manually implement each code fragment.

```
kelevra update TestProject -d appium
```

Now we can generate abstract test cases from the model. Kelevra generates a test suite called "MainSuite", using Graphwalker, which consists of 10 abstract test cases, where each test case achieves 100% transition (edge) coverage by a random walk.

```
kelevra generate TestProject -r Home.graphml \ 
-s edge_coverage:100 -g random -t MainSuite -n 10
```
Now that we have an abstract test suite, we can instantiate those test cases to concrete executable test cases.

```
kelevra instantiate TestProject -t TestSuite -d appium
```

The tool generates the executable test cases and stores them in a sub-folder of the Appium folder.

```
/TestProject
  | abstract
  |   | MainSuite
  |   |   | Testcase1.abs
  |   |   : ...
  |   |   | Testcase10.abs
  | models
  |   | Layer1.graphml
  |   | Layer2.graphml
  | targets
  |   | MainModel.graphml
  | executable
  |   | appium
  |   |   | assets
  |   |   | MappingTable.py
  |   | testsuites
  |   |   | MainSuite
  |   |   |   | Testcase1.py
  |   |   |   : ...
  |   |   |   | Testcase10.py
  |   | options.cfg
```

Prior to running the test cases in the "MainSuite" test suite, one has to have set up the framework that is under test or is used during the execution. In this example, we will run the tests for a mobile application. Therefore, we have to set up either the Android or iOS platform (or both), as well as other prerequisites. A documentation for the prerequisites for each test driver is provided with the tool. We also have to make sure that either a mobile device or an emulator is set up.

Now that we have set up for mobile testing, we can execute the test suite.

```
kelevra run TestProject -t TestSuite -d appium
```

After the tests have run, we analyze the test cases, especially the ones that fail. Currently, there is no additional support for analyzing test results other than reading the logs from the command-line.
Now we have successfully applied MBT on a SUT, from models to test execution. We could continue the effort by incrementally updating the model and mapping table, generating new test cases and executing them on the SUT.
Chapter 5

A Mobile Case Study Using Kelevra: QuizUp

5.1 About QuizUp

QuizUp is a mobile trivia game that allows its users to challenge each other on several hundred topics (e.g. arts, science, sports) using almost a quarter million questions [29]. The users, who are people from all over the world, participate in a social experience by communicating and competing against friends or strangers in real-time trivia. The application, initially released for the iOS platform, turned out to be a overnight success [4]. To this day, QuizUp is still the fastest growing mobile game in history, with over 3.5 million users having registered in the first three weeks\(^1\). Within a year since its initial release, QuizUp released an Android client of the application [21], and massed to over 20 million users combined for the two platforms (Android and iOS).

Testing the application is a complex task due to its data-driven design, its built-in complexity and the business flexibility it provides. The data-driven design is embodied by the real-time dependency of large amount of data to display. The application communicates with the QuizUp servers, through web services, to fetch data (i.e. HTTP GET query) from the QuizUp databases, as well as posting new or updated data to the databases (i.e. HTTP POST query). For most of the scenes in the application, a significant portion of the data being displayed depends on data from the QuizUp databases. Thus, as testers, we can only control data, or the order of data, beforehand in limited scenarios assuming we do not have any authority over the databases. QuizUp is a user-based application, where users have to authenticate themselves prior to accessing the core features of the

\(^1\) http://nymag.com/daily/intelligencer/2013/12/quizups-growth-secrets.html
application. Therefore, much of the data displayed is user-dependent. Data, such as profile information, game history and message inbox, are examples of user-dependent data, which is unique for each user.

The built-in complexity is largely embedded in the game-play scenes of the game. A user can compete against other users (a.k.a. opponents) in any of QuizUp’s topics. After the user has requested to play a trivia game on a particular topic, the QuizUp servers search, in real-time, for an opponent who has also requested to play that particular topic at that particular moment. An alternative for playing a random user on a topic is to challenge a specific user, or stranger, on a topic of choice. Depending on whether or not the opponent is active when the challenge arises, the game can either occur synchronously or asynchronously. The challenger can start the game without the opponent being present, or wait until the opponent is present. If the opponent accepts a challenge, where the challenger has already played the game, the previously recorded performance of the challenger is simulated when the opponent plays that particular game.

The most substantial example of the application’s business flexibility is the availability of multiple mobile platforms. The Android and iOS clients are native clients implemented by separate teams. The Android client is implemented in Java while the iOS client is implemented in Objective-C. Nevertheless, the clients should follow the same set of business rules and requirements, and therefore behave conceptually in the same way. In addition, QuizUp also has support for multiple natural languages. Currently, the application supports English, Spanish, German, French and Portuguese.

Due to the above mentioned testing challenges, the QuizUp team conducts large amounts of testing. The team has been able to address ambitious testing goals, resulting in rapid and astounding success of the application, but with a high cost and a very substantial effort in thorough Quality Assurance (QA). The testing effort can be divided into several categories. The QuizUp team has developed a set of executable test cases, written in Calabash [10], that address some of the testing goals. However, the test suite is manually written and limited to short and common scenarios for the iOS client, and not for the Android client. The team has no automated tests for the Android client; instead it manages a beta group for Android users that provides rapid feedback prior to new releases. The team also includes five QA members who constantly verify that new versions and updates of the application meet its business rules and requirements through manual regression testing. In addition to the QA members, the team outsources a large set of end-user acceptance tests to a contractor company that assists them with QA. Thus, the overall testing effort is significant.
Due to the fact that the QuizUp team is interested in improving their testing processes through automation, the primary goal of the study reported in this chapter was to study the feasibility of using MBT on the QuizUp application. Similarly to the GMSEC team, the QuizUp team was interested in understanding the learning curve related to MBT for practitioners (who do not hold a doctoral degree). This interest was sparked by the possibility of the team maintaining the MBT models and infrastructure themselves after the study. However, the learning curve for applying MBT on a mobile application, such as QuizUp, was unclear. Thus, another goal of the study was to clarify the needed effort.

A major question for the study was: Can the QuizUp application be modeled in such way that test cases can be automatically generated to test the core features of the application? Another question was: Can we design the MBT approach in such way that it can test both the Android and iOS client without modifying the model? Since the two clients should represent the "same" QuizUp application, it was desirable to maintain a single QuizUp model instead of maintaining two separate models, which is typically the case in similar situations.

Since the QuizUp team has thorough testing processes, we consider the MBT approach to be successful if it were able to detect non-trivial issues.

The scope of the study

For this study we decided to test the Android client of QuizUp through its GUI. Although we did not test the iOS client, the implemented testing approach was designed in such way that the iOS client testing effort could be easily integrated in future work. We chose the Android client over the iOS client because it was more accessible for test automation due to Apple’s hardware restrictions, and due to the fact that the QuizUp team did not have any existing automated tests for the Android client.

The derived tests were run on mobile emulators using the Genymotion Android emulator [18]. Although current emulator technology supports the emulation of the physical state of mobile devices, such as network reception, sensors, battery status, etc., we did not test for those activities and events.

The study was mainly performed at QuizUp’s facilities. Questions were asked to the QuizUp team during the process when the documentation was not specific enough or ambiguous. Apart from that, the effort was carried out independently of the QuizUp team. The findings were reported to the QuizUp team during and after the study.

2 Apple requires automated tests for iOS applications to run on Mac OS X systems. We did not have such a system in this study.
5.2 Core features of the QuizUp application

Based on the application's documentation, as well as conversations with the QuizUp team, we extracted the core features of the QuizUp application to be tested. The application is divided into various scenes, which can be further divided into two categories: access scenes and in-game scenes. The access scenes, depicted in Figure 5.1, are scenes for identifying the user, through log-in or sign-up, prior to using the in-game scenes of the application.

Below, we provide light-weight descriptions of the core features. It should be noted that most scenes in the QuizUp application contain sub-scenes, which might even contain sub-scenes of their own, handling specific functionality within a particular scene. Not all sub-scenes of the core features will be described specifically.

Access scenes

The Welcome scene, given in Figure 5.1a, is the opening scene that is shown after starting the application. Since QuizUp is a social application, it offers users to use their existing accounts at either Google or Facebook to log in. In order to log in to QuizUp using these accounts, a user has to be logged into Google or Facebook, respectively, on the
mobile device in use. If a user is not logged in, he will be prompted through the QuizUp application to log in to his account at Google or Facebook.

A user can log in via an email account that has previously been signed up. By pressing the "Login now!" button in the Welcome scene, the user is brought to the Email Log-in scene, in Figure 5.1b, where a user can input his credentials (email and password). Inputting invalid information, such as a badly formed email address or an invalid password, will result in the application returning an appropriate error message that describes the fault. If a user has forgotten his password, he can enter the Forgot Password sub-scene, from the Email Log-in scene, to retrieve it.

A user, who does not have a pre-existing account, can sign up to QuizUp via email. By pressing the "Sign up with email" button in the Welcome scene, the user is led to the Email Sign-Up scene, as shown in Figure 5.1c. The user has to provide general information (e.g. display name, birthday), as well as an email and password of choice in a later phase of the scene (not shown in Figure 5.1c). Entering invalid information, such as an email that is already in use, will result in the application returning an error message.

Although QuizUp provides these different alternatives for accessing the application, it should be conceptually the same for all types of users.
In-game scenes

After a user has successfully logged in, he is brought to the Home scene of QuizUp. The Home scene, in Figure 5.2a, provides an overview of various aspects of the game such as topics-of-the-day (e.g. Naruto), challenges from other users (e.g. Jokes from Cherry123), and the user’s favorite topics (e.g. Entrepreneurs). The displayed data is mostly data that is fetched using the QuizUp servers. The topics-of-the-day category, for example, displays different data daily, while the user’s favorite topics are calculated server-side based on the user’s history of games played.

The user can, in most circumstances, press the sidebar button in the top-right corner to navigate between in-game scenes. The sidebar, in Figure 5.2b, displays a list of the following key scenes of QuizUp: Profile, Home, Topics, Friends, History, Messages, Discussions, Achievements, Store, and Settings. The displayed names of scenes in the list are all hard-coded except for the Profile scene. The user’s name and title are displayed instead (the top element of the list). Note that if the sidebar button is not available from some scene in the application, the user can always navigate back by pressing the Android back button until he arrives at a scene where the sidebar button is available. For this study we did not consider the Friends scene, the Discussions scene, the Achievements scene and the Store scene as core features of the application.

The Profile scene, in Figure 5.2c, displays user-dependent data such as the user’s name, title, country, friends and achievements. The user can change some the data in the profile tab of the Settings scene. If he does so, the changes should be present in the Profile scene. A shortcut button is available for accessing the profile tab of the Settings scene, in the top right corner below the sidebar button, instead of using the sidebar to navigate between the scenes. Additionally, the scene allows the user to examine his in-game statistics such as total wins, draws and losses, as well as favorite topics (not shown in figures).

The Topics scene, in Figure 5.3a, displays all categories of topics currently available in the game, where each category contains dozens of topics. Choosing a category (e.g. Educational) will display a list, similar to the list of categories, of the corresponding topics for the category (not shown in any figure). Choosing a topic (e.g. Math:General) will result in a sub-list for the topic being displayed where the user can access the Gameplay scene to play a trivia against a random user in the chosen topic. Both the list of categories and each category’s list of topics are data fetched from the QuizUp servers before the scene is loaded. An alternative for choosing a category and then a topic is to search for a topic directly by inputting the name of a particular topic using the search bar.
located at the top. The application then filters the displayed list of topics by the results of the search.

When the Game-play scene is accessed from the Topics scene, the user is matched up against an opponent (not shown in figures). The matched opponent will be a random QuizUp user provided by the QuizUp servers. After a user has been matched with an opponent, the scene displays the first question out of the seven questions to be displayed in the trivia game. The questions are fetched in real-time using the QuizUp servers. As shown in Figure 5.3b, each question has four available answers where only one answer is correct. Each user gets at most ten seconds to answer each question. At the top of the screen, the aggregated score for each user in a particular game is displayed. During any of the seven rounds of questions, the user can surrender resulting in the early termination of the game. When a game has finished, the scene moves to a statistical screen about the played game, shown in Figure 5.3c. Scores are calculated based on the results from the game and reported using the QuizUp servers.

The History scene, in Figure 5.4a, displays a list of the user’s history of played games against opponents. The list should be ordered based on when the games were played, such that the last played game is at the top of the list. The history is fetched from the QuizUp servers.

The Messages scene, in Figure 5.4b, displays a list of the user’s conversations with other QuizUp users. Choosing a particular conversation (e.g. conversation with Jeffrey) will
lead to QuizUp’s Chat scene for that conversation. In the Chat scene, the user can view each message in the conversation, as well as send a new message.

The Settings scene contains a Settings tab and a Profile tab. The Settings tab (not shown in figures) displays different configurations that the user can control, such as the natural language chosen, as well the option to log out of the application. The Profile tab, in Figure 5.4c, displays editable fields for the user-dependent data also displayed in the Profile scene. If the user updates his information, the update should be noted globally within the application and reported using the QuizUp servers.

Rules of QuizUp

In addition to the functionality of each scene, there are also a number of other behaviors related to the sequence of actions and the navigation between scenes that are important to describe as rules because they partially dictate the type of testing that needs to be performed.

- First-time log-in: Users logging in for the first time should see introductory information displayed in the Messages and History scenes.
- Clean up upon logging out: When a user logs out through the Settings scene, the session data should be cleaned up. That means that there should be no evidence of the user after the session.
• Profile changes should be recognized in all relevant scenes: Changing profile information, such as the user's name or title, should enforce an update in all other scenes that display that particular data.

• Games played should be stored in the History scene: After playing a game against an opponent, a record of that particular game should be stored in the History scene where the user can view information and statistics about that game.

• Sent messages in Chat conversation should be stored in the Messages scene: A message history with a particular opponent can be accessed through multiple different scenes in the application. The conversation should always be up to date independently from which scene the conversation was entered.

• The scene prior to entering the Settings scene matters: The Settings scene has two tabs, a settings tab and a profile tab. Which tab should be set depends on the scene prior to entering the Settings scene. The profile tab should be set when entering the Settings scene by using the shortcut button in the Profile scene. Entering the Settings scene from any scene other than the Profile scene and the Settings scene itself, the settings tab should be set. Attempts to enter the Settings scene while the Settings scene is open should be ignored.

5.2.1 Testing questions

From the above mentioned scene descriptions and rules, we derive the following testing questions. Together, the rules and the testing questions are mostly specific to QuizUp. However, some of the general characteristics (e.g. log-in, sidebar) of the application can apply to other mobile applications.

• Q1. Are there issues related to the access scenes displaying correct error messages when invalid information is input?

• Q2. Are there issues related to the log-out clean-up functionality?

• Q3. Are there issues related to displaying the correct scene headers for any given scene?

• Q4. Are there issues related to the question phase of the Game-play scene?

• Q5. Are there issues with the information (scores and statistics) shown in the Game-play scene after playing a game?

• Q6. Does the History scene at any point show an outdated history?
Q7. Are there issues related to stored messages in the Messages scene?
Q8. Are there issues related to updating user information?
Q9. Are there issues with navigating to the correct tab in the Settings scene?

In the remainder of this chapter we will describe how we used MBT to answer the above questions for the QuizUp application.

5.3 Applying MBT on QuizUp

Modeling of the QuizUp application was driven by the documentation of the core features of the application, the associated rules, and the derived testing questions. The model was then used to generate abstract test cases, which were then automatically translated into a set of concrete test cases that were executed on QuizUp’s Android client.

After our experience in the GMSEC study, we decided to conduct MBT using EFSMs as the notation to represent the model. After examining the core features of QuizUp we learned that implementing test oracles using EFSMs should be preferred as opposed to explicitly modeling finite scenarios using FSMs. The reason is that modeling using FSMs would require a large number of states to be added in order to address the test oracles for QuizUp’s behaviors, which would contradict our goal of generating random interaction sequences.

Over the last few years several tools have evolved that offer different mechanisms for UI test automation for mobile devices, each with certain advantages and disadvantages. For this study, we used the Appium UI test automation tool [41]. A description of the tool and how we use it are provided in more detail below.

We used the Kelevra command-line tool, which is a part of the thesis work and was implemented prior to this study, to trigger the phases of MBT. We recorded the detected defects and effort. The process and the collected data are discussed in more detail below.

The Appium UI test automation tool

We chose the Appium UI test automation tool, or test driver, because 1) it is cross-platform and therefore can be used for both Android and iOS test automation, 2) it is open-source, and 3) it is well documented and, 4) has a very active community [41]. Appium extends the well-known Webdriver JSON wire protocol specified by Selenium [43]. Tests can be written in any language that has a WebDriver library. Languages such as Ruby, Python,
Java, JavaScript, PHP, and C# all include a WebDriver library. For this study we chose to use Appium’s Python implementation of the Webdriver library.

At its heart, Appium is a web server that exposes a REST API. It receives connections from a client, listens for commands, executes those commands on a mobile device, and responds with an HTTP response representing the result of the command execution. At the lowest level, Appium uses the UI automation tools from the Android and iOS SDK’s to inject events and perform UI element inspection. Namely, the Android UIAutomator and iOS UIAutomation tools [3, 5].

Using the Appium library, UI element objects can be retrieved by UI inspection. Gestures, clicks and keyboard inputs are examples of methods that can then be applied to the retrieved UI element objects.

The modeling goal

The primary modeling goal was to design the QuizUp model in such way that the derived test cases would answer the testing questions and determine whether or not QuizUp is behaviorally consistent with the requirements of the application. The goal was also for the test cases to represent realistic, but possibly unusual, interaction sequences. Note that we did not model attempts to access in-game scenes without being logged into the application as a user. We were aware that Appium (the test driver in use) would crash when attempting to interact with UI elements that would not be present.

A second modeling goal was to design the model in such way that the derived tests from the model could run on QuizUp’s production server, or similar. The production server hosts the live version of the application that global users can access. That means that we, as testers, do not have full control of the data space. New users can emerge, the QuizUp team can update the list of available topics, and messages can arise randomly from real users. Thus, we would have to design the model in such way that the implemented test code in the mapping table would not be dependent on specific data in the application, but rather implemented to identify types of data and select elements dynamically. An alternative for a production server would be a closed server where we could have full control of the data, as well as being able to inject (or update) data. That would require having only test users and not real users. However, such server was not in place for this study.

The third modeling goal was to handle users with different levels of maturity. That is, the derived tests should not be dependent on a particular test user or his status in the game. An arbitrary test user could therefore vary from being a new user who just finished signing
up to being an advanced user who has played hundreds of games. For this purpose, a service called QTDS (QuizUp Test Data Service) was implemented. The service, which is currently very minimal, stores the credentials and key information (e.g. name, title, country) of different test users in a JSON file.

**The QuizUp model as hierarchical EFSMs**

We modeled the QuizUp application as a collection of EFSMs that were structured in a hierarchical fashion using five layers to manage complexity. It is worth mentioning that some scenes in the application can be accessed as sub-scenes from many different scenes. For example, the Game-play scene can be accessed from scenes such as the Home scene, Topics scene and more. Thus, determining the layer depth of the Game-play scene itself can vary depending on which scene provides access to it. That being said, five layers to manage complexity means that the model is of at most five layers for any sequence in the QuizUp model. Additionally to the rules for state colors in Section 2.3, a green state is an entry state from the 3rd layer or lower to the 4th layer or lower. That is, all entry states for layers lower than the 3rd layer are marked as a green state because of the fact that some scenes can vary in layer depth.

The highest layer (see Figure 5.5) of the model is concentrated on using the Email Log-in or Email Sign-up scenes before entering the application’s In-game scenes, where each state for these scenes serves as an entry state to the second layer of the model. The design, thus, explicitly ensures that the "dummy" user (the test case) has logged in prior to using any of the core in-game features. Unusual but valid sequences, such as repeatedly entering and exiting these scenes are possible outcomes from a random traversal of this model. We chose not to model the options of logging in using a Google or a Facebook account in this study because that required maintaining such dummy "users" as well. However, since there is no conceptual difference between any of QuizUp’s log-in options regarding the In-game scenes of the application, adding support for Google or Facebook accounts requires only small additions to our model. The goal is to add support for them in the near future.

As a result of how we designed the highest layer of the model, the second layer of the model is both concentrated on the step-by-step actions of logging in or signing up, as well as accessing the core features of the in-game scenes. The second-layer model for the Email Log-in scene is shown in Figure 5.6. The model is designed in such way that it explores realistic, as well as unusual, sequences for inputting the users credentials (email and password) in order to access the application. Prior to inputting the credentials, a
"dummy" test user is retrieved from the QTDS service. Additionally, the user maturity level is logged by using a helper function, since some in-game scenes (e.g. Messages scene) display different information depending on the maturity of the user. The design of the model permits evil testing (testing of invalid sequences). Based on the documentation, we categorized the types of emails and passwords that a user can input. There are three types of emails to input; a valid email, an email for a non-existing user (invalid), and a badly formed email (invalid). There are two types of passwords to input; a valid password and an invalid password. The model permits any combinations for inputting these different types of emails and passwords. Each transition for these email and password types has an associated helper function that helps us identify which combination was traversed. We are then able to embed the appropriate test oracle for validating the traversed combination. The "validEmailLogin" guard only returns true if both a valid email and a valid password provided as input. Otherwise, the guard returns false. The "invalidEmailLogin" guard then allows us to understand which type of error message we should expect to be displayed depending on the type of invalidity of the input combination.

The second-layer model for the Email Sign-up scene, which will not be described specifically, has a similar structure and emphasis as the model for the Email Log-in scene.

The second-layer model for the in-game scenes of the application is shown in Figure 5.7. The model includes entry states for the Profile, Home, Topics, History, Messages and Settings scenes, as well as a state for the application’s sidebar. Each of the entry states has a sub-model (in the third layer of the model). Therefore, the model is designed as an intermediate layer for navigating between different in-game scenes using the sidebar. When the "dummy" user is led to a new scene during the traversal, the helper function "Scene" is used to add that particular scene to an array which contains the history of traversed scenes for a particular traversal. The function allows us to later embed assertions into our model that rely on the history. As described in Section 5.2, a user can log-out of
Figure 5.6: Sub-model for the Email Log-in scene in the second layer of the QuizUp model.

the application from the Settings scene. Thus, a transition is available from the Settings scene to log-out that will bring the user back to the Welcome scene (in the first layer of the model).

Ensuring that the model is not dependent on specific data enforces a certain design. For example, to test the Topics scene (in Figure 5.8a) requires choosing a category and then a particular topic within that category (in order to access the Game-play scene). Modeling each category as a separate transition would require a significant maintenance effort because each added category to test would result in an additional transition. It would also be a fragile approach because if one of the categories under test were removed by the QuizUp team, the derived tests that take that transition would fail. Therefore, we only have a single transition to represent the action of choosing any of the displayed categories, as opposed to having a separate transition for each category to test. The assumption is that the implemented code fragment for the transition will identify all displayed categories and select one from the list. Thus, the Topics sub-model is very abstract and simple.

Using EFSMs we were able to control the flow of the model traversal through the Game-play scene, as shown in Figure 5.8. During a traversal, we ensure using a helper variable ("round") and a guard that the traversal will loop for seven rounds of questions. In addition, the model explicitly gives the "dummy" user an option to surrender the game while
Figure 5.7: Sub-model for the In-game scenes in the second layer of the QuizUp model.

playing in any of the seven rounds. However, to ensure that surrendering of games would not occur often, we used *Graphwalker’s* weight option so that the transition only has a 5% chance of being taken each time it is available.

**Generate abstract test cases**

We used Kelevra’s `generate` command, which interfaces with *Graphwalker*, to generate 100 abstract test cases with built-in assertions for the core features. Kelevra was configured in such way that these test cases achieved 100% state and transition coverage using *Graphwalker’s* random-path generation algorithm.

**Mapping abstract labels to concrete Appium code fragments**

We used Appium’s Python library to communicate with the QuizUp application. The code fragments for each label in the QuizUp model were manually written. However, we used Kelevra’s `update` command to semi-automate the effort. The command tracks whether new states or transitions have been added to the model, and if so, generates a template code fragment for new labels.
We learned from the GMSEC case study that non-executable mapping tables can be messy and also that maintaining a single mapping table becomes challenging when the model size increases (see Section 3.4). Thus, we decided for this study to implement a class structure which would allow us to have a separate mapping table for each scene. Each scene class inherits a scene base class. The base class includes generic code fragments for actions accessible from all scenes (e.g. pressing the Android back button). Therefore, it was unnecessary to implement code fragments for such actions in each scene class.

**Code Example 5.1** Sample code fragments in the InGameScenes class

```python
class InGameScenes(Scene):
    def Sidebar(self):
        assertTrue(self.get("userName").text == Scene.state.user.name)
        assertTrue(self.get("userTitle").text == Scene.state.user.title)

        # more assertions for the sidebar list ...

    def openTopics(self):
        self.get("topicsSidebarBtn").click()

        # more code fragments ...
```
The base class also includes an object, called the state object, that the scene classes can access to fetch shared test data. The state object was vital in order to ensure that the QuizUp model is not dependant on specific data. For example, when a "dummy" user is retrieved using the QTDS service prior to logging in via email, the user's credentials and information will be stored in a user sub-object of the state object. This allows us to embed assertions in other scenes that are dependent on the data. When opening the sidebar, for example, the user name and title should be displayed for navigating to the Profile scene. Thus, the code fragment for the "Sidebar" state, in Code Example 5.1, includes assertions that check that the text values of those UI elements are equal to the values stored in the user sub-object of the state object.

Although Appium is a cross-platform test driver, the identification of UI elements is native for Android and iOS, respectively, because they have different UI layout architecture. Thus, different Appium methods are used to identify Android UI elements as opposed to identifying iOS UI elements. To minimize complexity in the mapping tables, we abstracted the actual Appium method calls for identifying UI elements to a simpler method called "get". The function retrieves necessary information for identifying a particular UI element, in our case for Android, from the UIObjects class (see Code Example 5.2).

**Code Example 5.2** Sample UI element variable in the UIObjects class

```python
class UIObjects:
    userName = {
        "Android" : {
            "id": "uiautomator",
            "method": "resourceId",
            "argument": "com.quizup.core:id/display_name"
        }
    }
    # iOS id and type could be added here for the button ...

    # more UI element identifications ...
```

The UIObjects class, which is a part of the implemented class structure, stores values for each of the UI elements being used in the code fragments in any of the mapping tables. The values (e.g. uiautomator) specify what type of a Appium method call will be used. The "get" function then returns the correct Appium method call to identify the UI element passed as an argument (see Code Example 5.3).
By implementing the class structure in this way we are able to maintain the same mapping table test code for both the Android and iOS clients of QuizUp. The reason is that we are able to fully utilize Appium as a cross-platform test driver by providing the abstraction for identifying UI elements. Thus, if we were to test the iOS client in future work, we would simply need to add values for iOS UI element identification in the UIObjects class for all UI elements used.

**Executing the concrete test cases**

Using Kelevra’s `instantiate` command we were able to automatically create concrete test cases from the set of 100 abstract test cases. We then used Kelevra’s `run` command with Appium configured as an argument to execute the test suite on two different Genymotion emulators. The Google Nexus 5 and Samsung Galaxy S4 devices were the chosen devices to emulate.

**Analyzing the results and effort**

A summary of the results from executing the concrete test cases is shown in Table 5.1. In Column 3, an answer to each testing question is provided. In Column 4, the issues that can be traced back to badly formed or missing UI elements are provided. In Column 5, issues that can be traced back to incorrect data are provided. The value in a particular cell indicates whether or not we detected issues related to a particular testing question. The table shows that some issues were detected using EFSMs.

**Sample issue related to updating user information** We detected an issue in the key scene list in QuizUp’s sidebar in relation to Testing Question 7. The issue originated in the Settings scene (Profile tab) when the "dummy" user updated his name. The application should update the name in all the scenes in which it occurs after the change. However, the name which is displayed in the sidebar list (e.g. magic36) did not update when the sidebar was entered shortly after, causing the assertion that validates the name to fail. The assertion, which is shown in Code Example 1, failed because the test data was

---

**Code Example 5.3** Sample Appium method call returned by the "get" function

```python
driver.find_element_by_android_uiautomator(
    'new UiSelector().resourceId("com.quizup.core:id/display_name")'
)
```
Table 5.1: Summary of detected issues in QuizUp for each testing question.

<table>
<thead>
<tr>
<th>ID</th>
<th>Testing Question</th>
<th>Answer</th>
<th>UI</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Are there issues related to the access scenes display correct error messages when invalid information is inputted?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q2</td>
<td>Are there issues related to log-out cleanup functionality?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q3</td>
<td>Are there issues related to displaying the correct scene headers for any given scene?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Q4</td>
<td>Are there issues related to the question-phase of the Game-play scene?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q5</td>
<td>Are there issues with the information (scores and statistics) shown in the Game-play scene after playing a game?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q6</td>
<td>Does the History scene at some point show an outdated history?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q7</td>
<td>Are there issues related to stored messages in the Messages scene?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q8</td>
<td>Are there issues related to updating user information?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Q9</td>
<td>Are there issues with navigating to the correct tab in the Settings scene?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

updated correctly, but the actual value of the UI element for the user’s name in the sidebar list did not.

As shown in Table 5.2, the EFSM model for QuizUp consists of 163 states, 241 transitions, 262 requests (method calls), and 145 assertions. The reason for the relevant difference between the number of transitions and the number of states is because actions such as opening the sidebar and navigating back using the Android back button were accessible from a majority of the states. The high number of assertions is due to the fact that we were able to embed appropriate test oracles using EFSMs.

The setting-up effort before modeling and testing the QuizUp application is shown in Table 5.3. We had learned MBT in the GMSEC study and, thus, knew MBT when this study started. We had also become acquainted with the Graphwalker tool from the GMSEC study. Therefore, no particular effort was recorded for getting to know MBT and the Graphwalker tool for this study. However, we were was applying MBT on a mobile application for the first time. We had to familiarize ourselves with the Appium test driver
Table 5.2: Summary on the QuizUp EFSM model.

<table>
<thead>
<tr>
<th>Layer</th>
<th>States</th>
<th>Transitions</th>
<th>Requests</th>
<th>Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st layer</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2nd layer</td>
<td>35</td>
<td>53</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>3rd layer</td>
<td>71</td>
<td>102</td>
<td>99</td>
<td>63</td>
</tr>
<tr>
<td>4 &amp; 5 layer</td>
<td>48</td>
<td>77</td>
<td>122</td>
<td>71</td>
</tr>
<tr>
<td>Total</td>
<td>163</td>
<td>241</td>
<td>262</td>
<td>145</td>
</tr>
</tbody>
</table>

for the first time. Prior to choosing Appium as a test driver, we also tried out other options, such as using the Android UIAutomator directly. We also set up the Genymotion emulators appropriately. Finally, we had to implement a test driver for Appium which could then be plugged into Kelevra.

Table 5.3: Setup effort in the QuizUp study.

<table>
<thead>
<tr>
<th>Task</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting to know several UI test automation tools (including Appium) and setting up the Genymotion emulators</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Implementing the Appium test driver for Kelevra, customized for QuizUp</td>
<td>1 week</td>
</tr>
</tbody>
</table>

Please note that two of the three weeks of set-up would have been the same if the goal was not MBT but simply test automation.

Table 5.4: EFSM effort in the QuizUp study.

<table>
<thead>
<tr>
<th>Task</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designing the EFSM model and mapping tables for QuizUp</td>
<td>6 weeks</td>
</tr>
<tr>
<td>Executing test cases and analyzing the results</td>
<td>1 week</td>
</tr>
</tbody>
</table>

We then spent time understanding QuizUp, incrementally designing the QuizUp EFSM model, and maintained the mapping table as states and transitions were added. During those five weeks we also implemented the test oracles for the EFSM model. There were 15 guards and helper functions (112 source lines of java-like code that is executed during model traversal) implemented for the QuizUp model. After we finished the model and mapping table, a week was spent on executing the generated test cases and analyzing the results.
Thus, it took 10 weeks for a programmer who was familiar with MBT (but not with mobile applications) to create a significant EFSM model, generate test cases, instantiate them, execute on two different emulators, and analyze the detected defects.

5.4 Lessons learned

Here we present our lessons learned from the case study of applying MBT on the QuizUp application.

Similarly to the lessons learned from the GMSEC study, some of these lessons are not necessarily unique to our work although we report them from our own experience of applying MBT, thus, supporting the empirical evidence reported in the existing literature.

QuizUp is very well tested

The constructed model and the overall testing effort for the QuizUp study was quite significant. The case study demonstrates that it is feasible to use MBT for a system like QuizUp since we were able to detect issues in the application. Overall, the effort showed that the application is very reliable. The results were not surprising since the QuizUp team conducts a lot of testing from different aspects and the system is in use. The addition of MBT to the QuizUp testing effort is of great value for the team since it allows for faster and more extensive validation.

MBT can be cross-platform for mobile applications

The UI element identification abstraction enabled us to fully utilize Appium as a cross-platform test driver. Maintaining only a single instance of the mapping tables for both Android and iOS test automation would save a lot of effort for the QuizUp team. The mapping table test code also became much simpler and readable using a wrapper function (the "get" function) to hide all of the different Appium methods in use. According to the QuizUp team, they felt much more confident writing test code using the implemented class structure rather than using the Appium methods directly.
Evaluating the correctness of UI element identification can be hard

Problem: Although we were able to implement executable mapping tables (as suggested after the GMSEC study), we realized that verifying the correctness of UI element identification before executing the test cases would be hard. The reason is that Appium does not evaluate the identification of UI elements until it executes the corresponding method call at run-time. When testing scenes in the 4th or 5th layer of the QuizUp model we often would have to wait several minutes before understanding whether the identification of some UI elements was correct or not.

Solution: We found it very useful to use an interactive Python shell, while implementing the test code, where we were able to execute single Appium method calls for the identification of particular UI elements. First we manually open the scene in the QuizUp application where the UI element to identify was present. Then we would execute the Appium method for our proposed identification of a UI element. If the attempt to retrieve the UI element object was successful, we would add those values to the element’s variable in the UIObjects class. This way we are able to save a lot of time by isolating the identification of UI elements.
Chapter 6

Discussion

In the Introduction, we described NASA GMSEC and QuizUp as two different flexible systems. A key difference is that GMSEC is built around a software bus which is accessible through APIs that are implemented in several programming languages, while QuizUp is a mobile application that has GUI clients for the Android and iOS mobile platforms. Another difference is that GMSEC is based on the pub-sub architectural style, while QuizUp is a custom mobile application. Therefore, the testing questions for the GMSEC study were based on testing a software bus that implements the pub-sub style, whereas for QuizUp the testing questions were based on the business rules and behaviors that are (mainly - there are some commonalities across all mobile apps) specific (or custom) for QuizUp.

Despite the above mentioned differences, the systems have certain similarities that enabled us to use the same MBT approach. The key similarity is that both systems are state-based systems where the user interaction can be described as sequences of events. The users interact with the systems by sending commands (or stimuli) and the systems react to the stimuli, depending on the state of the system, by issuing a detailed response. This stimuli-response pattern allowed us to model the systems as state machines, which is the key component of the MBT approach used in this study. Most systems can probably be described in the same way. However, some systems do not provide a detailed enough response to all stimuli, making it difficult to determine the state of the system.

Prior to the GMSEC study, we had only limited experience with testing in general and had no experience with MBT. It follows that we did not have any experience with MBT tools (e.g. Graphwalker) and close to no experience with state machines as model representations, except for some lectures at the university. Therefore, much of the effort spent in the GMSEC study was devoted to learning and applying MBT for the first time. A large
portion of the study (10 weeks out of 14 weeks) was spent in setting up and applying MBT using FSMs, which provided very limited results. However, spending only an additional 4 weeks for applying MBT using EFSMs provided drastically improved results.

Actually, we were able to take advantage of much of what was learned in the GMSEC study in the QuizUp study. At the time of the QuizUp study, we had gained experience in MBT which meant that we were much more efficient in setting up the QuizUp study (3 weeks compared to 7 weeks in the GMSEC study). Based on the results from the GMSEC study, we were also able to immediately determine that EFSMs would be the most appropriate model representation for QuizUp without first experimenting with FSMs.

When comparing the model size (i.e. numbers of states and transitions) for GMSEC and QuizUp, it is clear that the QuizUp model is much larger. The GMSEC model consisted of 49 states and 62 transitions (see Table 3.7), whereas the QuizUp model consisted of 125 states and 192 transitions (see Table 5.2). The reason for the difference is twofold. Firstly, we simply tested more features for QuizUp than for GMSEC and, secondly, in QuizUp, we were able to retrieve more detailed information about the state of the system at any given time. In QuizUp, for example, error messages (text strings displayed in the GUI) were separate for different scenarios or sequences. However, for GMSEC, we only used simple API return codes to assert whether an action should be successful or not.

There is also a significant difference in the number of requests and assertions. There are 8 requests and 15 assertions for the GMSEC model, but 87 requests and 74 assertions for the QuizUp model. The reason for the great difference is that a state in GMSEC reflects the expected behavior of the system after a particular API method call, whereas a state in QuizUp is comprised of dozens of UI elements that formed a view or a scene in the application. Therefore, for some states in QuizUp, we had to validate multiple UI elements, and data, in order to be confident that we were in fact in a particular state.

To summarize the effort spent in the two case studies, the efficiency in the QuizUp study was higher than in the GMSEC study. We spent less time setting up the testing infrastructure and performed a much more extensive effort without spending more time in total for the QuizUp study. We can argue that MBT started paying off as soon as we had applied MBT, for the first time, using FSMs for GMSEC. A similar but stronger argument can be made for QuizUp because due to our experience we could model more in shorter time.

Although we were able to successfully apply MBT on both systems under test, there were a few aspects that we found interesting related to testing (and MBT in particular) that we did not address in this study. Since both GMSEC and QuizUp rely on interactions between
components (or users), distributed MBT would have been an interesting extension to our work. By distributed MBT, we mean that the generated test cases would represent multiple "dummy" applications (or users) interacting with one another. This would be a much more realistic use of the systems. However, there were both limitations and challenges that we could not address during the study. For example, Graphwalker does not support the generation of abstract test cases from multiple model instances, where each instance would represent a single "dummy" application. Even if Graphwalker would have such a feature, we would have to construct very elaborate test oracles that could handle a shared data space for the "dummy" applications. Assume that we have a "dummy" application called A and another "dummy" application called B. If A would perform an action that would affect B, B would have to be aware of that in order to address its correct state appropriately.

An interesting aspect that arose during the QuizUp study was to create a protocol for modeling mobile applications. Since mobile applications are built using platforms such as Android and iOS, we could provide a standard protocol for modeling common patterns and UI behaviors. The protocol would probably only cover a small subset of possible patterns, but it would be convenient to use the suggested models provided by the protocol as templates when modeling. However, we did not construct a protocol for modeling mobile applications because the projected effort was out of scope for this study.

In addition to a mobile model protocol we discussed adding a simple textual language to describe the present UI elements and patterns in a particular view or scene in a mobile application. For example, 'Click Button "Ok"' could be a statement that would be compiled, or interpreted, to a transition in a model. This could especially help inexperienced individuals in MBT with little modeling experience. However, since we did not construct a mobile model protocol, we decided not to design and implement a textual language (because the language should be compiled in way that complies to a certain style or protocol).

### 6.1 Common lessons learned

Here we present our common lessons learned from the case studies of applying MBT on GMSEC and QuizUp.
Analyze existing test cases - they are valuable inputs to MBT

Problem: We did not know how to use the GMSEC API or QuizUp GUI in detail, which is necessary to construct valid models. The SUT documentation might not be complete or might not be written in a way that is easy to quickly understand for the independent testing team. The risk is that it takes a long time to build the first model and generate the first executable test cases.

Solution: We have found it efficient to first review existing test cases in order to understand the SUT, to build the model, and to define the mapping between actions of the model. In GMSEC case we had existing unit test cases, whereas for QuizUp we had existing end-user acceptance tests. We have also found that a model that is based on the information contained in the original test cases can be created very quickly and can generate many more test cases and provide higher coverage than the original ones that the model was based on. It is clear that without the existing test cases, it would have taken us more effort to conduct these studies. Thus, we recommend leveraging existing test cases, if they exist, as a start, both to understand the details of how the SUT works and the as a start of the initial model.

Embed debugging information in the generated test cases

Problem: When a generated test case fails, the tester has to review the test case and try to understand the state of the system and what caused the failure. For example, if unsubscribe fails in GMSEC’s case, the tester has to manually reconstruct the subscription table. This process can be time consuming especially if the generated tests are lengthy.

Solution: We found it useful to embed debugging information such as the state of the subscription table, which is created during the model traversal time, into the generated test cases. We found that such debugging information is of great value to understand the root cause of test failures and to characterize the test failures to developers. While it is true that adding the debugging information will make the test cases more complex, the benefits are overwhelmingly useful to reason about test failures.

We also found it useful to provide a way for the tester to navigate from the place in the test case where the assert failed in the generated test cases back to the corresponding place in the abstract test case. This helps the tester reason about the sequencing of the test instructions, since the implementation details are omitted in the abstract test cases. This is often necessary when trying to understand the test case and the corresponding failure in the concrete test case.
Models also benefit from style guidelines

Problem: One benefit of MBT is that the model is a precise documentation of the SUT’s behavior. However, the model can be difficult to understand if it does not have any style guidelines.

Solution: We have found it useful to ensure that state names and transitions clearly explain the intention and follow a consistent naming convention. We also found it useful to structure the layout of the model in a consistent way. Such models help others to understand the scope of what is tested and not tested, identify weaknesses in the model, etc.
Chapter 7

Related Work

There are many MBT books in the literature. In [9], a somewhat theoretical overview is provided of what formalisms are being used as the underlying formal models for MBT. In [48], a practical introduction to MBT is provided, including how to write models for testing purposes and how to generate test suites. There are also many MBT case studies in the literature. For example, in [49], a similar case study is performed and empirical evidence gathered for the effectiveness of MBT. It is also worth mentioning that Harry Robinson has created a website on MBT [39] with a collection of papers, presentations, etc. Our work is at the intersection of software architectural styles, requirements and MBT. That is, we discussed how to leverage architectural style or requirement constraints to develop a model for test case generation, thus complementing the resources in [39].

Architecture-based Testing. There are a few other research approaches (e.g., [32] and [7]) on architecture-based testing. Most of them deal with the problem of how to use the data and control flow structures to derive test plans and test cases from the software architecture. In our opinion, those approaches are generally not end-to-end, meaning that they do not discuss how to construct executable specification models from software architecture or how to automatically derive ready-to-run test cases. The SEI has an initiative on architecture-based testing [15]. Our work shares the same goal of leveraging architectural styles for software testing. However, while the SEI initiative develops a fault model for each architectural style, we study how to build models for test automation using MBT to test systems based on a specific architecture style. They discuss what faults are related to a certain style in a general context while we discuss specific models for testing of pub/sub and mobile systems using MBT.

Testing of the Pub-Sub systems and the software bus. In [30], the authors present an approach for unit testing publish/subscribe applications using the Linear Temporal Logic
(LTL) and the Java Markup Language (JML). They model the expected behavior in LTL and convert to JML, which contains pre-condition, post-conditions, invariants for each unit under test. Our approach complements their approach because our objective is to automatically test the pub-sub infrastructure, whereas their objective is to test pub-sub applications by mocking the pub-sub infrastructure. In addition, their paper does not discuss reusable models for the pub-sub style. In [44], the authors present their experiences of testing a software bus using MBT. Their models are represented using advanced process algebra specifications, which are used as the input for the JToRiX tool [6] in order to automatically derive test cases on-the-fly. Our tests are offline, meaning that we do not run the tests during the model exploration phase simultaneously as in [6]. As a result, our test cases are repeatable because they are static and do not change during run-time and are convenient for testing multiple languages as we do in this study. However, our oracle cannot handle some of the issues arising from non-determinism. For example, in the GM-SEC there is no guarantee of the First-in-First-out (FIFO) policy for message delivery, thus the actual delivered message will be known only at run-time, making it difficult to embed the expected message as part of the test oracle at modeling time.

**Modeling notations.** Several modeling notations for MBT exist [8]. Many are declarative languages for modeling. E.g., in Microsoft’s SpecExplorer tool, models are specified as C# programs. SpecExplorer (and other similar tools) are for testers who can program. In contrast, our models are basic FSMs/EFMSs that require a short learning curve, and based on our experience are more appealing to practitioners. However, this comes at the cost of less power and expressiveness. We wanted to start with basic modeling notations and understand their effectiveness and drawbacks. Work is underway to use SpecExplorer for testing the GMSEC to better understand how state machines are different from declarative model-based notations.

**Testing fundamentals.** In [46], the authors point out the crucial interrelationship between tests, programs, and oracles. In the context of MBT, models encode the test oracle and thus the generated tests have built-in assertions for test automation without human intervention to decide the verdict. In practice, obtaining the precise oracle is difficult for non-trivial scenarios. In our first case study, we reviewed the existing test cases, API documents, and interviewed developers of the GMSEC. While most of the oracle problems were resolved in that process, MBT pinpointed several specification ambiguities, for example, in the GMSEC study, we realized that it was not well specified whether or not one should be able to repeatedly connect and disconnect as well as whether the connection object should be reset after disconnect. However, such specifications are crucial in order to build models for MBT.
Random testing, Symbolic execution and model checking In [36] and [22], the authors apply random testing, symbolic execution, and model-checking techniques to generate test inputs for achieving different code coverage criteria. Such methods are complementary to MBT because the generated test input is to cover the code in contrast to MBT’s focus on checking whether the expected behavior and the actual behavior are consistent with each other. Code coverage is, of course, very important but checking whether the software actually implements the specified requirement is arguably as important as achieving code coverage.

A tool of interest is QuickCheck [12]. The tool aids Haskell programmers in formulating and testing properties of programs. The great value of Quickcheck is that properties can be described as Haskell functions, and those properties can be automatically tested on random input. Moreover, it is possible to define custom test data generators. We have started exploring how one might use Quickcheck in an MBT approach such as ours.
In this thesis, we presented an empirical study where MBT was applied to two different flexible systems, namely, the APIs of NASA GMSEC’s software bus and the Android client of QuizUp through its GUI. The main goal of the study was to examine whether MBT could be used, in an effective and efficient way, to test flexible systems using the same approach. Our hypothesis was that we would be able to use the same MBT approach on different flexible systems, using GMSEC and QuizUp as examples of such systems. Moreover, we studied whether this would be feasible with reasonable effort.

The study shows that we were able to use the same MBT approach to test two different flexible systems, thus, supporting our hypothesis. Although GMSEC and QuizUp were indeed different (e.g. QuizUp has a graphical user interface, GMSEC does not), they shared certain similarities that allowed us to use the same MBT approach. The most notable similarity is that both systems are state-based systems where the user interaction can be described as sequences of events. We found that maintaining a single behavioral model for each system was key in order to test these flexible systems in an efficient way.

The study also shows that it is feasible to use MBT for flexible systems such as GMSEC and QuizUp. This was demonstrated, for example, by the fact that the test cases were able to detect non-trivial issues in systems that were already well-tested. Much of the effort in the GMSEC study was devoted to learning and applying MBT for the first time, whereas our gained MBT experience allowed us to set up faster and perform a more extensive modeling effort for QuizUp. The effort data and the detected defects show that MBT started paying off as soon as we had applied the process for the first time.

Applying MBT using FSMs in the GMSEC study provided very limited results, whereas using EFSMs we found much more detected defects. Therefore, we chose to solely use EFSMs as a model representation in the QuizUp study. Our conclusion is that EFSMs are
a preferred model representation over FSMs, assuming that the person constructing the models has a background in programming.

We also found that MBT does provide a systematic way to test systems, and even though there are still manual steps involved, a high degree of automation is possible to achieve with reasonable effort for someone who has no previous experience with MBT. A possible extension to our work would be to minimize the manual steps even more. Constructing a protocol for modeling mobile applications, for example, would be beneficiary to standardize the modeling effort. Another option would be to implement a textual language to describe a SUT and its possible actions. For mobile applications, for example, we would describe the UI elements and patterns in a particular view or scene under test. The constructed syntax in the language could then be compiled, or interpreted, to a model representation. We refer the reader to Chapter 6 for further avenues for future work.
Bibliography


