‘User Interface Test Automation and its Challenges in an Industrial Scenario’

Name of student: Ligaj Pradhan (lpn09006@student.mdh.se)

Supervisor: Sasikumar Punnekkat (sasikumar.punnekkat@mdh.se)

Examiner: Sasikumar Punnekkat

Date 2011-11-05
Abstract

The growing demand for UI test automation has triggered the development of many tools. Researchers and developers have been continuously working to further improvise the existing approaches. If we look at GUI test evolution we can observe a clear progress from manual testing towards complete automation. Numerous approaches have been made to automate the GUI testing process. Record and playback tools, key-word driven methodologies, event flow exploration strategies, model based approaches are continuously evolving with higher level of automation. Similarly, new ideas and strategies to make these tests efficient are also emerging. Optimization of this resource consuming activity is another very important aspect in this area. Dependencies between different tests can create deadlock scenarios, while running larger test suites. A concept of Ordered Test Suite can be used to cope with such dependencies. Following the Model Driven Architecture initiative by Object Management Group, a new global trend of Model Driven Engineering is creating a big sensation in the field of model based software development. Using the same principle, studies have also been made to automatically generate tests from models. Behavioral models can be made using the model driven approaches and these models can be analyzed to generate tests automatically. This master thesis addresses different approaches made for Graphical User Interface test automation, some optimization issues and solutions, a case study done at a software company to automate User Interface testing and a model driven approach for automatic test case generation.
Preface

I would like to thank all the people who helped me with this master thesis. Thanks to ‘Catalysts’ for providing a chance to work with the GUI test automation implementation. Thanks to Dmitri Pissarenko for supporting and guiding me at ‘Catalysts’. I am especially thankful to Antonio Cicchetti from Mälardalen University for guiding me to implement and better understand the concepts of Model Driven Engineering. I am also thankful to Prashanta Paudel for helping me in exploring different Model Driven Engineering tools. Thanks to my thesis advisor Professor Sasikumar Punnekkat for advising and supporting me throughout the thesis process. Finally, I am very grateful to ‘EURECA Project (www.mrtc.mdh.se/eureca) funded by the Erasmus Mundus External Cooperation Window (EMECW) of the European Commission’ for providing me the opportunity to pursue my master’s degree in Software Engineering at Mälardalen University.

Thank you all!

Ligaj Pradhan
Table of Contents

1 GLOSSARY .............................................................................................................................................. 6

2 INTRODUCTION AND PROBLEM FORMULATION .................................................................................. 8
   2.1 BACKGROUND ........................................................................................................................................ 8
   2.2 PROBLEM FORMULATION ...................................................................................................................... 10
   2.3 STRUCTURE OF THE THESIS ................................................................................................................. 11

3 EXISTING MODELS FOR GUI TEST AUTOMATION ........................................................................... 12
   3.1 CAPTURE/PLAYBACK TOOLS ................................................................................................................ 12
   3.2 KEY-WORD/TABLE-DRIVEN FRAMEWORKS ....................................................................................... 13
   3.3 EVENT-FLOW MODEL IMPLEMENTATIONS ........................................................................................ 14
   3.4 MODEL BASED APPROACH .................................................................................................................. 16
   3.5 GUI TESTING USING COMPLETE INTERACTION SEQUENCES ....................................................... 19

4 DIFFICULTIES IN GUI TESTING ............................................................................................................. 22

5 OPTIMIZATION EFFORTS IN GUI TESTING ............................................................................................. 24
   5.1 CALL TREES ......................................................................................................................................... 24
   5.2 MODEL BASED ..................................................................................................................................... 24
   5.3 CALL-STACK COVERAGE ..................................................................................................................... 25
   5.4 OPERATIONAL EXTRACTION .............................................................................................................. 25
   5.5 ALL-USES CRITERIA .............................................................................................................................. 25
   5.6 SELECTIVE REDUNDANCY ................................................................................................................... 25
   5.7 PRIMARY AND SECONDARY CRITERIA ............................................................................................ 26
   5.8 CLUSTERING ALGORITHMS ............................................................................................................... 26
   5.9 EVENT BASED COVERAGE ............................................................................................................... 26

6 SOME POPULAR GUI TESTING TOOLS ................................................................................................... 27

7 IMPLEMENTATIONS AT CATALYSTS ....................................................................................................... 29
   7.1 IMPLEMENTATION TO TEST THE IN-HOUSE PROJECT CALLED ‘TASKMIND’: .................................. 29
      7.2.1 Background .................................................................................................................................. 30
      7.2.2 SilkTest ........................................................................................................................................... 30
      7.2.3 XPath .............................................................................................................................................. 31
      7.2.4 OTS Algorithm ............................................................................................................................. 32
      7.2.5 General structure of the project .................................................................................................. 33
      7.2.6 Running the test cases: ................................................................................................................. 34
      7.2.7 Results: ......................................................................................................................................... 34
      7.2.8 Overall impact of the project: .................................................................................................... 35

8 MODEL DRIVEN TECHNIQUE FOR GUI TEST AUTOMATION ....................................................... 36
   8.1 INTRODUCTION ..................................................................................................................................... 36
   8.2 EMF ....................................................................................................................................................... 37
   8.3 TEFKAT ................................................................................................................................................ 38
      8.3.1 Basic structure of Tefkat transformation specification: ............................................................... 38
1 Glossary

AJAX – Asynchronous JavaScript and XML
API – Application Programming Interface
AUT – Application Under Test
CIS – Complete Interaction Sequences
CNL – Controlled Natural Language
CSV – Comma Separated Values
EMF – Eclipse Modeling Framework
ESEEs – Event Space Exploration Strategies
FSM – Finite State Machine
GUI – Graphical User Interface
IDE – Integrated Development Environment
LTS – Labeled Transition Systems
MCDC – Modified Condition and Decision Coverage
MDA – Model Driven Architecture
MDE – Model Driven Engineering
MOF – Meta Object Facility
OMG – Object Management Group
OTS – Ordered Test Suite
PIM – Platform Independent Model
PIT – Platform Independent Test Model
PSM – Platform Specific Model
PST – Platform Specific Test Model
QVT – Query View Transformation
RCP – Rich Client Platform
SMC – Sequence of Method Call
SUT – System Under Test
SWT – Standard Widget Toolkit
TSL – Test Script Language
UI – User Interface
UML – Unified Modeling language
VB – Visual Basic
VBA – Visual Basic for Applications
WPF – Windows Presentation Foundation
XMI – XML Metadata Interchange
XML – Extensible Markup Language
2 Introduction and Problem formulation

2.1 Background
Graphical User Interface (GUI) has become the most conventional way to interact with the software. The GUI reacts to various user events like mouse clicks and keystrokes. This allows the user to communicate with the underlying application. GUI in turn communicates with the user via method calls or some kind of messaging system. Large portion of the code is dedicated to implement and maintain GUI. Research has shown up to 60% of the total software code has been used for implementing GUI [3]. However till recent years the GUI testing approach for the functional correctness of software had been largely neglected. The use of GUI in safety critical systems is also increasing rapidly which also emphasizes the necessity of GUI functional testing. The proper testing of GUI can significantly contribute in the total safety level, robustness and usability of the entire application.

GUI functional test means validating GUI objects, checking functional flows by operating GUI objects and verifying output data which are generated in backend and then displayed in front pages [2]. People have ventured to perform these operations following different models and techniques which can range from fully manual to semi-automatic. However the tendency is to automate as much as possible so as to make it very fast and have a huge coverage which would otherwise take a tremendous time for a human.

Several tools are developed and numerous approaches are put forward to achieve this goal. The simplest of the kind are semi-automatic types which is kind of incomplete, ad hoc and to a very large extent manual. They can be systems with record-playback capabilities which could record the mouse and keyboard events as scripts, when a user interacted with the GUI. This process is generally followed by a tester who can playback the recorded test cases. The tester can even play around the script and manipulate the script to make some modifications. This process is extremely labor intensive and largely relies in the ability of the test designer [5]. It is huge and expensive to manage and quite a lot of work to generate the entire test cases.

Numerous automation strategies and solutions are being proposed with the objective of automating the GUI functional testing. Many techniques are created to support test case design and generation. Virtual presentation with formalized models tends to be getting pretty popular in this regard [2], [10]. Various tools are also developed for supporting automatic test execution like Rational Functional Tester from IBM and Quick Test Professional from Mercury [11]. Some researchers have also tried to integrate all these modules for test case generation and selection, test execution and test reporting together [2]. Slightly different approach by constructing an event flow model or the integration tree for the application is also being used for the same purpose [3]. This involves defining event space exploration strategies (ESEs) and using it for test case generation, test-oracle creation, coverage evaluation and regression testing.

Researches in automatic test case generations has demonstrated that a higher level of support could be achieved by automatically generating the test cases although still a programmer should code all possible
decision points in the GUI [5]. Easing and facilitating more automation, systems able to automatically generate and select test cases, execute them and report the result are coming into realization [2]. The idea behind test automation is mimicking what a human can do using the GUI and test data i.e. interact with the system and see if the desired conditions are achieved. In this process there are several challenges. One of the earliest and most important challenges is identifying the GUI objects and their actions and events. It is followed by the test case design, selection and their execution and reporting.

The main reason for GUI test to be difficult to automate is because the interaction between user and application is interactive. Several authors have expressed that any automated tool to overcome this difficulty should be able to provide with the following services [1]:

a. Record and playback of physical events in the GUI
b. Screen capture and comparison
c. Shell scripting to control and execute the tests

People have faced many pitfalls in the process of automating GUI testing [5]. Compared to the traditional software testing, it faces many more difficulties. Even the Simple task to determine the coverage criteria faces complications because traditional code-based coverage criteria do not necessarily apply in this domain. Criterion like ‘event coverage’ has been introduced which might be like all the reachable events in the application should at least be called once. Deciding how much event coverage is to be done can be very important because often if we try to cover all the possible branching and paths it can be too large. Similarly verifying if the GUI test has passed or failed might also be tricky. Generally the process involved is a separate program called ‘test oracles’, which compares the expected results to the actual results and decides if the test is passed. But still, to decide what to check to make this decision can be difficult. For example the test case might lead to a screen which no longer contains a certain level but the entire state of the application is still correct. Now the test may fail only because it didn’t find the label in its correct position. As the project advances and new GUI emerges, the test project is also to be extended so that all areas of GUI is covered and tested. Regression testing also possesses special challenges for GUI testing. GUI tests might depend on the layout and as the application is developed minor positions can change very frequently. This in turn can change the outcome of many older tests. Likewise, many test outputs used by test oracles can be obsolete. It might be very costly to detect frequent modifications and adapt older tests for these changes.

The prime purpose of this thesis is to explore how these goals can be achieved. A careful and through research is to be made on what researchers and developers have been doing to deal with all these pitfalls of GUI testing and achieve superior automation.

The second phase of the thesis should involve the challenges being faced by a software company and what solutions are being adopted in this regard. Especially when the amount of test cases grows too large for a large system it will become very difficult to manage and maintain. Simply to run the test cases it will take huge time and memory. The GUI testing process becomes cumbersome and inefficient. It is also expected to find out how this problem is dealt with in today’s software industries.
A rather new approach based on Model Driven Architecture (MDA) has been a matter of great interest in the field of software verification and validation in the last decade. Traditionally it is used to develop applications without writing any codes. First a model of the system is developed in Unified Modeling language (UML) or similar format and by defining some mapping rules, the code for the system is self-generated. There are many MDA based tools available today. Eclipse Modeling Framework is an Object Management Group (OMG) initiative to address MDA. This has been a popular tool to work in this area. Today, many researchers have been trying to use this approach in software validation and verification for automatic generation of tests. Impressed with the growing popularity the final section of this thesis is dedicated to explore the possibilities of automatic test case generation using MDA tools. Research works previously conducted in this area are also closely followed. An approach purposed in [31] is carefully studied and even a prototype is made to observe the possibility of this new upcoming approach.

2.2 Problem formulation

The general intension of this thesis is to address works that are conducted in GUI test automation. In this process the thesis aims to explore several key aspects as explained below:

1. Software testing has become a very important activity in the world of software development. Though we find many ways to automate the testing of the source code, automating GUI testing is still not pursued very seriously. However, with the increasing application of GUI to interact with the software systems, it has been realized that GUI testing is a vital step when it comes to software verification and validation. As such the initial task in this thesis is to provide a background and discuss on the need for automating GUI testing.

2. Many researchers and developers are working on this issue of automating GUI Testing. Our second task is to perform a literature review on sate of art techniques that exists today to address GUI test automation.

3. The main reason for GUI test automation to be still neglected is because it is difficult and not so straight forward as source code testing. Our third task is to discuss various difficulties faced in this process.

4. Simply creating and running the GUI test cases is not sufficient. Normally they consume long time period for execution and need large resources, which may not be generally available. Thus they have to be efficient and optimized. Thus the fourth task is to discuss optimization in GUI testing.

5. Our fifth task is to present how a company in Austria called ‘Catalysts’ is trying to optimize the time taken to run massive set of GUI test suites. Several problems like multiple object generation, excess memory usage and dependency between test cases are to be handled in this implementation. This implementation is to be presented briefly.

6. Model Driven Development is creating a new sensation in building a software using models. The same techniques are now tried to generate test cases and test the GUI of a system. Our final goal would be to explore and implement Model Driven Techniques to be used for automatic test case generation.
2.3 Structure of the thesis

This thesis report is divided into several chapters. Chapter 1 provides the extensions of the main acronyms used in this report. Chapter 2 introduces the context and background of the thesis followed by problem formulation and a description on the organization of the report. Existing models and relevant theories are discussed in chapter 3. Chapter 4 elaborates the different kinds of difficulties that might be faced while testing GUI of a system. Some optimization approaches and test reduction techniques are discussed in chapter 5. Some popular GUI functional testing tools are briefly explained in chapter 6. Implementations regarding GUI test automation, that were carried out in a software company are presented in chapter 7. Chapter 8 introduces and describes thoroughly how the tools and techniques of MDE can be used in automating GUI testing. This chapter also illustrates the practice with an example. Finally a brief summary and conclusion of the thesis is provided in chapter 9.
3 Existing models for GUI test automation

3.1 Capture/Playback tools
Most of the GUI testing tools use the popular capture/playback concept as an easy way to record test scripts. Capture/Playback tools are designed for mimicking how a human interacts with the AUT. A tester carefully interacts with the AUT and records every action. Generally, all low-level mouse clicks and keystrokes are recorded. Some tools recognize GUI objects as objects and some rely in the coordinate system of the screen or the mouse pointer. Tool specific test scripts are also generated. Verification points are to be inserted at different states to keep track if the AUT is behaving as desired. After the recording is completed, the tester can playback the recording automatically. During playback, the actual values and properties of selected objects are compared with the expected values and properties to decide if the test passes. If the test case does not behave as desired when it is rerun, it fails. Nothing needs to be changed in the script, if the exact same actions are to be repeated. However, if the desired testing behavior is to be achieved, for example with different sets of data, then we need to change the recorded script. This is where the tester has to undergo lot of manual work. Though traditionally claimed to be an automatic process, it is often questioned if it is indeed an automatic tool in reality [39]. This is because of the huge manual work required to record tests by interacting with the AUT, insert verification points and modify the test scripts to fix changes.

The first action that is to be performed is recording the test steps. For this, a tester needs to interact and record the interactions manually. This process is generally labor intensive, time consuming and very tedious. The tester who is recording the tests also needs to be well acquainted with the AUT, as he should be able to interact with it effectively. While recording, the tester has to manually insert all the verification points and specify with what data the actual results are to be compared. User intervention is not finished even after the recording of the tests and the generation of the test script is completed. If the AUT is to be tested with different sets of data, then again the recorded script is to be edited. During the life cycle of a software development, it is continuously modified. When the system functionality changes the recorded test scripts becomes useless. It is to be re-recorded or it has to be changed by some trained professional. Often, a failed test is to be fixed before continuing with other test runs. All these pitfalls while working with the capture/playback tools suggest that huge human intervention is involved in this concept. It is very difficult to write robust test cases with this method. Most of the time is spent on manipulating the tool and editing the recorded test scripts. This often results in less time and effort in designing effective and efficient test cases to find bugs.

Despite these sorts of nuisance, some testers have successfully used capture/replay tools to test their systems. It is generally helpful to create small modular test scripts. It is very fast to create such simple test scripts. Although maintaining these test scripts can be a tremendous task, recapturing them may be very simple and fast for many systems [39]. This sort of test automation is also effectively used during the prototyping phase of a system along with mock-up objects. It is fast and as the system grows, the script can be edited and updated. Although this method possess a lot of labor-intensive effort, it can be useful if it is used carefully by an experienced tester.
3.2 Key-word/table-driven frameworks

It would be best if we could describe the test case in human natural language. But it is not understandable for the machine to automate those test cases. As such a less complicated solution of using key-word driven framework prevails. It is one type of Controlled Natural Language (CNL) [12] that is used to specify test case requirements. The main idea is each step in the specification contains a verb and zero or more arguments. Generally the verb is described as the key-word. For example Table 1 demonstrates the use of SET and VERIFY as two basic key-words to specify test cases. In addition to the key-word they have three other parameters viz. Object, Value and Description. Objects are the GUI controls. Values are the properties of the object and Description is additional text for comprehension purpose. As in [2] each test steps in a four dimensional vector form i.e. <Key-word, Object, Value, and Description>. This test steps are generally presented in a table, so this model is also popularly known as ‘table-driven framework’. Table 1 shows a sample for describing test cases using key-word framework.

<table>
<thead>
<tr>
<th>Step</th>
<th>Key-word</th>
<th>Object</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SET</td>
<td>TextBox1</td>
<td>100</td>
<td>Input 100</td>
</tr>
<tr>
<td>2</td>
<td>VERIFY</td>
<td>Label1</td>
<td>200</td>
<td>Verifying output</td>
</tr>
</tbody>
</table>

*Table 1: A sample for describing test cases using key-word framework*

There is a mapping for the key-words to what actions should be done so that the testing system can interpret the activities and automate the test cases.

This model is said to divide the whole GUI test into two different phases, viz. Planning Phase and Implementation Phase. This further makes it easy to divide work between domain specialist and the programmer. Generally planning is done by the one who has clear idea about the domain but may not be good at programming. It is possible because he only needs to know about the key-words which are pretty easy to understand. Likewise the implementation i.e. the actual coding of the driver where what happens in code when particular key-words are used is the job of the developers.

Key-word driven GUI testing has several advantages:
1. Maintenance is low if we consider long run. This is because test cases are concise, clear, easily readable to the testers, easy to modify and easily reusable.
2. Key-words can be reused in multiple test cases.
3. It is not dependent on any particular tool or language so even those who do not have programming background can use it.
4. Division of labor is easy, as testers with domain knowledge can plan the test cases and developers with programming knowhow can implement to map the effect of the key-words in the actual code.
5. Abstractions of the layers are clear and very logical.
3.3 Event-flow model implementations

The main idea behind this model is to represent the event flow of an application i.e. all the interactions and events in an event-flow model. The main advantage of event based model upon other model based approaches is that other model based approaches generally has a limited applicability as they are built with one particular type of GUI testing activity in mind like test case generation, test oracle generation or regression testing. Event flow based model is presented in [3] as a general purpose model which can be used to serve different purposes intended by the GUI tester using proper techniques. However they are pretty expensive to create.

The event flow model basically represents all possible sequences of the events that can occur in an application. In [3] a hierarchy of model dialogs is described in an integration tree. It shows all the execution paths in the dialogues. Individual events are represented by using preconditions and effects.

Event flow model presents a general solution to GUI testing. In [3] it is explained how a wide variety of testing tasks can be performed using event flow by defining special techniques called ESES.

**Event flow model:** The event flow model consists of two parts viz. preconditions and effects. The precondition is the state where the event is executed and the effects are the changes to the state when
the event is executed on it. The effects consist of all possible sequences of events as represented by the interaction tree. It has been used for various kinds of GUI testing like goal-directed test case generation, test-oracle generation, checking models, graph-traversal-based test case generation, test-coverage evaluation, genetic-algorithm-based test case generation and Bayesian-network-based test case generation [3]. Memon also shows how event flow can be used to detect two kinds of faults viz. when the software doesn't perform as documented and when it shows unacceptable behavior like crashing and freezing. In [3] GUI is considered as a hierarchical, graphical front-end of a software that accepts fixed event inputs from human interaction and the system and produces deterministic graphical output. It consists of many other graphical objects which have definite values or attributes during execution and the collection of all these attributes defines the state of the GUI. Thus a GUI state is likely to contain the collection of all the graphical objects that it holds at that instance and all the properties associated with them. In [3] events in a GUI are modeled as operators represented by their preconditions and effects. For example like if ‘S’ is the state and then set of events E= {e_1, e_2, e_3, e_4, e_5, e_6 ...e_n} are the state transducers that can act on the objects present in S and change it to another state. Now the main idea of event flow is to show all possible interactions in the GUI. A graph model can be constructed where each vertex represents the events and the edge joining one vertex with other suggests that the second event can be executed right after the first one. This can also be represented using a state model, where each vertex represents the state and the edge would represent which child state will follow the parent state. In [3], Memon discusses how hierarchy of modal windows can be represented in such a model. He also presents how to construct an event flow graph and integration tree by identifying modal dialogues and their invoke relationships. An approach to automatically generate the event flow graph and integration tree by combining reverse engineering techniques and the algorithm to compute the set of succeeding events is also presented. The tool developed by Memon is called 'GUI Ripper' and it automatically runs the System under Test (SUT) opening the application’s windows in depth first manner. It also extracts all the widgets and their properties. This information is used to construct the event flow graph and integration tree.

After defining the event flow graph, it becomes possible to carry out various aspects of GUI testing by defining simple suitable ESES. Memon [3] describes 3 simple ESES which we discuss as follows:

1. **Goal directed search for model checking:** The main idea behind the model checking involved here is checking if a model allows the transition from a valid state to an invalid state. Memon describes an AI planning tool that uses the event flow graph and integration tree to parse from one given valid state to its branches. It is provided with a valid state and an invalid state. Now if it is able to reach the invalid state, then the model is incorrect. If it cannot find ways to reach the invalid state from the given valid state then the model is correct.

2. **Graph exploration for test case generation:** A GUI test case is generally in the form \{S_0; e_1, e_2, e_3, e_4, e_5, e_6 ...e_n\}, where S_0 is the chosen state to start the test and e_1 to e_n is the set of events that can execute in that state. Now a tester can use several graph traversal techniques to generate numerous amount of test cases from the event flow graph satisfying criteria like code coverage or branch coverage. Using event flow graph it would be very easy to automate instead of manually capturing the initial state and all
the events following it. It can be suitable to discover faults like crashing or freezing and to check if the application is behaving as specified with correct results.

3. Operator execution for test-oracle creation: A test oracle is the knowledge what we use to determine if the test has run successfully. Now the oracle’s form can be defined in the sequence of states $S_1, S_2, ..., S_n$ such that $S_i$ is the expected state after the event $e_i$ in the GUI. Now as such it becomes very easy to explore the event flow graph and determine what might be the state of the GUI after another event. It’s just the changes in the graphical widgets and their attributes. Summing these changes to the current state we get another state that is the expected state after the execution of certain event. Again it becomes fairly easy to automatically create test oracles by exploiting event flow graphs.

As such event flow models presents a very promising scenario in case of automatic GUI testing. Though in the initial preparation phase it might be little more work, it can be utilized in multiple GUI testing activities fairly cheaply and with much lesser effort. The general purpose nature of this kind of models makes it an interesting approach in GUI testing.

3.4 Model based approach

The main concept behind the model based testing is that the actual application is represented by a model representation and the test cases are written in this level as well. Such test cases are often referred as ‘abstract test suite’. To execute them or to let them interact with the real system they are to be first converted in to ‘executable test suite’. Today people are trying to reduce the time and effort spent in development and testing by introducing advanced model based process and tools. One of them is model driven development. With models development process is raised to a different level of abstraction. The upcoming idea is to use this approach to develop the test cases and test the GUI of the SUT using these models. People have ventured to use UML to model systems and also develop test cases from it automatically [31]. Model based development paradigm is a new upcoming technique of software development focusing in domain models rather than on computing algorithmic approach. It is getting very popular in the recent years as it has some huge potential advantages. It is specially getting popular in real time and safety critical systems where failure after field deployment is simply unacceptable. There are many advantages for working with Model Based techniques like, easiness in communication and prototyping, correctness of the model can be established mathematically, automatic code generation, etc. With this technique, we can represent our systems in terms of different models and use them for different purposes like formal verification of correctness, automatic code generation and automatic test generation. This form of testing is able to find bugs already during the time of model building. It is specially considered better when the SUT changes frequently as only the model has to be changed, requiring less change in the test suites. Using models it is clearer and easier to communicate than text. The correctness of the models can be established and maintained mathematically [26]. Another very significant advantage of this type of testing is that it separates the domain logic and the basic programming details. This leads the testers to focus on domain logics and
behavior directly without getting disturbed by the complex programming details. In scenarios where the domain experts lack detailed programming knowledge, which is generally true, they can validate the GUI based on their domain expertise and directly start playing with abstract models of the system [31]. Thus there is a trend evolving in automatic GUI testing with model based approaches. One interesting venture to use model based test generation tool to generate automatic test is described in [26]. It was a joint venture between Aalborg University and NOVO Nordisk. The input to the tool was the UML state-machine model and the output was a test suite that satisfied some preferred testing criterion like edge or state coverage. The main advantage that was observed from the venture was that time required for test generation was significantly reduced. In addition to this, another significant advantage was an increased coverage was achieved with reduced test scripts. The main purpose of this venture was to develop a test generation tool for system testing of a GUI using UPPAL model checking tool. NOVO Nordisk had an assignment of system testing a GUI to determine if all the preferred interactions in one of their product were working properly. The graphical representation of the behavioral specification was traditionally provided in Visio. Previously the testers needed to study these Visio representations and generate set of test cases that could test the preferred behavior of the GUI. The process is presented in Figure 2 as in [26].

![Figure 2: Old approach for GUI test](image)

This process had several problems like creating the test cases manually was tedious, difficulty in determining the coverage of the test cases, changes in the SUT required the whole process to be changed, etc. As such UML Statemachine was chosen as the input model and UPPAL was adapted to accept these models. The later was achieved using XMI to represent the UML Statemachine which was further converted into UPPAL models. Then the UPPAL engine was used to generate a test suite with either the edge coverage or state coverage. Finally the resultant test cases were converted into scripts that could be run in the target machine [26]. Now the new process for GUI testing is represented in Figure 3 as provided in [26]:
There is another interesting approach to exploiting model based approaches in automatic GUI testing using Labeled Transition Systems (LTS) in conjunction with action words and key-word techniques [32]. The GUI testing is represented by the key-word and action word techniques. Key-words correspond to small GUI actions like key presses and menu navigation. Action words describe higher level abstraction of user actions like opening a file. This approach separates the business logic with GUI navigation and let the domain experts to design the test cases even before the implementation of the system. The main idea was to design test models and generate test runs from them automatically. The test models were to be described as LTS whose transitions are corresponding to the action words which in turn describes the GUI navigation. The action words are again mapped into corresponding sequences of key-words and then fed into a general purpose GUI testing tool. This tool has the capability to interpret the key-words and walk through the test models, verify results and handle the reporting. GUI test generation by modeling the system with UML activity diagrams is explored in [30]. They are then annotated with categorized data and test scripts are generated from the models. The application behavior is represented by the use of UML Use Case diagrams. Activity diagrams describe the relationship between the use cases and the actors that are active in the system. In other words they cover the functionality workflow of the system. They form the basis for testing the different functionalities and business rules described in the use cases specification. They are further refined first by enhancing them to capture the information needed for generating useful test cases and then by annotating with additional test requirements typically test data. This test data is referenced for branching purposes. In the test generation phase, an activity’s text will become a test step. They use a tool made at Siemens Corporate Research, the Test Development Environment using UML. The output from this tool is a set of XML-based files that can be formatted for presentation as a set of textual test procedures or executable test scripts. Several coverage criteria like graph and data coverage are used further to determine how deep to how much test should be generated to how much the system should be tested.

Figure 3: New approach for GUI test
3.5 GUI testing using Complete Interaction Sequences

GUI testing has been a difficult task because of the vastness of the input field. With all the combinations of the input user interface objects and selections it soon grows into an enormous task. There have been several approaches to reduce this input explosion and make GUI testing feasible in reality. One of such approaches called Complete Interaction Sequences (CIS) is proposed in [6] utilizing all the user sequences of GUI objects and selections which collaborate to produce a desired response for the user.

![Finite State Machine (FSM) for the Edit-Cut-Paste-Copy-Paste CIS Sequence](image)

*Figure 4: The Entire Finite State Machine (FSM) for the Edit-Cut-Paste-Copy-Paste CIS Sequence [6]*

FSMs are commonly utilized to test these CIS. Concept of responsibility is utilized which means that the activity involves GUI objects and others to produce an observable and desirable effect [6]. Each identified responsibility is represented with corresponding CIS. With the introduction of the concept of
responsibility the approach has already vastly controlled problem that might be created by input explosion. Next the amount of required tests can further be greatly reduced by identifying components of CIS that can be tested separately. A number of transformations can be applied as by [6] which will reduce the FSM into reduced FSM. In [6] they reduce the FSM by using following algorithms:

1. Abstracting strongly connected components
2. Merging CIS states that have structural symmetry

The main concept behind implementing the above reduction operations is to replace the given subset of states and all transitions between them as a superstate (single node) in the FSM for the CIS. Then these subsets will be added to arbitrary tests of the reduced FSM so that they would occur in at least one test of the entire FSM.

![Reduced FSM for the FSM presented in Figure 4][6]

Figure 5: Reduced FSM for the FSM presented in Figure 4 [6]

One example presented in [6] demonstrated how the reduction transformations will simplify the testing by reducing the FSM representing the CIS. This is the effect of reducing the overall tests required for the CIS. The CIS represented in Figure 4 is an Edit-Cut-Paste-Copy-Paste sequence which contains two open
file sequences. If all the combinations of the possible paths are considered then 50 design tests are required. However after applying the reduction transformations 2 structured symmetry components and one strongly connected component are observed. It is reported that 2 design tests are required for components 1 and 3 and 4 design tests are required for component 2. These are all depicted in Figure 5. As a result of the reduced FSM the total number of design tests required is dramatically reduced to 8. In some other empirical investigation they also reported that no additional faults were detected by testing all possible paths over reduced FSM for any CIS considered in their investigation.
4 Difficulties in GUI testing

GUI testing is far more difficult than the conventional software testing that is done for the source code. Many new and complex problems arise in the process.

1. It is very common for GUI to have large number of states. There are many situations of state explosion as well. Especially with user interfaces with lot of choices and branching, the number of states that is generated can be huge, often very impractical to be tested in full. In such scenarios performing GUI testing can be extremely difficult. Several coverage criteria are to be fixed to limit the amount of branching to be covered.

2. The GUI controls can be inter-related, synchronized and can be inter-dependent. This would lead to great difficulty in testing the individual parts or certain sectors in the GUI in isolation. To test a small part of the GUI it might be necessary to include a huge part of the software. This would add further complexity to the test case design and would be very resource intensive. Inter-dependencies between GUI objects are also to be thoroughly understood and applied in generating tests in this situation.

3. The inputs and outputs are graphical and might depend on the position of the object in the screen displayed. This is a significant problem when tools that rely in the proper position of text and graphics in the GUI. The data can be there in the GUI but the tests may not find them as they are not displayed in the specified position. Moreover, changes in the position of buttons, menus and other layouts can fail the already generated test cases.

4. Regression tests are very important to check if the new modifications made have introduced any bug in the software. They are supposed to be executed regularly, often nightly for agile development environments. GUI test can be significantly affected by the regular change in the layout and they can consume significant resources and time. As such regression testing is very difficult.

5. It is continuously changing as it is scaling up during the development time [19]. When we introduce new layout, change the existing layout or add additional GUI items the existing test cases are also to be updated. Regular updates might be a continuous process for such a system. In case of incremental development process this might create great challenges.

6. Automated or not automated GUI test case generation and execution generally consumes a lot of time and resources. This might not be always readily available. Making these processes efficient is always a challenge.

7. There are several different kinds of coverage criteria which can be used as guidelines to determine the testing adequacy for code. However traditional coverage criteria like line coverage, branch coverage, block coverage, event coverage, etc might not be suitable for determining the testing adequacy in GUI testing. The GUI of a software system differs in abstraction level with the
underlying application code which makes the mapping between the GUI events and the underlying code not so straight forward [5]. This might hinder the use of traditional coverage criteria infeasible for GUI testing. Confusion to decide what coverage criteria are to be used to check that the application GUI is adequately tested.

8. When it comes to determining if a GUI test has passed it might be very confusing. In situations when an unexpected GUI state generates an unexpected screen it is difficult to compare the generated result with the expected result. This makes the further execution of the test case useless. There might be situation where it is difficult to pinpoint errors when the final output is correct but the intermediate outputs are incorrect. Similar situations make it difficult to determine if the test case passes or fails.

9. Combinatorial explosion might be a huge problem when the software deals with a lot of branching and choices. One screen might generate many other screens and this might continue as we traverse into the GUI. Considering all the choices and branching can generate a huge combination of screens and make it very impractical to test the GUI.
5 Optimization efforts in GUI testing

Optimization is often an important issue while running huge set of tests. Especially when it comes to GUI testing, the tests are longer and more resource intensive. Reduction of such test cases becomes a vital task in order to make their use practical and more efficient. A huge volume of test cases that takes days to run can be a problem when it has to be executed regularly in an iterative and rapid development environment. Nightly builds and smoke tests will not be possible in such scenario. So the volume of test cases must be reduced. But there is a big challenge in this process. Reduction in the volume of test cases should not significantly hamper the fault detection effectiveness of the tests. Similarly the test coverage is also desired to be very high. Many efforts are being made to reduce the amount of test cases. Some of the reduction techniques with various approaches in GUI testing are discussed in this section.

5.1 Call Trees
The tree model of the method calls and program behavior are constructed during testing. After the construction of this call Trees, algorithms such as ‘overlap aware greedy search’ is applied on it to reorder the tree so that the equal amount of test coverage is achieved faster than the original tree. The metric to evaluate for prioritization can be coverage effectiveness as used by [15]. The experimental results from [15] show the implementation to construct such tree took 13% more time in testing. But as a result achieves equivalent coverage much faster, reduces 45% test suite reduction and consumes 82% less time.

5.2 Model Based
The concept of Model based development is that the development process is centered on a formal description or specification of the proposed system. Thus it is also called specification based development. This specification can be subjected to various kinds of analysis like completeness and consistency analysis, model checking, theorem proving and test case generation. It is possible to predict that the model is correct by validating it through manual inspections, formal verification, simulation and testing [14]. The implementation can then be automatically generated. It is also possible to automatically generate specification based test from the models. Generally large number of such conformance tests is generated. Reducing such large number of tests without hampering the coverage or fault finding capability of the tests can significantly reduce the cost involved in generating, executing, storing and maintaining such test suites. Test results from [14], shows that such automatically generated conformance test suites can be significantly reduced while the desired coverage is maintained. However there is some reduction in the fault finding capability. It depends on the type of application developed how much tradeoff is possible against test case reduction and the fault finding capability. In [14], the effect on test suite size and fault finding capability were tested against various kinds of coverage criteria like Variable Domain Coverage, Transition Coverage, Decision Coverage, Decision Usage Coverage, Modified Condition and Decision Coverage (MCDC) and MCDC Usage. They observed that all the coverage criteria reduce the size by more than 80%. But MCDC with Usage seemed to be the least
sensitive to the effect on fault finding capability. Its fault finding capability seemed to be reduced by only around 7%. However due to the reduction in the fault finding capability, they suggest that only test size reduction maintaining some coverage criteria is not sufficient.

5.3 Call-Stack Coverage
A call stack is a sequence of active calls pushed into a stack-based architecture. The main idea is to push the methods in the stack when they are called and then pop them out later. It is considered that 2 same test cases will generate the same set of call stack and one can be removed in order to save the resources and time consumed [13]. Call stack coverage has a special benefit of encapsulating the context in which the methods are called, over the other practices such as line or edge coverage. Several other additional advantages are also discussed in [13].

5.4 Operational extraction
Harder et al. [21] uses dynamic invariant detection techniques to reduce the test suites. While running a program they maintain an ‘operational extraction’ which is the mathematical picture of the program’s dynamic behavior. The main concept in this technique is to run the tests in turns and if the test doesn’t change the ‘operational abstraction’ the test case is discarded. The special feature of this technique is that it uses the dynamic program behavior of the application rather than the syntax of the program.

5.5 All-uses criteria
Apart from using some structural coverage criteria to reduce the test suite size some people have ventured to use some data flow coverage criteria. One of such work is presented in [22]. They use all-uses criteria to reduce the test suites. All-uses coverage criteria basically mean that all paths from a location where a data variable is defined to every location where the variable is used are to be covered. As a result they observed that reducing the size of the test suite keeps the all-uses coverage constant barely affects the fault finding capacity of the test suites.

5.6 Selective redundancy
The reduction in the size of the test suite intuitively gives a feeling that the fault finding capacity of the test suite will decrease. Looking at the past works conducted by researchers all over the world, it is still confusing to decide if the fault finding capacity is seriously hampered or not. Some report that it is hampered and some report that it is not. So in such situation an approach that seems to be in the middle can be applied. One such work can be found in [38]. They identify different kinds of redundancies in the test code. Some redundancies seriously hamper the fault finding capacity and some redundancies don’t. So the idea is to only remove such redundancies that barely hamper the fault finding capacity.
5.7 Primary and Secondary criteria
Jeffery and Gupta [38] present a new approach of combining 2 different coverage criteria as primary and secondary to introduce selective redundancy and improve the fault detection effectiveness. For example call stack can be used as secondary and branch coverage can be used as primary coverage to reduce the test suites. The main idea behind this approach is to use the primary criteria to find out the redundant test cases and to use the secondary criteria to decide to either remove or not to remove that test case.

5.8 Clustering algorithms
Podgurski [24] and Dickinson et al. [25] applies clustering algorithms to reduce the test suites. The idea behind it is to use some test profiles to arrange the test cases from the pool into some clusters. Certain dissimilarity metric is used to arrange the test cases into clusters. Now test cases that are in the same clusters are considered to be very similar and thus only one test case per cluster or certain amount of test cases from each cluster is executed.

5.9 Event based coverage
Event based coverage in GUI is also a very effective criteria where test cases are designed as sequences of events. As such event coverage criteria like event (each event in isolation is the coverage requirement) and event-interaction (unique pairs of events are the coverage requirement) are also used to reduce the test suites [13].
6 Some popular GUI testing tools

In this section, some popular GUI testing tools currently available in the market are discussed. A brief review of their major features is also done. Most of the tools available today in the market involve capture and playback capabilities. Generally, humans interacts with the SUT for recording the mouse and key events into scripts and secondly with the test tool to insert verification points, specify data storage and edit test scripts.

1. IBM Rational Functional Tester: IBM Rational Functional Tester is developed by the Rational Software division of IBM [40]. This test software is to be used by quality assurance teams to write tests that mimics the human actions. It is primarily intended for regressing testing of the SUT. The recorder records the user actions and events and generates either Java or Visual Basic .net applications. Along with the scripts, it also generates a series of screen shots to present a visual storyboard. Testers can edit either the recorded scripts or these screen shots to create tests that fit the actual requirement. The tester has to insert various verification points where different properties are compared with the values recorded when the tests were captured. If the values are same then the test passes or else it fails. All the test runs and results are logged in Rational Functional Tester log. It uses an object map to identify and find objects in the application during testing. This map is automatically prepared during the recording phases of the user actions upon the SUT. Generally, object recognition is difficult when the GUI is updated and the existing GUI objects are changed. They are different during test execution as compared to what they looked like, when they were recorded. IBM Rational Functional Tester uses a special functionality called ‘ScriptAssure’ to ignore small differences in the GUI objects and identify them from the object map. The level of allowed discrepancy is set by the tester. Data driven testing is also supported as multiple data can be supplied to the tests by adding to the test data pools. This prevents changing scripts when running the tests with additional test data.

2. HP WinRunner: WinRunner is another GUI test automation tool that can record and playback user interactions as test scripts. It uses a proprietary Test Script Language (TSL) to record the tests. This script allows easy customization and parameterization of user inputs to alter the recorded tests and modify it according to how the tester’s wish. It can also be used together with HP Quick Test Professional. It was first developed by Mercury Interactive, which was later acquired by HP.

3. HP Quick Test Professional: HP Quick Test Professional is a tool developed by Mercury Interactive, which was later acquired by HP [43]. It is suitable for functional and regression tests. It is a part of HP Quality Center tool suite. It supports keyword and scripting based GUI testing. It can be used for enterprise quality assurance. It uses Visual Basic Scripting Edition (VBScript) for recording user interactions with the SUT.

4. RIA Test: RIA Test is a GUI test automation tool for Adobe Flex applications [41]. It is built on top of Flex automation framework, and supports Flex 2, Flex 3, Flex 4, Flex 4.5 and AIR applications. It has
an Action Recorder to record all the user interactions with the SUT. It also has a Component Inspector to identify the GUI objects in the SUT. It records the user interactions in an actionscript-like RIA script. It also has a built-in script debugger with breakpoints, stepping and variable value lookups.

5. Test Partner: Test Partner is GUI test automation software from Micro Focus. It allows two primary methods to automate GUI tests. The first one is a code-oriented approach using Visual Basic for Applications (VBA). With VBA, the tester is provided with a rich Integrated Development Environment (IDE), which facilitates the tester with excess to core VB library and capabilities like debugging and error handling. The second method is to use a visual storyboard based environment. It is also known as Visual Navigator where the tester is provided with different screenshots of the SUT. The tester also has the ability to add various types of test logic using these screenshots.

6. Selenium: Selenium is a record and playback software-testing tool for web applications [45]. It also provides an IDE to manage and edit recorded tests. A domain level language called ‘selenese’ can be used to write scripts in different popular programming languages like C#, Java, Groovy, Perl, PHP, Python and Ruby. It is also compatible with most of the modern browsers and can be deployed on Windows, Linux and Macintosh platforms.

7. Maveryx: Maveryx is an automated functional GUI and regression test tool for Java applications [44]. It is very different from other similar automated test software because it does not use an Object Map to recognize GUI object from the SUT. Maveryx’s search engine supports advanced fuzzy mapping algorithms to recognize the GUI objects in execution time. The approximate matching approach of Maveryx has two main advantages. The first one is that tests can be prepared even when the application is not fully prepared. The second advantage is that it is resilient to frequent application changes.

8. SWTBot: SWTBot is an open source, java based functional testing tool for the Standard Widget Toolkit (SWT) and Eclipse based applications [46]. It provides record and playback tests and integrates very well with Eclipse. It can run on all the platforms where SWT can run. SWTBot provides very simple APIs so that it can be used by anyone, not just developers.

9. SilkTest: SilkTest is a functionality-testing tool for enterprise applications [42]. SilkTest offers various clients. SilkTest Classic uses a domain specific language, which is an object-oriented language. Silk4Js is another client that supports automation in Eclipse using Java as a scripting language. Similarly, Silk4Net can be used to automate test in Visual Studio using Visual Basic (VB) or C#. SilkTest identifies all windows and controls of the SUT as objects. It can also identify mouse movements and keystrokes in the SUT. Its record and playback or descriptive programming can be used to capture dialogs or other GUI controls in the SUT.
7 Implementations at Catalysts

Catalysts GmbH is a fast growing software company, located in Austria. As an industrial application of the GUI testing concepts some work were carried out in Catalysts during this thesis preparation. The GUI test automation was tried in two projects. These two implementations are described in this section.

7.1 Implementation to test the in-house project called ‘taskmind’:

The first implementation is introducing GUI testing one of Catalysts’ in-house projects called ‘taskmind’. It is a web application prepared in adobe flex which promotes cooperation in a team by basically helping managing task assignment between people working in a huge organization. It has an html login screen which redirects it to the main flex application. The main flex application contains panels for specific purposes like searching projects, contacts, detail viewing of tasks, navigation, etc. Each panel has various controls like labels, buttons, text areas, checkbox, etc.

‘Silk4J’, the SilkTest plug-in for Eclipse is used to perform User Interfaces functional testing, as it supports flex applications. Silk4J’s powerful test automation capabilities make it the perfect solution for regression, cross platform and localization testing across a broad set of application technologies including AJAX and Web 2.0, Rich Client Platform (RCP) or .NET and client/server.

The basic approach used here is to use the ‘Record Locater’ functionality to automatically obtain the ‘XPath’ of the controls and then use it to obtain the handle to the UI control while running the test cases. A library which consists of all the panels in the GUI is maintained. Each panel class contains handles to all the controls inside the panel and the methods that can be called using those controls which will call actions like click the button, type in the textbox, check the checkbox, etc. This is entirely a manual process.

Test suites are maintained for each type of activities which contain test cases related to that activity, e.g. ‘settingsTest case’ contains test cases related to changing password, changing email, etc. These test cases are again maintained manually trying to imitate the flow of the UI as far as possible. Each test case gets the handle of the controls in the UI via the panel class they belong to and can operate in the GUI with actions like mouse clicks and keystrokes.

The hard work paid off during running these test cases. It is entirely automatic and all the programmatic clicks and keystrokes are very well carried out, mimicking a user working around the application GUI. If the asserted statements doesn’t find the actual value similar to the expected value of text or other attributes of the UI controls the test fails. This can be repeated any number of times. Creation of some items is carefully followed by its deletion to allow this functionality. As the project is carried out in an agile team, continuous testing is very important. This automatic testing serves very well in this scenario and immediately reports any flaw during the development. As the project advances and new GUI emerges, the test project is also to be extended so that all areas of GUI is covered and tested.
7.2 Implementation 2: GUI test automation implementing Ordered Test Suite (OTS)

7.2.1 Background
An extensively large transportation and logistics company, with their own group of developers was building software. They would perform their entire task through this software. Obviously there were huge number of requirements and the quality of the software was of utmost importance. The development started and soon the system became very huge as there were lots of user requirements that should have been addressed by the software. Because of the size and complexity it was very difficult to keep track of what features were implemented and what implementations were working properly. It was almost chaos. As such the main objective of the project was to construct GUI test cases and keep a track of what all features and requirements are fulfilled.

The SUT was a client/server application. It was an eclipse RCP application. As the development of the system progressed, they started writing tests and soon a group of testers were assigned the responsibility to prepare functional tests and test the software. The functional tests were maintained in word files which described what steps were needed to be followed and what should be verified for the test to pass. Each tester would run the functional test manually simulating the condition as if it was run by the real user. Gradually the amount of test cases grew very large and it became impossible to be executed manually by humans. For instance there were 2500 functional test cases simulating user using the system and a group of 12 testers took a month to complete the testing. This time was simply too much and often by then another iteration would have been finished and it would me time to run the tests again, making the testing useless. This problem was only growing and it showed no signs of reduction until the tests were automated instead of running them manually.

As such the task was to coordinate with the existing team, write tests, scripts and develop a platform making automatic GUI test implementation. The main requirement was that our automatic GUI testing should dramatically reduce the time taken in the process.

Further we had a situation where a test case could be blocked because its predecessor test cases might be blocked or missing. For example if we have A -> B -> C, meaning C depends on B and B depends on A then if A was missing or blocked then both B and C would fail. Similarly if B was missing or blocked then C failed. It would create a big problem unless we knew which test case to run and fix first. In this scenario A was to be run first, then B and C respectively. So some kind of mechanism was also to be implemented to solve this dependency problem.

7.2.2 SilkTest
SilkTest is a powerful software test management, automation and performance testing tool from Micro Focus. It is very useful for automating regression testing as it delivers advanced test automation capabilities. It is widely used by companies to replace the cumbersome manual testing. Silk4J was first introduced by SilkTest in 2008. Developers and testers can use Java as the test script to develop tests
using Silk4J package. It is an eclipse plug-in and very useful in testing enterprise software applications. It supports testing of a broad range of technologies like:

- Adobe Flex applications
- Java SWT applications
- Windows Presentation Foundation (WPF) applications
- Windows Application Programming Interface (API)-based client/server applications
- xBrowser applications

It provides a java runtime library which is compatible with JUnit. This means we can leverage the JUnit infrastructure making it possible to use JUnit to create tests. Among its many features that support rapid test development one very important is its Standalone Action Recorder for rapid test creation. It is basically record and playback. But this is not how we proceed in our implementation. We use a special feature inside this action recorder which is called locator spy. Once this function is started then we get the XPath location/address of each UI object in the application screen. The idea is to use these paths to access the GUI objects from our java code. Its object recognition approach is very robust and this also helps to decouple the graphical unit of the Application Under Test (AUT) from the test scripts that we write for functional testing. This feature is also to be exploited in our implementation.

7.2.3 XPath

W3C describes XPath as a language for addressing parts of an XML document. Its main purpose is to address parts of an XML. It also allows basic strings, booleans and numbers manipulation. It uses compact and non-XML syntax to make its use possible within URIs and XML attribute values. As it is used to denote navigation in a hierarchical structure of an XML document, it is named XPath.

Silk4J supports a subset of the XPath query language. Silk4J provides a tool called Locator Spy to identify the caption or the XPath locator string for GUI objects. We can use these XPath strings with methods like Find and ‘FindAll’ to locate and use those GUI objects in the test cases. The use of Locator Spy makes it sure that the XPath we are using is correct. It relies in the current context i.e. the current location in the hierarchy on which the find method is called. All XPath expression depends on this position similar to a file system. Thus we get the power of Dynamic Object Recognition with XPath in Silk4J. Dynamic object recognition works very well in certain situations.

- When the GUI is changing: For example, to test clicking a button in a dialog where both the dialog and the button name are changing, using dynamic object recognition enables you to test the button without worrying about their names. We can however verify if we are testing the current button.
- Web application that includes dynamic tables or text: In situations where we want to test a table which is displayed on mouse over to a certain text, we can use XPath to address the table without displaying it or worrying about the mouse over that text.

It has other huge advantages as it is a common XML based language defined by the W3C and it needs only single object at the time instead of a pool of GUI objects which can be pretty resource intensive at times.
Some examples of XPath strings are:

<table>
<thead>
<tr>
<th>XPath String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desktop.find(&quot;/Shell[@caption='SWT Test Application']&quot;)</td>
<td>Finds the first top-level Shell with the given caption.</td>
</tr>
<tr>
<td>desktop.find(&quot;//MenuItem[@caption='Control']&quot;)</td>
<td>Finds the MenuItem in any hierarchy with the given caption.</td>
</tr>
<tr>
<td>myShell.find(&quot;//MenuItem[@caption!='Control']&quot;)</td>
<td>Finds a MenuItem in any child hierarchy of myShell that does not have the given caption.</td>
</tr>
<tr>
<td>myShell.find(&quot;Menu[@caption='Control']/MenuItem[@caption!='Control']&quot;)</td>
<td>Looks for a specified MenuItem with the specified Menu as parent that has myShell as parent.</td>
</tr>
<tr>
<td>myShell.find(&quot;//MenuItem[@caption='Control' and @windowid='20']&quot;)</td>
<td>Finds a MenuItem in any child hierarchy of myWindow with the given caption and windowid.</td>
</tr>
<tr>
<td>myBrowser.findAll(&quot;//td[@class='abc*']//a[@class='xyz']&quot;)</td>
<td>Finds all link elements with attribute class xyz that are direct or indirect children of td elements with attribute class abc*.</td>
</tr>
<tr>
<td>myBrowser.find(&quot;//FlexApplication[1]/FlexButton[@caption='ok']&quot;)</td>
<td>Looks up the first FlexButton within the first FlexApplication within the given browser.</td>
</tr>
</tbody>
</table>

Table 2: Examples of XPath

7.2.4 OTS Algorithm

The main idea behind the OTS algorithm is creating a dependency graph. Some test cases might be dependent on other test cases. For example test case ‘T1’ might create some resource. Test case ‘T2’ may need this resource to run. Now it is not possible for ‘T2’ to run before the execution of ‘T1’. This created a necessity that we need to determine these dependencies and run the test cases in some sort of order. When we know which test case is dependent on which create a dependency graph. Finally we run the test cases which are not dependent on other test cases first and then those that depend on these test cases. The test cases are given ranks according to their dependency levels. Lower the rank lesser the dependency.

1. Add all test cases as nodes to a directed graph.
2. Add all dependencies between test cases as edges between the nodes. An edge from A to B means that A requires B.
3. Set variable rank to 1.
4. If there are nodes in the graph, do following:
   a. For every node without outgoing edges do following:
      i. Set its test rank to the value of rank variable.
      ii. Remove the node from the graph (including its edges).
   b. Increment rank.
5. Create a list with all test cases.
6. Sort test cases by test rank.
For example in the above directed graph, let the nodes represent the test cases and the arrows between
the nodes show the dependencies. If an arrow is heading from A to E, it means that A depends upon E. Now, let us see how these test cases from A to G can be ordered based upon the above algorithm. First of all we set we set rank=1, and get all the nodes without outgoing arrows i.e. D, G, F and C. We set their ranks to be 1 and remove them from the graph. We increase rank=2. Next we have E and B as nodes without arrows going away. So we set their ranks to be 2 and remove them from the graph. We increase rank=3. Now only A remains with no arrows going away so we set its rank to 3. Now we sort all the nodes with increasing ranks. This gives us the ordered test suite.

<table>
<thead>
<tr>
<th>Order</th>
<th>Test case</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Ordered Test Suite

7.2.5 General structure of the project
The SUT here was an Eclipse RCP application. It’s a client server application. Silk4J Eclipse plug-in was used to write the test. Basically there were around 2500 test cases which were in word documents explaining the state, action, list of test cases that the test case under execution depends upon and the intended results of each test. The first thing was to define the base state of the application. With Silk4J it was possible to define the base state fairly easily. This was what the application would look like when it was first loaded. Then, the crucial job was to represent these manual tests into java code that can be launched automatically. Our approach was to keep the logic part and the data part separate. So we had property files which were simple text files to hold data necessary for each test and separate java file that
handled all the logic i.e. things like what menu to click, what text to be filled, what buttons to be clicked and what results to compare. These java classes also had a set of XPATH locations of the GUI objects that was needed to be accessed from the code. This was obtained by using the spy locator tool provided by SilkTest. Silk4J can access the GUI objects of the SUT using these XPATH strings and do operations like clicking and typing on them. The tests implemented ‘JUnit Test’ so we could run these tests as JUnit test cases exploiting the JUnit assert statements to compare the expected and obtained results and determine if the tests passed or failed.

Now test cases were organized according to the usecases. Each usecases were defined as a folder with special names like ‘22_04_01’ and there were a number of test cases under this folder representing different actions related to this particular usecase. A test case was named combining the usecase name and its serial number. For example a test case that belonged to the use case 22_04_01 may be 22_04_01_45. Each test case also had a set of property files. These were different set of data that is to be used by the test case. Each test case had several data files so that various combinations of data could be tested with the test case. These property files also contained the names of the other test cases which should be run prior to the current test case. These test cases were mentioned before in the word file based test cases provided to us.

7.2.6 Running the test cases:
There was a special file which contained the list of the test cases and usecases that would be executed when running the test. If the name of the test case is listed then only this test case was run, but if a usecase name was listed then all the non-blocked and implemented test cases are executed. This file gave us the control on what test should be executed. When the tests are run, each time the application would start from the base state i.e. the state which appears when the application is first started. Then as per the commands written in the java code different menus were opened and data was fetched from the property files and inserted and action was performed in the GUI. This automation of the GUI test was visually seen. Finally the desired output was anything like text displayed in the screen, status of the application, the count of the data displayed, etc. This was again read from the GUI and compared with the desired values. This determines if the test passes. As JUnit was used with the test application, we simply used the JUnit assertions to determine if the test passed or failed. Before running the test cases all the other test cases that were needed to be executed first were determined. A dependency tree was created and ranks were given to the test cases. These ranks were used to execute the tests in order of their dependencies which makes sure no tests were blocked because they depended upon other test cases.

7.2.7 Results:
The result of the GUI tests was saved in a CSV file. This file contained details like the test case name, rank in the dependency graph, status like if it was blocked or should run, missing dependencies i.e. the list of the test cases that were to be run before executing it, pass or fail status after its execution, message regarding why a test case failed if it failed, etc. After looking at this file we knew if the test passed or if it failed then we also know the reason why it failed. It also produces a readme file which contains other results which give us the bigger picture of the test run. It shows the start and the end times in hours, total number of test cases run, total number and percentage of test cases that was
blocked because the test cases that they depended upon were not run before them, total number and percentage of the executable test cases, total number and percentage of the successful and failed test cases, total number and percentage of test cases that didn’t run because of errors (in the GUI or the program logic) and the most frequently failure reasons.

7.2.8 Overall impact of the project:
1. It was lot easier for the client to understand what usecases were implemented and functioning properly.
2. The manual work that was not possible to be finished in a month by 12 dedicated testers was now run automatically and completed every night.
3. The dependency issue blocking many test cases to run was also solved and test cases ran smoothly in order of their ranks in the dependency graph.
4. The system allowed testing with different combinations of data fairly easily, as data was provided via separate property files and we could have many property files for a single test case.
8 Model Driven Technique for GUI test automation

8.1 Introduction
Model Driven Engineering (MDE) is an approach to develop software systems using models to represent the whole system or parts of a system instead of writing code. Model Driven Architecture (MDA) is an initiative by OMG for the realization of MDE. Its main purpose was to support interoperable, portable and reusable software systems [31]. Models, metamodels and transformations form the basis of MDE. According to Bezivin [33], the basic principle in MDE is that ‘Everything is a model’. Two terms ‘representation’ and ‘conformance’ describe relations in MDE. A system can be represented by a model and a model should conform to a metamodel. The software is developed in a number of steps as a chain of model transformations. There are various levels of models of the system that can be transformed into each other. MDA consists of concrete specifications and process descriptions for model driven software development which form ‘an approach to using models in software development’ [34]. It can be used for purposes like abstraction and automated development of an application. MDA framework defines how models in one language can be transformed to models in another language. One basic use of MDA is to model a system in a platform independent modeling language e.g. UML and transform this platform independent model (PIM) into platform specific model (PSM) by using transformation specifications which is actually a mapping between the PIM and some programming language like Java. This basically separates the platform independent and platform specific system specifications. This approach has several advantages like

a. As system developers and designers can get better overview of the system using models, systems are easier to understand.

b. Application is rather transparent for domain experts.

c. Already developed systems are easier to maintain [35].

d. Automated generation of most of the code is possible from models which results in less code to be written by humans [31].

The transformation between models of different abstraction levels is performed by a transformation engine by executing a transformation model instance [35]. This transformation model should have definitions of how to transform input models to output models and the knowledge of the metamodels of both the source and the target models. In MDA all metamodels should also conform to their higher level metamodels. MDA’s Meta Object Facility (MOF) is a metamodeling architecture that provides a means to define the structure or abstract syntax of a language or data. MOF can be viewed as a Domain Specific Language (DSL) that can be used to define metamodels. Metamodels basically describes models, allowed metaelements, their properties and relationships and insures if the instantiated models are well formed. Each model should conform to its metamodel. Practically, they are an abstract syntax for models. OMG also has a standard set of languages for model transformation called Query/View/Transformation (QVT). It covers transformations, views and queries together.
In the field of software verification and validation, the trend has been to test the software implemented based on the system specifications. Today, many research works has been going on in using model based approaches in software testing. Model Driven Testing is a form of model based testing that is based upon MDE or in other words that uses metamodels, models and a set of model transformation rules. Model Driven Testing is defined as the application of MDA principles to software testing [36]. It has a big advantage as test can be automated and testing can be shifted to rather earlier phase of the system development. It further reduces the development time and reduces the maintenance effort. The test cases that are developed are even independent of any specific implementation. We can use models to define test at various levels of abstraction similar to MDA. Platform Independent Test model (PIT) specify tests on high abstraction level which is transformed to Platform Specific Test model (PST) by using transformation rules. This contains information of the technology upon which the tests can be executed. The final phase is to generate the code from PST in the specific programming language so that we can run the tests.

8.2 EMF

An Eclipse based MDA tool called Eclipse Modeling Framework (EMF) is a powerful tool in the field of MDE. It facilitates developing model based applications in Java. ‘The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model’, [29]. Basically EMF facilitates model development, model transformation and Java code generation [31]. It can be used to model domain models. Before instantiating domain models a metamodel of a model is to be specified. The metamodel basically describes the structure of a model. A model can be viewed as an instance of a metamodel. Models are specified using XML Metadata Interchange (XMI) and with EMF we can generate java classes for the model, edit them or view the model graphically. Metamodels can be created using annotated java, UML, XML or other different modeling tools and imported into EMF. This is possible because EMF provides strong interoperability with other EMF-based tools. EMF based transformation engines can be used to transform EMF based models and finally to generate source code or test code.
8.3 Tefkat

Tefkat is an EMF based model transformation engine. It is implemented as an Eclipse plugin which uses the EMF to handle models based on MOF, UML2 and XML Schema [4]. It has a simple syntax which is very similar to SQL. It is specially designed for writing scalable and reusable transformations using high level domain concepts. ‘The Tefkat language is declarative, logic-based, and defined in terms of a MOF metamodel’, [36]. It uses meta-models, source models and transformation rules to generate target models. These transformation rules are basically mappings from source meta-model to target meta-model. The input to the engine is the source metamodel, source model and a set of transformation rules. The output from the engine is the target model.

8.3.1 Basic structure of Tefkat transformation specification:

A Tefkat transformation is normally a single file with an extension ‘.qvt’. The start of the file is a definition of the transformation. A transformation is a named entity with input and output parameters. The input is the source metamodel and the output is the target metamodel. Figure 8 shows a general structure of this definition. It shows only one input and one output metamodels but they can be both multiple metamodels expressed by comma separation. The default context for all FORALL and WHERE expressions will be the first source instance and the default context for all MAKE and SET expressions will be the first target metamodel.

```
TRANSFORMATION smc2xunit : smc->xUnit

NAMESPACE http://smc
NAMESPACE http://xUnit
```

*Figure 8: Transformation definition*

The transformation definition is followed by any number of class definitions, rules, pattern definitions, and template definitions. First comes the class definitions. The transformation engine can only resolve the class names if we specify the set of models that define those types with a ‘NAMESPACE’ followed by the URL that resolves to the EMF Ecore instances or in other words the models that are used in the transformation. This is also shown in Figure 8. A basic example of a Tefkat rule is shown in Figure 9.

```
RULE Model_2_TestModel(model, testSuite, interaction)
   FORALL Model model, Interaction interaction
   WHERE interaction.owner = model
       MAKE TestSuite testSuite FROM model2testSuite(model),
               TestCase testCase FROM interaction2testCase(interaction)
       SET testCase.name = interaction.name,
           testSuite.testCase = testCase,
           testSuite.name = model.name
;
```

*Figure 9: Rule definition in Tefkat*
Tefkat rules are primary action elements. Basically they consist of source and target that share variables. The rules match and check for constraints in number of objects from the source model and then create a number of objects in the target models. Tefkat rules are named and consist of optional parts like FORALL, MAKE, SET and WHERE. The FORALL key-word selects all the elements of the specified type in the source model for which the rule has to be triggered. The MAKE key-word creates an object in the target model for each selected element of the source model when the constraints checked by the WHERE key-word is satisfied. The SET key-word sets the values to the created objects in the target model. For example in rule definition shown in Figure 9 it propagates through all the interactions of all the models as suggested by the FORALL key-word and checks where the interaction’s owner is the model. When this condition is satisfied it creates a TestSuite of type TestSuite and test case of type Test case as suggested by the MAKE key-word. Then finally with the SET key-word it sets the name attribute of the test case as the name attribute of the interaction, test case attribute of the TestSuite to the test case created and the name attribute of the TestSuite created as the name attribute of the model.

8.4 MOFScript

MOFScript is a model to text transformation language. It supports generation of implementation code or documentation as textual output from models that are based on metamodels. An Eclipse plug-in of this tool is also available. The tool is based on EMF and Ecore as a metamodel framework. MOFScript basically requires the source model, its metamodel and a set of rules to map the metamodel to the text to be generated. When executed it applies the rule on the input model step by step. The output can be carefully tailored so that it can be syntactically correct to be implemented in specific environment. The script can add and play around with text as per the requirement in specific language. With careful manipulation of text any operations like declaring variables, making method calls or declaring loops and iterations can be achieved. Figures 10 to 13 show some examples presented in [37] which explains the basic syntax MOFScript. Detail guidance on using MOFScript is presented in [37]. A textTransformation key-word defines the name of the module as shown in Figure 10. It can be any name. Parameters are passed to it, which are the input metamodels. Then the entry point rule which is named ‘main’ like in other programming languages, defines where the transformation execution should start. It has a context which defines the element type of the metamodel that will be the start point for execution.

```
textTransformation UML2Java (in uml:uml2)

uml.Model::main () {
  self.ownedMember->forEach(p:uml.Package)
  {
    p.mapPackage()
  }
}
```

*Figure 10: MOFScript module definition and main entry point for the transformation*

The rules in MOFScript are logically similar to functions. They have a context type which is a metamodel type and optionally a return type. A rule returning some values can be reused in expressions inside other
rules. While returning a value result assignment statement is used. The body of the rule contains statements. Figure 11 demonstrate how rules are defined.

```cpp
uml::Package::mapPackage () {
    self.ownedMember->forEach(c:uml::Class)
        c.mapClass()
}

uml::Class::mapClass() {
    file (rootdir + package_dir + self.name + ext)
        self.classPackage()
    self.standardClassImport ()
    self.standardClassHeaderComment ()
    <%
        public class %> self.name <% extends SomeClass %>
    self.classConstructor()
    <%
        /*
            * Attributes
            */
    %>
    self.ownedAttribute->forEach(p : uml::Property) {
        p.classPrivateAttribute()
    }
    newline(2)
    <%}%>
}
```

*Figure 11: Rules in MOFScript*

A property is a constant string value assignment and a ‘var’ is a variable whose value can be changed during the execution of the module. They can be either local, i.e. defined inside rules and blocks, or can be global i.e. defined inside the transformation definition but outside of the rules and blocks.

```plaintext
property packageName = "org.mypackage"
var myVariable
```

*Figure 12: Defining constants and variables*

The output is basically a file which is defined using a key-word ‘file’ and passing the location of the file as the parameter to it. Another key-word called ‘println’ can be used to write on the output file. Figure 13 demonstrates how the output to a file is done in MOFScript. Iterators can also be used to iterate over collections of model elements. Very common way to iterate over a collection of model elements is using the forEach key-word. Conditions can also be set for the forEach iterator which acts like filters and only selects the elements when the condition set is true. If statement is be used to support conditional branching. In Figure 13 the first forEach loop iterates through all the objects in the collection of the type Operation. However the second foreach loop is supplied with additional condition. So it iterates over only those objects in the collection of type Operation which name starts with the letter ‘a’.

40
The methodology followed in the thesis is proposed in [31]. They have tried to apply the principles of MDA in software verification and validation using sequence diagrams. Sequential diagrams are well known to capture the behavioral elements in a UML design. They give a clear picture of how the software behaves and how the components in an application interact with each other. In a use case driven software development process, sequential diagrams are perfect tool to understand the interactions between components. They propose to write the test cases using sequence diagram and finally generate platform specific test cases from it using MDA.

Following this approach, the generation of test cases is done in two steps. The first step is to form a model from the sequence diagram and transform the modeled UML sequence diagram into an intermediate testing model which is a general xUnit test case model. Tefkat is used for this purpose. This kind of transformation is also known as horizontal transformation. In this kind of transformation, the abstraction level of the model is maintained i.e. in this case PIM to PIM. This step generates generic test cases from sequence of method calls that are extracted from the sequence diagram. The second step is
to transform this xUnit model into executable, language specific JUnit or SUnit test cases. In this thesis we will only work to generate JUnit test cases that can run only in Java. MOFScript is used for this purpose. This kind of transformation is also known as vertical transformation. In this kind of transformation the abstraction level of the model is not maintained i.e. in this case PIM to PSM. The data for the test cases are taken from a file in [31]. In this thesis we implement another model called data model to supply data to the test cases that are to be generated. Both xUnit and data models are fed to the transformation in this step as inputs to generate JUnit test case as output. After the second step is completed we can execute the JUnit test cases to check if they pass. The test results are checked by comparing the expected values against the actual values returned. An example is created to check the application of this approach. The advantages of using this model driven approach to generate test cases are also discussed in [31].

Figure 15: An overview of the methodology [31]

Figure 15 shows the overall methodology involved while generating test cases from sequence of method calls that is part of the sequence diagram. Component 1 and component 2 are metamodels of the sequence of method call (SMC) and xUnit respectively. Component 3 and component 4 are the rules or mappings to transform SMC model into xUnit model and xUnit model into executable test code. Component 7 is the source model from which we need to generate test cases. Component 5 and 6 are the MDA tools used in the process. Component 5 represents the Tefkat transformation tool which is used for horizontal model to model transformation. Component 6 is the MOFScript transformation tool which is used to generate the executable test cases. Tefkat tool needs SMC metamodel, SMC model, xUnit metamodel and the model to model transformation rules to generate an intermediate output model represented by component 8. This is xUnit model and is an input to the MOFScript transformation tool. Component 9 is the test data which contains the parameter values to be passed to the methods and the expected return values from the method calls. Component 10 is the header files which contain code information. For the sake of simplicity, in case of the example presented in this
thesis, component 9 and 10 are disregarded and this information is hardcoded while generating the test cases. So, the input to the MOFScript transformation tool are the xUnit metamodel, xUnit model generated by the Tefkat tool and the model to text transformation rules. MOFScript transformation tool then finally generates the executable unit test cases.

8.6 SMC Metamodel
The metamodel of sequence of method calls is the same as the one used in [31] only with some minor adjustments. They have created this metamodel with a view to use UML sequence diagrams only for generating test cases. More complex aspects of sequence diagrams such as connectors of message events are ignored. The simplified metamodel for the sequence of method calls from [31] with some minor adjustment is shown in Figure 16. All SMC models have to conform to this metamodel.

![Figure 16: SMC Metamodel [31]](image)
Figure 17: Tree view of the SMC metamodel shown in Figure SMC viewed in editor mode in Eclipse

The SMC metamodel contains ‘Interaction’ which represents a specific part of a sequence diagram. The ‘Message’ is contained in ‘Interaction’ and is equivalent to a method call. Now like a method, ‘Message’ can have ‘Parameter’ and also optionally ‘ExpectedValue’. Conventionally methods are always associated with their owner class. Here, the ‘Message’ is also associated with an ‘OwnerClass’ which owns it. This ‘OwnerClass’ can have ‘ConstructorParameter’ which is useful when the ‘OwnerClass’ is to be instantiated. Now ‘Parameter’, ‘ExpectedValue’ and ‘ConstructorParameter’ can be either of ‘SimpleAttribute’ or ‘ComplexAttribute’. ‘SimpleAttribute’ means those of atomic types like integer, float and string. ‘ComplexAttribute’ means those representing data structures. For the sake of simplicity and clarity only ‘SimpleAttribute’ is considered in this thesis.
8.7 xUnit Metamodel

After we transform the SMC model using the Tefkat tool we obtain the intermediate output. The intermediate output in the transformation process is the xUnit model. This is a platform independent and generic test case. The Tefkat tool transforms the SMC model to this intermediate output. The metamodel for this is shown in Figure 18. The metamodel used in this thesis is the same one that was derived in [31] with some minor changes to be accepted in the current versions of the MDA tools used in the thesis. Any xUnit model that is generated by the Tefkat tool has to conform to this metamodel.

In this metamodel the ‘TestSuite’ is the container for the ‘Test case’. The ‘Test case’ further contains ‘Assertion’. An ‘Assertion’ is a condition check that should hold true after the execution of the test case. JUnit has many different kinds of assertions like equal, not equal, true and false assertions. ‘Assertion’ contains a ‘Method’ and ‘ExpectedValue’. While executing a test case the method which is the part of the code is executed and the return value is compared with the ‘ExpectedValue’. For an ‘Equal’
assertion, the test passes if the return value is equal to the expected value. ‘Method’ which is similar to the ‘Message’ from the SMC metamodel is owned by an ‘OwnerClass’. ‘Parameter’ can be supplied to a ‘Method’. The ‘OwnerClass’ can be supplied with ‘ConstructorParameter’ which is used during the time of class instantiation while generating the test code. Now ‘Parameter’, ‘ExpectedValue’ and ‘ConstructorParameter’ can be either of ‘SimpleAttribute’ or ‘ComplexAttribute’. ‘SimpleAttribute’ means those of atomic types like integer, float and string. ‘ComplexAttribute’ means those representing data structures. For the sake of simplicity and clarity only ‘SimpleAttribute’ is considered in this thesis.

Figure 19: xUnit metamodel in tree view as viewed in the Eclipse editor
8.8 Transforming SMC to xUnit model

TRANSFORMATION smc2xunit : smc->xUnit

NAMESPACE http://smc
NAMESPACE http://xUnit

RULE Model_2_OwnerClass(model, testSuite, interaction, testCase, message, assertion, method, c1, c2)
    FORALL Model model, Interaction interaction, Message message, ::smc::OwnerClass c1
    WHERE interaction.owner = model AND message.owner = interaction AND
    c1.owner = message
    MAKE TestSuite testSuite FROM model2testSuite(model),
    TestCase testCase FROM interaction2testCase(interaction), Assertion assertion
    FROM message2assertion(message),
    Method method FROM message2method(message), ::xUnit::OwnerClass c2 FROM c(c1)
    SET c2.name = c1.name,
    method.ownerClass = c2,
    assertion.name = append( append(interaction.name, " "), message.name),
    method.name = message.name,
    method.static = message.static,
    assertion.order = message.order,
    assertion.assertionType = message.assertionType,
    assertion.method = method,
    testCase.assertion = assertion,
    testCase.name = interaction.name,
    testSuite.testCase = testCase,
    testSuite.name = model.name

RULE ConstructorParameter_2_ConstructorParameter(model, interaction, message, method, c1, c2, cp1, cp2)
    FORALL Model model, Interaction interaction, Message message, ::smc::OwnerClass c1,
    ::smc::ConstructorParameter cp1
    WHERE interaction.owner = model AND message.owner = interaction AND
    c1.owner = message AND cp1.owner = c1
    MAKE ::xUnit::OwnerClass c2 FROM o(c1), ::xUnit::ConstructorParameter cp2 FROM
    o2(cp1)
    SET cp2.name = cp1.name,
    cp2.type = cp1.type,
    cp2.setter = cp1.setter,
    c2.constructorParameter = cp2

RULE SimpleAttribute_2_ConstructorParameter(model, interaction, message, method, c1, c2, cp1, cp2, sel, se1)
    FORALL Model model, Interaction interaction, Message message, ::smc::OwnerClass c1,
    ::smc::ConstructorParameter cp1,
    ::smc::SimpleAttribute sel
    WHERE interaction.owner = model AND message.owner = interaction AND
    c1.owner = message AND c1.owner = c1 AND sel.owner = cp1
    MAKE ::xUnit::OwnerClass c2 FROM o(c1), ::xUnit::ConstructorParameter cp2
    FROM o2(cp1),
    ::xUnit::SimpleAttribute se2 FROM cp2se(se1)
    SET se2.name = sel.name,
    se2.type = sel.type,
    se2.setter = sel.setter,
    cp2.simpleAttribute = se2
The SMC model is transformed into xUnit model using the rules shown in Figure 20. It is provided to the Tefkat tool as a .qvt file, along with SMC metamodel and xUnit metamodel. This is basically the mapping of model elements from source to target model. Tefkat uses these rules for transformation of SMC.
model to xUnit model. A careful study of these rules shows that the SMC model name is mapped to TestSuite name in the target model. Interaction name is mapped to test case name. Assertion element is created and method element is added which is again mapped form the message element in the source model. Message owner name is mapped to method owner name. Likewise every element in the source model is mapped to the elements in the target model.

8.9 JUnit test case generation from xUnit

Once the xUnit model is generated by the Tefkat tool, MOFScript rules are used to generate JUnit test case using the generated xUnit model as the input model. The chunk of MOFScript rules used in this thesis is shown in Figure 21. The first line is the module definition. Two models are passed to it. The first one is the xUnit model and the second one is the data model. To pass data for test case generation i.e. the values of parameters and variables, a separate model called the data model is created. The metamodel and an instance of this model are shown in Figure 22 and 23 respectively.

The main function in the MOFScript is the main entry point to the transformation. The first action here is to loop inside the elements of type ‘Message’ in data model whose name is ‘SETUP_1’. ‘SETUP_1’ contains constructor parameters of the methods that are to be called from the test cases. So, first we initialize these parameters with the function ‘getConstructorParam’. The name of the parameter value that is to be read from the data model is hardcoded for the sake of simplicity and clarity. For example ‘getConstructorParam("numberOfCoke")’ gets the initial value for the number of coke when the Vending Machine is started in the example explained in section 8.10 in this thesis.

The code headers are written to the test case file in the next step. Packages that are required to locate and use the classes required by the application to run the test cases are imported. Then a forEach loop which loops for all the elements of type ‘Test case’ in the xUnit model maps all the test cases to the JUnit code. Actions like test case definition, mapping of assertions is done in this step. Mapping action propagates further inside till all the aspects of the xUnit model like the methods, parameter and constructor parameters are mapped and their corresponding code is generated in the test file. Finally a test file which is ready for execution is created.
Figure 21: MOFScript rules to generate JUnit test cases from xUnit mode
Figure 22: Metamodel of the data used for the generation of the test case

Figure 23: Data model used for defining values for parameters during test code generation
8.10 Example

In this thesis to apply and check this methodology on a simple application implementing a simple vending machine. Actions like inserting a euro coin, turning the crank of the machine, dispensing and releasing a bottle of coke are the main methods. The simple vending machine has several states which are governed by the user actions and the amount of coke in the machine. Now to test the methodology a simple interaction with the machine is modeled in the SMC model. The interaction consists of several actions like initializing the machine with a fixed number of cokes, inserting a coin, turning the crank and dispensing the coke. This model is presented in Figure 24.

Figure 24: SMC model used for the example

The model is named SimpleVendingMachine_Model and contains one interaction called Generated_Model. This interaction contains several messages. The first one is the SETUP_1 message. It contains the constructor parameter for the initial number of coke the machine should have when first started. Similarly the second message is the insertCoin message. This represents action of inserting a coin in the machine. It takes a coin parameter, an expected value and states that the owner class of this message is the SimpleVendingMachine class. Likewise the other two messages in the model are trunCrank and dispense. They also state that their owner class is the SimpleVendingMachine class.
This model is input for the Tefkat tool along with the SMC metamodel, xUnit metamodel and the transformation rules. Tefkat tool then generates a xUnit model after horizontal transformation. Both of these models are still platform independent. The xUnit model that is generated is shown in Figure 25.

![Diagram](platform/resource/myFirstMBT/instances/xUnit_simpleVendingMachine.xmi)

Test Suite SimpleVendingMachine Model
- Test Case GeneratedTest
  - Assertion GeneratedTest_SETUP_1
    - Method SETUP_1
      - Constructor Parameter numberOfCoke
  - Assertion GeneratedTest_insertCoin
    - Method insertCoin
    - Parameter coin
    - ExpectedValue expectedCoin
    - Parameter coin
    - ExpectedValue expectedCoin
    - Method insertCoin
  - Assertion GeneratedTest_turnCrank
    - Method turnCrank
    - Constructor Parameter numberOfCoke
  - Assertion GeneratedTest_dispense
    - Method dispense
    - Constructor Parameter numberOfCoke

Figure 25: xUnit model for the interaction chosen from the example

If we closely compare model in Figure 24 and 25, it can be noticed that the ‘Interaction’ element is mapped as ‘Test Case’ and for each ‘Message’ we have an ‘Assertion’ element with the proper method (i.e. with exactly the same name as the ‘Message’) inside it. This is a generic form for unit test cases. This model is the intermediate output for the transformation. It has to be further transformed and mapped into text code to get executable code.

8.10.1 Results

Finally the MOFScript rules discussed above in Figure 21 is used to transform this xUnit model in to JUnit test cases. A data model which contains the values of the parameters used in the interaction is also provided to the transformation. Values for parameters like the ‘numberOfCoke’, ‘coin’ that is inserted in the machine, ‘expectedCoin’ i.e. expected value for the coin that is inserted in the machine, etc are read from this data model. For the sake of simplicity all the package definitions and imports of the classes that are necessary to run the intended interaction are hardcoded in the text file. Without this the test case would be useless as the classes of the actual program that is under test is provided through these imports. The generated test case is shown in Figure 26. This test case is now ready to be executed. It is now transformed to a platform dependent entity and will need a JUnit support to be executed. The assertion statement simple compares the inserted coin and the expected value of the inserted coin. If this matches the test passes, otherwise the test fails. When we want to test the interaction with different set of data, simply a different data model can be provided. This allows us to test the interaction with huge range of data without the need of any modification what so ever by hand.
This is a simple auto generated test case. But it shows how we can auto generate similar test cases by using MDA techniques. When huge numbers of test cases are to be generated and if the application is changing regularly, this method might be very useful. It is also suitable when the domain experts need to analyze and design the system without any development knowledge. The system can be designed in UML models. The behavior of the system can also be modeled in UML as shown in the above sections and then finally with model transformations, executable test cases are generated.

```java
package com.test;
import com.simpleVendingMachine.*;
import junit.framework.*;

public class Test_GeneratedTest extends TestCase {
    public static void main( String args[] ) {
        TestSuite testSuite = new TestSuite(Test_GeneratedTest.class);
        testSuite.run( new TestResult( ) );
    }

    public void testGeneratedTest( ) {
        try {
            //Prepare constructor parameter object
            int numberOfCoke = 5;
            SimpleVendingMachine simpleVendingMachine = new SimpleVendingMachine(numberOfCoke);

            //Prepare parameter object
            int coin = 1;

            //Prepare expected object
            int expectedCoin = 2;

            int return_expectedCoin = simpleVendingMachine.insertEuro(coin);
            assertTrue( expectedCoin == return_expectedCoin );

            simpleVendingMachine.turnCrank();
            simpleVendingMachine.dispense();
        } catch ( Exception exp ) {
            System.out.println( exp.toString() );
            fail("Exception occurred during test case execution");
        } //End of Test Method
    } //End of TestCase

Figure 26: Test case generated for the chosen interaction from the example
9 Future Work

In this thesis, different models and approaches in GUI test automation are explored. An implementation in a software company is also provided which uses one of the described techniques. However, other models are only described. As such, the effectiveness of these various models can be tested and compared, by developing simple prototypes.

Another possible future work can be to extend the prototype used to demonstrate the use of MDE in software functionality test automation. Currently the prototype is small and may be insufficient to justify that this approach can really work in real world complex scenarios. Similarly, for the sake of simplicity, some portions while generating the test code, like the headers, are hardcoded. This kind of hardcoded portion of the generated code should also be automated to obtain a complete automated test code.

The tools used for model transformations, while using MDA techniques to demonstrate test automation, can be replaced by more recent tools. New tools can be more flexible and powerful. There might also be more support available for newer tools. Thus, this aspect can also be included as future work.

A more detailed research on emerging approaches to automate test case generation using MDA techniques can be an interesting work. Today, many researchers are involved in the task of automating software testing using MDA techniques. It would be an interesting to address these works and compare their effectiveness.

Thus, the thesis lays the foundations to bigger research. If time and opportunity allows, it would really be interesting to carry this research to higher levels.
10 Summary and Conclusion

GUI has become a very important way to interact with the software system. GUI makes it easier to interact with the software and today developers dedicate large portion of their code in its implementations. More over uses of GUI in safety critical systems are also growing. Thus proper testing of the GUI has become a very important part of software testing. However testing GUI is not a very easy task. It faces additional difficulties that the conventional software testing does not have to face. One of the major difficulties is in automating the generation and running of GUI tests. Manual running of GUI test cases takes a lot of time and possess further problems. Several approaches are being purposed and practiced to achieve automation in GUI testing. If we carefully observe the development in GUI test automation, we can see it evolving slowly with increasing automation. First record and play application tools were popular. Slowly tools with more automation came into existence which supported scripting once the GUI objects are captured by the tool. Then further automation was added. Tools that could automatically generate test cases and test oracles after carefully analyzing the system software were also developed. Approaches using Key-Word/Table-driven Frameworks, Event Flows and Interaction Sequences are pretty popular in such scenario. Similarly Model based approaches are also very popular and efficient. In the recent years software testing using model driven techniques gained a huge popularity. This technique explores the use of MDA in software testing. Test cases are automatically generated from UML models which can be readily executed in the specific programming environment. A simple prototype is developed following this approach in this thesis. Sequences of method calls are modeled and test cases were automatically generated by using MDA tools and transformation rules. This thesis is an effort to realize what all is happening in the world of GUI testing. If focuses mostly in how automation can be applied in the process of GUI testing. Model Driven Testing is extensively studied, discussed and implemented during the final stages of the thesis. An industrial experience with GUI test automation in a software company called Catalysts GmbH is also shared in the thesis. The main target was to execute a large number of GUI test cases automatically, replacing manual testing for an enterprise application. GUI test automation has been continuously worked upon and significant progress has been made. However, the automation is still not easy and people have to largely rely on manual efforts. Significant effort must be continuously made to make GUI automation practical and easily applicable.
11 References


