THEMIS: Framework for Automated Testing of Graphical User Interfaces

Mark Stobbe
INF/SCR-07-105

September 2, 2008
Summary

GUI testing has been a problem ever since they were introduced. A GUI has meaning for a human, but computers do not care so much about fancy visual interfaces. Not only do they not care, they are also incapable of understanding a GUI, let alone validating. It is safe to say that only humans can really validate a GUI completely. However, humans are not computers and often make mistakes. This thesis will outline an automated GUI testing framework to help developers test their application’s GUI in the early stages of development.

Special thanks go out to Itude who offered me the opportunity to perform this research under their guidance. Also thanks to my parents and sister for supporting me during these five years and especially the last summer vacation.
Chapter 1

Introduction

In today’s world a lot depends on computers and the programs installed on them. These applications range from complex management systems for power plants to simple messenger programs. All these systems have an interface to interact with the user. This interface is extremely important, because it is the main interface between the user and the program. Some applications use a command line interface where the user enters single commands one after the other, while other applications have complex graphical screens with lots of options and facilities. This interface is known as the user interface of a system and enables the user to interact with the system. In this thesis we will focus on the most common type of user interface nowadays, the graphical user interface.

Several years back all programs only had a command line user interface. This way of interacting with a program is keyword-driven. From a developer point of view this is a straightforward mechanism to implement. The system asks the user to enter a command. This command is then interpreted, calculated by the system and the result is printed back to the user.

Graphical user interfaces (GUIs) work different, because they are event-driven. The user is presented with a “visual control panel” where all the actions and results are displayed. When a user triggers an action it will actually generate an event. This event is captured by the application, which calculates the results and displays it to the user. An event-driven system allows the user to determine the flow of the program. This is different from a keyword-driven interface where the developer determines the flow of the program. Take for example a simple email client. In a command line interface the system would typically ask the user for the recipient, subject and body in that order. While an event-driven system allows the user to start by typing the body, switch to the recipient, maybe type some more of the body and finally enter the subject.

Event-driven systems provide a lot more flexibility for the user but also increase the complexity of the application. Moreover, when we look at large applications such as Eclipse with thousands of events, it becomes close to impossible to see all possible scenarios a user might create. However, since the GUI is the only way for the user to interact with the system, we do want to make sure everything works appropriate.

Testing of the GUI is usually done by test engineers. They perform manual acceptance tests on the system according to the application’s specification. More precisely, testing an application’s GUI means performing all possible actions a user could ever possibly do and verifying that the system acts appropriately with respect to the specification.

1Technically there are more ways to communicate with an application. For example through its API, command line, speech recognition, etc. For this thesis we will always refer to the interaction between a user and application through its graphical user interface only.
CHAPTER 1. INTRODUCTION

In many cases, the specifications are incomplete in which case the tester uses common sense to decide whether the application behaves as expected.

This is labor-intensive work for small business applications, but even more for large systems with a GUI consisting of thousands of widgets. Testers often first focus on the parts that are important to the user. This ensures that at least the basic functionality of the application works correctly. The tester checks each text field for different inputs, clicks all buttons, selects items in a list, etc. Each of these actions is simple on its own, but when put together they form a complex sequence of actions. For most applications it is impossible to test all possible sequences and the way a tester works is by creating “smart” sequences. These “smart” sequences try to cover most of the application’s functionality at once. Often you will see that testers start by performing actions which should not alter the application’s state followed by the correct actions to get to the next screen.

These preliminary tests, entering invalid data, submitting incomplete forms, etc. are similar for all types of applications and require little domain knowledge. However, for the developer these tests are often the ones that find bugs. Common phrases in this context are: “Oops, forgot to check for null!”, “Oh! You should only enter integers.”, etc.

Our goal is to automate the preliminary GUI tests and improve the quality of the software send to the test department. We propose a generic framework which is mainly intended for use by developers to track down obvious bugs. However, testers may also benefit from our framework as it provides a basic test coverage.

Our solution is based on the concept of T2, which is a trace-based random testing tool for Java classes. We use the trace mechanism within T2 to record and simulate user interactions with the GUI. This automatically gives us regression support. T2 also has a built-in mechanism to detect non-termination which we can use. Section 2 describes in detail how this is done.

The framework is intended for Java GUI toolkits and can easily be extended to accommodate new toolkits. The current version comes with implementations for AWT/Swing and JavaServer Faces. Actions are automatically extracted from the application’s GUI at runtime. Custom strategies can direct the tests to focus on specific parts of the GUI. The developer can improve the tests by inlining specifications using assertions. These will automatically be enabled by our framework and tested. For reasons that will become clear later we also provide a record/replay mechanism. This provides the developer with another way to direct tests and inject domain specific knowledge.

In the next section we will dive deeper into the problem and distinguish between levels of automation. In section 3 we will discuss related work in this area. Section 4 describes our solution from a top-level view, while section 5 goes into detail about some of the problems we encountered during the implementation. Section 6 lists the results we got with the several implementations and section 7 concludes this thesis.
Chapter 2

Problem Description

GUIs are a must for most applications currently developed. There are visual editors that enable developers to construct GUIs using drag & drop in literally seconds. However, this comes with a price. GUIs easily grow into large and complex hierarchies of windows, dialogs and other widgets. The developer is responsible for coupling all these widgets to the business logic which make up the application’s functionality. This glue between the GUI and business logic is one that is similar for all applications and often error-prone. Many GUI toolkits therefore strive to minimize the code needed for this coupling by providing good widget interfaces and special widgets such as typed input fields.

Nevertheless, developers need to be aware of all possible scenarios for a user to interact with the system and account for all of them. A good indicator for the complexity of an application’s GUI can be found by looking at the number of widgets and their dependencies. If a window contains only one button, it will inherently be easier to test than when it consists of many input boxes and buttons which are all connected to each other.

In these situations automated GUI testing will assist the developer to perform initial tests. Common bugs in validating input, catching incomplete or empty data are typical tests which can be automated. With the automation of these tests the developer can focus on testing the main functionality. The automated test framework will handle other tests such as entering different strings into input fields, selecting items from a list and clicking buttons. This will quickly track down basic mistakes which can then easily be solved by the developer.

Before we dive further into the problem, we will first clear up some of the terminology used throughout this thesis. In the introduction we used the term widget. A widget is nothing more than an element in the GUI. GUIs consist of many widgets each representing part of the functionality of the application. Some of these widgets do nothing more than grouping of widgets, such as panels, while others generate events, like input fields and buttons. Often ‘widgets’ are also called ‘controls’ or ‘components’ depending on the context.

We also used the term action quite a lot. An action, in our context, is an interaction between the user and the application’s GUI. For instance, a single mouse click which pushes a button is an action, but also composing an email can be seen as a single action. However, the latter can also be seen as a sequence of smaller actions. Namely, enter the recipient’s address, enter a subject, type the email message and push ‘Send’. The level of detail has a great impact on the system’s performance.

However, it is not immediately clear what level to choose. If we choose clicks and key presses as elementary actions, then we might end up with a lot of actions, without
accomplishing much. While in the other extreme case, if we choose to see every use case of the application as a single action, then we do not have a lot to work with. Moreover, in some cases we would like to have larger actions, for example to fill in a form, while after that we might need smaller actions again. In the next section we will explain how we solved this.

Another term which requires clarification is application. When we use the term application, we usually mean the target under test. We say ‘target’ here, because our framework does not enforce any requirements upon the application. Moreover, we may also test other targets such as single classes or database queries. This depends completely on the implementation of our framework and is by no means limited to GUI testing. Actually, in section 5 we will also discuss an implementation for unit testing single classes.

Now that we have the terminology in place, let us look at what automated GUI testing is and what it entails. There are several decisions to be made when automating GUI testing. The first thing is to what extent the process of testing a GUI should be automated. For each GUI test we identify two stages: generation and execution. In the generation stage we choose the action we want to execute and generate the input values for this action such as a string for input fields and selection indexes for lists.

The generation stage is also the trickiest part. First of all we need to know what actions we can choose from and what these actions require as input. The latter influences the number of data types we should generate. For example, if the developer uses a typed input field for one of the forms, then we can safely generate input for that type only. Moreover, if certain inputs are constantly rejected, then we could anticipate by generating other type of inputs for that action. In the implementation section we will see that the AWT/Swing toolkit only has a few typed widgets, whereas JavaServer Faces provides lots of typed widgets.

After the generation stage, we enter the execution stage. In the execution stage we execute the prepared action on the application. In general this is not so difficult because GUI toolkits offer special interfaces for simulating button clicks and text input. However, an important issue with respect to GUI testing is whether the execution should be visible on the user. In other words, do we show windows opening and closing, buttons being pressed and text being entered? And if we were to hide this from the user, how can we accomplish such a thing without modifying the source. This problem relates to how we decide to represent the target application. In [2] they discuss three possible ways to represent an application’s GUI and we will argue each of them.

The first representation suggests using the application and its GUI toolkit itself as target. The GUI toolkit defines the interface for each of the application’s widgets and holds the glue to the actual application. With this representation we stay as close as possible to the way a user interacts with the application.

Typically we will extract all widgets from the GUI in each step and trigger one of the actions of such widget. Many GUI toolkits allow the extraction of widgets at runtime using introspection. However, a problem we face with this approach is that it is difficult to modify the application’s behaviors such as visibility of windows, because there is no abstraction from the real application.

The second approach solves this problem by requiring a separation between the application’s business logic and its GUI. Sometimes you will hear the term User Interface Management System for these kind of applications. Typically this approach allows us abstract the application into a simplified model. This simplified model can then easily be adapted to form a good testing target.

The problem with this approach is that applications often do not have a full separation of their GUI and business logic. This requires the framework to work with a partial
UIMS or transform the application in such a way that the GUI can be decoupled from the business logic. However, this differs for each application and therefore requires manual manipulation in most cases.

The last suggestion uses formal models to represent the application’s GUI. The GUI is described in a formal language suitable for simulation and verification. The advantage of simulating the application’s GUI during the design phase helps to quickly spot mistakes and possibly bugs. Formal models also allow automated testing, but does require the developer to include specifications in the formal model. This is time consuming and non-profitable for many applications. Especially since modern GUIs are often dragged together in literally seconds with the use of visual editors. Formal models are common practice for development of high risk applications such as car electronics but less practical for our purpose.

In our framework we have chosen the first approach, but did not exclude the second straightaway. The final decision as to how to model the target is left to the engine implementation. In theory we could even choose the last approach, but this was not taken into account during the implementation phase. The engine implementation for AWT/Swing uses the first approach and uses introspection to extract the GUI. As for the JavaServer Faces implementation we use the UIMS approach. JavaServer Faces, by default, allows other “views” to be used. This makes it possible to plug in our framework and work directly on the business logic.

Role of T2

We mentioned in the introduction that our framework is based on T2. T2 is an automated unit testing tool for single Java classes. In each test session T2 will stress the class by generating invocations and field updates. The test data for these operations are automatically generated from a domain set. The thing that makes T2 special is that it will sequence these operations, effectively modifying the state of the test object. This generates a different situation for method invocation, one which might fail. T2 also has the ability to pass the current test object as parameter to one of its own methods, as well as take object from a pool which contains previously generated objects.

The developer can also write pre- and post conditions using the assert statement to check for properties an class invariants. This will make testing even more accurate. We will not go into all the details of the T2 framework, but for those interested we included a more elaborate introduction in appendix C. Next we will focus on the integration between T2 and our framework.

Proof of concepts have shown that the integration with T2 is difficult to achieve without altering its source code. Inspection of the sources led us to the conclusion that too much functionality is collected into only a few classes. Also, specific parts of the implementation are hard coded, which are certain to require different behavior. Large parts of the engine are coded into static methods, which cannot be overridden.

Apart from a few implementation challenges, there are also differences in definition between T2 and THEMIS. T2 defines a target as a single class. However, a GUI is typically tested as part of the complete application. In the case of a web application it becomes even more complicated. Web applications are typically accessed through a browser, which performs requests to the server for its view. Here there are only classes used in the application internally.

Other definitions have a more direct mapping to GUIs. A class invariant for a GUI is just a GUI invariant. A typical invariant is that there should always be at least one possible action. Pre- and post conditions are less common in a GUI, because the GUI is responsible for expressing these conditions. Take for example a text field where
the user has to fill in his/her age. One could argue that the precondition for this field is that it should contain a positive integer value. However, it is the responsibility of the application to validate the input of the user, not that of the precondition. So even though we enter an invalid input, the trace is still continued, because no violation occurred.
Chapter 3

Related Work

In the past there has been a lot of research into automated GUI testing. Nowadays it has drifted to the background, but there are still interesting developments taking place. In this section we will give an overview of existing tools for GUI testing. These tools can roughly be divided into two types of automation: automated regression and automated test case generation. The latter implies the first as automated generated test cases offer regression support by default.

Automated regression automates the process of replaying test cases each time the application is modified. Often test cases are generated with the aid of a capturing tool. These tools allow the developer to generate a test case by means of a normal interaction with the application. Typically it will capture the triggered events and transform them into appropriate actions within the GUI. There are some differences between the tools which will be explained in section 3.1.

The other type of automation, automated test case generation, also automates the process of generating tests. There have been several research projects ranging from tools using genetic algorithms to using formal descriptions. We will look at a few of them in section 3.2.

3.1 Automated regression tools

There are many record/replay tools available: Abbot, Marathon, QF-test, Jemmy, JFCUnit, etc. We now look at several of these and pay close respect to their differences.

Abbot is a GUI testing framework which allows the developer to write unit tests to simulate user behavior. Abbot is developed for AWT and Swing. The developer is given a specification language for expressing GUI actions, like entering text in a field or performing a button click. For this Abbot uses a robot class, which uses either Java’s Robot class or inputs events in the AWT event queue. The properties of the GUI can be checked with simple assertions, similar to JUnit. Next Abbot compiles and executes these scripts to test the user interface. An extension to the Abbot framework is Costello. Costello records the events raised by the user and generates an Abbot test script.

An important part of Abbot is component identification. Abbot needs to store a unique identifier for a component to be able to replay the test script. It uses several custom attributes of a component to distinguish between them. For example, for buttons they use the text attribute, and for a frame the title attribute. Combining several of these custom attributes gives Abbot the possibility to track components.
Marathon\textsuperscript{4} has a clean separation of phases. These are event capturing, finding and naming components and resolving components. Marathon captures and replays events using an event monitor. This in contrary to Abbot, which uses a robot class to mimic user behavior. The naming strategy Marathon uses is fairly similar to the one Abbot uses. However, Marathon also uses the component hierarchy to decide which component will record the action. For example, a JSpinner\textsuperscript{5} has two small buttons. If the user clicks on one of these buttons, an event will be generated for that button. The action, however, is recorded to the parent JSpinner component. Now Marathon can save access and save the state of the JSpinner component.

Marathon packs all components in an \texttt{MComponent}. This allows them to treat AWT and Swing components the same. For replaying it also becomes easier, because all \texttt{MComponent}s have a method \texttt{play()} which simply performs the recorded action.

QF-Test\textsuperscript{5} uses an RMI connection to communicate events. Events are captured using a custom event queue. Replay is done by triggering the events on the corresponding components. QF-Test supports Java/Swing and Eclipse/SWT applications with a graphical user interface (including Applets, WebStart, ULC).

In contrast to Abbot, QF-Test chooses to use the hierarchical information for each component. Each component is part of a tree and each node is unique with relation to his parents. As where Abbot decides to not use this information, because created test cases break easily when the structure is changed, QF-Test thinks different.

### 3.2 Automated test case generation tools

The typical tools in this category are model-based testing tools which work on an abstract version of the target, the model. There are many different definitions of the term ‘model’. In the rest of this section a model is a simplified abstraction of the target application. Typically these simpler targets provide better flexibility and are easier to grasp when implementing difficult algorithms.

There are however also tools which do not have a model such as THEMIS. Instead they work on the actual target application using language features or source code. Another such tool is JWalk which works on classes instead of GUIs. There is a large resemblance between JWalk and THEMIS with respect to their design.

JWalk employs a novel testing technique which they call lazy systematic unit testing. The tool starts by generating sequences of a fixed length and asks the user to verify correctness of each sequence. The results of these questions are then used for further testing, slowly building a specification for the class under test. An unique feature of JWalk is the ability to analyze test results of the current test session. This can be used to perform selective pruning on the test sequences.

The core of JWalk is similar to THEMIS as will become clear in chapter \textsuperscript{4} and \textsuperscript{5}. Basically JWalk allows own definition of strategies, listeners and employs similar design patterns like Strategy, Observer and Template Method. However, as said before, JWalk allows user interaction during testing which is not available in THEMIS.

The rest of this section focuses on GUI testing tools which take a different approach to automation. So does XTest\textsuperscript{2} use genetic algorithms to generate test cases. PATHS\textsuperscript{6} uses planning techniques to direct the tests and SpecExplorer\textsuperscript{7} is again more like THEMIS with the exception that it works on a model of the target application.

XTest focuses on generating “novice-like” interactions. When a user is confronted with an application for the first time he/she needs time to learn how to use the interface. This novice does not just randomly click the buttons and menu. He/she has an objective, like “Create a new file” or “Turn of high-lighting”. XTest uses genetic algorithms to
3.2. AUTOMATED TEST CASE GENERATION TOOLS

During the testing process, the algorithm learns more about the GUI model and the paths to perform certain actions become shorter. XTest uses the application itself as specification for the GUI. They argue that reverse engineering GUIs to get the corresponding UIMS is error-prone and formal models are nice, but they lack tool support for translating abstract specifications to real applications. XTest uses the GUI model to find possible actions to create test sequences. To cut down the branches XTest uses its genetic algorithm. However, XTest does not evaluate test sequences, it only creates “novice-like” test scripts using a genetic algorithm. Verification of the tests needs to be done by the test engineer.

PATHS uses planning techniques from the field of artificial intelligence to generate directed tests. First PATHS defines a formal model for representing a GUI. At any time in point, the GUI is represented as a set of objects \( O = \{o_1, o_2, ..., o_n\} \). These are the current components the GUI contains. Typically these are windows, buttons, text fields, labels, etc. Next, properties are also a set \( P = \{p_1, p_2, ..., p_n\} \) where each property \( p_i \) is an \( n_i \)-ary boolean relation, with \( n_i \geq 1 \). This allows us to model GUIs, for example, a simple window with a single text field: \( O = \{w_5, b_7\} \), with properties:

\[
P = \{\text{window}(w_5), \text{field}(t_7), \text{title}(w_5, "MyWindow"), \text{text}(t_7, "Hello!"), etc. \}.
\]

A GUI can thus be represented by two sets consisting of the objects and a set of properties which are always true. This makes up the state of the GUI. Over time this state will change and certain property relations may not hold. For this PATHS introduced actions. A set of actions \( A = \{a_1, a_2, ..., a_n\} \) are the transitions from one state to the other. An action can be parameterized, e.g. \( \text{setText}(b_7, "Bye!") \).

Next PATHS defines a GUI test case as a pair \( <s_0, a_1; a_2; ...; a_n> \), where \( s_0 \) is the initial state consisting of the current objects with their properties, and \( a_1; a_2; ...; a_n \) represent a legal sequence of actions to be taken. Next PATHS introduces operators as a 3-tuple \( \langle \text{name}, \text{preconditions}, \text{effects} \rangle \). Every operator is identified by their name; a set of positive preconditions \( p(a_1, ..., a_n) \) where \( p \) is a \( n \)-ary property; a set of effects \( p(a_1, ..., a_n) \) where \( p \) is a \( n \)-ary property. In the example below you can see how to define an operator \( \text{setText} \).

**Name:** \( \text{setText}(t_x: \text{field}, s: \text{String}) \)

**Preconditions:** \( \text{enabled}(t_x) \)

**Effects:** \( \text{text}(t_x, s) \)

With these operators we can derive an expected state after a sequence of actions performed on an initial state. PATHS can now compare actual states against expected states. They identify 3 types of verifications: changed, relevant and complete. These differ in what properties they verify. Changed-Properties Verification only checks expected property changes according to the effects of an operation. Relevant-Properties Verification checks all properties in the reduced property set. This set contains only those properties that were used for construction the GUI. And last, the Complete-Properties Verification checks all properties made available by the GUI toolkit.

**SpecExplorer** uses a formal model of the GUI. This model is extracted from a set of UML diagrams, translated to the specification language Spec\# and then refined to make it executable. The resulting model allows for consistency checks and model animation. Next, SpecExplorer extracts a finite state machine (FSM) from the Spec\# model. This FSM is then used to generate test cases. This set is limited by a coverage criteria, namely full transition coverage. This means that it covers all the possible transitions from one state to the other. This criteria can be changed to other coverage
settings, and can also be limited to the degree of coverage one would like to achieve. After generating test cases, SpecExplorer only needs to map the model back to the implementation and is ready to go.
Chapter 4

Solution

In this chapter we give a top-level overview of the THEMIS framework. The next chapter we will focus on problems encountered during implementation as well as provide details about the concrete engine implementations.

4.1 Requirements

Our framework must solve the problem of automating GUI testing. This will help developers to track down bugs in the early stages of development. From a practical point of view can it be useful to stream GUI testing together with unit testing. This requires the framework to work without human interaction and verbose report generation. T2 already supports report generation in a simple text format. We intend to take this one step further and generate XML documents which list taken actions, their context and other useful information.

Because we do not want to put any constraint on the underlying GUI toolkit, our framework must be extensible. Moreover, extensibility also plays a role when defining custom strategies for choosing actions, generating test data and execution of actions. Especially when the GUIs grow larger, it becomes more important to allow customization of all parts of the framework.

Developer also need to have the ability to add “checks” within the code such as pre/postconditions and invariants. By default T2 allows developers to insert assertions in the source code. This should also be available in our framework. We also introduce a validator which is also available to the developer to verify the state of his/her application.

Another requirement later added came from practical experience. This is the ability to record and replay actions during execution. We came across this requirement when trying to test a login system. A typical login system has two input fields and a login button. We can perfectly test this, but we have little chance of moving on to the next screen. The developer can in such case create a record which fills in a valid username and password pair, which can then be used to logon to the system.

4.2 Design

Next we discuss the design of our generic framework for automated GUI testing. Our framework focuses on developers to test their application’s GUI in an early stage. We provide feedback about the tests taken, regression support and customization options.
We created three implementations of the framework for AWT/Swing, JavaServer Faces and unit testing. Each implementation can be used without source-availability, which is useful for testers who wish to use our framework too.

At the basis of our framework is the T2 framework. The T2 framework initiates new test sequences, triggers execution of actions and provides regression support. Our framework forms the glue between T2 and implementations for specific GUI toolkits. Each implementation implements several smaller steps which are the ingredients for the framework to create a trace, where a trace is defined as a sequence of actions.

Each trace starts by resetting the state of the target application. This is necessary to ensure that we get the same result when performing regression on a test. Next, we build a context for the initial state of the application. A context should contain all factors which may possibly influence the state of the target. For GUI applications this is typically the state of all widgets. However, we cannot ensure that the context captures all dependencies. Applications may depend on the availability of specific files or dynamic data. This lies outside the scope of the context and therefore we cannot fully reset the state of the target application. We discuss this point further in section 5.2.1.

Now that we have constructed a context for the target, we use a strategy to select an action. This action will be the one that we perform on the target for this step. Before we can do this, we have to prepare the action. Typically an action requires preparation in the form of generating test data. If we want to test an input field we need to supply the action that enters some string into the field. In the case of GUI testing we can suffice with mostly primitive types. T2 provides special domain maps which we use to generate primitive test data. These domain maps contain possible values for each type such as -1, 0 and 1 for integers and “line\nline”, “!@#$%ˆ*”, etc.

After the action has been prepared we save a description of what this action will do and then execute it. We need to save this description, because if the action throws an exception or in some other way fails to complete we would like to know what went wrong. We could have chosen to do this the other way around. Instead of saving the action beforehand, we could have captured all exceptions when performing an action. This would simplify the implementation, but an action could have changed parts of the state before throwing an exception. This would corrupt the report when we choose to generate this after the error occurred.

When an action is executed, it should be bound to an element in the context. In most cases the action will use that element as its target. For example, an action for clicking a button takes as target the button wrapped in an element. When the action is executed it will retrieve the button from the target element and perform a click. This is only possible when the action is bound to an element which contains a button. We also have unbound actions. These actions contain an identifier of the element they should be bound to. When an unbound action is chosen for execution, we first search an element with the specified identifier inside the context, bind the action to this element and then execute the action like any bound action.

Unbound actions allow us to support replay/record. When an action is executed we unbind it from its element. This unbound action can be stored to disk. This action can later be reloaded and dynamically bound to an element in the context. Note that we are not required to store the complete context, as the action is bound to an element identifier. If the context changes, but our target element is still available, then the action can still be executed.

---

2Web applications often support a Date type. Technically this is not a primitive type. However, for web applications all types are converted from and to strings anyway. We also support complex types through its constructors. This is further explained in section 5.
The execution of actions depends on the underlying GUI toolkit. The framework does not make any assumptions and simply passes control to the action. When a violation occurs during execution, our framework will notify T2. T2 replies by resetting and replaying the violating trace. This is noted by our framework and during this trace we will generate an XML report.

4.3 Quality assurance

The three base requirements are extensibility, scalability and reusability. These requirements are difficult to test and therefore we choose to focus on this question: how effective is our approach with respect to automated GUI testing?

For this purpose we will use code coverage. First the target application is instrumented with “counters” which will track what parts are executed. Next we feed the target application to our framework and analyze the results. Code coverage will show us how much and which parts of the target application was covered by our tests. We can use this to our advantage to improve reachability, but can also be used by developers to determine what unit tests to write.

Next we also want to make sure our framework actually finds bugs. We do this by modifying the target application with intended bugs. We run our framework on the modified version and determine the number of found bugs. This measurement gives us a good idea on how effective our framework is. We will perform this experiment with two different strategies, fully random and event coverage strategy. The latter is a technique in which all possible combinations of a certain length are executed.

As for the extensibility and reusability requirements, these were tested during development. The framework has undergone several refactoring steps as well as the different engine implementations. In section 7 we will discuss the difficulty of implementing the engines and whether this can be done by the average developer.
Chapter 5

Implementation

The THEMIS package consists of a small core framework and three concrete engine implementations. This section will first explain the details of the core package followed by the different engine implementations. Implementors need little knowledge of the internals to be able to implement their own engine. However, it can be informative to see how THEMIS is structured and have a look at the sample implementations.

The first engine is the unit engine. This engine illustrates that THEMIS can also be used for other targets than GUIs. It tests single classes by invoking methods and updating fields similar to T2. Pre/postconditions and invariants are programmed using the assert statement and validators.

The second engine is the AWT/Swing engine. This engine is able to test AWT/Swing applications by starting the application through the main method and decomposing the windows into elements and actions. A special feature of this engine is that all windows are hidden during test runs.

The third engine is the JavaServer Faces engine. This engine takes a JavaServer Faces web application as its target. The user adds a small phase listener plugin into the web application and feeds the initial request page to the engine. The engine will decompose the webpage and create elements and actions. Next an action is chosen and executed by sending a new HTTP request to the server. A special feature of JavaServer Faces is its support for typed widgets which we can use to limit the amount of test data.

5.1 Core

5.1.1 Structure

The THEMIS core package consists of essentially six classes: Proxy, Dummy, Engine, Context, Element and Action. The other classes help to make THEMIS more flexible. Figure 5.1 shows the relations between the different classes. In this diagram the reader may recognize several design patterns. The observer pattern is used for the TraceListener. The builder pattern for the similar named Builder class. The Engine class can be seen as a template method pattern and the Proxy class is more like a mediator.

The main class in our framework is the Proxy class which is a singleton. This class is the connecting glue between the user, T2 and the engine. It parses the command line, creates the engine and sets up T2 for our dummy class. Furthermore it holds global
CHAPTER 5. IMPLEMENTATION

Figure 5.1: Class diagram of the core package

information about the target and several pass-through methods from the dummy to the engine.

The Dummy class acts as target class for T2. All calls to the dummy are forwarded to the proxy, which translates them into a fine-grained call structure on the engine. The advantage of having a dummy class as target is that we can control the information available to T2. When a violation occurs, T2 will automatically replay the trace and output the values of all fields of the target class. Therefore our dummy only contains an “info” field which holds the description of the current action. This will make sure that only that piece of information is printed. Another possibility is to suppress T2’s output and write our own. In section 5.1.3 we will take a closer look at the connection between THEMIS and T2.

The Engine class holds the real target and its context. Before we take a closer look at the engine class, we will first explain what the context of a target is. The Context represents all the relevant parts of the targets state. The default context is implemented as a tree data structure. We have chosen for this data structure because many GUI toolkits have their own component tree, which then maps nicely to our context. However, implementors are free to choose whatever data structure is suitable for their targets.

As said before, each node in a context stores an element which represents a part of the target. This can be a field, method, component, etc. anything that extends the Element class. Each element must have a unique identifier with respect to that context and can have actions attached to it. The identifier is used to bind and unbind elements from the context. By default the Element class will generate unique identifiers. It does so by taking the name of the element and appending a unique number. This number is not shared between different traces as a new trace has a new context and thus new elements.

2The Proxy class is lazy initialized and contains a setInstance method to reset the proxy. This is purely done for practical reasons and not a requirement.
There are two types of actions, *bound* and *unbound*. An action must always be bound to an element within the current context to be executable. Unbound actions can be bound to their element by supplying the context in which the element can be found. The `bind` method in an unbound action does nothing more than lookup the element with the specified id and copy the stored values into the bound action. Unbound actions are necessary to support record/replay functionality. They allow us to store actions which need to be executed later on when the action’s target element becomes available. The following section describes the execution phase in detail.

### 5.1.2 Execution

![Flow diagram of the core package](image)

Figure 5.2: Flow diagram of the core package

Figure 5.2 shows a flow diagram of the framework. Each execution is triggered by T2 through instantiation of the dummy. This event is forwarded to the engine where the context is reset and the target reloaded. The `Builder` instance is retrieved and invoked to build a context.

Now, first the queue is checked for any pending actions. Typically the queue contains actions when a certain sequence of actions need to be replayed. Actions can be added to the queue either in a strategy or at the start for replaying a stored action sequence. If there are no actions pending, then the strategy is instantiated and passed the current context. The `Strategy` class is responsible for choosing an action from the context. If the context does not contain any actions or no action could be chosen, then we issue a warning stopping the current execution.

The `Strategy` class is an unique feature of THEMIS. It allows developers to plugin their own testing strategy. Moreover, the same strategy can be used for different engine because its in the core package. Actions are extracted from the context and are the main input for a strategy. However, we do allow strategies to maintain a state over the duration of a trace. This allowed us to develop the event coverage strategy using in section 6.
If an action was selected, either from the queue or by the strategy, then this action will now be prepared. Each action has a list of state variables. These state variables make up the variable part of an action. For example, a text field action typically has a state variable ‘value’, which holds the new value for that field if the action is executed. State variables can either be fixed or not. A fixed state variable has a concrete value assigned to it, while ‘unfixed’ variables point to a magic constant: State.ANY. Preparing an action is nothing more than fixing all its state variables. This is done using an instance of the Generator class.

The next step is to bind the action to the current context. This is only relevant if the action was taken from the queue. Nevertheless, actions taken from the context are re-bound. The action is now bound and its state is fixed. This means it is ready for execution, which is the next step. The action will take values from its state and use them in its execution. For a text field this means that its text is set to the value stored in the ‘value’ variable.

After successfully executing an action, the context will be validated by all registered validators and the trace step ends. If an exception is thrown during execution or validation, then this means a violation has occurred. All exceptions are propagated to the dummy class. The dummy class will check each exception to determine their origin. If this exception was thrown within the THEMIS framework, then there is a problem within THEMIS. We will wrap the exception in a T2Error to notify T2 of the problem. In any other case the exception is just forwarded to T2 without further modifications.

5.1.3 Integration with T2

The core package is responsible for the integration with T2. We would like to have T2 do most of the work. However, after looking through the documentation and source code of T2, it became clear that it would be difficult to make a clean extension to the framework. Several problems came to the surface at first glance.

T2 has two main classes, RndEngine and Trace, which both contain large parts of the functionality of T2. Moreover, most of that is static code, which makes overriding difficult. Values like what engine to use and the type of target are hard-coded into these static parts. We would therefore have to rewrite these pieces to accommodate for different engines and targets.

Another issue has to do with the way actions are collected. For a single class all actions are fixed for the complete duration of a test. T2 will analyse the class when its loaded for the first time and extract all methods and fields. However, for GUI applications the interface dynamically changes when new windows are shown or hidden, widgets are added or removed, etc. As stated before, T2 only analyses the class during load-time, which means it does not alter its set of actions and cannot be used for GUI testing in this way. We notified the author of T2 about this problem and in the next release it will be possible to manipulate the action sets.

There are several ways to solve these problems. The best way would be to refactor the T2 sources. We would distribute the functionality into multiple smaller classes with a set of fine-grained methods to stimulate inheritance. This enables new engines to be plugged in with different target types as well as other features like dynamically changing interfaces. This option was discussed with the author of T2 but due to a lack of time we had to dismiss this option.

Another solution is to ignore the problems listed above and try to tie our framework to T2. Instead of inheriting from the RndEngine we could inherit from the Engine class and implement similar functionality in our own engine. Next we also have to replace
the tracing functionality to accommodate for dynamic interfaces and other features such as choosing strategies and record/replay actions. This solution requires us to re-implement most of the core functionality of T2. Moreover, we would also have to re-implement parts of the functionality which were also implemented in the T2 engine.

Instead we chose to create a dummy class which will be used by T2 to control the tests. T2 instantiates the dummy class with a random value. This value will be used as seed for our test data generator. It is important to let T2 make this random value because T2 will guarantee that the same value will be passed the next time we replay that test. The instantiation of the dummy will trigger the engine to reset the real target. Next T2 will only invoke the step method, which triggers the engine to choose, prepare and execute an action.

After the action is finished T2 invokes the classinv method which will trigger validators to be executed. This is similar to a class invariant for classes. When a violation occurred T2 will be in charge of printing the description listed in the dummy class.

This is only the string in the dummy class which contains a nicely formatted report of the taken action and its context. Other features like non-termination, step and trace count, etc. are controlled by T2 and have no significant role in THEMIS.

5.2 Unit Engine

The unit engine has several features. The engine automatically enables assertions, has three ways to instantiate the target and support for base domains and pools. In many aspects the unit engine is similar to T2 itself and in some parts it even surpasses T2’s functionality.

At the start of each execution the target must be instantiated. The obvious way is to instantiate a class through one of its constructors. T2 generates values for the constructor arguments and invokes the constructor. If the constructor throws an exception, then new values are generated and instantiating is retried. T2 will only try this a limited number of times and if no object could be acquired, then it will fail.

Our engine takes a broader approach. Instead of taking only the constructors, it also finds all methods and fields which return an object of the target type. These are all put together and one of these is chosen as instantiation method. If instantiation fails, then it will try a different method, instead of retrying with different arguments.

We use a special class loader to enable assertions for the target class. The user benefits from this since he/she does not have to manually enable assertions on the command line. The class loader can also reload an already loaded class. This effectively resets the static state of the class.

Next introspection is used to find the fields and methods which will be available for execution. T2 is a lot more flexible here. T2 offers options to include or exclude various class members, allow private and protected members to be tested, and more. The current unit engine does not have this flexibility and only allows public members to be tested. We can change this simply by changing the builder.

The base domain and object pool are similar to those of T2. We actually reuse the base domain and pool of T2. In short, the base domains contain test data for several primitive types and the pool contains the previously generated objects. For example, when we generate objects for a method invocation we add these objects to the pool. As for primitive types we get those values from the base domain.

The current unit engine does not implement several of the advanced features employed by T2 such as application models and typed collections. These features can also be incorporated into the unit engine, but we have focussed on automated GUI testing.
5.2.1 Discussion

An important aspect for T2 to be successful is regression. Regression can only be done if precisely the same sequence can be replayed. However, you also want to try random sequences of actions. So how does T2 replay the same sequence then? If a violation occurs T2 saves the initial state it used to create the sequence. When T2 is asked to replay the same sequence it loads the stored state and starts the engine. Since the engine will be in exactly the same state it will result in the same violation. This allows the developer to solve the problem and regress the violating trace to see whether his changes really solved the problem.

A critical point is that everything that happens for the duration of the execution is deterministic. If this is the case, then when we restore the initial state and start the execution it will replay precisely the same sequence with the same result. However, per definition this is impossible, as every action we do has an effect, which we can only undo by doing another action, which in turn has its own effect again, etc.

Now you might think nothing is deterministic so you cannot perform regression, thus T2 cannot be successful. We can however restore the relevant state in most cases. Take for example the following class:

```
class Gateway
{
    private static boolean allowed = true;

    public void enter()
    {
        if (allowed)
        {
            allowed = false;
            throw new RuntimeException("Now you cannot enter again!");
        }
    }
}
```

Table 5.1: Gateway Example

Let us first discuss why this piece of code poses a problem. For convenience we will assume this class is in fact the target class passed to T2. In practice THEMIS enforces another indirection, but this indirection is invisible to the user. T2 will start with instantiating this class by calling its constructor. Next T2 uses introspection to find out that the only available method is the enter method. The allowed field is private and should not be modified outside the class. The first time T2 invokes the enter method, it throws a RuntimeException. According to T2 this is a violation and T2 replays the same sequence again to explain to the user what went wrong. T2 does this by creating a new instance of the Gateway class and execute the enter method again.

However, because the allowed variable is static, it will now be set to false. Thus, in this case the runtime exception is not thrown and no violation is found. The user is presented with a seemingly correct trace of the program, which is obviously not the case.

T2 presumes that traces will not interfere with each other. However, this does not allow us to use any static variables or other persisted state objects such as files and databases. We think that static variables should be allowed and therefore our framework implements a way to reset the variables. We do this by reloading the class before
each new trace. This will effectively reset all static variables back to their defaults. We take a small performance penalty because reloading a class takes time, but this is insignificant in the GUI engines explained in the following chapters.

Our solution can still create awkward situations, where first a violation is found, but later on it cannot be reproduced. These situations are detected by THEMIS and a warning will be issued together with a log of all taken actions. This enables the user to manually inspect each step and possibly reproduce the problem.

## 5.3 AWT/Swing Engine

The AWT/Swing engine takes a GUI application modeled with AWT and/or Swing toolkit as its target. The GUI is decomposed into components. Each component can have multiple actions attached to it. All actions are invisible to the user, that includes opening windows and dialogs.

The engine starts by preparing the target application. All class files are passed to an aspect weaver. The weaver takes a single class and weaves it with the predefined aspects. These aspects simulate all visibility related functionality, like setVisible, isVisible, etc. This stops the application from showing any windows, while still providing valid return values for method which query the visible state of a window.

The aspects we use are written in AspectJ. AspectJ is the leading aspect-oriented extension language with an active community supporting development. A short introduction to AspectJ can be found in appendix A as the reader may be unfamiliar with programming aspects.

The engine keeps track of all shown components using a map. When a window is shown for the first time, it will be added to the map with the value true. If the window is hidden, then this value becomes false. Note that the window is not removed from the map, as we could then have used a set. By tracking windows in this fashion we know when the window is opened for the first time. This is useful as Java normally fires a WindowOpened event when this happens. Other events such as ComponentShown and ComponentHidden also depend on this map to decide when they should be fired. Windows are removed from the map again when they are disposed. The user can still create the window, but this will count as a new window and thus a new WindowOpened event will be fired.

In AWT/Swing, windows and components differ quite a lot from each other with respect to their visibility. By default components are visible, while windows are not. Also the displayability of a component, whether it is bound to its peer, differs. A window is displayable when it is visible, while components require all parents to be displayable and visible.

Dialogs require even more attention. There are two types of dialogs, modal and modeless. Modal dialogs are commonly used to display errors and warnings, but also open and save dialogs are often modal. If a dialog is modal, then it means that its parent and all its ancestors cannot be accessed as long as the dialog itself is visible. Moreover, execution is halted when after setVisible(true) has been called. Only after the dialog closes, execution will be resumed. This is best illustrated with the example listed in [2]

If we execute the above piece of code, first dialog 1 will be shown, followed immediately by dialog 2. Execution of this code will now halt until dialog 2 is closed.

---

2We have chosen to combine AWT and Swing due to the large overlap, i.e. Swing is an extension to AWT. In practice it is a bad idea to use both AWT and Swing in the same application. However, this only applies to GUI development and the way components are visually layout. We are not going to build any GUI, and therefore it seems plausible to combine the two into one implementation.
public class MyDialog
    extends Dialog
    implements WindowListener
{
    public MyDialog(int num)
    {
        ... System.out.println("Construct dialog "+num); ... }

    public void windowOpened(WindowEvent we)
    {
        ... System.out.println("Dialog "+num+" opened!"); ... }

    public void windowClosed(WindowEvent we)
    {
        ... System.out.println("Dialog "+num+" closed!"); ... }
    ...
}

dialog = new MyDialog(1);
dialog.setVisible(true);

dialog = new MyDialog(2);
dialog.setModal(true);
dialog.setVisible(true);

dialog = new MyDialog(3);
dialog.setVisible(true);
...

Table 5.2: Modal Dialog Example

Assuming we close dialog 2 at some point, execution is resumed and dialog 3 is opened. The output of the above corresponds to:

Construct dialog 1
Construct dialog 2
Dialog 1 opened!
Dialog 2 opened!
Dialog 2 closed!
Construct dialog 3
Dialog 3 opened!
Dialog 3 closed!
Dialog 1 closed!

Since we do not call setVisible anymore, we must also simulate the halting mechanism. There are several solutions for this. We could drop the requirement that we never want to see any window, but this is a last resort. Another solution is to open modal dialogs in a separate thread, pausing all other threads. Although we did not try this solution, we are confident that it makes the system complex and error-prone. We might be better off dropping our initial requirement in such case.

Luckily we had another solution which turned out to be a lot simpler. All that is required for execution to be continued again, is that the modal dialog blocking execution is closed. We therefore choose to make an internal loop waiting for the dialog to close.
Normally T2 triggers the engine to make a step, but in this case we bypass T2 and trigger a step ourselves. This will force the engine to choose a new action to be performed upon the last created modal dialog. This action might close the dialog, in which case we break out of the loop and resume execution.

This perfectly simulates the behavior of modal dialogs, but also has its disadvantages. Because we bypass T2, T2 will not be notified of any steps we take in the mean time. T2 depends heavily on its regression support, so we definitely want to make sure we do not break this. We solved this by counting the nesting of our steps and appending the action description to the dummy when nesting is more than zero.

Related to dialogs are Swing’s JOptionPane s. Each JOptionPane encapsulates a dialog, which can have various forms and properties. The main problem is that JOptionPane uses the setVisible method within its static methods. For example, a call to JOptionPane.showMessageDialog will show a default message dialog, even with our aspects turned on. This is because we cannot weave any of Java’s core classes. This includes the Swing toolkit. For this reason we created another aspect which captures all calls to JOptionPane.show*Dialog methods. This aspect mimics the JOptionPane implementation, only this time we can capture the call to setVisible, because it is not part of the Java core.

Next we will explain the context used for this engine. The context reassembles all known components. Components become known when their window parent becomes visible. We capture this event with the aspects and decompose the window using the builder. The relevant components are added as elements to the context together with their actions.

For example, all components which extend the AbstractButton are wrapped inside a SwingButton, which is by default equipped with a ‘click’ action. This action performs the basic click operation on a button. It is possible for a component to have multiple actions, in such case the element will also have multiple actions attached to it. The current supported components and their actions are listed in the table below.

<table>
<thead>
<tr>
<th>Component</th>
<th>Action(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractButton</td>
<td>click</td>
<td>simulates a mouse click on the button</td>
</tr>
<tr>
<td>JTextComponent</td>
<td>change</td>
<td>changes the text of the field</td>
</tr>
<tr>
<td>JList</td>
<td>selection</td>
<td>changes the selected rows in the list</td>
</tr>
<tr>
<td>JTable</td>
<td>selection</td>
<td>changes the selected rows in the table</td>
</tr>
<tr>
<td>JComboBox</td>
<td>item</td>
<td>changes the selected item</td>
</tr>
</tbody>
</table>

Table 5.3: Supported components

Many components do not have actions, e.g. panels, labels, progressbars. Although they may contain other components which do have actions. The builder for this engine will recursively build the context for these components. In some cases this requires us to specify how to continue. Take for example a JSplitPane. The component itself does not have any actions of interest, but its children may have. Therefore the builder continues to build a context for the leftComponent and rightComponent. A slightly more interesting example is a JMenuBar. The JMenuBar is decomposed into its child JMenu components. These are again instances of an AbstractButton. However, if the child component is a JMenu instance, then it can have its own list of child components again. So each JMenu component is again decomposed into its child JMenuItem components, etc.

It can be the case that a component is not supported by the engine. The engine will then issue a warning that the component is not supported and recurse all its child
components. These can be accessed for each component that is an instance of the `Container` class. If the component does not extend the `Container` then it will be wrapped in a `SwingElement` and recursion stops.

5.4 JavaServer Faces Engine

JavaServer Faces is a web technology based on JavaServer Pages, which is again based on Java Servlets. Each new layer adds more functionality to the previous layer. JavaServer Faces includes some useful functionality for automated testing. If the reader is unfamiliar with JavaServer Faces then he/she is advised to first read appendix B. This section will first highlight some of JavaServer Faces relevant features and then describe the implementation details.

Web-based applications are text-based by nature. This follows naturally from the underlying HTTP protocol which is also text-based. However, within the application itself it is often preferable to work with all sorts of types. For example, if the user is asked to enter his/her age into a form field, then this number will be converted to a string and sent over the network. At the server-side the developer has to convert the string to a number again.

Many languages do provide ways to make this task easier with default parsers and regular expressions. However, it would be ideal if this was completely invisible to both the user and developer. JavaServer Faces achieves this functionality by making input fields typed. When the user now submits a form it will be validated and converted to the correct type. Moreover, the developer can even register custom validators on an input field for which special constraints apply.

The implementation of this engine is divided into a client and server part. The client part controls the test run and is initiated through the command line. The server part hooks into the JavaServer Faces framework to collect information about the GUI state.

The engine requires a `phase listener` to be installed on the web-application. Phase listeners follow the well-known observer pattern. Before and after each phase of the request life cycle all registered phase listeners are invoked. Registration of a phase listener is done using a configuration file. JavaServer Faces allows multiple configuration files to be included in one web application, which makes it easy to hook in our phase listener.

We use a phase listener to build the context tree. The context tree for this engine is a simplified version of the component tree, annotated with type information. For the current engine the type information is limited to only the type. In the future this will also include information about the registered validators.

The target for this engine is the URL where the web application is deployed. This URL is used to perform an initial GET request. The server will return a session id and initialize the context tree. The context tree is transferred to the client through the `JsffMetadataServlet`.

Next, several steps might be taken before the engine decides to follow a link. Only then the client will collect the values from the input fields and perform a POST request. After the server has processed the request the meta data is requested again and a new context tree is constructed.

JavaServer Faces maintains several scopes for the beans of a web application. The developer configures his beans to work in a specific scope. The available scopes in JavaServer Faces are: request, session and application. Similar to the discussion in section 5.2.1 we have to draw a line for the trace state. We decided to start a new
session for each trace. This means that changes made to beans configured in the application scope are retained.

The phase listener can also be replaced by a custom rendering kit. The default rendering kit returns HTML pages. In our case we are not interested in the HTML, but only in the component tree annotated with type information. A custom rendering kit can output this information in some format and parsed by the client to build the context. Other engine implementations are encouraged to try this approach.
Chapter 6

Results

6.1 Coverage testing

We performed a simple experiment to test the effectiveness of our approach. We want to know whether automated GUI testing is feasible. We used an example application which tracks training statistics for different users and sports. It consist out of 3 main windows and several smaller dialogs.

The first screen shows the different users and the associated sports. After selecting a user and sport the main window is shown. This window contains a large table with all the training records. These can be filtered and sorted with appropriate fields and buttons. If the user selects several records he/she can also open up a graphical representation where he/she can decide which values are mapped onto what axis. The user can manage his/her records using the add, edit and remove functionality.

The application makes a good candidate. It contains many different components, requires validation of input fields, uses modal dialogs and is of considerable size. The application has been thoroughly tested and has been used for over 3 years now by several people.

We tested the application with the default settings for T2. However, since the application requires a database, we performed two experiments, with an empty and filled database. We measure the coverage using code coverage tool EMMA\cite{EMMA}. EMMA supports 4 levels of coverage per class, method, block and line. Here blocks are defined as consequitive statements without any jumps or jump labels.

In the first experiment we used the default settings. That is, for each session T2 generates 500 traces with each 3 steps. However, because there are modal dialogs it can be that some traces are count more than 3 steps. There is a timeout of 2 minutes which will cut off the session. We repeated the experiment 10 times. The cumulative results were generated separately by using 10 sessions and merging the coverage statistics. This experiment was repeated 5 times.

Before we could start all the tests we implemented actions for the components and added logging functionality. On the first test the application actually failed. We found a bug in the test application. The bug had to do with trying to graph an empty list of records. After we resolved this bug, we continued the experiment.

During the testing we encountered a bug concerning record storage. Each record is stored in a simple text file per line. The record fields are delimited by semicolons. A problem of this storage method occurs when writing comments with new lines. These new lines corrupt the database and make it impossible to read the file again.
In the next experiment we filled the database with dummy records. This ensures all actions are possible without having the framework try to add records, which is typically domain specific.

The results show that we can actually find bugs, as we did find them during testing. We do require the user to “startup” the testing process by adding domain data as in our experiment. The coverage is pretty good and by closer inspection we see that most of the uncovered parts are related to the graph function. This functionality cannot be fully tested because there are no windows for the application to paint on and thus related methods are not invoked. Consequently there are several domain classes which have not been fully covered, as this functionality is only required for the graphical representation.

### 6.2 Mutation testing

We also performed mutation testing with a toy application. This application has two lists and two buttons. The lists contain various persons which can be parented with the use of the “...make child of...” button. New persons can be added by clicking the “New person” button which popups a modal dialog asking for a name and age.

We generated mutations of the original program with $\mu$Java\(^\text{II}\). The mutations are simple modification in the original program such as $j++$ becomes $j--$, etc. $\mu$Java generated 54 mutations, most of them located in the ‘make-child’ action. To get to this action we need perform at least 3 actions. Namely, select one or more persons from
6.2. MUTATION TESTING

<table>
<thead>
<tr>
<th>nr.</th>
<th>[class, %]</th>
<th>[method, %]</th>
<th>[block, %]</th>
<th>[line, %]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90.00 %</td>
<td>67.00 %</td>
<td>68.00 %</td>
<td>74.00 %</td>
</tr>
<tr>
<td>2</td>
<td>90.00 %</td>
<td>64.00 %</td>
<td>63.00 %</td>
<td>68.00 %</td>
</tr>
<tr>
<td>3</td>
<td>90.00 %</td>
<td>67.00 %</td>
<td>68.00 %</td>
<td>74.00 %</td>
</tr>
<tr>
<td>4</td>
<td>92.00 %</td>
<td>70.00 %</td>
<td>70.00 %</td>
<td>77.00 %</td>
</tr>
<tr>
<td>5</td>
<td>90.00 %</td>
<td>67.00 %</td>
<td>69.00 %</td>
<td>74.00 %</td>
</tr>
<tr>
<td>6</td>
<td>88.00 %</td>
<td>66.00 %</td>
<td>66.00 %</td>
<td>72.00 %</td>
</tr>
<tr>
<td>7</td>
<td>86.00 %</td>
<td>63.00 %</td>
<td>55.00 %</td>
<td>60.00 %</td>
</tr>
<tr>
<td>8</td>
<td>88.00 %</td>
<td>64.00 %</td>
<td>64.00 %</td>
<td>70.00 %</td>
</tr>
<tr>
<td>9</td>
<td>90.00 %</td>
<td>67.00 %</td>
<td>68.00 %</td>
<td>74.00 %</td>
</tr>
<tr>
<td>10</td>
<td>90.00 %</td>
<td>65.00 %</td>
<td>61.00 %</td>
<td>68.00 %</td>
</tr>
<tr>
<td>avg.</td>
<td>89.40 %</td>
<td>66.00 %</td>
<td>65.20 %</td>
<td>71.10 %</td>
</tr>
<tr>
<td>std.dev.</td>
<td>1.65 %</td>
<td>2.05 %</td>
<td>4.59 %</td>
<td>4.86 %</td>
</tr>
</tbody>
</table>

Table 6.3: Filled database, single run, 3 steps

<table>
<thead>
<tr>
<th>nr.</th>
<th>[class, %]</th>
<th>[method, %]</th>
<th>[block, %]</th>
<th>line, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90.00 %</td>
<td>67.00 %</td>
<td>70.00 %</td>
<td>75.00 %</td>
</tr>
<tr>
<td>2</td>
<td>92.00 %</td>
<td>69.00 %</td>
<td>71.00 %</td>
<td>77.00 %</td>
</tr>
<tr>
<td>3</td>
<td>90.00 %</td>
<td>67.00 %</td>
<td>70.00 %</td>
<td>75.00 %</td>
</tr>
<tr>
<td>4</td>
<td>56.00 %</td>
<td>33.00 %</td>
<td>30.00 %</td>
<td>33.00 %</td>
</tr>
<tr>
<td>5</td>
<td>66.00 %</td>
<td>39.00 %</td>
<td>33.00 %</td>
<td>37.00 %</td>
</tr>
<tr>
<td>avg.</td>
<td>78.80 %</td>
<td>53.00 %</td>
<td>54.80 %</td>
<td>59.40 %</td>
</tr>
<tr>
<td>std.dev.</td>
<td>16.65 %</td>
<td>17.49 %</td>
<td>21.30 %</td>
<td>22.33 %</td>
</tr>
</tbody>
</table>

Table 6.4: Filled database, five cumulative runs, 3 steps

the left list, one or more persons from the right list and click the “...make child of...” button. We therefore decided test with a trace depth of 3.

Next we implemented a new strategy which performs “event coverage”. This is similar to a brute force strategy as it generates all combinations of actions of a fixed length. [12] argues that event-triple coverage positively influences the test suite coverage. We expect this to be true for our mutant tests as well.

We expect this to be more stable and successful than our random strategy. Below are the results of the different tests. The column “alive” means that the mutation went unnoticed, “killed” means we found the mutation and “no-term” means the application did not terminate and thus can be considered killed. We repeated this experiment five times for both strategies.

A note is in order here. The generated mutations do not always corrupt the program. For instance, the toy application contains a main frame class. This class is only created once and therefore modifying references to this class from non-static to static does not influence the functionality. Unsurprisingly, our framework does not detect these changes and specialized unit tests are required. There are several more mutations which do alter the functionality but do not necessarily break the application. These were left in the test suite, because they do not interfere with the relative percentage of the different experiments.

The differences between the percentage of killed mutations for the event coverage test can be explained by the random values chosen for each action. Since each action combination is only tested once, it can happen that we did not use the correct values
### Table 6.5: Random strategy, 3 steps, 500 traces

<table>
<thead>
<tr>
<th>nr.</th>
<th>alive</th>
<th>killed</th>
<th>no-term.</th>
<th>perc. killed</th>
<th>avg. time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>24</td>
<td>3</td>
<td>44.44 %</td>
<td>68095</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>25</td>
<td>2</td>
<td>46.30 %</td>
<td>67263</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>22</td>
<td>3</td>
<td>40.74 %</td>
<td>68821</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>26</td>
<td>1</td>
<td>48.15 %</td>
<td>59318</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td>24</td>
<td>2</td>
<td>44.44 %</td>
<td>63299</td>
</tr>
<tr>
<td>avg.</td>
<td>29.80</td>
<td>24.20</td>
<td>2.20</td>
<td>44.91 %</td>
<td>65359.20</td>
</tr>
<tr>
<td>std. dev.</td>
<td>1.48</td>
<td>1.48</td>
<td>0.84</td>
<td>3.16 %</td>
<td>3994.80</td>
</tr>
</tbody>
</table>

### Table 6.6: Event coverage strategy, 3 steps

<table>
<thead>
<tr>
<th>nr.</th>
<th>alive</th>
<th>killed</th>
<th>no-term.</th>
<th>perc. killed</th>
<th>avg. time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>26</td>
<td>2</td>
<td>48.15 %</td>
<td>55403</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>26</td>
<td>2</td>
<td>48.15 %</td>
<td>57905</td>
</tr>
<tr>
<td>3</td>
<td>29</td>
<td>25</td>
<td>2</td>
<td>46.30 %</td>
<td>57524</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>26</td>
<td>2</td>
<td>48.15 %</td>
<td>62220</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>26</td>
<td>2</td>
<td>48.15 %</td>
<td>57697</td>
</tr>
<tr>
<td>avg.</td>
<td>28.25</td>
<td>25.75</td>
<td>2.00</td>
<td>47.69 %</td>
<td>58263.00</td>
</tr>
<tr>
<td>std. dev.</td>
<td>0.50</td>
<td>0.50</td>
<td>0.00</td>
<td>0.93 %</td>
<td>2858.42</td>
</tr>
</tbody>
</table>

However, we can conclude from the triple-event coverage results that it’s a stable strategy as well as being quite fast on average. This can be explained by the systematic testing approach. It executes each combination of 3 actions covering all basic execution paths in the application. This acknowledges the observation made in [12], who argue that triple-event coverage influence the test suite coverage. However, we would like to stress that the fixed length for a trace is determined by the application. Our simple required 3 actions to activate several parts of the application. This may very well differ in other applications in which case the number should be increased to be able to test these parts.

The random strategy can still be useful for a quick test. The advantage of the random strategy is the control over the number of sequences. This value directly influences the run time of the session. However, coverage is consequently lower due to the limited amount of time. For illustration purposes we ran a simple experiment where we slowly increased the number of traces we generate. Figure 6.1 clearly shows the relation between the number of bugs found and the time spent.

The reader should not conclude from this figure that we cannot track down more than 50% of the mutations. The reason for this is that the toy application has no specification. For instance, if we only parent a child when the age is lower than the parent, then a mutation which negates this condition will go undetected. We never checked whether the child was really parented. This observation does not influence the conclusion we made with respect the difference between the strategies.
Figure 6.1: Relation between number of traces and time spent
Chapter 7

Conclusions

We have succeeded in building an automated GUI testing framework for different GUI toolkits. Our experiment has shown that it can in fact detect bugs and achieve a pretty good coverage. We decided to hide all GUI activities from the user using aspects. This solved the problem of modal dialogs and improved usability for the developer. However, we were forced to simulate most of the window show and hide functionality. For this we used aspects which gives an elegant solution.

During the experiment we encountered a shortcoming for custom components. The coverage percentage for custom components is low because they are never painted on the screen. We did not solve this problem because of time constraints. However, we think it can easily be solved by identifying these custom components and then painting them on a mockup screen. This will effectively invoke the appropriate painting logic and increase coverage again.

The mutation tests have shown that random testing is not always effective, but does offer a way to quickly check an application for simple bugs. The event coverage strategy offered a steady coverage percentage but requires the developer to determine a good trace depth. If this value is set too low then we will not be able to catch all bugs, but when set to high it becomes closer to brute forcing which can be infeasible for high values. However, our tests have shown that in our case triple-event coverage as discussed in [12] is indeed effective.

Another solution is to revise our decision of hiding GUI activity. Instead the developer will be presented with live display of all the actions taking place. However, we still require aspects to stop modal dialogs from blocking our framework.

As for the integration with T2, THEMIS functions as a target of T2. This gave us a lot of flexibility, while maintaining most of the functionality of T2. In the future we can decide to drop T2 all together and replace the functionality by our own ‘driver’. The main reason for integrating our framework with T2 in such fashion were definition differences and implementations issues. T2 defines a target as a single class, while GUI testing is typically performed on the whole application. Another problem was the dynamically changing interface of GUIs which was incompatible with the static introspection done by T2.

The implementation of the different engines was relatively easy. We used an iterative process of refining the framework interface and in the end it only took several hours with average knowledge of the GUI toolkit to successfully implement an engine. Also the ability for creating custom strategies functions very well. We could easily implement the protocol/event coverage to improve coverage.

THEMIS also provides several other advanced functions which allow developers to
customize the testing process. We integrated replay functionality to circumvent domain specific parts of the application such as login screens. We think that this can even be taken a step further by allowing the developer to annotate parts of the application. The engine will then use the annotations instead of the domain map for testing.

While developing THEMIS we encountered a use of our framework to test *usability*. T2 can limit the length of traces by passing an argument to the framework. We can use this argument to see what parts of the target application are covered within a specified number of steps. The way to do this is to instrument the target application with a coverage tool and run THEMIS with the specified step count. The result will be a coverage report where uncovered parts require more steps than specified. This may result in a redesign of the application as some of the functionality is hard to get to.

All in all we succeeded in building a framework for automated GUI testing. Automated GUI testing can be effectively used to track down bugs in the early stages of development without too much work. The developer can quickly scan the application for validation errors and other basic mistakes without having to manually work his/her way through the GUI. There are still parts of automating GUI testing that we did not cover and we have hinted at a few of them already. However, our framework provides a good basis for further research and development in this area.
Appendix A

AspectJ

Aspect-oriented programming (AOP) is designed to capture crosscutting concerns in an object-oriented design model. AOP solves crosscutting problems such as security, logging, persistence, etc. For this AOP introduces the notion of aspects. An aspect is made up of two parts: an advice and joinpoints. An advice is a piece of functionality written in plain Java. This piece of code can be inserted anywhere in the program by defining injection points, or rather joinpoints.

For each joinpoint the advice can decide to insert the code before, after or in place (around) that joinpoint. These joinpoints are defined by pointcuts. A pointcut can either be a regular expression-like statement or using annotations.

We will now demonstrate the power of aspects using a simple example. We have a method which can draw primitive characters using only lines (see A.1). However, the programmer of this piece of code did a poor job and forgot to add coloring and scaling. This would mean that we would have manually check and modify over 100 lines. Since we are lazy programmers we try to solve the problem using aspects.

```java
public void drawCharacter(Graphics g, char c, int x, int y)
{
    switch (c)
    {
        case 'A':
            g.drawLine( x, y, x, y-5);
            g.drawLine( x, y-5, x+4, y-5);
            g.drawLine(x+4, y-5, x+4, y);
            g.drawLine(x+4, y-3, x+4, y-3);
            break;
        case 'B':
            ...
            ...
    }
}
```

Table A.1: Draw Character Example

We want to color each character using two colors, one color for horizontal lines and one for vertical lines. For this we calculate the slope in the x and y direction and color the lines with a larger x slope blue and the more vertical lines will be red.
Table A.2: Color Aspect Example

The aspect listed in A.2 shows the necessary code. We define an aspect ColorA which contains a single before advice. The pointcut associated with this advice defines our joinpoints and states that before each call to drawLine on the object g of type Graphics, we want to insert the advice. In this advice we can use the arguments passed to the drawLine to calculate the slopes and use the target object to set the correct color.

Table A.3: Enlarge Aspect Example

The aspect in listing A.3 shows another example of how aspects can be used to capture crosscutting concerns. Here we replace the call to drawLine with our own call to drawLine which uses the old values multiplied by a scaling factor. Now we can change the FACTOR variable to scale our characters.

The reader might wonder what !within(EnlargeA) means. This part of the pointcut definition ensures that only method calls to drawLine outside the aspect are affected by this advice. If we were to leave this out the advice would also apply to the method call within the advice itself and cause an infinitive loop.

We have shown that aspects are a useful addition to object-oriented programming to capture crosscutting requirements in a program. We have only scratched the surface of what is possible with aspects and there is quite a lot of research going on about how to improve and encourage developers to use aspects. For further details about AOP we direct you to the website of AspectJ [http://www.eclipse.org/aspectj/].
Appendix B

JavaServer Faces

JavaServer Faces is a server-side user interface framework for web-based applications. It builds an extra layer on top of the somewhat older Java servlet technology and goes hand-in-hand with JavaServer Pages, see figure B.1. The current version is 1.2 which ships with Java EE 5. JavaServer Faces technology separates the business logic and view representation further apart. It includes several key factors in web application development namely: representation of UI components and their state, event handling and input validation, page navigation and support for internationalization and accessibility.

![Diagram of JavaServer Pages and JavaServer Faces](image)

Figure B.1: Web technologies

In figure B.2 we see the life cycle of a request processed by JavaServer Faces. When a request comes in the server retrieves the view identifier from the request. The view is created, if it does not exist. The components retrieve their values updating the UI component tree. In the next two phases the values are validated and the business model (backing bean) is updated. The fifth phase invokes business logic stored in the backing beans to handle form submission. In this phase the backing beans are also responsible for specifying the next view. For this purpose the outcome of a method is used, which determines the next page based on the configured navigation rules. The last phase renders the components in their current state and a response is returned.

JavaServer Faces has a few nice features in relation with automated testing. Validation and conversion between request parameters and the business model is handled by the server. This means that we have type information available for text fields. Instead of generating all sorts of strings we can focus test data generation to a specific type. Moreover, JavaServer Faces gives the developer even more control during validation, which may very well be used to the advantage of automated testing.

Another feature of JavaServer Faces can be traced back to the way pages are rendered in a web browser. Every input fields and submit button is given an identifier during
rendering, if none was specified. JavaServer Faces uses this information to restore the view of the web page on the server. This information can also be used by us to identify components and generate trace reports.
Appendix C

T2

T2 is a unit testing tool which automatically generates test cases through a series of method calls. Test generation can be influenced by adding pre- and postconditions to the class. These can be specified using the `assert` statement inside the class methods.

A typical test iteration in T2 starts with the creation of a `target object`. This target object will be used to invoke member methods. This will most likely change the internal state of the object, which can then be passed or targeted for the next invocation. Each invocation can throw assertion exceptions. These exceptions can be tagged with `PRE`, `POST` or `APP`.

If T2 encounters an exception with a `PRE` tag, it will discard the invocation, because the precondition was not met. An `APP` tag signals that the created situation is not possible according to the application model. For example, it could be the case that a method a is never called before method b. This can be expressed using an application model. And last, if an `POST` tag is encountered it will output the violating trace leading to this exception. Furthermore, a `class invariant` can be specified, which is checked after each method invocation.

During an iteration objects are created and modified. These objects also live inside a pool. When T2 needs to have an object for a method invocation it will decide whether to pick one from the pool or create a new one. This effectively combines several invocations together. T2 combines methods in such a way that the correctness of a class is expressed in terms of the conjunction of all its method specifications.

T2 also supports replaying of previous traces. When a violation occurs T2 will save the trace to disk. The developer can fix the class and replay the violating trace to check whether he solved the problem.

T2 differs from other tools largely because it allows on-the-fly testing, which makes it more interactive. Some other tools also put a constraint on the specification language, whereas T2 allows specifications to be written in plain Java.
Bibliography


