FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Generalization of the model to implementation mapping tool

Marcos Augusto Ribeiro da Fonseca Guerra Liberal

Report of Project/Dissertation
Master in Informatics and Computing Engineering

Supervisor: Ana Cristina Ramada Paiva (Eng.)

29th June, 2008

Generalization of the model to implementation mapping tool

Marcos Augusto Ribeiro da Fonseca Guerra Liberal

Report of Project/Dissertation

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: António Augusto de Sousa (Eng.)

External Examiner: José Francisco Creissac Freitas Campos (Eng.)

Internal Examiner: Ana Cristina Ramada Paiva (Eng.)

31st July, 2008

Resumo

O teste de Interfaces Gráficas é uma área de investigação em crescimento. O uso generalizado de Interfaces Gráficas e o aumento da sua complexidade levaram à necessidade de desenvolver novas técnicas e ferramentas que auxiliem os "testers" no processo de testes. Uma equipa de investigação da Faculdade de Engenharia da Universidade do Porto (FEUP) desenvolveu uma extensão para o Microsoft Spec Explorer chamada GUI Mapping Tool que adiciona novas funcionalidades à ferramenta para que esta possa suportar o teste de Interfaces Gráficas. O Spec Explorer é uma ferramenta que permite a geração e execução automática de casos de teste. Esta ferramenta implementa uma técnica de testes chamada Model-Based Testing (MBT) em que os casos de teste são derivados directamente de um modelo da aplicação que queremos testar, modelo este que tem de estar escrito numa qualquer notação formal. Actualmente, a Mapping Tool desenvolvida pela equipa de investigadores possui alguns problemas que dificultam o seu uso num vasto leque the aplicações e que tornam o seu desenvolvimento bastante complicado. Entre outros problemas, a ferramenta usa uma framework obsoleta e actualmente inacessível para interagir programaticamente com Interfaces Gráficas e não foi feita para supportar aplicações Java e Web.

O objectivo deste trabalho é providenciar soluções para os problemas encontrados na antiga Mapping Tool e desenvolver uma nova ferramenta que implemente essas soluções.

Este trabalho permitiu concluír que não existe uma técnica ou uma ferramenta para o teste de Interfaces Gráficas que seja adequada em todas as situações de teste, e que uma das escolhas mais importantes a fazer quando se pretende desenvolver uma ferramenta para automação de testes de Interfaces Gráficas é a escolha da framework usada para interagir programaticamente com estas, isto porque é a framework que define o que o ferramenta vai ou não poder fazer.

Abstract

Graphical User Interface (GUI) testing is a growing research area. The widespread use of GUIs and increase of complexity lead to the necessity of new techniques and tools that aid the testers on their testing process. A research team from Faculdade de Engenharia da Universidade do Porto (FEUP) developed an extension to the Microsoft Spec Explorer called GUI Mapping Tool that adds new functionalities to the tool so it can support GUI testing. Spec Explorer is a tool that provides a way of automatically generating and executing test cases. It implements a testing technique called Model-Based Testing (MBT) in which test cases are derived from a formal model of the application we want to test. Currently, the Mapping Tool developed by that team of researchers has some problems that make it difficult to use it on a big set of GUI applications and that make the development of the tool harder. Among other problems, it uses an obsolete and currently inaccessible framework for programmatically interact with GUIs and it was not made to work with Java and Web applications.

The objective of this work is to provide solutions for the problems found on the old GUI Mapping Tool and develop a new one that implements those solutions.

This work concluded that there is not a technique or a tool for GUI testing that fits on all testing purposes, and that one of the most important choices that has to be made when building a tool for automated GUI testing is the choice of the framework for programmatically interact with GUIs. This is because the framework defines what the tool can and can not do.

Acknowledgements

I would like to thank my supervisor, Professor Ana Cristina Ramada Paiva Pimenta, from Engineering Faculty of Porto University, for her guidance, determined search of resources, unforgettable mentoring and encouragement that made this dissertation possible.

A special thank to my co-supervisor Professor João Carlos Pascoal de Faria, also from Engineering Faculty of Porto University, for his inputs, enthusiasm and his invaluable perceptiveness in the discussions we had that enriched my perspective.

I whish to thank my friends Francisco Santos and Telmo Couto for their support and feedback about my work.

I wish to express my gratitude to my parents, Armindo Liberal and Ana Clara Liberal, and to my sister Ana Helena Liberal for all their support and comprehension.

And a very special thank to Ana Mafalda Correia for all support, feedback, patience and encouragement.

Contents

List of Figures

List of Tables

Abbreviations and Symbols

API	Application Programming Interface
CLI	Command Line Interface
FEUP	Faculdade de Engenharia da Universidade do Porto
FSM	Finite State Machine
GUI	Graphical User Interface
HTML	HyperText Markup Language
MBT	Model-Based Testing
SUT	Software Under Test
UI	User Interface UML
Unified Modeling Language	
VDM	Vienna Development Method
XML	eXtensible Markup Language
WPF	Windows Presentation Foundation

Chapter 1

Introduction

1.1 Context

Nowadays, most of the software systems provide a Graphical User Interface (GUI). This has become the most accepted way of interacting with software. Users prefer GUIs over Command Line Interfaces (CLI) because they provide an easier way of interacting with software and because they can present output in a more readable and understandable manner. Due to the growing importance of GUIs and because of their properties, worries about their correct functioning are growing too. To raise the confidence on the correct functioning of GUIs we must submit them to testing.

GUI testing is a time consuming and difficult task mainly because there are many ways to interact with GUIs. A user can use the mouse to click on a GUI object (like a button), insert text with the keyboard on a text box, and generate events with sequences of key strokes. Besides that, each sequence of user actions can put the GUI on different states, so a single user action may have to be tested on all possible states because for each state it can have a different behavior. This raises exponentially the number of possible test cases.

Because of the importance of GUIs and of the difficulties involved on testing them, GUI testing has become an important research area and recently we have witnessed the emergence of many tools and approaches that help dealing with particular problems of GUI testing and that provide some degree of automation to the process. Building tools to automate GUI testing processes raises itself many problems in comparison with building tools which automate the testing of software through an API (Application Programming Interface). Automating regression testing on GUIs is problematic because the physical characteristics of the GUI may change many times without even changing the software underneath and this may turn useless some scripts created to automate the execution of test cases. The automatic generation of test cases suffers from the problem of the huge number of possible GUI states (as shown on the previous paragraph). Testing software through its GUI requires the use of some kind of framework which provides methods for interacting with GUIs by simulating user actions. Finally, when testing GUIs, we only have access to the state of the underlying software through the GUI layer.

Currently on the market, we have Capture/Replay tools like WinRunner, which record user interaction with GUIs and save it in scripts so they can be replayed later. These tools are most useful for regression testing. Unit Testing [Bur02] frameworks of the xUnit family used together with GUI test libraries are able to automate test execution but not test generation. There are even some tools that generate and execute random test cases, but most of the tools available do not cover the aspects of test case generation and test case evaluation. Some researchers have already attempt to cover this flaws. Ana C. R. Paiva in [PFTV05] developed an extension to the Microsoft Spec Explorer [Resa] tool so it can support GUI testing. Spec Explorer is a testing tool developed by a Microsoft Research team that already supports automatic test case generation and execution for API testing, but it lacks support for GUI testing. It uses a Model-Based testing [UL07] approach which consists on checking if a piece of software conforms to a formal specification (or model) of itself. For automatic test case generation and execution it becomes necessary to bind model actions with the corresponding concrete ones. The extension developed in [PFTV05] adds support for GUI testing to Spec Explorer by assisting testers relating abstract model actions to concrete ones on the GUI, filling the gap between model and implementation. Atif M. Memon in [MPS99] presented a new technique for automatic test case generation based on Artificial Intelligence planning techniques and developed an experimental tool to prove the concept.

The work described on this dissertation comes from the need to improve the state of the art of GUI testing, and is part of the AMBER iTest project. This project is being developed in partnership by FEUP and Critical Software. The main goal is to develop a set of tools and techniques to automate specification based GUI testing, so that they can be used in industrial environments. This work is one task in the project and aims to build a new extension to Spec Explorer that covers limitations of the previous extension developed by Ana C. R. Paiva.

1.2 Motivation and Goals

The motivation for this work comes from the need for better tools and techniques to support GUI testing. Available tools provide a small amount of automation and testers still have a lot of manual work while testing software through their GUIs which makes the process hard, expensive and sometimes neglected.

The main goal of this work is to build a new extension to Spec Explorer. This new tool shall support the following aspects:

- Recognize the most common controls used in GUI applications;
- Capable of testing Windows, Java and Web applications;
- Support one-to-many relationships between abstract model actions and concrete ones, which means that a single model action may map to a sequence of physical GUI actions;
- Save mapping and GUI objects information in XML [W3C] files with a predefined format;
- Make use of an existing, well supported and well documented library that provides methods to simulate user actions on a GUI.

Besides that, a well defined (and unambiguous) set of rules for mapping abstract model actions into concrete user action on the GUI must be defined.

1.3 Structure of the report

This dissertation contains four more chapters. On chapter 2, it is described the State of The Art of GUI Testing. Chapter 3 provides a detailed description of the problems involved on this dissertation and a high level (architectural) description of the solutions. Chapter 4 provides details about the implementation of the solutions. Finally chapter 5 shows the conclusions of this work and suggests future work.

Chapter 2

State of the Art

2.1 Introduction

This chapter describes the state of the art on GUI Testing. The most important techniques and tools currently available are here shown and explained as well as some problems and development opportunities.

2.2 GUI Testing techniques and tools

2.2.1 Capture/Replay

These tools build test scripts while the user is interacting with the GUI. The steps performed are saved into the script that can be replayed later for testing purposes. These tools are commonly and more effectively used in regression testing.

The test scripts can be made more general by using variables but some of these tools do not support this feature. They identify GUI objects through their physical properties which makes it difficult to adapt the scripts to GUI updates which may require build again all scripts.

There are currently on the market some good tools that use this testing approach. The most widely known commercial tool of this kind is HP/Mercury Interactives WinRunner [HP]. There are some free tools too, like Abbot [Wal].

2.2.2 GUI Unit Testing

Unit Testing [Bur02] is a well known software testing practice. It consists on testing individual software units to check if they satisfy all the expected requirements.

The most well known framework which supports Unit Testing is the xUnit family, but originally this framework family supports only testing software through its API and provides no support for GUI testing. Almost every major programming language has an implementation of xUnit. This framework family is code-driven, which means that when using an implementation of the xUnit like JUnit (implementation of xUnit for the Java programming language), the tester creates test cases as programs which can be automatically executed. An example of a test case in JUnit can be:

```
@Test public void simpleAdd() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

Some extensions to this framework which provide GUI support are available on the web, for example, the JFCUnit and Abbot which are extensions of JUnit with methods for simulating user actions.

As the previous technique, GUI Unit Testing requires that the tester creates the test cases manually but it facilitates regression testing because the code of the test cases can be run any time the tester wants.

2.2.3 Random Input Testing

This technique is also known as Monkey Testing. It consists of automatically executing the GUI with randomly generated inputs. These tools are relatively easy to build and there is no effort required to generate test suites.

The main problem of this technique is of probabilistic nature. If we have an input domain with 1000 elements and there is one bug related to one of these elements, there is a 1/1000 probability of finding that bug.

2.2.4 Goal-driven Approach to Generate Test Cases for GUIs

Atif M. Memon [MPS99] developed a technique for automatic test case generation based on well known Artificial Intelligence planning techniques. He developed a tool that analyses the hierarchical structure of a GUI and creates a set of operators that correspond to the possible user actions on the GUI. Then, the tester has to provide a set of initial and goal states to the tool which produces a set of plans as an output. Each plan is a sequence of operators which takes the GUI from an initial state to a goal state.

This technique has some scaling problems in large applications and the coverage of the test plans generated is too dependent on the provided set of initial and goal states (there is no systematic method to create this set). Despite of these flaws, this is a step on the automation of test case generation.

2.2.5 Model-Based GUI Testing

Model-Based testing [UL07] is a testing technique in which test cases are directly derived from a formal model of the software under test (SUT). Typically when using this approach, the first thing to do is to build an abstract model of the system. Building an early model of the system is considered a good software development practice [Sch06] mainly because models are normally at a higher abstraction level, which encourages developers to think earlier about high level aspects of the system like the structure, functionality and temporal behavior. By thinking about these aspects earlier on the development process, developers may detect design errors earlier which in turn make the correction of these errors cheaper than correcting them in later phases. Models are built using a modeling language which can be semi-formal like standard UML [SV05][Gro], or formal like VDM++ [VDM][FLM+05] or Spec# [Resb][ea05].

The main advantage of model-based testing is the potential it has for automation. If a model is built using a formal language, there are tools, like Spec Explorer, that automate the processes of test case generation and execution. These tools require a formal model as an input, and then they automatically generate abstract test cases. These test cases cannot be directly executed with the SUT because they are in a different level of abstraction, which means that a new set of concrete test cases has to be derived. To do so, a mapping between the model and the implementation must be created. To guide the creation of the test cases, the tester may define their coverage criteria too. Finally, abstract and concrete test cases can be executed and results compared. If the results on both test suites are equivalent, we may say that the software conforms to the specification under the defined coverage criteria.

Figure 2.1 illustrates the model-based testing process.

This technique can be used on GUI testing. Ana C. R. Paiva on [PFTV05] developed an extension to the Microsoft Spec Explorer to add GUI testing support to this tool. Spec Explorer is a Model-Based testing tool, but it requires that the application we want to test is implemented in a .Net [WSW03] language and built as a .Net assembly. The actions of the model are then mapped to methods on the implementation. This raises some problems when we want to test GUIs, because the application may not be implemented in a .Net language (it can be, for instance, a Win32 application) and the application functionality may only be available through the GUI layer. In this case we need a way to map model actions with concrete user actions on the GUI. The extension developed in [PFTV05] by

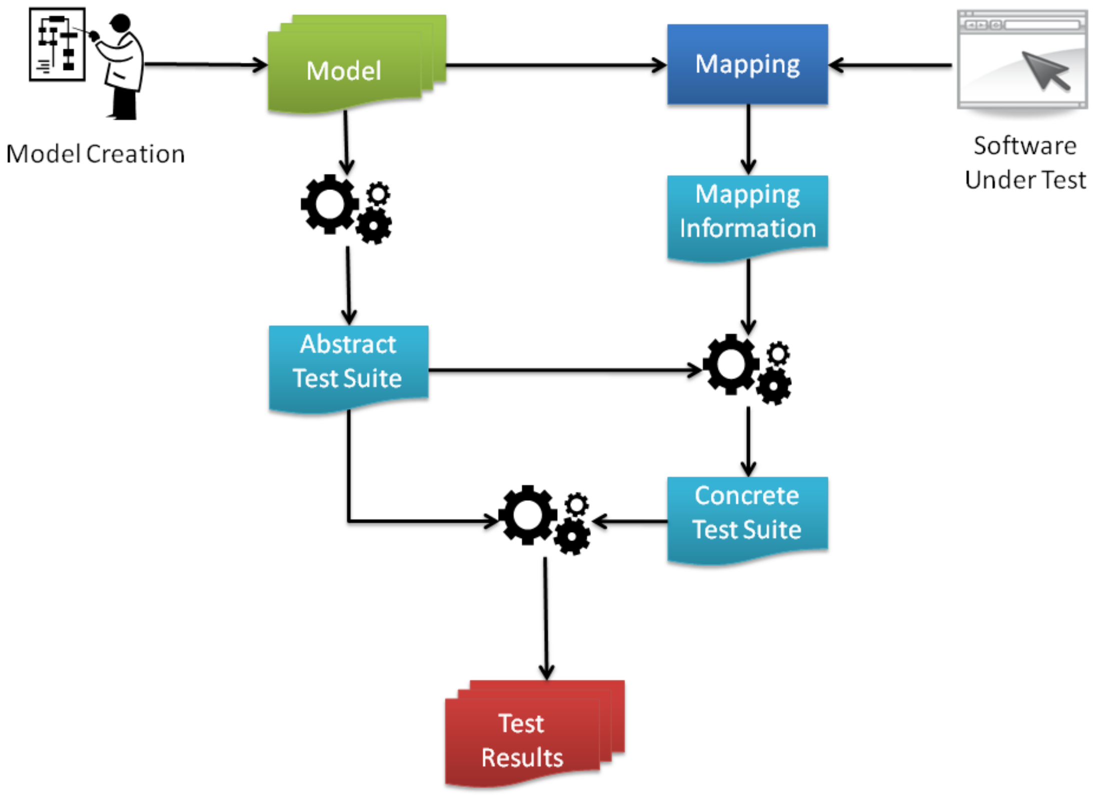


Figure 2.1: Overview of the Model-Based Testing process

Ana C. R. Paiva assists the user in this task and provide some guidelines on how to build GUI models in the Spec# modeling language.

Model-Based testing is a testing technique that has the potential to provide a high degree of automation and it allows a more exhaustive testing of applications, because we can automatically generate test cases for virtually every possible state of the application under test. The main problem of this technique is that for large applications testing every possible state is not viable, because the number of states may be too large (state explosion problem). The need for a formal model of the application can be seen as a problem if we do not want to build one, but on the other hand, building early models has been proved to be a good software development practice [Sch06].

2.2.5.1 Spec Explorer

This tool was developed by a Microsoft Research team with the purpose of creating a tool that supports Model-Based testing, which means that it must be able to automatically generate abstract and concrete test cases from a formal model of an application, and must be able to run them for conformance checking. The models (or software specifications) are built in the Spec# formal modeling language which is a super-set of the C# language [JF03] that adds support for non-null types, pre and post conditions, as well as object invariants. In Spec#, user creates models by defining a set of state variables that represent the state of the model and a set of functions or methods that represent the behavior (or functionality) of the model. Below, an example of a Spec# model is presented:

```
namespace Test;

/* State variables */
bool Running = false;
bool Scientific = false;

/* Methods */
[Action] void SetRunning(bool newRunning)
  requires Running != newRunning;
{
  Running = newRunning;
}

[Action] void SetScientific(bool newScientific)
  requires Running;
  requires Scientific != newScientific;
{
  Scientific = newScientific;
}
```

The functions of the model may have preconditions (defined by the requires keyword) which define the conditions necessary so that the function may be applied.

This is a very simple explanation about how to build models in Spec#. The language has much more functionalities including the ones we see in most object-oriented programming languages (like classes). A full description of the language can be found on the Spec# project page [Resb].

After building the model, the user may use it with Spec Explorer to generate test cases. First Spec Explorer translates the model in Spec# to a Finite State Machine (FSM). This is an exploration process in which model actions are executed, and the resulting states are recorded. The transitions between states on the FSM are calls to model actions, and each FSM state is the resulting state of the model when an action is called at some point in the exploration process. The user may specify how this process is to be executed by using one or more of the following techniques supported by the tool to prune the exploration:

- **State filters** — these are Boolean expressions that are run on each state reached in the exploration process. If the expression is false, the exploration process do not continue from that state;
- **Restriction of the domains** — the user may define the possible values of each state variable;
- **Equivalence classes** — in this technique states are partitioned in equivalence classes. If in the exploration process a state belongs to an equivalence class, and a predetermined number of representatives of that class have been already reached, further exploration from that state is prevented.

From the FSM, Spec Explorer derives the abstract test cases which are in turn possible paths between two states of the FSM. To be able to create and execute the concrete test cases, the user has to map each model action to the corresponding method on the implementation. The implementation must be a .Net assembly and must be accessible by the Spec Explorer.

Finally the test cases may be executed. The conformance checking is done by comparing the results of the abstract and concrete tests. If the result of each abstract test case is equal to the result of the corresponding concrete test case, then we can say that the implementation conforms to the specification.

2.2.5.2 NModel

NModel [NMo] is a model-based testing tool like Spec Explorer. An executable formal model acts as a specification of an implementation. A model can be analyzed with special tools that help on finding defects on the specification, test cases can be automatically generated from it, and it acts as a test oracle that provides the correct results when testing the implementation.

On NModel, models are written in C# (in contrast to the Spec Explorer that uses models written in Spec modeling language). A model consists of actions and state variables defined on a single namespace.

An example of an NModel C# model can be:

```
namespace MyModelProgram
{
static class Contract
{
static bool MyStateVariable1 = true;

[Action]
static void Reset() { /* ... */ }
}

class Client : LabeledInstance<Client>
{
bool isEntered = false;
// ...
}
}
```

NModel is not a single application but a set of independent applications and libraries. The NModel package comes with:

- A library with attributes and data types for building models in C#;
- Two different model visualization and analysis tools;
- A test generation tool;
- A test execution tool that runs the tests and checks conformance between specification and implementation.

NModel also permits the definition of pre-conditions on actions of the model by defining functions with the same name as the actions, but ending with Enabled, as the following example shows:

```
static void FooEnabled() { return mode == Mode.Start; }
static void FooEnabled(int x) { return x > 0; }
static void FooEnabled2(int x) { return x < 100; }

[Action]
static void Foo(int x)
{
mode = Mode.Running;
}
```

It is also possible to apply state filters and domain restrictions to the state variables to prune the exploration process (this process is also similar to the one described on the previous section). This is achieved by using some attributes like [StateFIlter] and [Domain] on functions.

More details about NModel and how to build NModel models can be found on the project page [NMo].

2.2.6 GUI Testing Frameworks

This section describes important frameworks that can be used in the development of applications for GUI testing. These frameworks provide ways of extracting properties and interacting programmatically with GUIs.

2.2.6.1 UI Automation

UI Automation [MSD] is an object-oriented framework for GUI manipulation. It was developed by Microsoft to be their new accessibility framework for the Microsoft Windows. The framework provides programmatic access to GUI elements. Programs can search for GUI elements, retrieve its properties and simulate user interactions on those elements.

In UI Automation, every element of a GUI has an associated AutomationElement object which exposes properties of that element that are shared by all types of GUI elements. An example of a property is the ControlType which defines the type of a GUI element, for example, a button or a check box. Each AutomationElement also provides a set of objects of the type ControlPattern. These objects provide methods to interact with GUI elements. There are different types of ControlPatterns, which provide different kinds of interaction with GUI elements, and the ones supported by an AutomationElement depend on the ControlType of that element. For example, a simple button on a GUI has an AutomationElement object that represents it. The ControlType of that element is button and it supports the InvokePattern (a type of ControlPattern) which provides a method to simulate a user clicking on the button.

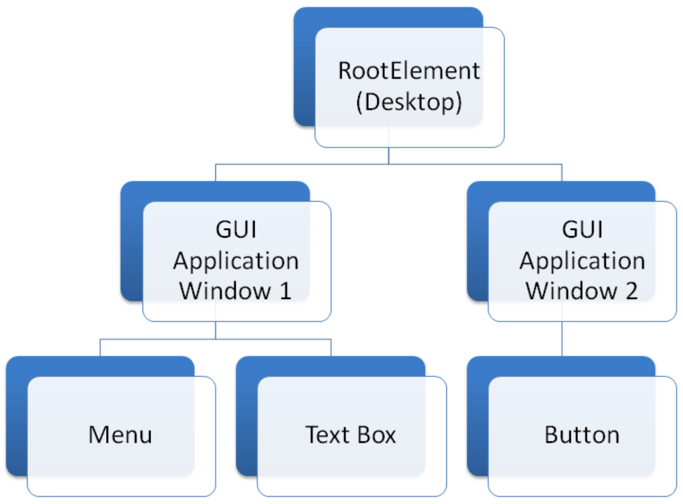


Figure 2.2: A possible arrangement of the UI Automation tree

UI Automation arranges AutomationElements hierarchically in a tree structure with the element representing the desktop as the root (figure 2.2). Elements descending directly from the root are application windows and following each of these elements are other elements which represent the controls of the application windows. Some of these controls may be containers which in turn may have more descendent elements which represent controls inside the container. The framework has classes which provide methods to navigate and search for AutomationElements on the tree.

UI Automation deals very well with GUIs that are built using different GUI platforms. Programmers may write their applications using UI Automation without having to worry if the GUI was built using only Win32 API, Windows Forms or Windows Presentation Foundation (WPF). For instance, UI Automation masks any differences found on property names of the different GUI frameworks. The Caption property of a Win32 button, the Content property of a WPF button and the ALT property of an HTML image are all seen in UI Automation as the Name property.

2.2.6.2 Abbot

Abbot [Wal] is a GUI testing framework for Java applications. Like UI Automation, it provides a way of accessing GUI elements programmatically, extract properties and simulating user interaction with them. Besides that, Abbot also comes with some tools which provide capture/replay functionalities.

Table 2.1: Comparative analysis of GUI testing techniques

	Automatic Test Case Generation	Automatic Generation of Test Case code	Automatic Test Case execution	Regression Testing	Conformance checking	Unit Testing
Capture/ Replay		x	x	x	x	
Random Input Testing	x		x			
GUI Unit Testing			x	x	x	x
Goal-driven Approach to Generate Test Cases for GUIs	x				x	
Model-Based GUI Testing	x	x	x		x	

Each type of GUI element in Java has an associated Tester class on the Abbot framework. So, for instance, for a JButton there is a JButtonTester class. These classes provide methods for simulating user actions on GUI elements and for making some assertions about their state. There are also classes that provide methods to search for GUI elements based on their properties.

Abbot also comes with capture/replay capabilities. It has a tool named Costello that is capable of recording sessions of user interaction with the GUI. These sessions are saved in scripts which can be edited on the same tool.

2.3 Conclusions

On GUI testing (as in all software testing), there is not one technique which overcomes all the other techniques in all situations. Each technique was created in response to a particular problem of GUI testing.

Table 2.1 and Table 2.2 show comparative analysis of tools and techniques based on several parameters.

Table 2.2: Comparative analysis of GUI testing frameworks based on the GUI platforms they support and on some other parameters

	Free	Win32	Windows Forms	WFC	Java AWT	Java Swing	Web	Accessible Documentation
UI Automation	x	x	x	x	x		x	x
Abbot	x				x	x		x

Chapter 3

Model-to-implementation Mapping Tool

This chapter makes a detailed description of the problem approached on this dissertation. A high level solution for the problem is also presented. The implementation details of this solution are presented on the next chapter.

3.1 The gap between model and implementation

On the previous chapter a description of the model-based testing process was presented and tools that apply this testing technique have been described. With this description we can see the potential that model-based testing brings in terms of automation.

When building model-based testing tools, one problem that rises naturally is how to make the connection between abstract model actions and concrete implementation actions. In model-based testing this connection is necessary because the tool must know which implementation action to call for each abstract action called in an abstract test case. These abstract test cases are generated from the model of the application and are made of sequences of abstract actions. When the mapping is done, each time an abstract action is called, the corresponding concrete action is called too and the results of both are compared.

Models can be built at different abstract levels depending on what we want to do with them. For instance, if we want to test a system or component at the API level, we will develop a model at the same level, which means that the model will consist on a set of actions and each action represents an API method that will be mapped to a concrete method on the implementation. If we want to test a system through the GUI, a more convenient model would consist in a set of abstract actions representing possible user actions on the GUI, like clicking a button or inserting text in a text box. To achieve this, the abstract model actions would have to be mapped somehow to these concrete user actions. The inconvenient of this last case is that we may not have access to an API with concrete user actions because GUIs provide all the functionality through the GUI layer (events generated by GUI controls).

On Spec Explorer, the implementation must be a .Net assembly (executable or dll) and the user maps the abstract model actions to methods on the assembly.

If we want to test an application through the GUI layer with Spec Explorer we need to create some intermediate code with the methods that simulate user actions. The abstract actions will then map to these methods. Building this code manually is a hard task and turns the process of testing GUIs with Spec Explorer unpractical, but some researchers have already built prototypes of tools to automatically generate this intermediate code like the one in [PFTV05].

3.2 GUI Mapping tool

As already mentioned, if we want to test applications through the GUI abstraction layer with Spec Explorer it is necessary to build some intermediate code that will simulate user interactions with the GUI.

The problem is that even for small applications, it is unpractical to build this intermediate code manually. To deal with this problem, a team of researchers already built a prototype of a mapping tool for the Spec Explorer [Resa] that helps mapping abstract model actions with GUI controls and automatically generates the intermediate code with the mapping information.

With the GUI mapping tool, a user connects abstract model actions to the corresponding GUI physical elements. The tool then uses a set of rules that take into account this mapping information to determine the code it has to generate.

The code consists on a set of methods that simulate user actions on the GUI. These methods are then connected on Spec Explorer to the abstract model actions as we were connecting model actions to methods of an implementation on a .Net Assembly.

The tool developed in [PFTV05] has some problems:

- It supports a small set of GUI controls;
- Only deals with Windows applications (has no support for Web and Java applications);
- It uses a private library provided by Microsoft. This library is not available to the public which makes more difficult the development and industrial acceptance of the mapping tool. It is also an obsolete library (as we are going to see, Microsoft has a newer, more powerful and free framework);

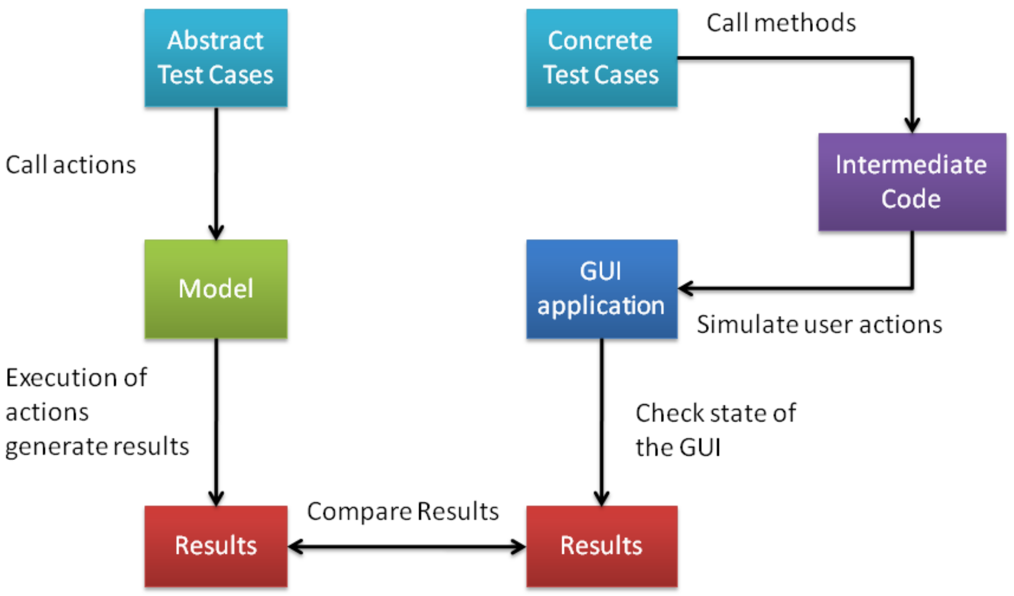


Figure 3.1: Execution of test cases for a GUI application on Spec Explorer with the intermediate code

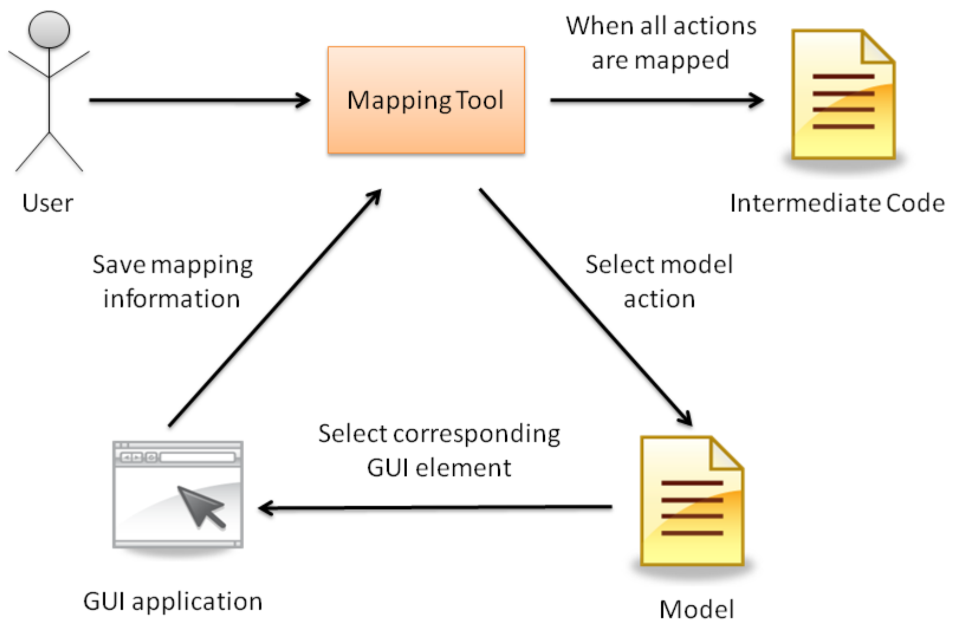


Figure 3.2: GUI mapping tool

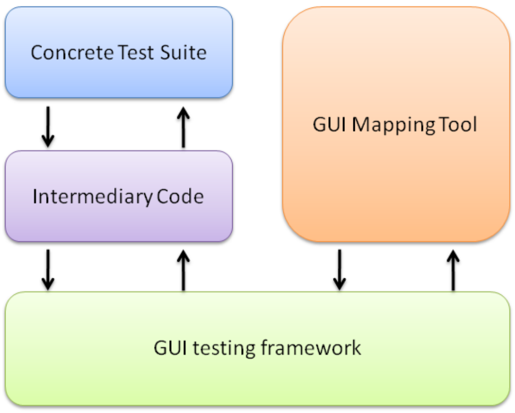


Figure 3.3: GUI testing framework

- Do not have a well defined rule (or set of rules) to uniquely identify a GUI control through the properties of that control;
- It only supports one-to-one relationships between abstract actions and concrete actions.

Descriptions of possible solutions to these problems are presented next on this section and on the following chapter.

3.2.1 Choosing a GUI testing framework

As said earlier, the intermediate code consists on a set of methods that simulate user actions on GUI elements. In order to accomplish this, the intermediate code must use a framework that provide ways of programmatically interact with GUIs. The same applies to the mapping tool itself, because it must be able to interact with GUIs so it can extract properties from the controls.

If we look at 2.2, we can see that on this research, a framework that supports all GUI libraries was not found. A solution that covers all GUI libraries must necessarily use both frameworks presented on the state of the art (UI Automation and Abbot). However, the tool developed on this dissertation only uses the UI Automation framework which means that currently it does not support Java Swing [Mic] applications.

3.2.2 Saving mapping information

The mapping tool must save mapping information for two reasons:

- Make decisions in the code generation process;
- The intermediate code needs access to the properties of the objects retrieved on the mapping process so it can search for them again on the test execution phase.

This means that two kinds of information have to be saved:

- All relations between model actions and concrete GUI objects;
- Properties of concrete GUI objects.

More details about how the mapping information is saved by the mapping tool and used by the intermediate code on the test execution phase and by the mapping tool on the code generation phase are given in the next chapter.

3.2.3 Uniquely identification of GUI elements

On the test execution process, each time a method of the intermediate code is called to execute a user action on the GUI, it must find the GUI element where the action must be performed. In order to do this, the method of the intermediate code uses the information about the objects that was saved on the mapping phase.

Searching a GUI application for a specific element through the properties of that element is a task highly dependent on the framework we choose to interact with GUIs. On the specific case of UI Automation (the framework used in this work), it provides access to a property called AutomationID. This property gives a unique name to each control inside the same container. A container can be, for example, a group box, a tab control or a panel.

Figure 3.4 shows a possible state of the UI Automation tree structure (see section 2.2.6.1). Each node that represents a GUI element has an AutomationID. The root node (desktop) and nodes that represent top-level windows do not have an AutomationID.

More details about how the mapping tool searches for GUI elements and what kind of information it uses are provided in the next chapter.

3.2.4 One-to-many relationships between model actions and concrete actions

Until now, the GUI mapping tool has been described as a tool that helps the user on the process of mapping each abstract action on a model to a single concrete user action on the GUI.

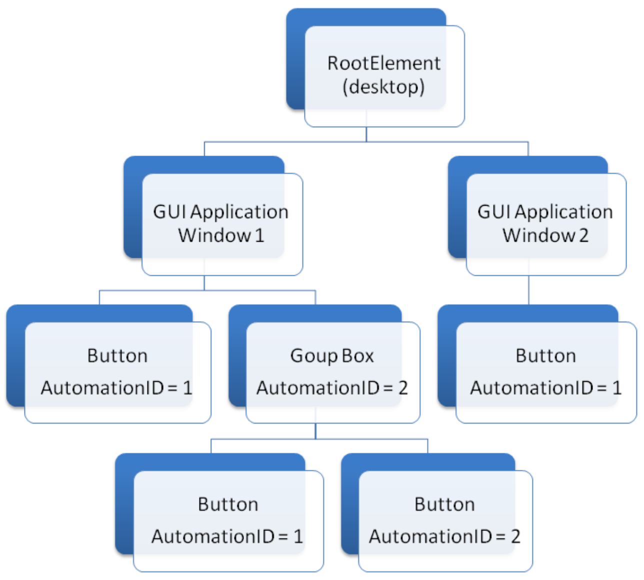


Figure 3.4: A possible state ofthe UI Automation tree with nodes representing UI elements and every element that is not a top-level window has an AutomationID

This idea of a mapping tool is enough for simple user actions like clicking on a button, inserting text on a text box, selecting an item on a list box or checking the caption of a label. But there are some situations in which we may have more complex user interactions like a drag and drop (a mouse down event on an object followed by a mouse move and a mouse up event) or some situations in which we want to model more high level actions like a login (inserting a username and a password on two text boxes and click a button). On these cases we need to be able to map a single model action to several concrete user actions.

More details about how this feature can be implemented will be provided on the next chapter.

3.3 Conclusions

On this chapter a solution to the gap between the model and the implementation on Spec Explorer was presented along with the proposal of solutions to some of the problems of the GUI Mapping Tool built on [PFTV05]. One of the most influent decisions presented on this chapter was the choice of the GUI testing framework. This framework is necessary to provide to the Mapping Tool and the intermediate code programmatic access to GUI objects. This means that the framework we choose determines what the tool can and cannot do with GUI objects.

The solution presented uses the UI Automation framework. By only using this framework any implementation of this solution will not support Java Swing applications, but if we look at the Abbot framework, it provides functionalities for GUI interaction very similar to the ones provided by UI Automation so a solution that uses the Abbot framework would look almost the same as the one presented here. Using both frameworks simultaneously raises some complicated architectural questions that go beyond the scope of this dissertation, because the two libraries work on two different platforms (Java and .Net).

Chapter 4

Implementation

This chapter describes the implementation developed on this dissertation for the solution presented on the previous chapter. As said earlier, a GUI mapping tool as already been developed by a team of researchers [PFTV05], but the tool has some problems that were listed on the previous chapter. The mapping tool built during this research work aims to cover the problems of the previous one.

4.1 GUI Model

Spec Explorer accepts models written in the Spec# modeling language. A model written in Spec# can be translated into a state machine where states are described by the model state variables, and transitions between states are actions of the model. Spec allows the creation of models at any level of abstraction depending on the needs of the modeler.

On this case, we want to test an application through the GUI layer, which means that we want to test how the application reacts when the user executes actions on the GUI. To achieve this, we need a model that specifies possible user actions on the GUI, so we can later map these specifications (or model actions) to the concrete user actions.

To help the user producing GUI models with meaning to be used by the mapping tool some guidelines to build GUI models will be provided next on this section. The mapping tool also uses some information about the names of the model actions in order to take some decisions, which means that some naming conventions must also exist. These naming conventions are also specified later on this section.

4.1.1 Guidelines

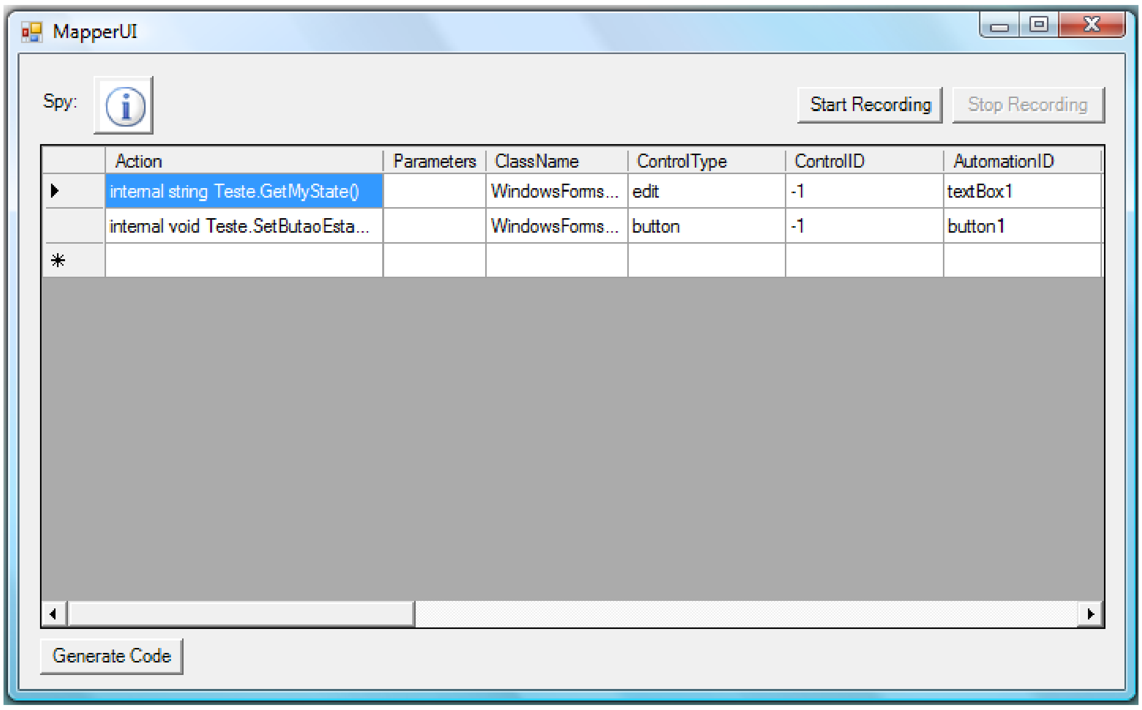
A model in Spec# is basically made of a set of functions (equivalent to functions or methods in a programming language) and a set of state variables. Functions which have the tag [Action] are possible transitions on the state machine generated in the exploration process by the Spec Explorer. The test cases are generated from the state machine which means that the sequence of actions on a test case is made of calls to these tagged functions.

If we want to test an application through the GUI layer, a good way of modeling it would be defining a set of actions which represent the set of possible user actions on the GUI and defining a set of actions which represent a user checking the state of the GUI.

An example of a GUI model can be:

```
namespace Test;

//State variable
bool myState = true;

// User clicking a button
[Action]
void SetButaoEstado() {
  myState = !myState;
}

// Probe action
[Action]
string GetMyState () {
  if(myState) {
    return true;
  }
  else {
    return false;
  }
}
```

This is a model of a very simple GUI which provides only two possible ways of interaction. The first action (SetButtonState()) represents a user clicking on a button which causes a change in the state. The second action (GetMyState()) represents a user checking the state of a GUI element. This last action is also called a probe action. Spec Explorer allows the definition of a special kind of actions called probe actions. These actions will always be executed whenever any other action that is not a probe action is executed. The



	Action	Parameters	ClassName	ControlType	ControlID	AutomationID
▶	internal string Teste.GetMyState()		WindowsForms...	edit	-1	textBox1
	internal void Teste.SetButaoEsta...		WindowsForms...	button	-1	button1
*						

Figure 4.1: GUI mapping tool

results of these actions are compared with the results of the corresponding methods on the implementation for conformance checking.

4.1.2 Naming conventions

One of the variables used by the GUI mapping tool to decide what code shall be generated for each model action, is the signature of the model action itself. It is required that the model actions written in Spec# follow some naming conventions:

- The model actions are constituted by a three character predefined prefix and a name;
- Prefixes are: Clk (click), Ins (insert text), Del (delete text), Set (override text), Get (read GUI state), Cmp (action that maps to several user actions);
- Actions related with the same GUI physical object must have the same name.

4.2 GUI Mapping tool

Figure 4.1 shows the UI of the mapping tool.

To map abstract model actions to concrete user actions, the user of the mapping tool must first say what GUI control will be the target of that action. To do that, the user drags the Spy button and drops it on the target control. When this is done, the mapping tool extracts the properties of the control and presents them on a table as shown on Figure 4.1.

When the mapping of model actions is done, the user may generate the intermediate code by clicking the Generate Code button. When this button is clicked the code generation process begins and the intermediate code is created. Files with mapping information are created along with the intermediate code as the result of this operation.

The Start Recording and Stop Recording buttons are used to map a single action to a sequence of user actions but this functionality will be later explained.

4.3 Logical names

Each physical GUI object is automatically associated with a logical name by the tool. The reason for this choice is to achieve implementation and specification independence, so the tool (and the user) can easily search for physical object information or mapping information.

Logical names are built by concatenating the namespace in which the GUI window is modeled, with the name of the abstract action (without the prefix).

```
namespace Calculator;

[action]
public void InsTextBox1(string s)
{
//...
}

[action]
public void DelTextBox1(int i, int f)
{
//...
}
```

The logical name created from the abstract actions presented above is:

```
CalculatorTextBox1
```

The abstract user actions, in the example, are modeling different user actions over the same GUI object. As it can be seen, if the name convention is effectively used, just one logical name will be generated for each GUI object. This means that the physical information of the GUI object will be saved just once.

4.4 XML files

On the previous chapter it was said that the mapping tool must save the mapping information. This information is saved into two XML files. One keeps information about the connection between model actions and GUI objects, and the other keeps information about the GUI objects (properties of GUI objects).

An example of a XML file with the mapping information is:

```
<Mappings>
  <Action>internal string Teste.GetMyState()</Action>
  <Parameters />
  <ControlType>edit</ControlType>
  <Return>string</Return>
  <LogicalName>TesteMyState</LogicalName>
  <Prefix>Get</Prefix>
  <Sufix>MyState</Sufix>
</Mappings>

<Mappings>
  <Action>internal void Teste.SetButaoEstado()</Action>
  <Parameters />
  <ControlType>button</ControlType>
  <Return>void</Return>
  <LogicalName>TesteButaoEstado</LogicalName>
  <Prefix>Set</Prefix>
  <Sufix>ButaoEstado</Sufix>
</Mappings>
```

This file contains a set of ¡Mappings¿ tags. Each ¡Mappings¿¡/Mappings¿ pair contains the information about a mapping between an abstract action and a GUI object. That information is contained within another set of tags described in the following list:

- <**Action**> — Contains the complete action prototype (return type, action name and parameters);
- <**Parameters**> — Contains the type of all the parameters of the action;
- <**ControlType**> — Contains the type of the GUI control to which the abstract action is connected;
- <**Return**> — Contains the return type of the abstract action;

- <**LogicalName**> — Contains the logical name of the GUI object to which the abstract action is connected;
- <**Prefix**> — Contains the prefix of the action;
- <**Sufix**> — Contains the name of the action (action without the prefix).

An example of a XML file with the properties of the GUI objects is:

```
<Objects>
  <LogicalName>TesteMyState</LogicalName>
  <ControlType>edit</ControlType>
  <ControlID>-</ControlID>
  <AutomationID>textBox1</AutomationID>
  <Caption />
  <ParentCaption>Teste</ParentCaption>
  <Path />
</Objects>
<Objects>
  <LogicalName>TesteButaoEstado</LogicalName>
  <ControlType>button</ControlType>
  <ControlID>-</ControlID>
  <AutomationID>button1</AutomationID>
  <Caption>Estado</Caption>
  <ParentCaption>Teste</ParentCaption>
  <Path />
</Objects>
```

This file, like the previous one, also contains a set of tags. Each ¡Objects¿¡/Objects¿ pair contains information about one GUI element. That information is contained on another set of tags. The following list describes the tags inside a ¡Objects¿¡/Objects¿ pair:

- <**LogicalName**> — Contains the logical name of the GUI object;
- <**ControlType**> — Contains the type of the GUI object;
- <**ControlID**> — Contains the ControlID property of the GUI object;
- <**AutomationID**> — Contains the AutomationID of the GUI object;
- <**Caption**> — Contains the caption of the GUI object;
- <**ParentCaption**> — Contains the caption of the window containing the GUI object;

- <**Path**> — Contains the AutomationID path of the GUI object. This path is the sequence of AutomationIDs of all the parent containers of the GUI object. This path uniquely identifies the GUI object and is used by the intermediary code to search for it.

4.5 GUI User Actions Library

GUI objects have well defined possible user interactions. We know that we can click on a button, we can insert text on a text box, and we can check/uncheck a checkbox. UI Automation provides a high level way of programmatically interact with those objects, but in order to facilitate the development of the application, a higher level library was created. For instance, for UI Automation, clicking on a button is different than clicking on a check box, but to the user is just a click. This new library tries to reach this level of abstraction with the objective of making future developments easier.

Having this in mind, a library with a set of functions to simulate the possible user interactions with GUI objects was created.

Besides having functions for interaction with GUI objects, the library also have some functions for getting properties of those objects. These functions are needed to check the GUI state and they connect to the probe actions on Spec Explorer.

The following example shows the structure of a method of this library:

```
public void Click(string logicalName)
{
//(1) read physical properties of the GUI
//object named logicalName

//(2)finds the concrete GUI object

//(3)performs a click on that GUI object
}
```

The code generated by the mapping tool consists on functions calls to this library.

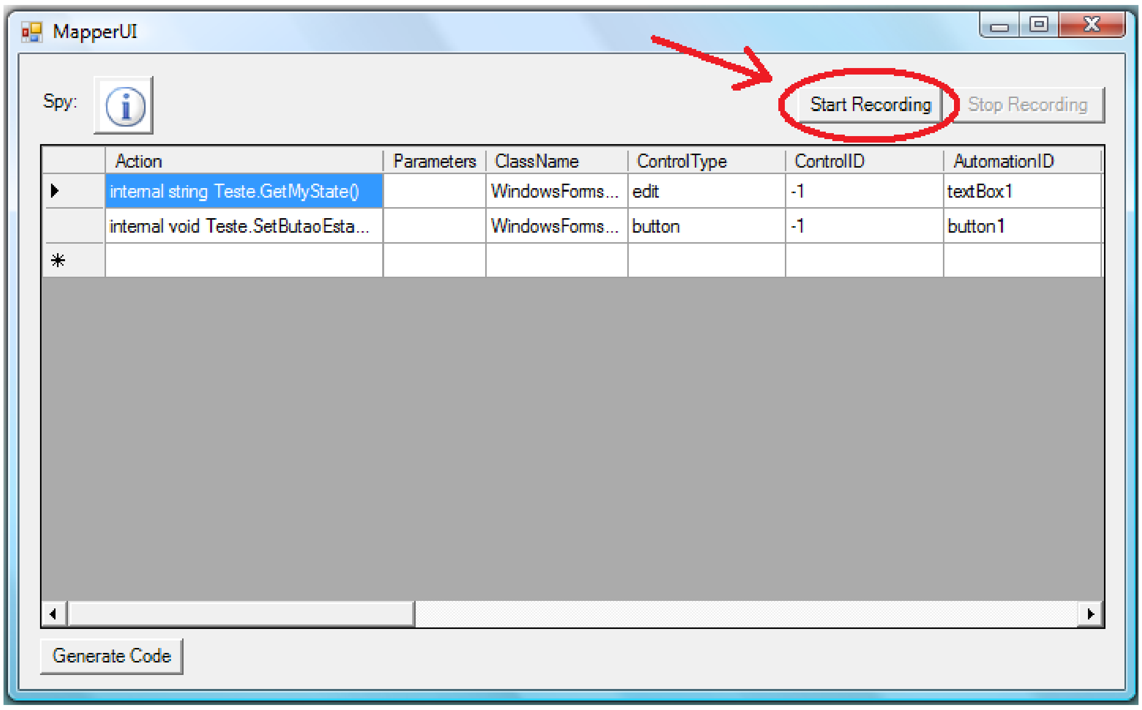
4.6 Rules for mapping abstract model actions into concrete user actions

The main task on the code generation process is to decide which functions of the GUI User Actions Library to call for each model action. In order to perform those decisions, a set of rules to map model actions to user actions must be defined. These rules are based on the signature of the model actions (return type, prefix, name and parameters) and on the type of the physical GUI objects associated with them. The type of a GUI object can easily be known by checking the ControlType property that is accessible with UI Automation. This property is retrieved by the mapping tool when a model action is related to a GUI object.

Some of the rules are:

- Model action with prefix “Clk”: when the model action has the prefix “Clk”, it is assumed that it is modeling a user clicking on the object. So, for instance, if the object is a button, it will be clicked, if it is a check box, it will be checked/unchecked;
- Model action with prefix “Ins”: when the model action has the prefix “Ins” and the GUI object is, for instance, a Text Box, a Masked Text Box or an editable Combo Box, the model action is modeling a user inserting text. In this situation, the model action must have a string parameter that will have the text to be inserted. It is not possible to perform text insertion on a button;
- Model action with prefix “Del”: when the model action has the prefix “Del” and is related with TextBox, a Masked TextBox or an editable Combo Box, the model action is modeling a user deleting some text on the GUI object. In this situation, the model action must have two integer parameters indicating the indexes of the first and last letters of the text that will be deleted;
- Model action with prefix “Sel”: when the model action has the prefix “Sel” and the GUI object is, for instance, a List Box or a Combo Box, the model action is modeling a user selecting one of the options of the GUI object (select one of the lines on a List Box or Combo Box). In this situation, the model action must have an integer parameter indicating the index (or line) selected by the user in the GUI object. That index must be valid, i.e., below the number of lines of the list;
- Model action with prefix “Get”: when the model action has the prefix “Get” it means that it is a probe action. If the corresponding GUI object is a Label, the model action is modeling a user reading the text in the Label. If the GUI object is a Check Box, the model action is modeling a user reading if the Check Box is checked or unchecked;
- Model action with prefix “Cmp”: when the model action has the prefix “Cmp” it means that it is an action that is going to be mapped to several user actions. An example of this can be the drag and drop or a user authentication (insert username and password in a dialog).

These are the prefixes that the tool currently supports and the rules that the tool currently implements. Following this line of though, more rules can be easily created and added.



Action	Parameters	ClassName	ControlType	ControlID	AutomationID
internal string Teste.GetMyState()		WindowsForms...	edit	-1	textBox1
internal void Teste.SetButaoEsta...		WindowsForms...	button	-1	button1

Figure 4.2: GUI mapping tool front-end with the button to start the capture/replay mechanism highlighted

4.7 Code generation

For each model action, a C# function with the same signature is created in a code file and bound to the model action in the Spec Explorer. That function in turn calls the function of the GUI User Actions Library chosen by the mapping rules defined before.

4.8 One-to-many relationships between model actions and concrete actions

To map a single model action to multiple user actions, the mapping tool uses a capture/replay mechanism. The user selects the action on the mapping tool and then clicks on the Start Recording button indicated on Figure 4.2.

When the button is clicked, all user interaction with the GUI application is recorded until the user clicks on the Stop Recording button. On the code generation process, the mapping tool creates a function on the intermediary code that will then map to the model action like it does with all other actions, but this function executes all the user actions recorded previously.

4.9 Case study

On this section it is presented a case study that shows the use of the GUI mapping tool with a Windows Forms application. To better illustrate the context of its usage and the importance of the tool, the case study goes through the entire model-based testing process with Spec Explorer.

4.9.1 Description of the application

The application for this case study is a very simple application that has two switches. A switch is a button that when clicked will change the state of a corresponding text box. The text boxes of the two switches have two possible states each: true and false. When pressed, a switch changes the state of the corresponding text box from true to false or from false to true.

The user can not interact with the text boxes directly. The only way to change their content is through the switches.

4.9.2 The model and implementation

The first step of the model-based testing process is building the model of the application we want to test. The model is built using the guidelines and naming conventions previously suggested on this chapter.

A possible model of the application described before is:

```
namespace Switcher;

bool myState1 = true;
bool myState2 = false;

[Action]
void ClkButtonState1() {
  myState1 = !myState1;
}

[Action]
void ClkButtonState2() {
  myState2 = !myState2;
}

[Action]
string GetMyState1 () {
  if(myState1) {
    return true;
  }
  else {
    return false;
  }
}

[Action]
string GetMyState2 () {
  if(myState2) {
    return true;
  }
  else {
    return false;
  }
}
```

There are two actions that model a user clicking on the two buttons of the application and other two actions that model the user checking the state of both text boxes.

The actions for checking the state of the text boxes (GetMyStateX()) are the probe actions, and they will be used only in conformance checking to check if the implementation conforms to the specification. These actions will be connected to methods on the intermediary code that will return the state of the text boxes. On the testing execution phase, every time the state of the model and of the implementation changes, the probe actions are executed and their results compared to the ones of the concrete methods to which they are connected. If they always match then the implementation conforms to the specification.

Figure 4.3 shows the GUI of the implementation. It is a simple Windows Forms application with two buttons and two text boxes.

4.9.3 Mapping model actions with concrete ones

Now that the model and the implementation are built, we may start mapping the abstract model actions to concrete GUI elements. To do that, select the “GUI Mapper” option on the “Tools” menu (figure 4.4) and the GUI mapping tool front-end appears.

Figure 4.5 shows the state of the GUI mapping tool when all actions are mapped.

When the mapping is done, the intermediate code may be generated by clicking the “Generate Code” button. The intermediate code file is saved on the folder of the Spec Explorer project. This code needs to be compiled into a .Net Assembly and a reference

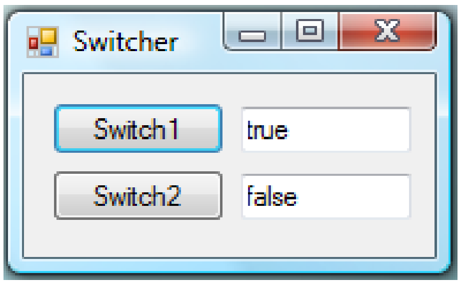
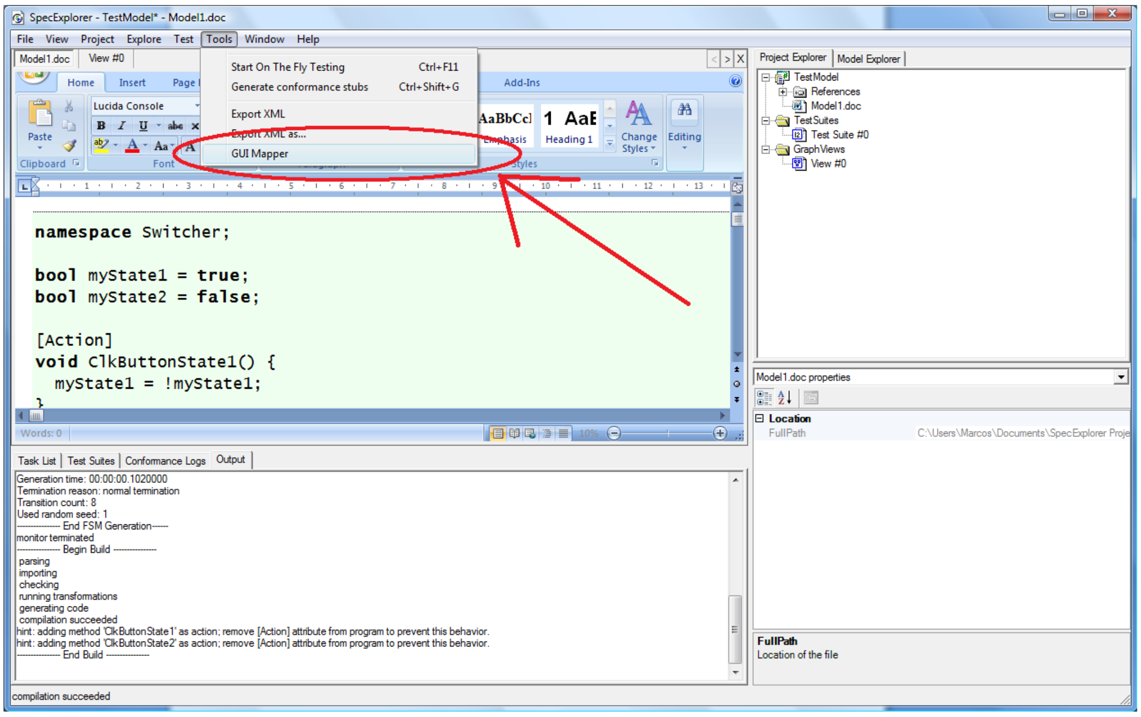


Figure 4.3: GUI of the Switcher application

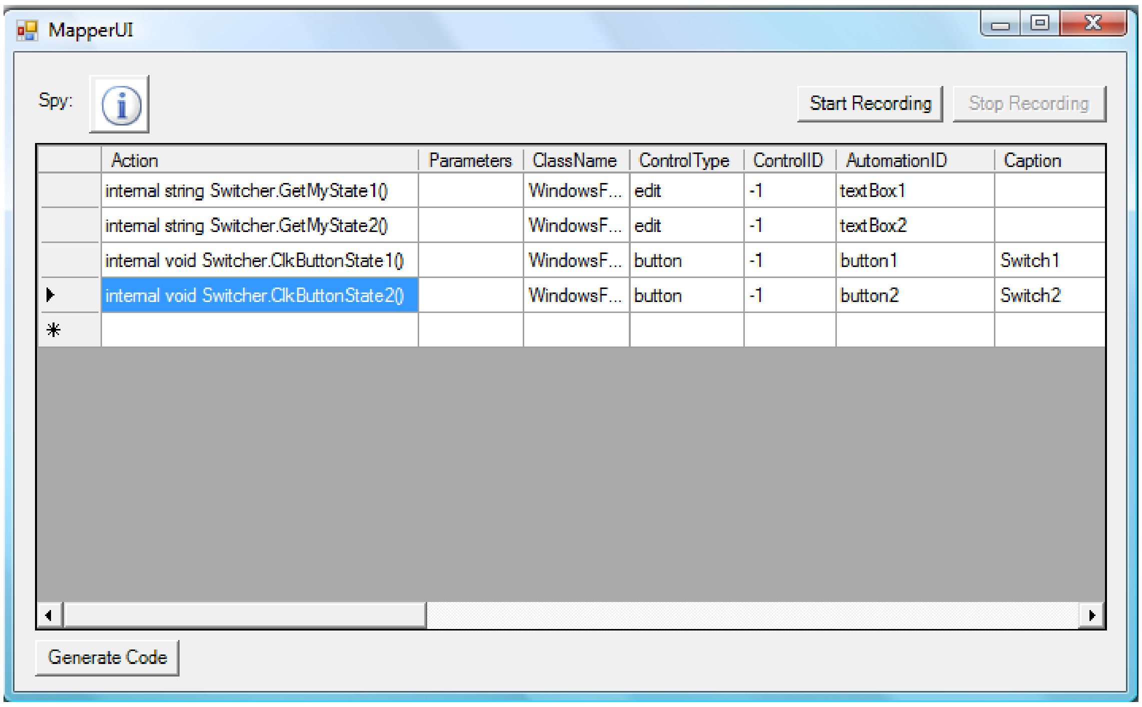


```
namespace Switcher;

bool myState1 = true;
bool myState2 = false;

[Action]
void ClkButtonState1() {
  myState1 = !myState1;
}
```

```
Generation time: 00:00:00.1020000
Termination reason: normal termination
Transition count: 8
Used random seed: 1
--------------- End FSM Generation------
monitor terminated
--------------- Begin Build ---------------
 parsing
 importing
 checking
 running transformations
 generating code
 compilation succeeded
hint: adding method 'ClkButtonState1' as action; remove [Action] attribute from program to prevent this behavior.
hint: adding method 'ClkButtonState2' as action; remove [Action] attribute from program to prevent this behavior.
--------------- End Build ---------------
```

compilation succeeded

Figure 4.4: Spec Explorer - opening the GUI mapping tool



	Action	Parameters	ClassName	ControlType	ControlID	AutomationID	Caption
	internal string Switcher.GetMyState1()		WindowsF...	edit	-1	textBox1	
	internal string Switcher.GetMyState2()		WindowsF...	edit	-1	textBox2	
	internal void Switcher.ClkButtonState1()		WindowsF...	button	-1	button1	Switch1
▸	internal void Switcher.ClkButtonState2()		WindowsF...	button	-1	button2	Switch2
*							

Figure 4.5: GUI mapping tool with mapping information

to this Assembly must be added on the Spec Explorer project. The next step is generating the FSM and the test cases. Figure 4.6 shows the FSM generated from the model and Figure 4.7 shows a description on the Spec Explorer of the test cases generated.

4.9.4 Results

The final step is the execution of the test cases. In order to do that the application under test must be opened. Figure 4.8 shows the results of executing the test cases generated previously as they appear on Spec Explorer. If the previous steps described in the case study were performed correctly, all the tests on the test cases generated shall pass.

4.10 Conclusions

On this chapter the implementation of the new GUI Mapping Tool was presented. Tests made during the development of the tool showed that the tool was able to detect and handle correctly the following controls on applications built using Win32, Windows Forms, WPF, Java AWT and Web platforms:

- Buttons;
- Labels;

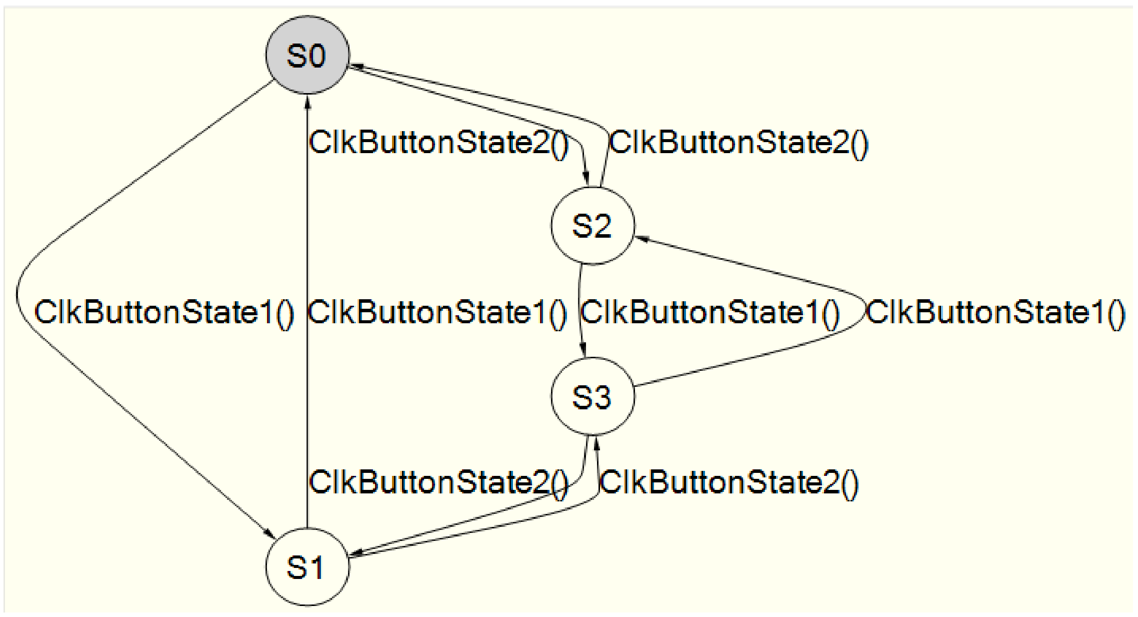
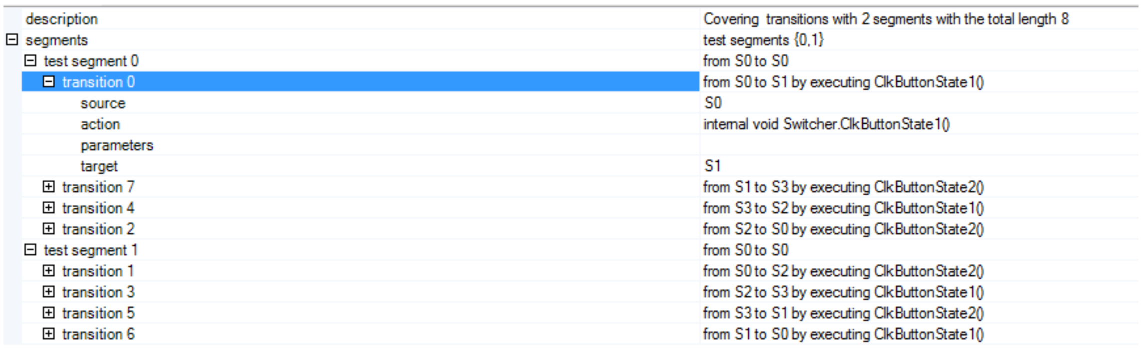
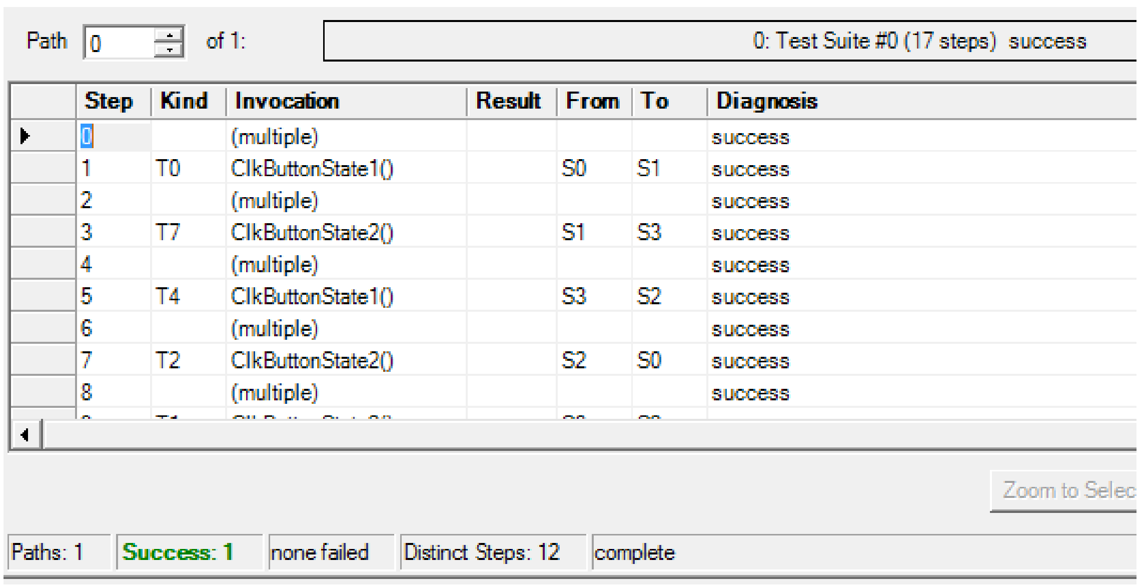


Figure 4.6: Finite State Machine generated from the Switcher model



description	Covering transitions with 2 segments with the total length 8
⊟ segments	test segments {0,1}
⊟ test segment 0	from S0 to S0
⊟ transition 0	from S0 to S1 by executing ClkButtonState1()
source	S0
action	internal void Switcher.ClkButtonState1()
parameters	
target	S1
⊞ transition 7	from S1 to S3 by executing ClkButtonState2()
⊞ transition 4	from S3 to S2 by executing ClkButtonState1()
⊞ transition 2	from S2 to S0 by executing ClkButtonState2()
⊟ test segment 1	from S0 to S0
⊞ transition 1	from S0 to S2 by executing ClkButtonState2()
⊞ transition 3	from S2 to S3 by executing ClkButtonState1()
⊞ transition 5	from S3 to S1 by executing ClkButtonState2()
⊞ transition 6	from S1 to S0 by executing ClkButtonState1()

Figure 4.7: Test Cases generated by the Spec Explorer from the Switcher model



Path 0 of 1: 0: Test Suite #0 (17 steps) success

	Step	Kind	Invocation	Result	From	To	Diagnosis
▶	0		(multiple)				success
	1	T0	ClkButtonState1()		S0	S1	success
	2		(multiple)				success
	3	T7	ClkButtonState2()		S1	S3	success
	4		(multiple)				success
	5	T4	ClkButtonState1()		S3	S2	success
	6		(multiple)				success
	7	T2	ClkButtonState2()		S2	S0	success
	8		(multiple)				success

Zoom to Selec

Paths: 1 | Success: 1 | none failed | Distinct Steps: 12 | complete

Figure 4.8: Results of the execution of the test cases

- Text Boxes;
- Check Boxes;
- Radio Buttons.

It is able to detect controls of those types, identify their type and interact with them with no problems detected.

The current tool only supports these five control types but as already said on previous chapters, UI Automation provides support for all native controls of the GUI platforms it supports, which means that the tool can very easily be extended to support more controls.

Chapter 5

Conclusions and Future Work

The research conducted on this work about the State of The Art of GUI testing shows that there is not a technique or tool that fits all testing purposes. Also, there is not a GUI testing framework which supports all GUI platforms. This last conclusion implies that a complete solution to the problem of this dissertation requires the choice of more than one GUI testing framework. The choice of a GUI testing framework is very important because it provides a programmatic way of interacting with GUIs and the framework we choose will influence our possibilities of providing good solutions to some of the problems approached on this work like, for example, the support for multiple GUI platforms. The solution suggested on this dissertation uses only the UI Automation framework which means that Java Swing applications are not supported, but since the Abbot framework (GUI testing framework which supports only Java GUI platforms) provides similar functionalities to the ones found on UI Automation, extending this solution to support Java Swing applications is possible but it raises some extra architectural questions because we cannot use Abbot from a C# application.

5.1 Main contributions of this work

The main contributions of this work are:

- A new GUI mapping tool that is easier to develop, extend and that supports a bigger set of applications than the previous one;
- A solution to the problem of mapping a single abstract model action to multiple concrete user actions on the GUI under test;
- A well defined set of rules for the automatic generation of the intermediary code by the GUI mapping tool.

5.2 Satisfaction of Goals

All the goals proposed for this dissertation where satisfied except the support for Java Swing applications on the new developed GUI Mapping Tool. Nevertheless, the solution proposed on chapter 3 can be extended to support Java Swing applications because Abbot provides similar functionalities to the ones found on UI Automation.

5.3 Future Work

The tool developed on this work can be improved:

- It can be extended to support a bigger set of GUI controls. UI Automation is able to recognize all standard controls of Win32, Windows Forms, WFC, Java AWT and Web platforms so the Mapping Tool can at least be extended to that point;
- Support to Java Swing applications can be added;
- The one-to-many relationship between model actions and user actions suggested on this dissertation do not solve all the problems of complex user interactions. Much work on this area is still needed;
- The construction of the formal model of the application is hard and still takes much time. It would be interesting to develop tools which aid on the creation of these models.

References

[Bur02] Ilene Burnstein. *Practical Software Testing*. Springer, 2002.

[ea05] M. Barnett et all. The spec programming system: Challenges and directions. *Proceedings of the VSTTE2005*, 2005.

[FLM$^+$05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2005.

[Gro] Object Management Group. Object management group - uml. http://www.uml.org/.

[HP] HP. Hp winrunner software - hp - bto software. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1074_4000_100__.

[JF03] Allen Jones and Adam Freeman. *C for Java developers*. Redmond, WA : Microsoft Press, 2003.

[Mic] Sun Microsystems. Trail: Creating a gui with jfc/swing. http://java.sun.com/docs/books/tutorial/uiswing/.

[MPS99] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for guis. *Proceedings of the Int. Conf. on Software Engineering, Los Angeles*, 1999.

[MSD] Microsoft MSDN. Microsoft ui automation. http://msdn.microsoft.com/en-us/library/ms747327.aspx.

[NMo] NModel. Nmodel - home. http://www.codeplex.com/NModel/.

[PFTV05] Ana C. R. Paiva, João C. P. Faria, Nikolai Tillmann, and Raul F. A. M. Vidal. A model-to-implementation mapping tool for automated model-based gui testing. *Proceedings of ICFEM'05*, November 2005.

[Resa] Microsoft Research. Spec explorer project. http://research.microsoft.com/en-us/projects/SpecExplorer/.

[Resb] Microsoft Research. Spec# project. http://research.microsoft.com/en-us/projects/specsharp/.

[Sch06] Douglas C. Schmidt. Model driven engineering. *IEEE Computer Society*, February 2006.

[SV05] Alberto Silva and Carlos Videira. *UML - Metodologias e Ferramentas CASE*. Morgan-Kaufmann, second edition, Maio 2005.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[VDM] VDM. Vdm information web site. http://www.vdmtools.jp/en/index.php.

[W3C] W3C. Extensible markup language (xml). http://www.w3.org/XML/.

[Wal] Timothy Wall. Abbot framework for automated testing of java gui components and programs. http://abbot.sourceforge.net/doc/overview.shtml.

[WSW03] Andy Wigley, Mark Sutton, and Stephen Wheelwright. *Microsoft .NET compact framework*. Redmond, WA : Microsoft Press, 2003.