A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing

Ana C. R. Paiva[1], João C. P. Faria[1,2], Nikolai Tillmann[3], Raul F. A. M. Vidal[1]

[1] Faculty of Engineering of the University of Porto, [2]INESC Porto,
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
{apaiva, jpf, rmvidal}@fe.up.pt
[3] Microsoft Research, One Microsoft Way
Redmond, WA 98052, USA
nikolait@microsoft.com

Abstract. This paper presents extensions to the Spec Explorer tool to automate the testing of software applications through their GUI based on a formal specification in Spec#. The Spec Explorer tool, developed at Microsoft Research, already supports the automatic generation and execution of test cases for API testing, but requires that the actions described in the model are bound to methods in a .Net assembly. The tool described in this paper extends Spec Explorer to automate GUI testing, by adding the capability to gather information about the physical GUI objects that are the target of the user actions described in the model, and automatically generate a .Net assembly with methods that simulate those actions upon the GUI application under test. It is described the overall GUI modelling and test process with these tools. The approach is illustrated with the Notepad application.

1 Introduction

Today's software systems usually feature Graphical User Interfaces (GUIs). GUIs have become an important and accepted way of interacting with today's software. This growing implantation of GUI software applications turns us more dependent on their correct functioning. However, testing of GUIs is difficult with very few tools and techniques available to aid in the testing process. Currently used GUI testing methods are almost ad hoc and require the test designer to manually develop test cases, identify the conditions to check during test execution, determine when to check these conditions, and evaluate whether the GUI software is adequately tested.

There have been efforts to automate the GUI testing process. Some tools, called Capture/Replay tools (http://www.testingfaqs.org/t-gui.html), are commercially available. They can be used to record user interactions and replay them later. However, they do not support the automatic generation of test cases.

There are also approaches to automate the generation of test cases. They can be generated from code but, in this case, testing is postponed until later phases of the software development process and expected outputs have still to be supplied by the user. A more effective approach can be generically named specification-based (or

model-based) testing. In this case, a GUI model has to be constructed and, depending on the nature of the model, different techniques can be used to generate test cases [1] in order to verify the conformity between an implementation and the specification. A formal model allows the automatic generation of expected outputs, besides test inputs. The process of writing a specification can also be useful to find inconsistencies and usability problems before the user interface is developed, that can result in time and money savings. Also, the construction of models enables the analysis of alternative designs without having to code them.

Spec Explorer (research.microsoft.com/SpecExplorer), from Microsoft Research, is an example of an advanced model-based conformance testing tool. It has the advantage of fully automating the generation and execution of test cases from an annotated model, and it provides a good integration between specification and implementation levels. However, when used to test GUIs, it still requires much user intervention to map the user actions described in the model to real actions upon the GUI under test.

The *main contribution* of this paper is the GUI mapping tool:

- It reduces the manual work required to test an application through its GUI.
- It bridges the gap between a model written in a high-level modelling language and the simulation of user events.
- It promotes a modelling pattern in which GUI components can be specified as reusable classes.

The tool has been developed as an extension to the Spec Explorer tool.

For comprehensiveness, the paper is organized along the activities of the test process: the next section presents an overview of the GUI test process supported by Spec Explorer and the new GUI mapping tool; section 3 explains how GUIs can be modelled adequately with Spec#; section 4 describes how test cases can be generated automatically with Spec Explorer; the main contributions are in section 5, where the new GUI mapping tool is described; and section 6 describes test execution. Related work is discussed in section 7 and the last section summarizes the results achieved and points out future work. The Notepad application is used as a running example to illustrate the approach.

2 Overview of the model-based GUI testing process

The goal of model-based testing is to check if an implementation of a software system conforms to the specification (or model) of that system. The specification captures the requirements and the conformity tests check if those requirements are fulfilled by the implementation. Given an executable implementation and a specification of a software system, the generic activities involved in model-based testing are test case generation (from the specification), test case execution, and comparison of the actual results obtained from the implementation with the expected results derived from the specification (which plays the role of a test oracle). To generate automatically the test cases and expected results from the specification, a formal specification is required. In case the formal specification is also executable (as is the case of Spec#) the expected results are obtained by executing the specification with the test inputs.

2.1 Automated model-based testing with Spec Explorer

Spec Explorer [2] is a software modelling and testing tool from Microsoft Research. A formal executable model can be written in the abstract state machine language (AsmL) (research.microsoft.com/fse/AsmL) or Spec# [3], which is a superset of C#.

Some of the methods in the specification are annotated as actions that represent possible transitions of a transition system. These actions can have pre-conditions, written as “requires” clause, that define the states in which actions are enabled.

From the specification, a Finite State Machine (FSM) is derived, from which in turn test cases (sequences of actions with actual input parameters) are automatically generated.

Conformance between the model and an implementation can be established by binding the model actions to implementation methods, executing the test suites on both the model and the implementation, and comparing their results. The implementation can be written in any language supported by the .NET framework.

Spec Explorer runs the test cases in an interleaved way, that is, it executes each test case action on the implementation and the specification, and compares the results obtained after each execution step.

Spec Explorer also supports on-the-fly testing. In this case, the test generation and test execution are connected into a single algorithm [4].

2.2 Automated model-based testing of GUI applications with Spec Explorer and the GUI mapping tool

Fig. 1 presents the main activities and artefacts involved in testing GUI applications with Spec Explorer extended with our GUI mapping tool.

As already mentioned, to perform conformance tests with Spec Explorer, a binding or mapping between the model actions and implementation methods in a .Net assembly must be provided. When the implementation is a .Net application, the mapping can be easily established since the model is written in a .Net language as well. For APIs exposed by other means, some glue code might be needed to map forth and back the data and method calls. However, when the application’s functionality is only exposed through its GUI, then the application must be driven through the GUI’s abstraction layer, by simulating the actions of a user interacting with it. That is the role of the GUI mapping code in Fig. 1.

In previous experiences of using Spec Explorer to model and test GUI applications [5], the authors realised that, even in the case of simple applications such as the Notepad application, the manual building of the GUI mapping code was unpractical and required too much effort. To solve that problem, a GUI mapping tool was developed and integrated with Spec Explorer.

The GUI mapping tool assists the user in relating the model actions ("logical" actions) to "physical" actions on "physical" GUI objects. A major difficulty that is solved by the tool is the identification of the GUI physical objects that the model ac-

tions refer to. The mapping code is automatically generated from high-level mapping information. Further information about this tool will be provided in section 5.

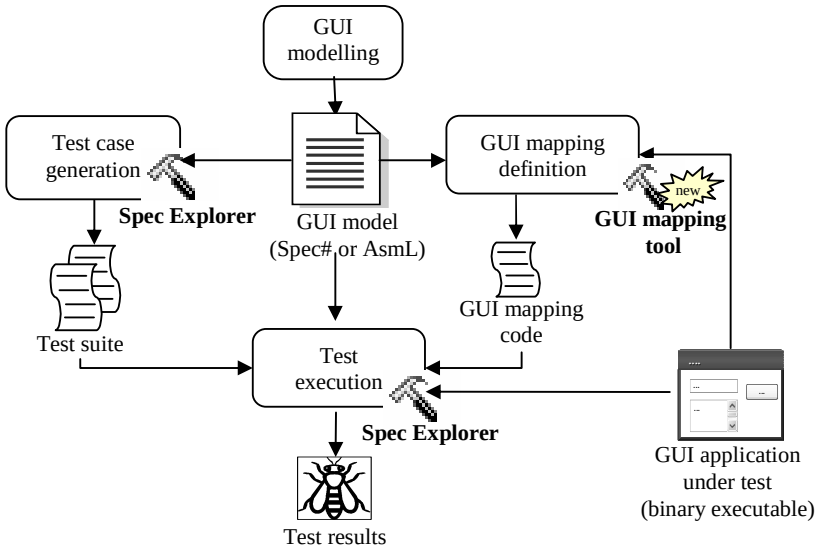

Fig. 1. Overview of the GUI modelling and testing process

3 GUI modelling with Spec# and Spec Explorer

The behaviour of GUIs can be modelled by state machines, with transitions triggered by user actions. State machines can be useful to guide the testing of software systems [6], although they suffer from the state explosion problem.

The way of modelling with Spec Explorer has been inspired by Abstract State Machines (ASMs). ASMs [7] provide a way to model any system at any level of abstraction. This is adequate for GUI modelling, because, depending on the context, one may want to model user actions at different levels of abstraction: at operating system level (where a click event is the sequence of pressing and releasing the mouse button), at API level (where a click event is seen as an atomic action), at user task level, etc.

Independently of the level of abstraction considered (lower level messages, or higher level messages that correspond to sequences of lower level messages), a GUI implementation places the messages in a queue and processes those messages in order. This behaviour can also be adequately modelled as an ASM with guarded actions which fire only when appropriate messages are fetched from the queue.

A model written in Spec# describes a possibly infinite state transition system. States are modelled by state variables. Some of the methods in the specification are annotated as actions that represent the possible transitions of a transition system. These actions can have pre-conditions, written as “requires” clause, that define the

states in which they are enabled. Thus, actions can be seen as the guarded update rules of an ASM. It is important to note that the states can have a very rich structure. In the case of GUIs, this allows to faithfully model the GUI's state from a user perspective. For example, a state variable can hold the textual content of a field. Methods annotated as actions can be used to model complex user actions (enter a string into a field, issue a command, loading content from a file, etc.) and describe its effect on the state of the system.

A simplified excerpt of a Spec# model of the Notepad application is shown in Spec. 1. A complete model can be found in www.fe.up.pt/~apaiva/MyNotepad.pdf.

```
namespace MyNotepad;

//State variables
string text="",selText=""; bool dirty=false; int posCursor=0;

// Start and close the Notepad application
[Action] void LaunchNotepad()
requires !IsOpen("Notepad"); {
  AddWindow("Notepad","",false);
  //... state variables initialization ...
}
[Action] void Close()
requires IsEnabled("Notepad"); {
  if (dirty) AddWindow("MsgClose","Notepad",true);
  else { RemoveWindow("Notepad");
           //... reset variables to initial values
       }
}
[Action] void MsgSaveChangesBeforeClose(string option)
requires IsEnabled("MsgClose"); {
  RemoveWindow("MsgClose");
  switch (option){
    case "No": RemoveWindow("Notepad"); return;
    case "Yes": AddWindow("Save"); return;
    case "Cancel": return;
    default: return;
  }
}
// change and query the content of the main window
[Action] string GetText()
requires isEnabled("Notepad"); { return text;
}
[Action] void InsText(string typedTxt)
requires IsEnabled("Notepad"); {
  text = text.Substring(0,posCursor-selTxt.Length) + typedTxt
       + text.Substring(posCursor,text.Length-posCursor);
  posCursor = posCursor – selTxt.Length+typedTxt.Length;
  selTxt = ""; dirty = true;
}
```

Spec. 1.: Excerpt of a Spec# model of the Notepad application

The excerpt shown also illustrates some techniques that are useful in modelling GUIs with Spec#.

For modularity reasons, except for trivial applications, the top-level windows of the application are better modelled in separate namespaces or classes. In the example shown, the `MyNotepad` namespace refers to the main window of the Notepad appli-

cation. The complete model has other namespaces corresponding to the Save, Open, Find and Replace dialog windows.

Inside each module (namespace or class) corresponding to a top-level window, state variables are used to model the abstract state of that window, and methods annotated as actions are used to model the possible user actions on that window. To enable conformance testing of the outputs displayed to the user, methods annotated as actions should also be provided to observe the state of the GUI that is exposed to the users' eyes. A query method can be provided for each observable state variable, with the name of the variable and a suitable prefix (e.g. `GetText` in the example above). Spec Explorer allows designating such actions as *probes*. A probe only observes the current state and does not change it. Probes are treated differently from ordinary actions during test case generation, as we will see later.

All the actions inside each module, except the one that launches the application, have at least one pre-condition: that the corresponding window is enabled. A window is enabled when it is open and doesn't have a child modal window on top. When a modal window is open (e.g. the Save and Open windows in the Notepad application), the other windows of the application are disabled. Since this is a common feature of GUIs, a separate reusable module – a window manager – was created to handle it.

The window manager provides the following self-explanatory helper methods:

```
AddWindow(windowName, parentWindowName, isModal)
RemoveWindow(windowName)
IsEnabled(windowName)
IsOpen(windowName)
```

When a method opens/closes a window it should add/remove that window to/from the window manager. When a window is removed, all its child windows are also removed. Message boxes are also registered in the window manager but need not be modelled as separate modules (e.g. `MsgClose` in the example above).

The window manager is part of the model, and its state is part of the model state.

4 Test case generation with Spec Explorer

Spec Explorer automatically generates test cases in two steps (Fig. 2) from a Spec# or AsmL specification. In the first step, a FSM is generated from a Spec# or AsmL specification. In the second step, test cases that fulfill coverage criteria are generated from the FSM.

The FSM is generated by bounded exploration of the state space of the model. Some techniques available to prune the exploration are:

1. state filters – boolean expressions that determine which states to explore;
2. restriction of the domains – the domains of the parameters are bounded to a finite set of possible values;
3. equivalence classes – this technique partitions states into equivalence classes and prevents further exploration from any state of such a class once a specified number of representatives has been reached;
4. reduction based on hierarchical structure – this technique was developed by the authors to reduce the size of the FSM obtained from a GUI model [5]. The FSM is

organized in a hierarchical model and that structure is the input to the FSM reduction algorithm. In our experiences, a reduction by around 50% of the number of states can usually be achieved.

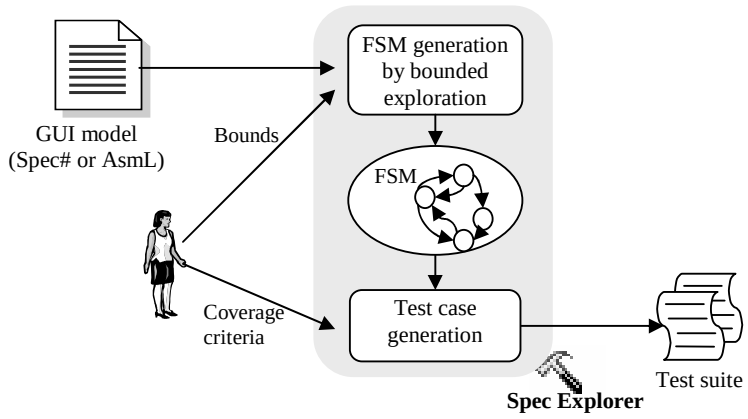

Fig. 2. Test case generation

The pruning of exploration becomes crucial when talking about modelling and testing GUIs. That's because testing an application through its GUI by simulating user events entails a significant overhead resulting in much slower test execution than testing an application through its API. The main challenge is to generate a test suit of manageable size while sill guaranteeing adequate testing.

As soon as the FSM is constructed, and the coverage criteria chosen, a traversal engine is used to unwind the resulting FSM to produce behavioural tests that fulfil the coverage criteria. The coverage criteria can be set to full transition coverage, shortest path to a set of user-defined states, or a random walk. Actions designated as probes are checked in every state of the resulting tests, and do not take part in coverage considerations.

In the Notepad example, the full transition coverage criterion was used to generate the test cases. Other coverage criteria, more adapted to GUI testing [8], can be easily added to the Spec Explorer tool through its API.

5 Model-to-implementation mapping with the GUI mapping tool

The aim of the GUI mapping tool is to reduce the manual work involved in model-based testing of software applications through their GUI.

As already mentioned in the overview, the GUI mapping tool assists the user in relating the logical actions described in the model to physical actions on physical GUI objects of the application under test (AUT).

The GUI mapping tool (Fig. 3) has a front-end (Fig. 4) that shows the mapping information gathered so far and gives access to the GUI Spy tool and the GUI Mapping Code Generator. The spy tool is used to get information about physical GUI objects in

the AUT, in a way similar to the Spy++ tool that ships with Microsoft Visual Studio. The code generator exports to XML files and C# the mapping information gathered. The C# code generated is based on calls to a reusable GUI Test Library. Further details will be provided in the next sections.

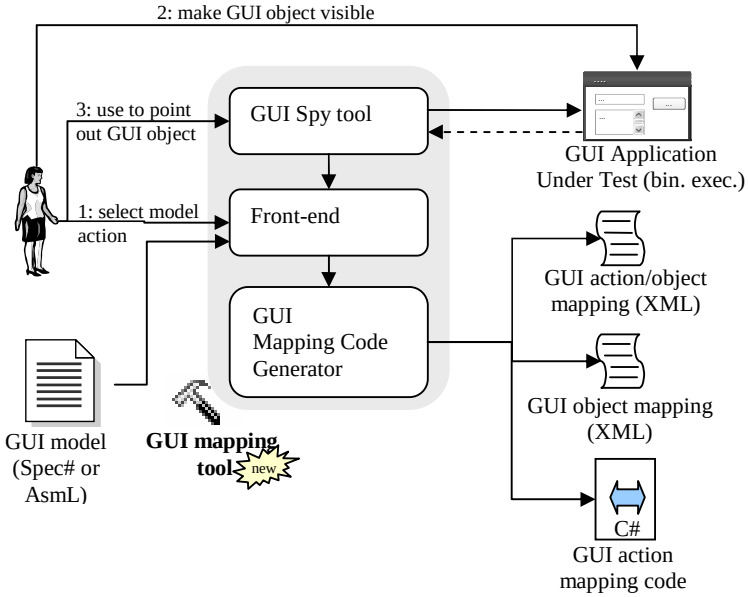

Fig. 3. Architecture of the GUI mapping tool

5.1 The GUI Spy tool

The GUI Spy Tool is accessible from the font-end of the GUI mapping tool (see Fig. 4). It allows the user to point out the physical GUI object that is the target of each logical action specified in the model.

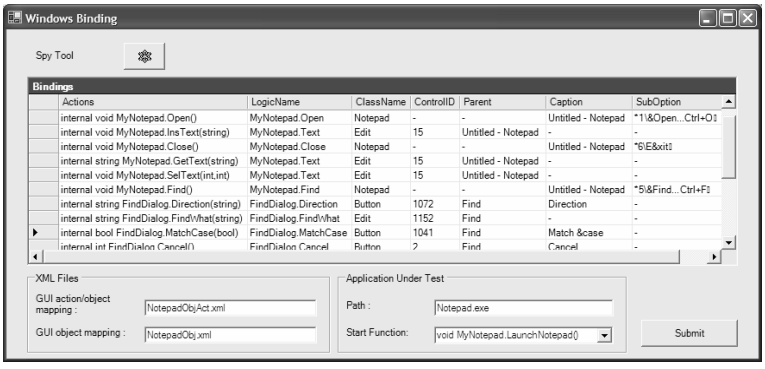

Fig. 4. Front-end of the GUI mapping tool

After selecting the logical action in the main grid (first column), the user drags and drops the spy icon on top of the corresponding physical GUI object in the AUT. If the desired GUI object is not visible, the user will have to interact also with the AUT in order to make it visible. The physical properties of the GUI object selected, as well as a logical name inferred by the tool (as will be explained in the next section), are then displayed in the grid (see Fig. 4).

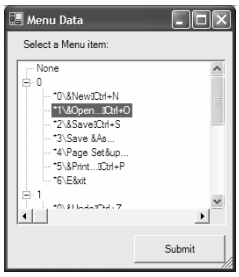

Fig. 5. Selection of menu options

The Spy++ tool that ships with Microsoft Visual Studio can only gather information about proper windows (or GUI objects with a window handle). Our tool goes a bit further: it can also gather information about window menus. So, when the tester wants to establish a relation between a specification method and an item inside a menu, he can drag and drop the mouse on top of the window that contains the menu at which time another window (at Fig. 5) is opened with all the submenu options, allowing him to choose the submenu option he wants. A similar option exists for controls like tab pages and toolboxes.

5.2 Logical names of GUI objects

Every physical GUI object is associated with a logical name. This keeps independence between specification and implementation levels and allows the generation of code more readable and easier to construct manually, if desired.

Default logical names are automatically generated by the tool. The logical name is equal to the namespace name followed by the name of the specification method without prefix (`Set`, `Get`, etc.). In order to obtain the same logical name for all the logical actions with the same target physical object, it is desirable that the names of those actions are constructed with a different prefix and the same suffix.

5.3 XML files generated

The mapping information captured is saved into two XML text files:

1. a file with the mapping between model actions and the logical names of the target GUI objects (GUI action/object mapping file in Fig. 3);

```
<Action id="internal void MyNotepad.Open()">
  <LogicalName>MyNotepad.Open</LogicalName>
</Action>
```

2. a file with the mapping between logical names and physical properties of GUI objects (GUI object mapping file in Fig. 3).

```
<GUIObject logicalName="MyNotepad.Open">
  <ClassName>Notepad</ClassName>
  <Caption>Untitled - Notepad</Caption>
  <SubClassName>menu</SubClassName>
  <SubOption>&Open...Ctrl+O</SubOption>
</GUIObject>
```

The mapping information needs to be gathered just once for each application. But if the specification is changed and the mapping information has to be updated, the XML files can be loaded by the GUI mapping tool for update. The XML files can also be changed directly by the user.

These XML files are also used for code generation and test execution as is explained in subsequent sections.

5.4 GUI Test Library

The C# code generated is based on calls to a reusable GUI test library that provides methods to simulate the actions of a user interacting with a GUI application and observe the content of GUI objects. This library was constructed in C# extending a previous existing library to best fit our needs.

The GUI test library provides three kinds of methods (Code. 1):

- methods that act upon GUI objects simulating the user, like sending text to a control that accepts text input (`SendText`). The target GUI object is identified by its logical name. Each method may have additional parameters with information needed to perform the action.
- methods that observe properties of GUI objects, like the text (`GetText`), insertion point (`GetInsertionPoint`), and selected text (`GetSelectedText`) of a text box. The target GUI object is also identified by its logical name. The return value conveys the information requested.
- methods that provide physical information about GUI objects identified by their logical names in order to identify those objects in the real AUT. This information may be loaded from a XML file.

```
// To act upon GUI objects
void Click(string GUIObjName);
void SendText(string GUIObjName, string txt);
void SelectText(string GUIObjName, int start, int end);
void SelectSubOption(string GUIObjName, string option);
void SelectCheckBox(string GUIObjName, bool check);
void SelectMsgBoxOp(string GUIObjName, string option);
// To observe properties of GUI objects
string GetText(string GUIObjName);
string GetSelectedText(string GUIObjName);
int GetInsertionPoint(string GUIObjName);
bool GetCheckBox(string GUIObjName);
// To map logical object names to physical objects
void LoadXMLObjMapping(string XMLFileName);
```

Code. 1.: Examples of methods implemented in the GUI test library

5.5 Rules for mapping logical actions into physical actions

Besides identifying the physical GUI object that is the target of each model action, it is also necessary to select the appropriate method from the GUI test library, which will simulate a physical action of the user on that GUI object.

The GUI mapping tool automatically infers the appropriate library method based on the type of the GUI object, and the signature of the model action.

Some of the rules that are applied are:

- When the sub option is filled in the mapping information, we assume that the logical action is modelling the action of a user selecting a sub menu option, a tab option or a tool button inside a toolbox (`SelectSubOption` method in the test library). This is the case of actions `Open`, `Close` and `Find` in Fig. 4.
- When a logical action has a string parameter and is mapped to a textbox, we assume that the action is modelling an event that sends text (`SendText` method in the test library). This is the case of actions `InsText` and `FindWhat` in Fig. 4.
- When the logical action is an inspection method, has a string as return value and is mapped to a textbox, we assume that it is modelling the eyes of the user looking at the content of the textbox, thereby retrieving the text (`GetText` method in the test library). This is the case of action `GetText` in Fig. 4.
- When the logical action has neither parameters nor return values, and is mapped to a button, we assume that physical action is to click the button (`Click` method in the test library). This is the case of action `Cancel` in Fig. 4.

5.6 Code generation

Spec Explorer requires that the actions in the model are bound to implementation methods (in a .Net assembly) with identical signatures (identical return type, number of parameters, and parameters' types). To fulfil this requirement, the tool generates C# code with methods with the same signature as the model actions, as illustrated in Code. 2. For each logical action, it is generated a method with the same signature, calling the method of the GUI Test Library inferred according to the rules described before, with the logical name of the target GUI object as an additional parameter.

```
#region automatically generated code
class GeneratedCode{
    public static void LaunchNotepad(){
      LoadXMLObjMapping("C:\\temp\\Notepad.xml");
      new App(@"Notepad.exe");
    }
    public static void Open(){
     UserEvents.SelectOption("Notepad.Open");
    }
    public static void InsText(string p0){
     UserEvents.SendText("Notepad.Text",p0);
    }
```

```
        public static string GetText(){
         return UserEvents.GetText("Notepad.Text");
        }
        //...
    }
    #endregion
```

Code. 2. Excerpt of the code generated automatically for the Notepad example

The start function launches the application and reads the mapping information between logical and physical GUI objects from the GUI object mapping file (in Fig. 3).

When executing the test cases only one instance of the AUT should be opened. Otherwise, the tool can pick the wrong window and the test case results can be compromised. To overcome that problem, some code is added manually to the start method (`LaunchNotepad`) to close all windows that were opened by the previous testing trace/path.

6 Test execution

As soon as the mapping code is constructed, compiled into a library and a reference to this library added to the Spec Explorer project, and the test cases are generated, it is possible to execute the test cases autonomously without user intervention.

Let's assume we have a deterministic model. Then, each test case consists of a sequence of steps. For each step, a specification action and its related implementation method are executed in locked step mode (e.g. the `Close()` method in Fig. 6). At the implementation level, each method does a call to a method defined in the generic GUI test library (e.g. `Click()` in Fig. 6) that interacts with the GUI AUT simulating the user actions. The query actions (with the `Get` prefix) get information about interaction objects' properties that are compared with the expected values obtained from the specification. The execution stops when inconsistencies are detected.

In GUI testing, inconsistencies between the specification and the implementation can arise for several reasons:

1. the model is trying to act on a control that is not enabled or cannot be found; or
2. the model is trying to act on a window that is not reachable or is not opened (e.g., a modal dialog is open and the window we want to reach is behind that dialog);
3. the expected result was not displayed (e.g., a text box does not display the expected content).

During the testing of Notepad, we discovered one sequence of actions which leads to an inconsistency between our intuitive model and the actual Notepad application:

1. Type text.
2. Search for text using the find dialog (Ctrl-F). Close the dialog.
3. Open the replace dialog (Ctrl-H). Close the dialog.
4. Press the F3 key (shortcut for "Find Next").

Then Notepad will search backwards instead of forwards. This is a sequence of events that manual test would probably miss since it isn't a common sequence of events.

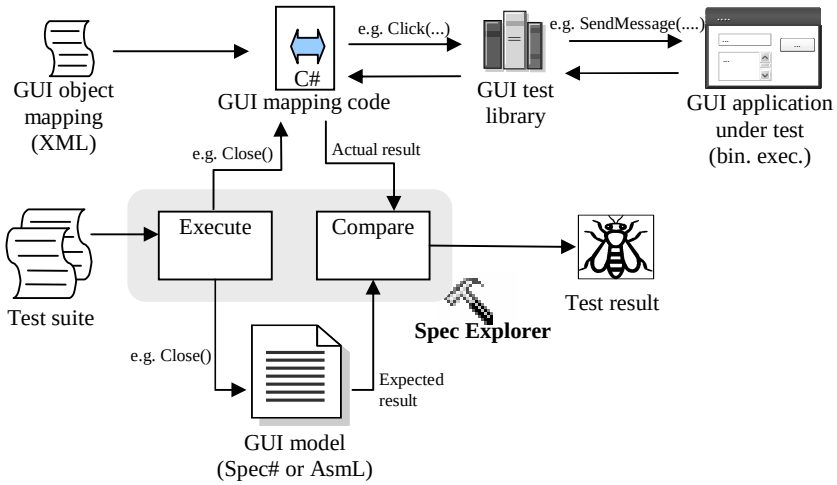

Fig. 6. Test execution

7 Related Work

Today, many tools exist to develop GUI applications visually, but they do provide neither support for specifying or modelling GUIs including their functional behaviour on a higher abstraction level nor for testing them in an effective way. Yet, testing GUIs represents a significant amount of overall testing efforts. To overcome this discrepancy, several kinds of testing tools were developed. These tools can be classified into capture/replay, random input testing, unit testing frameworks, and model-based.

With capture/replay tools, the test script is constructed by the tester interacting with the GUI that records his actions in order to replay them later. This kind of tools shifts the testing activity to the final phases of the software development process since they can be only used when the GUI, or part of the GUI, is already constructed. Also, these tools don't provide any support to evaluate the test script constructed according to coverage criteria. Examples of these tools are WinRunner (www.mercury.com) and Rational Robot (www.ibm.com). Changes to the implementation usually require the re-capturing of all affected test scripts.

The goal of random input testing tools is to crash the system under test. They generate test cases randomly and ignore any unexceptional outputs of the system. This category of tools is not well suited to find the defects related to the incorrect behaviour, but is the most cost-effective for finding defects that crash the system. Panorama C/C++ (www.softwareautomation.com) is an example of this category.

Another possible approach is to program the test cases. Frameworks like JUnit (www.junit.org) and NUnit (www.nunit.org) are of great help in organizing and executing test cases, particularly for API testing, but not in generating those tests. In the case of GUI testing, many bugs can only be uncovered through particular sequences of actions, which might arise in the daily use of the GUI. Unit tests however are usually a

few hand-written sequences of actions, which tend to be very short. Thus, there is a high probability to miss these kinds of errors. Another disadvantage of these tools is the required extra programming effort.

Model-based testing tools can be used to test conformity between an implementation and the specification. A high level of automation can be achieved with these tools since the test case generation, the test case execution, and the comparison of the expected results with actual results can all be automated.

The specification/model is used to generate test cases that fulfil a given coverage criterion. The techniques used to do it depend on the kind of specification used.

Belli, in [1], uses FSMs and regular expressions to model GUIs. He expands the original model with illegal behaviour and generates test cases that can bring the system into legal or into faulty states.

Shehady, in [9], uses Variable Finite State Machines (VFSM) to model GUIs and to cope with FSM scaling problems. VFSMs are FSMs with an added condition associated to each transition. The VFSM is converted into a FSM to generate test cases using the partial W algorithm [10]. The test cases are applied to the GUI and the results obtained are compared with the results expected. The comparison is performed at the end of the test case execution so that, even if the inconsistencies are found at the beginning of the test cases, the execution of an entire case is required.

Memon, in [11], uses a hierarchical structure to model GUIs. He defines a set of operands that correspond to user actions. These low level operands can be combined to form the upper levels. A technique based on Artificial Intelligence gets a set of operators, an initial state, and a goal state to produce a sequence of operators.

When the source code of the software application is available, white-box testing can be applied by analysing the source code and applying coverage criteria on the implementation to measure the quality of tests. However, often source code is not available, and black-box testing must be performed. In these cases, using model-based testing allows to apply coverage metrics on the model as a quality measurement. Although model-based testing can have many advantages like the automatic generation of test cases, it also often suffers from the gap between the modelling paradigm and the implementation interface. In addition to absent source code, often the access to the actual functionality of the software application is barred, in our case by a GUI that represents the only interface to the software.

The tool presented in this paper overcomes these limitations of black-box testing GUIs with the automatic generation of the mapping code that allows interacting with a software application.

8 Conclusions and Future Work

We have presented a tool which reduces the effort to test applications through their GUI based on a formal specification in Spec#. This tool is an extension of the Spec Explorer tool, developed by Microsoft Research that already supports the modelling, test case generation, and test case execution. An overview of the GUI model and test

process is provided and the components of Spec Explorer as well as the components of the tool extensions are described.

Spec Explorer together with the GUI mapping tool can be used to test existing software applications, or it can be used to assist the development of new software applications and to test them through their GUI. In the former case, a reverse engineering process could be useful to construct a model, or part of the model, of an arbitrary application exhibited by its GUI. In the latter case, the specification of the application (or part of the application) is constructed and afterwards the application is implemented and tested using automatically generated mapping code.

The Notepad application was used as a running example to illustrate the approach. During our testing efforts, an inconsistency with the intuitive model was found.

The tools presented in this paper work only on a specific technology, but the underlying concepts, architecture, and process can be applied to any technology.

Our future work will be the construction of a tool to reverse engineer a GUI application, by automatic exploration of the application through its GUI and automatic generation of a Spec# model, or part of that model, in a way similar to the one presented by Memon in [12]. This will allow us to apply our approach to bigger application without the effort of constructing their models from scratch.

References

1. Belli, F. *Finite State Testing and Analysis of Graphical User Interfaces*. in *ISSRE 2001*. 2001: IEEE Comp. Press.
2. Campbell, C., et al., *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. 2005, Microsoft Research. p. 34.
3. Barnett, M., K.R.M. Leino, and W. Schulte. *The Spec# Programming System: An Overview*. in *CASSIS*. 2004.
4. Veanes, M., et al., *On-The-Fly Testing of Reactive Systems*. to appear as a Microsoft Research Technical Report.
5. Paiva, A.C.R., et al. *Modeling and Testing Hierarquical GUIs*. in *ASM 2005 - 12th International Workshop on Abstract State Machines*. 2005. Paris - France.
6. Gurevich, Y., *Evolving Algebras 1993: Lipari Guide*, in *Specification and Validation Methods*, E. Börger, Editor. 1995, Oxford University Press. p. 9-36.
7. Grieskamp, W., et al. *Generating Finite State Machines from Abstract State Machines*. in *ISSTA 2002, International Symposium on Software Testing and Analysis*. 2002.
8. Memon, A.M., M.L. Soffa, and M.E. Pollack. *Coverage Criteria for GUI Testing*. in *8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*. 2001.
9. Shehady, R.K. and D.P. Siewiorek. *A Method to Automate User Interface Testing Using Variable Finite State Machines*. in *27th International Symposium on Fault-Tolerant Computing*. 1997: IEEE Press.
10. Fujiwara, S., et al., *Test selection based on finite state models.* IEEE Transactions on Software Engineering, 1991. **17**(6): p. 591-603.
11. Memon, A.M., M.E. Pollack, and M.L. Soffa, *Hierarchical GUI Test Case Generation Using Automated Planning*. IEEE Transactions on Software Engineering, 2001. **27**(2).
12. Memon, A., I. Banerjee, and A. Nagarajan. *GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing*. in *WCRE2003*. 2003.