Reverse Engineered Formal Models for GUI Testing*

Ana C.R. Paiva[1], João C.P. Faria[1,2], and Pedro M.C. Mendes[1]

[1] Engineering Faculty of the University of Porto, [2] INESC Porto
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
{apaiva, jpf, pedro.mendes}@fe.up.pt
http://www.fe.up.pt

Abstract. This paper describes a process to reverse engineer structural and behavioural formal models of a GUI application by a dynamic technique, mixing manual with automatic exploration. The goal is to diminish the effort required to construct the model and mapping information needed in a model-based GUI testing process. A skeleton of a state machine model of the GUI, represented in a formal pre/post specification language, is generated automatically by the exploration process. Mapping information between the model and the implementation is also generated along the way. The model extracted automatically is then completed manually in order to get an executable model which can be used as a test oracle. Abstract test cases, including expected outputs, can be generated automatically from the final model and executed over the GUI application, using the mapping information generated during the exploration process.

Keywords: Reverse engineering; model-based GUI testing.

1 Introduction

GUI testing, with the purpose of finding bugs in the GUI or in the overall application, is a necessary but very time consuming V&V activity. The application of model-based testing techniques and tools can be very helpful to systematize and automate GUI testing. An example of a model-based GUI testing approach, based on the Spec# pre/post specification language [1] and extensions to the Spec Explorer model-based testing tool [2], is described in [8,9,10].

However, the effort required to construct a detailed and precise enough model for testing purposes (in order to be able to generate not only test inputs but also expected outputs), together with mapping information between the model and the implementation (in order to be able to execute abstract test cases derived from the model on a concrete GUI), are obstacles to the wide adoption of these techniques. One way to relief the effort mentioned is to produce a partial "as-is" model, together with mapping information, by an automated reverse engineering process. This model will have to be validated and detailed manually, in order to obtain a complete" should-be"

* Work partially supported by FCT (Portugal) and FEDER (European Union) under contract POSC/EIA/56646/2004.

S. Leue and P. Merino (Eds.): FMICS 2007, LNCS 4916, pp. 218–233, 2008.

model at the level of abstraction desired. Some defects in the application can be discovered in this stage. Overall, the goal is to automate the interactive exploratory process that is commonly followed by testers to obtain a model for an existing application.

In this paper we present a dynamic GUI reverse engineering approach to achieve such goal. The application under test (AUT) is automatically explored through its GUI to discover as much as possible the GUI structure and behaviour and to generate a corresponding GUI model in Spec#, together with mapping information between the model and the implementation. Automatic exploration can be intermixed with manual exploration to allow accessing functionalities that are protected by a key or are in some other way difficult to access automatically. During the exploration process, the intermediate code of the AUT is instrumented with Aspect-Oriented Programming (AOP) techniques in order to be able to recognize and capture a wider range of GUI controls and events, beyond native ones. The model generated automatically is subsequently validated and completed manually.

An "Address Book" application (Fig. 1) built in Java with the Standard Widget Toolkit (SWT) provided by the Eclipse/Rich Client Platform (RCP) will be used as an example to illustrate the approach proposed.

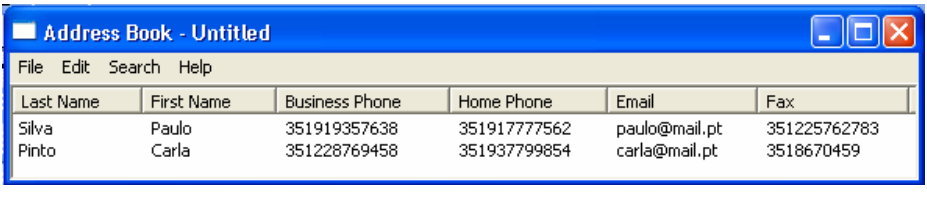


Fig. 1. Address Book main window

This paper is structured as follows: next section gives an overview of the reverse engineering and model-based GUI testing process; section 3 describes the desired characteristics of the target GUI model; section 4 describes the GUI reverse engineering process, while the details of the generation of the GUI model and mapping information are presented in section 5; section 6 describes model validation techniques; section 7 describes related work and the last section presents the conclusions and future work.

2 Overview of the Reverse Engineering and Model-Based GUI Testing Process

The goal of model-based testing is to check the conformity between the implementation and the specification (model) of a software system. The main activities of the model-based GUI testing process proposed are presented in Fig. 2.

The starting activity proposed is the construction of a preliminary GUI model by a reverse engineering process supported by the new REGUI2FM tool. This tool produces a preliminary GUI model in Spec# [1] and mapping information between the model and the implementation.

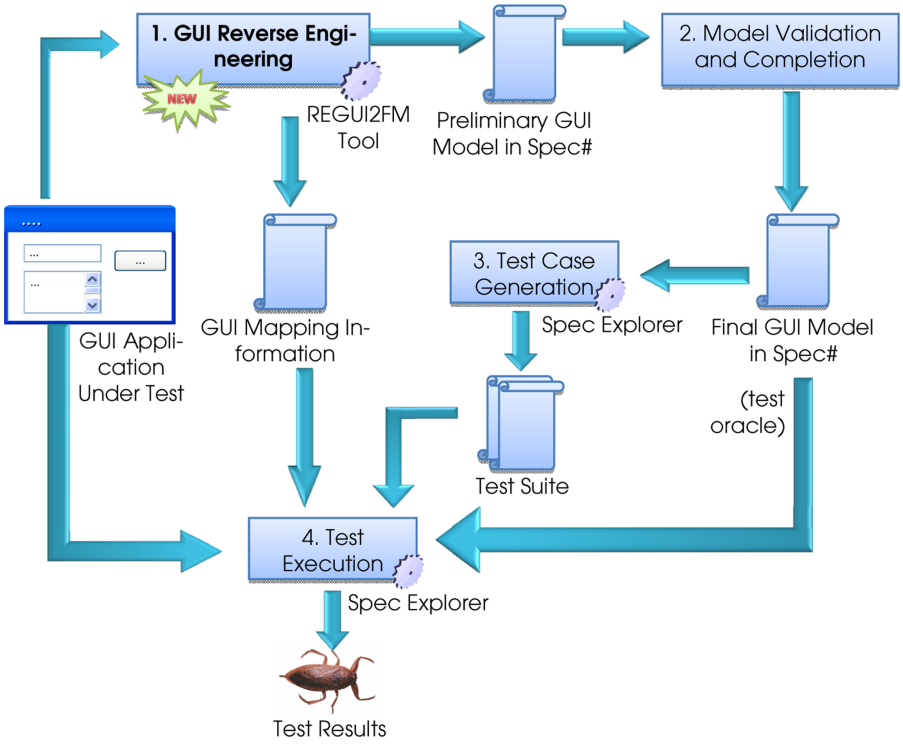


Fig. 2. Overview of the model-based GUI testing process with reverse engineering

The model obtained by the reverse engineering process captures structural information about the GUI (the hierarchical structure of windows and interactive controls within windows and their properties) and some behavioural information. The model describes at a high-level of abstraction the state of each window and window control (enabled/disabled status, content of text boxes, etc.) and the actions the user can perform on the window controls (e.g., press a button, fill in a text box). Besides the signature of each action, some pre and post-conditions are also generated, describing the states where each action is available (in the pre-condition) and navigation among windows caused by user actions (in the post-condition).

The mapping information comprise a XML file (GUI object map), describing physical properties of the GUI objects, and adaptor C# code with methods that simulate the user actions over the GUI, which are automatically bound to model actions. This information is needed to execute abstract test cases derived from the model on a concrete GUI.

In order to assure that the model is consistent with the application requirements and can be effectively used for test case generation and test output evaluation (as a test oracle), the preliminary model obtained by the reverse engineering process must be validated and completed manually with additional behaviour specifications.

Typically, executable method bodies must be added manually. Completeness and correctness of the model can be checked visually by defining views with the help of the Spec Explorer tool.

The final GUI model is then used to generate a test suite automatically, using the Spec Explorer tool [2]. Spec Explorer automatically generates test cases from a Spec# specification in two steps. In the first step, a finite state machine (FSM) is generated from the given Spec# specification. In the second step, a test suite that fulfils some coverage criteria is generated from the FSM (e.g., full transition coverage, shortest path or random walk). A test suite is a set of test segments with sequences of operations that model user actions (with input parameters) interleaved with operations to check the outcomes of those actions.

Test execution is also supported by the Spec Explorer tool. Conceptually, during test case execution, related actions (obtained from the mapping information) run in both the specification and implementation levels, in a "lock-step" mode, being their results compared after each step. Whenever an inconsistency is detected, it is reported.

3 Characteristic of the Target GUI Model

In the approach proposed, the main output of the reverse engineering process is a preliminary GUI model in Spec#, which is subsequently refined and provided as input to the Spec Explorer model-based testing tool. Hence, before explaining in more detail the reverse engineering process, it is important to describe how GUIs can be adequately modelled in Spec#, for model-based testing purposes.

Spec# is a pre/post specification language that extends the C# programming language with pre-conditions (written as *requires* clauses), post-conditions (written as *ensures* clauses), invariants, logical quantifiers, and other high-level constructs. A specification written in Spec# is executable: besides method pre/post conditions, one can write executable method bodies (also called model programs). This allows the specification to be used as a test oracle. Models written in Spec# can be the input for the Spec Explorer model-based testing tool.

For conformance testing purposes, a model written in Spec# can be seen as a description of a possibly infinite transition system. The states of the transition system are given by the values of the state variables. The transitions are executions of methods annotated as *Action*. Method pre-conditions indicate which actions are enabled in each system state. Spec# provides four kinds of actions: *controllable*, *probe*, *observable* and *scenario*. *Controllable* actions (the default ones) describe actions that are controlled by the user (or test driver) of the AUT; these actions may update the system state. *Probe* actions describe actions that only read the system state; they are invoked by the test harness in every state where their pre-condition holds, to check the actual AUT state against the model state. *Observable* actions are asynchronous and describe the spontaneous execution of an action in the AUT possibly caused by some internal thread; they are not used in the context of this work. *Scenario* actions describe composite actions; they are useful to drive the system into a desired initial state or to reduce the size of the test suite. When a scenario action is explored for test case generation, Spec Explorer records the sequence of atomic actions called and the intermediate states traversed.

Spec# can be used to model the GUI structure and behaviour at any level of abstraction desired. We describe next a choice of level of abstraction that we found appropriate for most form-based GUIs. Top-level windows of the AUT are modelled as separate classes or modules (namespaces). The internal state of each top-level window (content of text fields, etc.) is modelled by state variables. The actions available to the user inside each top-level window (enter a string into a text field, press a button, select a menu option, etc.) are modelled as *Action* methods. Pre-conditions describe the states in which the actions are available to the user. Post-conditions and method bodies describe the effect of the user actions on the system state. *Probe* methods are used to model the observation of GUI state by the user. *Scenario* methods are optionally used to model typical usage scenarios – sequences of steps (execution of lower level scenarios and atomic actions) the user should follow to achieve a goal. These scenarios need not represent end-to-end usage sequences, because scenario actions and atomic actions can be intermixed in the test cases generated.

An example of part of a Spec# model of the "Find" dialog box (Fig. 3) of the Address Book application is shown in Fig. 4. The "*Find what*" text box is modelled by a state variable (`findWhat`) and `set` and `get` action and probe methods. Check boxes are modelled by Boolean state variables (`matchCase` and `matchWholeWord`) and associated `set` and `get` methods. The *"Direction"* radio group and the "*Column"* combo box (with a closed list of options) are modelled by state variables (`direction` and `column`) of enumerated types and associated `set` and `get` methods. The "*Find*" button is modelled by an action method. All methods modelling user actions over GUI objects have a pre-condition that checks if their container window is enabled, and may have additional pre-conditions. For example, the *Find* button has an additional pre-condition to represent the fact that it is enabled only when the "*Find what"* text box is filled in. The example also includes calls to a reusable window management library that keeps track of the collection of windows open and of their enabled/disabled status (when a modal window is opened the other windows of the AUT are disabled).

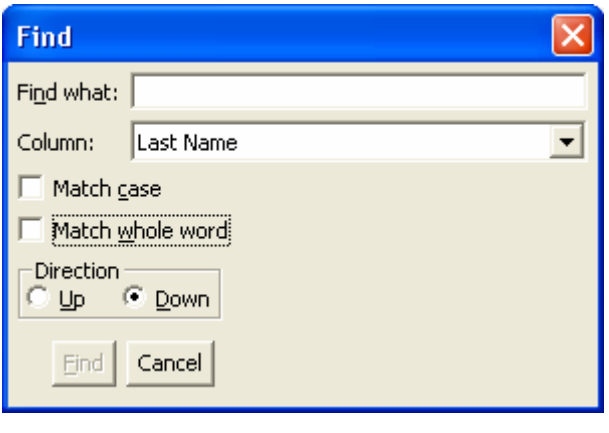


Fig. 3. "Find" dialog window of the Address Book application

```
namespace Find;

enum DirectionEnum { "Up", "Down" };
enum ColumnEnum {"Last Name", "First Name",
        "Business Phone", "Home Phone", "Email", "Fax"};

var string findWhat = "";
var bool matchCase = false;
var bool matchWholeWord = false;
var DirectionEnum direction = "Down";
var ColumnEnum column = "Last Name";

public string FindWhat {
  [Action(kind=Probe)] get
  requires IsEnabled("Find"); { return findWhat; }
  [Action] set
  requires IsEnabled("Find"); { findWhat = value; }
}

// similar properties for Column, MatchCase, MatchWholeWord
// and Direction

[Action] public void Find()
requires IsEnabled("Find") && findWhat != ""; {
  AddressBookWnd.FindNext(findWhat, column, matchCase,
                          matchWholeWord, direction);
}

[Action] public void Cancel()
requires IsEnabled("Find");
ensures IsClosed("Find"); {
   RemoveWindow("Find");
}
```

Fig. 4. Spec# model for the "Find" dialog window of the Address Book application

4 The GUI Reverse Engineering Process

The aim of the GUI reverse engineering tool (REGUI2FM) is to reduce the effort involved in the construction of the GUI model. As already mentioned in the overview, the GUI reverse engineering tool extracts structural and behavioural information about the GUI under test by a dynamic exploration process that mixes automatic and manual exploration. Its architecture is depicted in Fig. 5. The tool provides a front-end that gives access to a GUI Spy&Act tool, for automatic exploration (exploration mode), and a GUI Record tool, for manual exploration (record mode).

The GUI Spy&Act tool captures information about the GUI objects that are present in the AUT, in a way similar to the Spy++ tool that ships with Microsoft Visual Studio, and, based on that information, acts on the GUI objects simulating a user (e.g., click a button, select a menu option, or fill in a textbox), in a way similar to a smart monkey testing [7] tool. Since the Spy&Act tool interacts with the AUT through the operating system window manager, it is independent of the development language of the AUT.

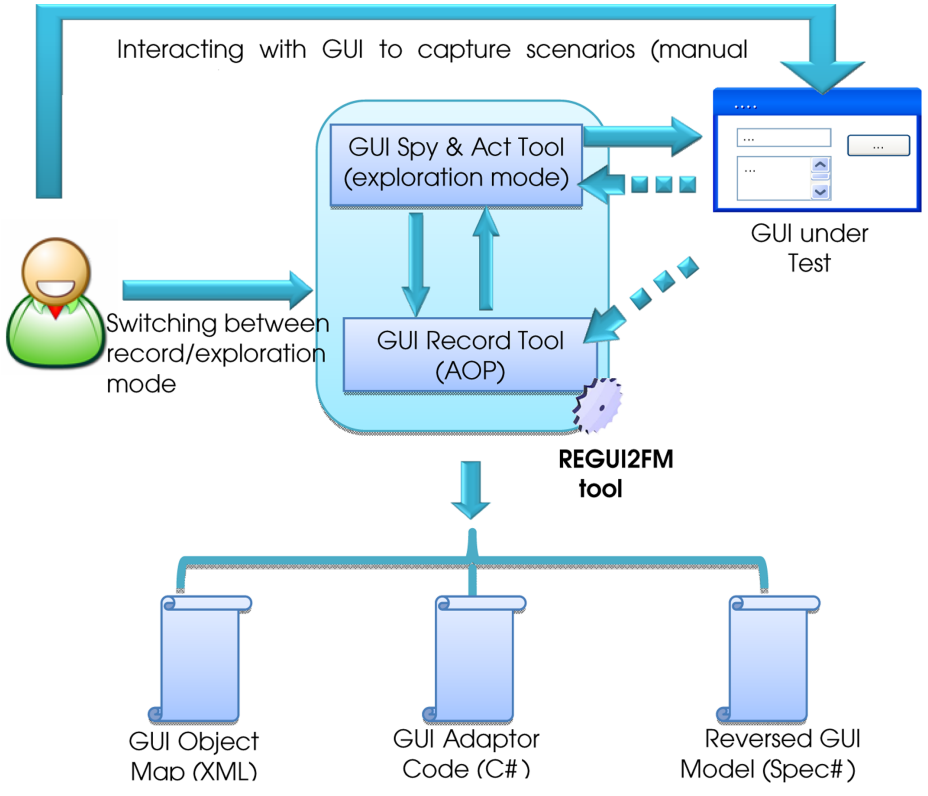


Fig. 5. Architecture of the reverse engineering tool (REGUI2FM)

A preliminary GUI model in Spec# is abstracted from the GUI states and transitions observed (in response to the actions performed), together with mapping information between the model and the physical GUI. Each window gives rise to a Spec# module. Interactive controls give rise to instance variables (e.g., there is a string variable for each text box) and methods (e.g., methods to read/write the text from/to a textbox), following the modelling style described in section 3 and illustrated in Fig. 4.

The Spec# state variables and actions (either *controllable* or *probe*) that the GUI Spy&Act tool should generate for each kind of GUI object can be configured in a XML file, as illustrated in Fig. 6. In this example, it will be generated a string instance variable and associated *set* and *get* methods for each textbox found, and a method corresponding to the *Click* action for each button found. This configuration information is also used by the tool to determine which kind of actions it should execute over physical GUI objects during the exploration. This file needs to be constructed only once and may be reused by other GUI reverse engineering processes.

```
<InteractiveObjects>
  <obj>
    <ClassName>TextBox</ClassName>
    <statevariable>string</statevariable>
    <controllable>set</controllable>
    <probe>get</probe>
  </obj>
  <obj>
    <ClassName>Button</ClassName>
    <controllable>Click</controllable>
  </obj>
  <obj> ... </obj>
  ...
</InteractiveObjects>
```

Fig. 6. XML configuration file

The GUI Spy&Act tool might not be able to reach application functionality that is protected by a key or is in any other way difficult to access without further knowledge. Two solutions are available to overcome this problem:

- the first one is to provide in advance some domain values that can be used during the automatic exploration process when interacting with controls;
- the other one is to switch to manual exploration mode, so that the user can interact with the GUI to supply the data or perform the steps required to access the hidden functionality, and switch back to automatic exploration thereupon.

The GUI Record tool captures the actions performed by the user, together with the GUI states traversed, in a way conceptually similar to a capture-replay tool. The sequence of actions performed by the user is abstracted to a Spec# method annotated as *scenario*.

At the end, the following files are generated:

- a XML file (GUI Object Map) gathering information about the windows and interactive controls detected, including physical identifying properties and logical names assigned;
- a Spec# file with the reversed GUI model, describing possible user actions over the GUI (behaviour model);
- a C# file (adaptor code) with methods to simulate concrete user actions over the GUI under test, corresponding to the abstract actions described in the Spec# model (needed for conformance checking during test execution).

The main activities and artifacts involved will be explained in next sections.

4.1 Automatic Exploration

The exploration starts from the application main window. In each step, it is captured information about interactive controls inside windows of the AUT. The information captured comprises: the hierarchical structure of GUI objects (windows and controls within windows); the type of each GUI object (window, button, textbox, etc.); values

of identifying properties (e.g., parent window and id), control state properties (e.g., enabled/disabled), and data state properties (e.g., text content) for each GUI object. The set of properties that should be captured for each type of GUI object can be configured by the user.

After identifying the interactive controls existing in an AUT window, the tool starts interacting with them simulating a user, e.g., click on buttons and menus, send text to textboxes, and select combo box options. The actions to explore upon each type of control and their input values can be configured by the user.

Some of the actions performed may cause navigation among windows (open a new window, close the current window and return to a previously visited window, etc.). In each step, the tool acts upon the window that has the input focus.

The tool keeps track of the collection of windows already reached and of the controls detected in those windows, using their identifying properties to avoid duplicates, as well as of the actions already performed on those controls.

The exploration process stops when all the relevant actions in all the windows reached have been explored, or when it is unable to make progress for some reason.

4.2 Manual Exploration

When the exploration algorithm stops before capturing information about all the windows of the application (e.g., there is a part of the application which is protected by a key), the user can switch to record mode.

In record mode it is assumed that the AUT runs on an AO (aspect oriented) enabled virtual machine and that it was built using the object-oriented programming paradigm [4,11]. When running the application in such an environment, the developed aspects extend the GUI object's "construction" process, by adding extra event listeners. These listeners are enabled only when an environment variable indicates that the current exploration mode is manual mode. The listeners intercept all possible user actions on standard GUI objects. Interaction with customized GUI objects extending standard ones requires adding some extra advice code using the chosen AO programming language for the purpose at hand. The advantage of this AOP technique is the ability to recognize a wide range of GUI controls, beyond native ones.

The advice code is responsible for logging user actions while interacting with GUI controls saving them in a Spec# scenario action such as the one that can be seen in Fig. 8.

5 Generation of the GUI Model and Mapping Information

5.1 Generation of the GUI Object Map (XML)

The GUI object map (Fig. 7) enumerates the GUI objects (windows and controls) of the AUT, and relates logical GUI object names with physical identifying properties. It is stored in a XML file. Logical names are assigned based on some heuristics (caption, nearest label, etc.).

```
<window name="Find">
  <caption>Find</caption>
  <class>#32770</class>
    <control name="Cancel">
      <caption>Cancel</caption>
      <class>Button</class>
      <id>2</id>
      <childPos>8</ChildPos>
    </control>
    <control ...>
      ...
    </control>
  ...
</window>
```

Fig. 7. GUI object map (XML)

5.2 Generation of the GUI Model (Spec#)

A preliminary GUI model in Spec# is generated by the REGUI2FM tool as explained in the next sub-sections.

5.2.1 Generation of the Overall Model Structure and Pre/Post Conditions

As already mentioned in section 3, the top-level windows of the application are modelled in separate namespaces or classes (for modularity reasons).

Inside each module (namespace or class) corresponding to a top-level window, state variables are used to model its abstract state and the state of the controls inside it. Each action that can be performed by the user within each top-level window (set/get the content of a control, press a button, etc.) is represented in Spec# by an *Action* method with a pre-condition that checks if the container window is enabled. In the case of actions that cause navigation among windows, it is also generated a post-condition that checks the open/closed and/or enabled/disabled status of the affected windows and a default method body (see the *Cancel* method in Fig. 4).

Besides keeping track of window navigation effects caused by the actions explored, the reverse engineering tool also keeps track of enabling/disabling and content update effects on GUI controls. Some of these dependencies among GUI controls can also be represented through pre and post-conditions, according to the type of dependency:

– "Setting the content of an object *A* to some condition enables an object *B*" – e.g., in the *Find* dialog shown in Fig. 4, the *Find* button is enabled when the *findWhat* text box is filled in. This dependency may be represented in Spec# by adding a pre-condition that checks the state of *A* to the methods that describe possible actions on object *B*. This avoids the addition of an extra state variable to represent the enabled/disabled status of *B*.

```
[Action] ... BMethods(...)
requires IsEnabled("WindowOfB") && A.State == S_A;
...
```

– "Performing an action on an object *A* updates the content of an object *B*" – e.g., pressing a *Clear* button may erase the fields in a form. This dependency may be represented in Spec# by a post-condition of the method that describes that specific action on *A*. A default method body is also generated.

```
[Action] ... AMethod(...)
ensures B.state == S_B;
{  B.state = S_B; ... }
```

In some cases, these dependencies can be discovered automatically by the exploration process. That would be the case of buttons that are only enabled when some text box is filled in, and buttons that erase the context of text boxes.

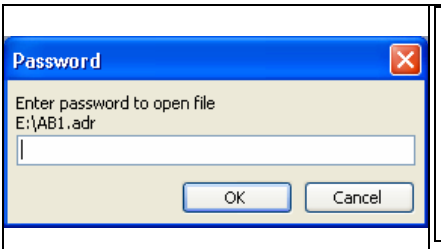
5.2.2 Generation of Default Method Bodies

Default method bodies are generated for set/get methods (see Fig. 4), and methods that cause navigation (see Cancel method in Fig. 4). Those method bodies must be checked and completed by the user, to take into account complex behaviours and side effects. For example, the method body of the *Find* action in Fig. 4 has to be constructed manually. All the other method bodies are generated automatically.

5.2.3 Generation of Scenario Methods

The sequences of actions performed by the tester while in record mode are captured by the REGUI2FM tool as *scenario* actions (Fig. 8) supported by Spec Explorer, as explained before. If desired, the scenarios generated can be subsequently edited by the modeller/tester. E.g., concrete values can be replaced by parameters to make the scenarios more generic and reusable.

Scenarios are useful for testing purposes in different ways: as a technique to drive the application into a desirable specific state, overcoming the problem of functionality protected by a key (Fig. 8), as explained before; as a way to describe test conditions that would be covered by manual tests and that can be seen as the minimum set of conditions to automatically test; and as a technique to prune the model exploration and test case generation process [2].




```
[Action(Kind=
ActionAttributeKind.Scenario)])
void PasswordScenario()
requires IsEnabled("Password");
{
  Password.Text = "mypassword";
  Password.Ok();
}
```

Fig. 8. Manual exploration sequence of a password dialog box recorded as a scenario action

5.3 Generation of the GUI Adapter Code (C#)

The C# code needed to execute the abstract test suit upon the real GUI, simulating the user actions, has a method for each (abstract) action described in the Spec# model. As an example, Fig. 9 illustrates the C# code generated for the portion of the Find name-space shown in Fig. 4.

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.

```
#region automatically generated code
  class GUIAdapter {
  public static void Find_SetFindWhat(string p0){
    UserEvents.SetText("Find.FindWhat, p0);
  }
  public static string Find_GetFindWhat(){
   return UserEvents.GetText("Find.FindWhat");
  }
  public static void Find_matchCase(bool p0) {
    UserEvents.SelectCheckBok("Find.MatchCase", p0);
  }
  public static void Find_matchWholeWord(bool p0) {
    UserEvents.SelectCheckBok("Find.MatchWholeWord", p0);
  }
  public static void Find_Find() {
    UserEvents.Click("Find.Find");
  }
  public static void Find_Cancel() {
    UserEvents.Click("Find.Cancel");
  }
  //...
}
#endregion
```

Fig. 9. C# code to simulate user actions for test execution

6 Model Validation

After completing the GUI model manually, it is possible to construct a view [10] of the navigation map using Spec Explorer (Fig. 10). Visual inspection of this map is a way to validate the model obtained.

Each state in Fig. 10 indicates the windows of the GUI that are enabled. In the presence of modeless windows, there may be more than one window enabled at the same time, in which case, a state may have more than one window name. This is the case of the *Find&AddressBook* state. While a modal dialog window is opened, as is the case of *Open*, *Save* and *Contact* windows in Fig. 10, user interaction with all other currently open windows of the same application is disabled.

The transitions visible at this level of abstraction are transitions that open/close windows of the GUI application. All transitions that occur inside a window/dialog are abstracted as one transition from the state that represents the dialog/window to itself.

This view can be expressed mathematically as the projection of the model states onto the state variable that holds the set of enabled windows.

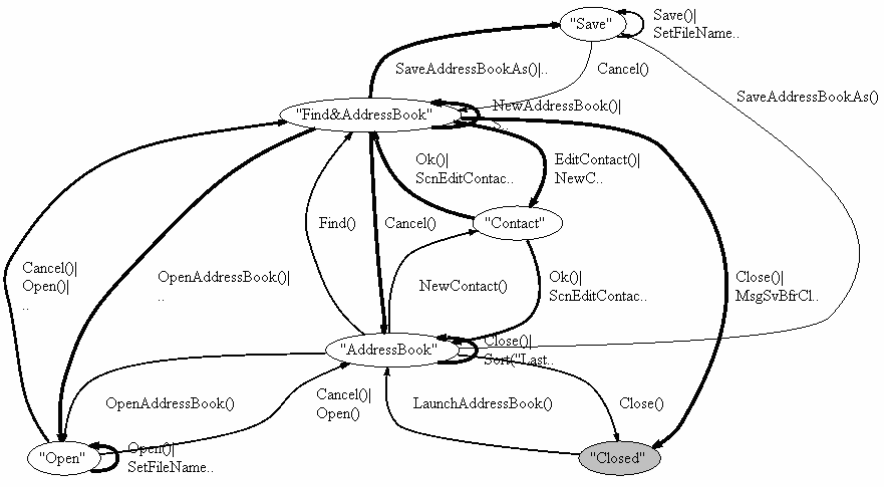


Fig. 10. Navigation map graph

7 Related Work

Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction [3]. It may be performed by static and dynamic analysis. Static analysis is performed on software code and does not involve its execution. Dynamic analysis extracts information from software by executing it.

Reverse engineering can be useful in several contexts like documentation, maintenance and specification-based testing. Another common application of reverse engineering is within a re-engineering process, for instance, to exchange legacy systems to different newer technologies.

The world is full of legacy systems. The technology is in constant change and some companies need to update their old systems. Reverse engineering tools can be used to build the model of existing applications that can be used by UIMSs to generate new GUIs with the same functionality of the older ones, but implemented in more recent technologies, or to be accessed from other computer platforms with specific characteristics.

One common example is the migration of legacy user interfaces to web-accessible platforms in order to support e-commerce activities. Stroulia et al. describe the CelLEST system within which a new process for migrating legacy systems for the Web was developed [12,13].

Vanderdonckt et al. describe a reverse engineering process of Web user interfaces [14]. The goal is to extract models of Web applications that were not constructed using a model-based approach and then use those models to generate UIs for other computer platforms, like palms, pocket computers, and mobile phones, without losing the effort deployed in the construction of the initial application.

The use of reverse engineering techniques to extract models to be used in a specification-based testing process is not so common. However, there is at least one example of this approach developed by Memon [5]. Memon claims that constructing a GUI model that can be used for test case generation is difficult, so he developed an approach to reverse engineer a model directly from an executable GUI. A so-called GUI ripping process opens automatically all the windows of the GUI under test and extracts their widgets, properties, and values. In the end, it is generated a GUI model that represents the GUI structure as a GUI forest, and its execution behaviour as an event-flow graph and an integration tree. However, the tool does not allow editing the model generated. Memon reports experiences where the ripping process is applied to extract a model from a correct GUI; the model extracted is then used to test an incorrect GUI. In industrial environments, such approach is helpful for regression testing.

Our approach distinguishes from the Memon's approach [5] mainly because of the expressiveness of the behavioural model that can be obtained, which is not limited to describe dependencies between pairs of user events (by an event-flow graph), but comprises also an explicit representation of the GUI state and pre/post conditions expressing dependencies between user events and the GUI state. This allows the same model to be used for the generation of valid test input sequences and outputs expected. In the Memon's approach, a separate pre/post model might be used as a test oracle, i.e., for the generation of the outputs expected [6], and the test sequences generated in first hand may not be valid because of state dependencies that are not taken into account.

8 Conclusions and Future Work

It was presented a dynamic GUI reverse engineering process, mixing automatic and manual exploration, with the goal of diminishing the effort required in constructing a GUI model for model-based testing purposes. Manual exploration mode is used to overcome situations when the automatic exploration process cannot progress because of dependencies that it cannot discover or because of functionalities that are protected by a password.

The outcome of the reverse engineering process is a preliminary behavioural GUI model in Spec#, together with mapping information between the model and the implementation (needed for test execution). The Spec# model describes at a high level of abstraction the actions available to the user and their effect on the GUI state. The mapping information comprises a XML file that stores information about the physical properties of the GUI objects, and a C# code file that bridges the gap between the abstract actions described in the model and simulated user actions upon the physical GUI objects.

The model generated automatically by the reverse engineering process has to be validated and completed manually so that it can be used as a test oracle. Test cases are generated from this model and then executed to check the conformity between the model and an implementation with the help of the Spec Explorer tool.

Preliminary results of using the reverse engineering tool (REGUI2FM) in small GUI applications show that the majority (around 70%) of the model can be built automatically.

We are currently enhancing the REGUI2FM tool to deal with more complex GUI applications and to use in the automatic exploration mode the same AOP mechanisms that are used to record user actions in manual exploration mode. By proceeding with this evolution, the scope of recognizable objects is expected to broaden, and the effort required to build the models is expected to be further reduced.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362. Springer, Heidelberg (2005)
2. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, Microsoft Research, MSR-TR-2005-59 (May 2005)
3. Chikofsky, E.J., Cross, J.H.: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software 7(1), 13–17 (1990)
4. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: M. A. a. S. M. (eds.) Proceedings of the European Conference on Object-Oriented Programming (1997)
5. Memon, A., Banerjee, I., Nagarajan, A.: GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In: Proceedings of the WCRE 2003 - The 10th Working Conference on Reverse Engineering, Victoria, British Columbia, Canada, November 13–16 (2003)
6. Memon, A.M., Pollack, M.E., Soffa, M.L.: Automated Test Oracles for GUIs. In: Proceedings of the FSE (2000)
7. Nyman, N.: In Defense of Monkey Testing (conferred in May 2006)
8. Paiva, A.C.R.: Automated Specification-Based Testing of Graphical User Interfaces, Ph.D, Engineering Faculty of Porto University (Ph.D thesis), Department of Electrical and Computer Engineering (2007), `http://www.fe.up.pt/~apaiva/PhD/PhDGUITesting.pdf`
9. Paiva, A.C.R., Faria, J.C.P., Tillmann, N., Vidal, R.F.A.M.: A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785. Springer, Heidelberg (2005)
10. Paiva, A.C.R., Tillmann, N., Faria, J.C.P., Vidal, R.F.A.M.: Modeling and Testing Hierarchical GUIs. In: Proceedings of the ASM 2005 - 12th International Workshop on Abstract State Machines, Paris - France, March 8–11 (2005)
11. Sabbah, D.: Aspect-Oriented software development. In: Proceedings of the Third International Conference on Aspect-oriented Software Development, Lancaster, UK (2004)

12. Stroulia, E., El-Ramly, M., Iglinski, P., Sorenson, P.: User Interface Reverse Engineering in Support of Interface Migration to the Web. Automated Software Engineering 10, 271–301 (2003)
13. Stroulia, E., El-Ramly, M., Kong, L., Sorenson, P., Matichuk, B.: Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach. In: Proceedings of the WCRE 1999 (1999)
14. Vanderdonckt, J., Bouillon, L., Souchon, N.: Flexible Reverse Engineering of Web Pages with VAQUISTA. In: Proceedings of the IEEE 8th Working Conf. on Reverse Engineering (2001)