Modeling and Testing Hierarchical GUIs

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

[1] Engineering Faculty of Porto University,
[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 a new approach to model and test hierarchical Graphical User Interfaces (GUIs). We exploit the structure of Hierarchical Finite State Machines (HFSMs) to reduce the number of states in the "flat" Finite State Machine (FSM) resulting from the exploration of the model. Firstly, independent dialogs are identified and highlighted in a HFSM built from the FSM. Then, the portion of the FSM that describes each dialog is reduced. To illustrate the approach, we construct a model of the Notepad application, which is part of Microsoft Windows. The model is written in Spec♯ and is converted automatically to a FSM using the Spec Explorer tool developed at Microsoft Research. The HFSM is then defined and the total number of states of the FSM is reduced. Spec Explorer generates test cases from the FSM, and also tests the conformity between the specification and the implementation.

1 Introduction

Today's software systems usually feature Graphical User Interfaces (GUIs). GUIs are the established medium of interaction with computer systems and can be a crucial point in the users' decisions to use or not use the system.

The user interface development process isn't mature yet. Integrated tools for specifying, implementing, and testing user interfaces are still lacking.

GUI testing is extremely time-consuming and costly. Applications are becoming bigger and more complex and manual testing of GUIs is labor-intensive, frequently monotonous, and is becoming an even more difficult activity. Currently used GUI testing methods are almost always ad hoc. Test cases are constructed manually, without guaranties of adequate coverage according to some criteria, and the evaluation decision whether the GUI is adequately tested is taken based on the developer's experience without theoretical foundation. When the GUI is modified, the developer needs to redefine the test suite and run the tests again.

Although Formal Methods have been criticized over past years for not being able to meet expectations, it is agreed upon that there should be a way of modeling user interfaces in an unambiguous form, supporting the formal evaluation of requirements.

The paper is organized as follows: the next section presents a way to model GUIs, outlines common Hierarchical Finite State Machines (HFSM) properties, and describes how to take advantage of them. Section 3 presents an approach to generate test cases from those state machines. Section 4 illustrates the approach by constructing the model of the Notepad application, generating the test cases and running the test cases to check the conformity between the application and the specification. Section 5 discusses related work and the last section summarizes the results achieved.

2 Modeling GUIs and Using Their HFSM Properties

State machines are well suited to model reactive systems. A state machine defines a set of states and transitions between states caused by actions. GUIs are reactive systems that respond to user actions. State machines can be very useful to guide the testing of software applications [7].

In general, the state of a GUI can be rich, e.g. include text typed by the user. Thus, the state machines are often infinite in size. Abstract State Machines (ASMs) [6] provide a way to model any system at an adequate level of abstraction. In our case of GUIs, the interface is naturally defined by events like user actions and program reactions. Still, the level of abstraction varies by what one considers as a single event: the pressing-down or releasing of a single key, the combined action "a letter has been typed", or the input of an entire sentence. In virtually all GUI implementations, such events are placed in a message queue and processed in order. This outer message loop can be seen as an ASM with guarded actions which fire only when an appropriate message is fetched from the queue.

A common problem with state machines is state explosion. One solution is to obtain a FSM from an ASM by exploring the state space while applying bounds. Here, the challenge is to find appropriate bounds.

HFSMs provide a way to deal with the state explosion problem. A HFSM is a FSM in which vertices can represent single states or groups of states (and transitions between the states of the group). These groups of states (and transitions) are themselves FSMs. Given a HFSM, it is possible to obtain a "flat" FSM by recursively substituting each group of states by its associated FSM.

HFSM are well suited to partition the behavior of a GUI: the hierarchical structure of the HFSM can mimic the hierarchical structure of objects and dialogs of the GUI. For example, a GUI might have a main window with a top menu (possibly with submenus) allowing the user to open modal dialog windows. While a modal dialog window is opened, user interaction with the other windows of the same application is disabled. This very common structure can be modeled by a HFSM with one group of states for each modal dialog. When we do not consider nested modal dialogs, then each modal dialog can be seen as an independent FSM.

The existence of independent modal dialogs allows us to reduce the number of states to consider. Assume we have an application with one main window, described by a FSM with m states, and k independent modal dialogs D_1, D_2, ..., D_k that can be accessed from the main window, with each D_i described by a FSM with n_i states. Without restrictions, the total number of states of the complete application would be the product $m \cdot n_1 \cdot ... \cdot n_k$ (because a state of the application is a combination of states of the main window and the dialog windows). But, if the D_is are modal, only one dialog can be open at a time, and fewer states have to be considered. Assume that, in the state machine that describes each dialog D_i, there is one distinctive state that represents the situation where the dialog is closed, and all the other n_i-1 states represent situations where the dialog is open. The possible states of the application can be grouped as follows:

- a group with $m \cdot 1 \cdot ... \cdot 1 = m$ states representing the situation where all the dialogs are closed and only the main window is active;
- for each dialog D_i, a group with $m \cdot 1 \cdot ... \cdot (n_i\text{-}1) \cdot ... \cdot 1 = m(n_i\text{-}1)$ states representing the situation where D_i is open and all the other dialogs are closed.

Summing up, the total number of states of the application is $m \cdot (n_1 + ... + n_k\text{-}k+1)$.

In the case of an application with modeless dialog windows, a similar reduction of the number of states cannot be achieved, because any number of modeless windows can be open at the same time. But, if the behavior of each dialog window is not affected by the state of the other dialog windows, then they can be considered independent. For testing purposes, it is not necessary to consider all the combinations of states of the different dialogs, as will be explained in the next section. Basically, it will suffice to fully test the behavior of one dialog, for only one particular state of all the other dialogs. Roughly, this corresponds to consider a reduced state machine similar to the one obtained in the case of modal dialogs, for testing purposes.

3 Test Case Generation

Generating test cases from FSMs is not a new research area [4],[7]. Coverage of all states and coverage of all transitions are common criteria used to generate test cases from a FSM. Since a HFSM can be converted to a FSM, those algorithms can also be applied. But, in this case, we are not taking advantage of the hierarchical structure to optimize the process. We will present an approach to generate test cases from HFSMs based on the notion of independent dialogs. We start by providing a more precise definition of independence based on the determination and analysis of variables manipulated by each dialog.

3.1 Determination of the Variables Manipulated by each Dialog

Consider an application with dialogs, D_1, D_2, ..., D_k, manipulating a set of variables V. W.l.o.g., we name the elements of this set as follows:

$$V = \{v_1, ..., v_{|V|}\}. \tag{1}$$

Without restrictions, the state space of the application, SSA, will be the Cartesian product of the values of the variables in the set V:

$$SSA = dom(v_1) \times ... \times dom(v_{|V|}). \tag{2}$$

From the complete FSM of the application, FSM_A, it is possible to obtain the subsets of FSM_A that describe each dialog D_i, FSM_{Di}, by state grouping according to a criteria provided by the developer. For example, FSM_{Di} could correspond to the group of states (and transitions between them) where dialog D_i is enabled.

Having delimited the state machine FSM_D that describes the behavior of a dialog D, it is possible to automatically deduce which variables are manipulated (read or written) by that dialog.

A variable v_i is written by (or is affected by) a dialog D if there is a transition in FSM_D that changes the value of v_i. Formally,

$$[\exists T \in Trans(FSM_D) \cdot Source(T).v_i \neq Dest(T).v_i] \Rightarrow v_i \textit{ is written by } D. \tag{3}$$

A variable v_i is read by (or influences the behavior of) a dialog D if at least one of the following conditions holds:

– there are two transitions T and T' in FSM_D and a variable v_k in V (not necessarily $i \neq k$) such that: (i) the source states of T and T' are different only in the value of v_i; (ii) T and T' have the same triggering action (name and arguments); (iii) the destination states of T and T' have different values of v_k; and (iv) at least one of the transitions (say T) changes the value of v_k;
– there are two states S and S' and a transition T with source S in FSM_D such that: (i) S and S' are different only in the value of v_i; (ii) there is no transition T' with source S' and the same action as T.

Formally,

$$\left\{ \begin{array}{l} (\exists T, T' \in Trans(FSM_D) \cdot Source(T) = S \wedge Source(T') = S' \wedge \\ S.v_i \neq S'.v_i \wedge (\forall j \neq i \cdot S.v_j = S'.v_j) \wedge Action(T) = Action(T') \\ \wedge \exists v_k \in V \cdot Dest(T).v_k \neq Dest(T').v_k \wedge S.v_k \neq Dest(T).v_k) \\ \vee \\ (\exists S, S' \in States(FSM_D) \cdot \exists T \in Trans(FSM_D) \cdot Source(T) = S \\ \wedge S.v_i \neq S'.v_i \wedge (\forall j \neq i \cdot S.v_j = S'.v_j) \wedge \neg \exists T' \in Trans(FSM_D) \cdot \\ Source(T') = S' \wedge Action(T) = Action(T')) \end{array} \right\} \tag{4}$$
$$\Rightarrow v_i \textit{ is read by } D.$$

Informally, this means that the response to user actions (and the actions available) in the context of dialog D depends on the value of v_i. In practice, this means that any implementation of dialog D must read (or query) the value of v_i when responding to user actions (or when determining which actions are available). A demonstration of the formula presented above is outside the scope of this paper, but can be found in www.fe.up.pt/˜apaiva/demo3.pdf.

For instance, consider an application with state variables $V = \{v_1,v_2,v_3\}$, and two dialogs D_1 and D_2 with the behavior described by the state machine of Fig. 1. The state machine also includes transitions (labeled A_2 and A_7) that do not belong to any of the dialogs, but allow switching between them.

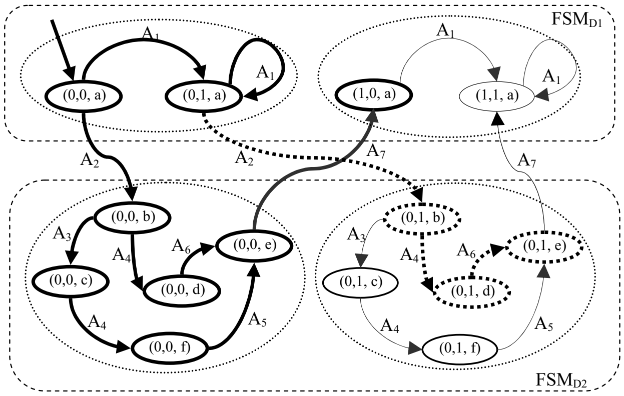

Fig. 1. State machine of an application with dialogs D_1 (action A_1) and D_2 (actions A_3 to A_6)

Dialog D_1 is enabled when v_3=a. Dialog D_2 is enabled when $v_3 \neq a$ and v_1=0. Given the transition $(0,0,a)\xrightarrow{A_1}(0,1,a)$ in D_1, we conclude, by (eq. 3), that v_2 is written by D_1. This is the only variable manipulated by D_1. From transition $(0,0,b)\xrightarrow{A_3}(0,0,c)$ in D_2, we conclude that v_3 is written by D_2. From transitions $(0,0,b)\xrightarrow{A_4}(0,0,d)$ and $(0,0,c)\xrightarrow{A_4}(0,0,f)$ in D_2, we conclude, by (eq. 4), that v_3 is also read by D_2. This is the only variable manipulated by D_2.

3.2 Independent Dialogs

Given two dialogs, if the set of variables written by one of the dialogs is disjoint from the set of variables manipulated by the other dialog, then they are independent. Informally, two dialogs are independent if the behavior of any of the dialogs is not affected by the state and interactions that occur in the other dialog. For example, dialogs D_1 and D_2 described above are independent.

3.3 Structuring the HFSM based on the Variables Manipulated by each Dialog

Under certain conditions[1], there is a relationship between the state variables manipulated by each dialog D_i and the structure of the FSM of the application (FSM$_A$), that allow us to structure FSM$_A$ into a HFSM.

[1] A sufficient condition is that the enabling condition of each dialog restricts the domain of each variable independently of the other variables.

Let PFSM_{Di} be the projection of FSM_{Di} onto the variables manipulated by D_i, as illustrated in Fig. 2. PFSM_{Di} describes the behavior of D_i.

In the opposite direction, FSM_{Di} is the union of the instances of PFSM_{Di} for all possible combinations of values of the variables that are not manipulated by D_i (restricted to the enabling condition of D_i). For example, FSM_{D1} (Fig. 1) has 2 instances of PFSM_{D1} (Fig. 2) with $v_1{=}0 \wedge v_3{=}a$, and $v_1{=}1 \wedge v_3{=}a$.

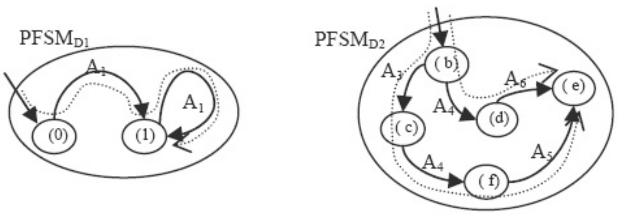

Fig. 2. State machines of dialogs D_1 and D_2 projected from the FSM at Fig. 1. Dotted lines represent test cases

Given this, FSM_A can be organized into a 3-level HFSM. The first (top) level has only two vertices, one for each dialog, as illustrated in Fig. 3. The criterion to group states together is given between square brackets. The intermediate level corresponds to Fig. 2, and the third (bottom) level corresponds to Fig. 1.

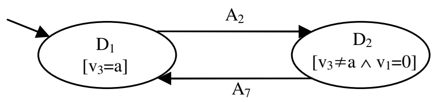

Fig. 3. Top level of the HFSM

3.4 FSM Reduction and Test Case Generation

Using the transition coverage criteria to generate test cases from the FSM of Fig. 1 (with 18 transitions), we would get 4 test cases (paths) with 21 steps.

But, since D_1 and D_2 are independent dialogs, they don't need to be tested every time there is a change on variables on which they don't depend. Only one instance of each dialog need to be tested. To test dialog D_i, the values of the variables that are not manipulated by D_i are fixed to a particular value, and the transition coverage criteria is applied to the PFSM of D_i to generate test cases. For example, to test D_1 we could fix $v_1{=}0$ ($v_3{=}a$ is already fixed) and generate the test case illustrated by the dotted line in Fig. 2. To test D_2 we could fix $v_2{=}0$ ($v_1{=}0$ is already fixed). With this approach, only 7 transitions are exercised, instead of 21. The instances of D_1 and D_2 that are tested are the ones shown on the left-hand side of Fig. 1.

To fully test the application, actions that do not belong to these dialogs, also have to be exercised. This is the case of actions A_2 and A_7 in Fig. 1. Applying the same approach to each of these actions (each one can be regarded as a dialog with a single action), we conclude that only one instance of each action need be tested in this case. For example, we can exercise (test) the instances of A_2 and A_7 shown with thick lines in Fig. 1. Overall, the transitions that need be exercised are all the transitions shown with thick lines in Fig. 1. Three test cases (paths), with a total of 10 steps, are enough to cover them. The size of the test suite is reduced from 21 steps to 10 steps.

In some cases, it is not sufficient to test only one instance of each dialog. After assuring that one instance is fully tested, a second instance may have to be traversed (usually only in part, by the shortest path) in order to reach some state or transition that has to be exercised. For example, assume that, with respect to Fig. 1, it is important to reach state (0,1,e), because it is the source of a transition that has not been tested yet (not represented in Fig. 1). In such case, the path shown with dotted lines in Fig. 1 also has to be included in the test suite.

In general, the selection of sequences in a way that all of the application's behavior is exercised, is a problem as hard as deciding the reachability of a state. Partial order reduction (POR) techniques used in model checking [10] address a very similar problem: Given a property of the system, e.g. a temporal property describing the reachability of a state, POR reduces the number of states that must be explored in order to decide whether the property holds for the entire state space. POR exploits redundancies of the state space like the commutativity of enabled transitions.

4 Example

In this example, the Notepad application, part of Microsoft Windows, is used to illustrate the approach to model and test GUIs. We have chosen Spec♯ as the specification language to encode the actions of the message loop; the Spec Explorer tool is used to test conformity between the specification and the implementation.

4.1 Spec♯ and Spec Explorer

Spec Explorer (http://research.microsoft.com/SpecExplorer) is a software modeling and testing tool developed at Microsoft Research. The model can be written in the abstract state machine language (AsmL) (http://research.microsoft.com/fse/AsmL), or Spec♯, which is a superset of C♯. Certain methods are annotated as actions, which represent possible transitions of a (possibly infinite) transition system. Preconditions, written as "requires" clauses, define in which states actions are enabled. The state of the system is comprised by the variables of the model program.

Consider the following example which could describe a part of a text editor. It defines the state as a text string. The TypeText method is marked as an action by an attribute. It is enabled when the combined length of the current text and the newly typed text does not exceed a specified maximum. When it is fired, it appends the newly typed text to the current text string.

```
string text = "";
[Action] void TypeText(string typedText)
  requires text.Length + typedText.Length < 100; {
  text += typedText; }
```

The transition systems discussed in this paper are deterministic. In this case, an explicit state model checker built into Spec Explorer systematically explores all transitions reachable within certain bounds. Spec Explorer can be configured to restrict the search space by various means including state filters and equivalence class groupings. The latter 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. This idea was first described in [5], a generalization can be found in [12]. The resulting FSM can be traversed using different coverage criteria, which yields a set of transition sequences forming a test suite. Conformance between the model and an implementation can be established by binding the model actions to implementation actions, executing the test suites on both the model and the implementation, and comparing their results. In general, Spec Explorer can handle non-deterministic transition systems, and the conformance relation is defined by interface automata [13]. An overview of the predecessor tool can be found in [2].

4.2 The model

We have constructed a complete model of the Notepad application but here only the Open and Replace dialogs will be used to illustrate the approach. The first is a modal dialog and the second is a modeless dialog. The model presented captures only the possible actions and sequences of actions the user can perform.

Variables like *NotepadObjAct*, *OpenObjAct*, and *ReplaceObjAct* are used to point out the interaction object or dialog that has the input focus. The other three variables, *FindWhat*, *FileName*, and *FileOpened* are needed because they influence the behavior of the application.

At the initial state, the Notepad application is closed. The only possible action at this state is to launch the application.

```
string NotepadObjAct="NotOpen";
string OpenObjAct="NotOpen", ReplaceObjAct="NotOpen";
string FileName=null, FindWhat=null, FileOpened=null;
[Action] void LaunchNotepad() requires NotepadObjAct=="NotOpen"; {
    NotepadObjAct = "ClientArea"; }
```

Spec. 1. Start the Notepad application

After launching the application, it is possible to interact with the client area typing text, open the File and Edit menus, and close the application.

```
bool NotepadEnabled {get {return NotepadObjAct notin
    Set{"NotOpen","OpenDlg", "ReplaceDlg"};} }
[Action] void TypeText(string text)
    requires NotepadObjAct=="ClientArea";{}
[Action] void File() requires NotepadEnabled ; {
    NotepadObjAct="FileMenu"; }
[Action] void Edit() requires NotepadEnabled; {
    NotepadObjAct="EditMenu"; }
[Action] void Close() requires NotepadEnabled; {
    NotepadObjAct="NotOpen"; ReplaceObjAct="NotOpen";
    FileOpened=null; FindWhat=null; }
```

Spec. 2. Possible actions after starting the application

When the File menu is opened, it is possible to select the Open option. This option will open the Open dialog. When the Edit menu is opened, it is possible to select the Replace option. This option will open the Replace dialog.

```
[Action] void Open() requires NotepadObjAct=="FileMenu"; {
    NotepadObjAct="OpenDlg"; OpenObjAct="FileName"; }
[Action] void Replace() requires NotepadObjAct=="EditMenu"; {
    ReplaceObjAct="FindWhat"; NotepadObjAct="ReplaceDlg";
    FindWhat = null; }
```

Spec. 3. Options inside the File and Edit menus

When the Open dialog is open, it is possible to type a file name, select a file type, and press the Cancel and Open buttons. The *OpenObjAct* variable indicates which control has the focus in the Open dialog.

```
[Action] void FileName_O(string fileName)
  requires NotepadObjAct=="OpenDlg"; {
    FileName = fileName; OpenObjAct = "FileName"; }
[Action] void FilesOfType_O(int index)
  requires NotepadObjAct=="OpenDlg"; {
    OpenObjAct = "FilesOfType"; }
[Action] void Cancel_O() requires NotepadObjAct=="OpenDlg"; {
    OpenObjAct = "NotOpen"; NotepadObjAct = "ClientArea";
    FileName = null; }
[Action] void Open_O() requires NotepadObjAct=="OpenDlg"; {
    if (FileExists(FileName)) {
      OpenObjAct = "NotOpen"; NotepadObjAct = "ClientArea";
      FileOpened = FileName; FileName = null;} }
```

Spec. 4. Actions inside the Open dialog

When the Replace dialog is opened, it is possible to fill the word to search for, to fill the word to replace for, and also to press the buttons Find Next, Replace, Replace All, and Cancel. Since the Replace dialog is modeless, it is also possible to switch to the Notepad main window and the other way around.

```
[Action] void FindWhat_R(string str)
  requires NotepadObjAct=="ReplaceDlg" ; {
    FindWhat = str; ReplaceObjAct = "FindWhat"; }
[Action] void ReplaceWith_R(string str)
  requires NotepadObjAct=="ReplaceDlg" ; {
    ReplaceObjAct = "ReplaceWith"; }
[Action] void FindNext_R()
  requires NotepadObjAct=="ReplaceDlg" && FindWhat!=null; {
    ReplaceObjAct = "FindNext"; }
// similar actions omitted for Replace and ReplaceAll
[Action] void Cancel_R()
  requires NotepadObjAct=="ReplaceDlg" ; {
    ReplaceObjAct = "NotOpen";
    NotepadObjAct = "ClientArea"; FindWhat = null; }
[Action] void SwitchToReplaceDlg()
  requires ReplaceObjAct=="NotActive" && NotepadObjAct!="OpenDlg";
  { NotepadObjAct = "ReplaceDlg"; ReplaceObjAct = "FindWhat"; }
[Action] void SwitchToNotepad()
  requires NotepadObjAct!="OpenDlg" && ReplaceObjAct!="NotOpen"; {
   NotepadObjAct = "ClientArea"; ReplaceObjAct = "NotActive"; }
```

Spec. 5. Actions inside the Replace dialog

As soon as the model is constructed, Spec. 1,..., Spec. 5, and the domains of the functions' parameters and variables are defined, the Spec Explorer tool generates automatically the corresponding FSM. By providing criteria to define equivalence classes among the set of states, it is possible to generate a HFSM with different levels of abstraction.

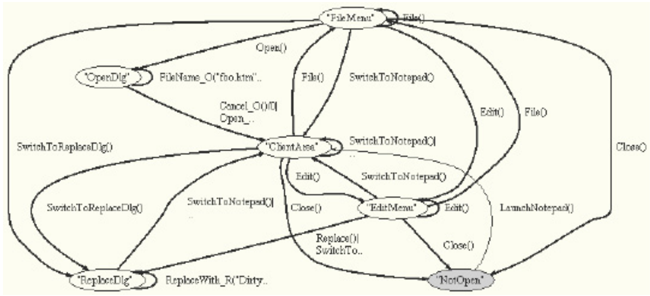

Fig. 4. Top level of the HFSM

The first level of abstraction, Fig. 4, will give an overview of the structure of the application. Each vertex corresponds to a different value of the *NotepadObjAct* variable. In this view, it is possible to see that the user can open the File menu or the Edit menu, interact with the client area typing text, open one of the two dialogs, Open and Replace, and it is also possible to switch the focus to the Replace dialog when this dialog is opened. The interaction inside those dialogs is detailed in the next level of abstraction, Fig. 5, and Fig. 6.

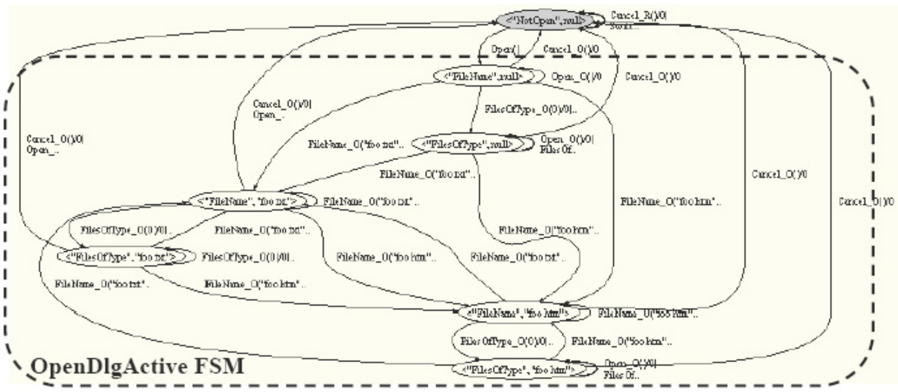

Fig. 5. State Machine of the Open dialog

The state machine of the Open dialog, Fig. 5, was obtained by grouping together states with equal values of the variables manipulated by the Open dialog (*OpenObjAct* and *FileName*). Each vertex shows the values of those variables. In this view, it is possible to see that the user can fill the file name, select the file type, open a file, and close the dialog. When the Open dialog is opened, the interaction object with the focus is the FileName textbox.

The state machine of the Replace dialog, Fig. 6, was obtained by grouping together states with equal values of the variables manipulated by the Replace dialog (*ReplaceObjAct* and *FindWhat*). The Replace dialog is a modeless dialog, from which it is possible to interact with other elements on screen; in particular, it is possible to switch to the Notepad main window.

The Replace dialog is in the *NotActive* state when it is opened but doesn't have the focus. This view shows that the user can fill the Find What and Replace With textboxes, and can press the Find Next, Replace, and Replace All buttons.

4.3 FSM Reduction

The complete FSM of our example has 399 steps and 69 states. Using the transition coverage criteria, without optimizing (reducing) the FSM, a test suite with at least 399 steps would be generated. This test suite would exercise the Open

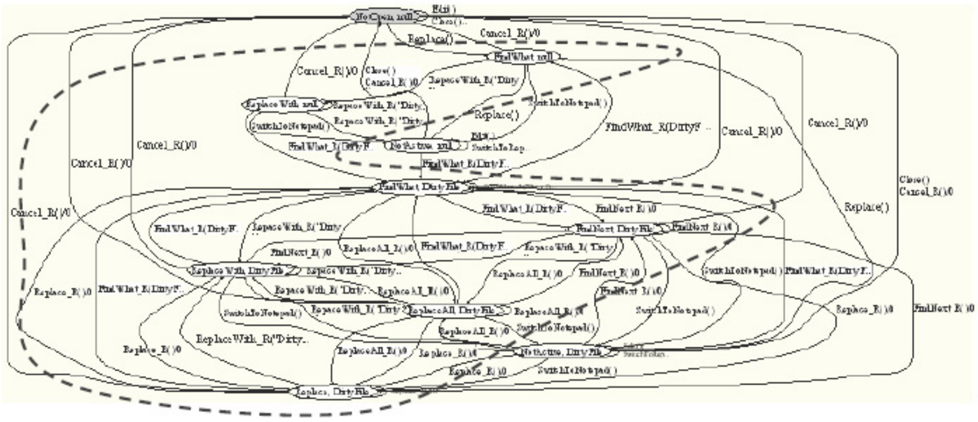

Fig. 6. State Machine of the Replace dialog. States where the dialog is active are surrounded by a dashed line

dialog 6 times and the Replace dialog 2 times. With our approach it is possible to reduce the size of the test suite.

The Open dialog occurs six times in the flattened FSM. Each instance of this dialog correspond to the states enclosed by a dashed line in Fig. 5, combined with the different values of the variables from which this dialog does not depend on (*ReplaceObjAct*, *FindWhat*, and *FileOpened*). So, the second or higher occurrence of the Open dialog can be substituted by the reduced state machine (with two states only) constructed with the minimum paths to reach the final states, as illustrated in the left-hand side of Fig. 7. Since the Open dialog has 6 states and occurs 6 times, the total number of states is reduced from 36 to 6+2*5=16. The number of transitions was reduced by 160.

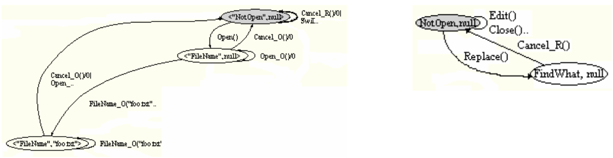

Fig. 7. Reduced FSMs of the Open (left) and Replace (right) dialogs

The Replace dialog occurs twice in the flattened FSM, for two different values of a variable on which this dialog does not depend on. The second occurrence of the FSM of the Replace dialog can be reduced to just one state, as illustrated in the right-hand side of Fig. 7. This leaves us with 9+1=10 states instead of 9*2=18 states. The number of transitions was reduced by 72.

Overall, the flattened FSM with initial 69 states and 399 transitions is reduced to a state machine with 69-(36-16)-(18-10)=41 states and 399-160-72=167 transitions.

4.4 Test Case Generation and Execution

The Spec Explorer tool generates automatically a test suite from the FSM covering all transitions.

To execute the test cases generated, it is necessary to bind model actions to implementation actions (methods) available in a .NET assembly. To serve as a bridge between the Spec♯ specification and the executable binary file of the Notepad application, we constructed some intermediate code in C♯, to execute and interact with the Notepad application simulating the user (trigger events like mouse clicks or keyboard keys).

Every time there is an inconsistency between the model and the implementation, the Spec Explorer tool stops and reports the error. An inconsistency can be obtained by several reasons:

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

In our experiment, no inconsistencies were found. This in not surprising since the Notepad application has been in use and tested for years already. Another explanation is that we modeled the behavior as we know it, accommodating any flaws or inconsistencies which we got used to.

5 Related Work

There is some research work addressing the automation of GUI testing.

Some tools for computer-based GUI testing, called capture & replay tools, are already commercially available. Examples of these tools can be found at (www.stlabs.com/marick/faqs/t-gui.htm). With these tools it is possible to record the user interactions (mouse clicks, mouse motions, keyboard input, ...) with a graphical user interface and replay them later. These tools can be helpful in several contexts like demonstrations; remote support; analysis of user behavior; macro functionality; and educational scenarios. But, for testing purposes, they are the subject of severe critics [14]: the test scripts are manually developed without coverage criteria concerns; the scripts contain hard-coded values; these tools store information at a low level of abstraction, capturing mouse positions, button clicks and storing bitmaps. Representing the information at such a low level of abstraction makes these tools very dependent on the physical properties of the use interface. A small change on the layout of the user interface might invalidate all the test cases.

There are several approaches to automate the generation of test cases from models of the GUI. Memon uses a model with a hierarchical structure in his work [8], to guide the generation of test cases, but not to reduce the size of the test suite. He defines a set of operators that are organized in hierarchies. The operators at upper levels are constructed from simpler ones at lower levels. These simpler operators correspond to user actions. Each operator has a pre condition that must be true before executing the operator, and the effect. Memon uses planning from Artificial Intelligence to generate test cases. Given a set of operators, an initial state, and a goal state, a planner produces a sequence of operators that will change the initial state to the goal state. He generates test cases from the upper hierarchical levels of abstraction and then nested invocations to the planner during abstract operator decomposition. Alternative test cases can be obtained by substituting the different test cases obtained for the lower levels into the high-level plan.

Andrews, at [1], uses HFSMs to model Web applications and uses constraints to reduce the set of input values and to help solving the state explosion problem.

Belli, in [3], presents an approach to model the legal and the illegal behavior of GUIs using FSA, Finite State Automata, and regular expressions. Belli starts identifying all legal sequences of user system interaction and then expands the model with illegal behavior. The final model is used to generate test cases that can bring the system into legal states, producing the desired system response, or into a faulty situation, producing an error message.

Shehady, in [11], uses VFSM, Variable Finite State Machines to model GUIs. VFSMs are FSMs with an added condition associated to each transition. The transition can be expressed by: name <state> <input> <next state> <output>. The VFSM is converted into a FSM to generate test cases using the partial W algorithm [4].

Some of the authors of this paper have already used a tool that preceded the Spec Explorer tool (the AsmL Tester tool) to automate the testing of GUIs [9], but the reduction of the FSM was n ot addressed.

A popular approach to test GUI is to code the test cases "manually" but, in this case, the developer has a hard work to test adequately the GUI behavior.

6 Conclusion

We presented an approach to model GUIs with HFSMs and to generate test cases from those models in an optimized way taking advantage of the hierarchical structure.

The Spec♯ specification language, developed by Microsoft Research based on Abstract State Machines, was used to construct the model of the application. This model was converted automatically into a FSM using the Spec Explorer tool. With the definition of expressions to construct state groups, it was also possible to obtain a HFSM. This tool was also used to generate test cases and execute them to perform conformity checks between a specification and an implementation. To test conformity between the specification and the implementation, an intermediate code in C♯ was constructed to simulate the user actions and interact with the application.

The approach was illustrated with a model of the Notepad application from Windows where it was possible to reduce the number of states of the initial FSM from 69 to 41 using the structure of the corresponding HFSM.

As future work, we plan to refine the approach described in this paper and automate the FSM reduction.

References

1. Andrews, A. A., Offutt, J., and Alexander, R. T.: Testing Web Application by Modeling with FSMs. presented at the SOftware SYstems Modeling. (2004)
2. Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., and Veans, M.: Model-Based Testing with AsmL.NET, presented at the 1st European Conference on Model-Driven Software Engineering, (2003).
3. Belli, F.: Finite State Testing and Analysis of Graphical User Interfaces, presented at the ISSRE 2001, (2001).
4. Fujiwara, S., Bochmann, G. v., Khendek, F., Amalou, M., and Ghedamsi, A.: Test selection based on finite state models, IEEE Transactions on Software Engineering, Vol. 17(6), (1991) 591–603.
5. Grieskamp, W., Gurevich, Y., Schulte, W., and Veanes, M.: Generating Finite State Machines from Abstract State Machines, presented at the ISSTA 2002, International Symposium on Software Testing and Analysis, July, (2002).
6. Gurevich, Y.: "Evolving Algebras 1993: Lipari Guide," in Specification and Validation Methods, E. Börger, ed. (Oxford University Press, 1995), pp. 9–36.
7. Lee, D. and Yannakakis, M.: Principles and Methods of Testing Finite State Machines – A Survey, Proceedings of the IEEE, Vol. 84, (1996) 1090-1996.
8. Memon, A. M., Pollack, M. E., and Soffa, M. L.: Hierarchical GUI Test Case Generation Using Automated Planning, IEEE Transactions on Software Engineering, Vol. 27(2), (2001).
9. Paiva, A. C. R., Faria, J. C. P., and Vidal, R. M.: Automated Specification-based Testing of Interactive Components with AsmL, presented at the 5th edition of the Quatic (Quality: the bridge to the future in ICT) international conference, Porto, (2004).
10. Peled, D., Clarke, E., and Grumberg, O.: Model checking (MIT Press, Cambridge, Massachusetts, 2000).
11. Shehady, R. K. and Siewiorek, D. P.: A Method to Automate User Interface Testing Using Variable Finite State Machines, presented at the 27th International Symposium on Fault-Tolerant Computing, (1997).
12. Veanes, M. and Campbell, C.: State Exploration with Multiple State Groupings, submitted to the 12th International Workshop on Abstract State Machines, (2005).
13. Veanes, M., Campbell, C., Schulte, W., and Kohli, P.: "On-The-Fly Testing of Reactive Systems," (to appear as a Microsoft Research Technical Report).
14. Zambelich, K.: Totally Data-Driven Automated Testing. Whitepaper, Automated Testing Specialists (ATS), retrieved at http://www.sqa-test.com/WhitePaper.doc.