A Generic Library for GUI Reasoning and Testing

João Carlos Silva
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
jcsilva@ipca.pt

João Saraiva
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
jas@di.uminho.pt

José Creissac Campos
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
jose.campos@di.uminho.pt

ABSTRACT

Graphical user interfaces (GUIs) make software easy to use by providing the user with visual controls. Therefore, correctness of GUI's code is essential to the correct execution of the overall software. Models can help in the evaluation of interactive applications by allowing designers to concentrate on its more important aspects.

This paper presents a generic model for language-independent reverse engineering of graphical user interface based applications, and we explore the integration of model-based testing techniques in our approach, thus allowing us to perform fault detection.

A prototype tool has been constructed, which is already capable of deriving and testing a user interface behavioral model of applications written in Java/Swing.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation (e.g.,HCI)**]: User Interfaces

General Terms

Design, Languages, Verification

Keywords

Graphical User Interfaces, Reverse Engineering, Models, Testing

1. INTRODUCTION

It is becoming increasingly important to ensure that Graphical User Interfaces (GUI) based applications behave as expected [14]. The correctness of the GUI is essential to the correct execution of the software [3]. Regarding user interfaces, correctness is usually expressed as usability: the effectiveness, efficiency, and satisfaction with which users can use the system to achieve their goals [19].

Tools are currently available to developers that allow for fast development of user interfaces [2] with graphical components. However, the design of interactive systems does not seem to be much improved by the use of such tools. Interfaces are often difficult to understand and use for end users [23]. In many cases users have problems in identifying all the supported tasks of a system, or in understanding how to achieve their goals. Moreover, the code produced by such tools is difficult to understand and maintain.

In the context of an ongoing effort to develop tools to support the automated analysis of interactive system designs[1], we are investigating the applicability of reverse engineering approaches to the derivation of user interface behavioral models amenable to verification of usability related properties.

Our objective consists in developing tools to automatically extract a model containing all possible GUI behaviors. The model must specify when a particular GUI event can occur, which are the related conditions, which system actions are executed and which GUI state is generated next. Additionally, we want to be able to reason and test this GUI model in order to analyse aspects of the original application's usability, and the quality of the implementation. Our first tool, named GUISURFER, is already capable of deriving and testing an user interface behavioral model of applications written in Java using a subset of the Swing toolkit.

This work will not only be useful to enable the analysis of existing interactive applications, but can also be helpful when an existing application must be ported or simply updated [13]. In this case, being able to reason at a higher level of abstraction than that of code will help in guaranteeing that the new/updated user interface has the same characteristics of the previous one.

In previous papers we have explored the applicability of slicing techniques [25] to our reverse engineering needs, and developed the building blocks for the approach (see [21, 22]) . The work was focused on extracting the models from Java/Swing source code. In this paper we extend that work in two directions. We present a generic model for language-independent reverse engineering of GUI-based applications, and we explore the integration of model-based testing techniques in our approach, thus allowing us to perform fault detection [12].

2. MOTIVATION

Throughout the paper we will use a Java/Swing interactive application as the running example. This application models an agenda of contacts: it allows users to perform the usual actions of adding, removing and editing contacts. Furthermore, it also allows users to find a contact through its name.

The interactive application consists of four windows, named *Login*, *MainForm*, *Find* and *ContactEditor* as shown in Figure 1.

The initial *Login* window (Figure 1, top-left window) is used to control the access of the users to the agenda. Thus, a login and password has to be introduced by the user. If the user introduces a valid login/password and presses the `Ok` button, then the login window closes and the main window of the application is displayed.

[1] http://www.di.uminho.pt/ivy

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii,U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

The final publication is available at the ACM DL via http://dx.doi.org/10.1145/1529282.1529307

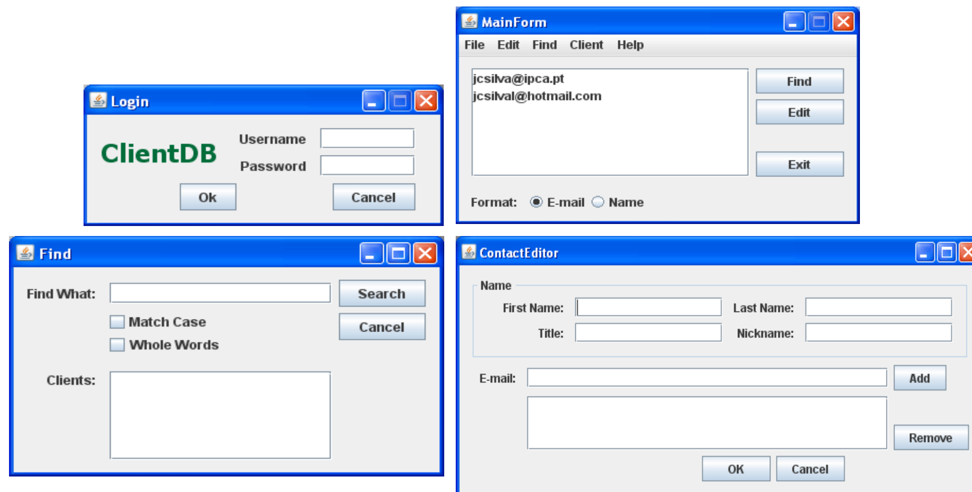

Figure 1: A Java/Swing application

On the contrary, if the user introduces an invalid login/password, then the input fields are cleared, a warning message is produced and the login window continues to be displayed. The Java fragment defining the action performed when the `Ok` button is pressed is as follows:

```
private void OkActionPerformed(...)
{if (isValid(user.getText(),pass.getText()))
 {new MainForm().setVisible(true);
  this.dispose();}
 else javax.swing.JOptionPane.showMessageDialog
           (this,"User/Pass not valid","Login",0);
}
```

where the method *isValid* tests the pair username/password inserted by users.

By pressing the *Cancel* button in the *Login* window, the user exits the application. Authorized users can use the main window (Figure 1, top-right window) to find and edit contacts (*Find* and *Edit* buttons). By pressing the *Find* button in the main window, the user opens the *Find* window (Figure 1, bottom-left window). This window is used to search and obtain a particular contact's data from his name. By pressing the *Edit* button in the main window, the user opens the *ContactEditor* window (Figure 1, bottom-right window). This last window allows the edition of all contact data, such as name, nickname, e-mails, etc. The *Add* and *Remove* buttons enable edition of the e-mail addresses list of the contact. If there are no e-mails in the list then the *Remove* button is automatically disabled.

Until now, we have informally described the (behavioral) model of our interactive application. Such descriptions, however, can be ambiguous and often lead to different interpretation of what the application should do. In order to unambiguously and rigorously define an application we can use a formal model. Moreover, by using a formal model to define the interactive application, we can use techniques to refactor, manipulate and test such applications. Figure 2 shows a possible formal model to specify the behavior of our running example: a finite state machine where states represent the GUI idle periods, i.e when there are no events or actions being executed (blue boxes such as state1, state2, ect), and the transitions between states are defined by the events associated to the GUI objects. These are modeled in Figure 2 by arrows like the labeled arrow *Press "Ok" button and valid user/pass*. Moreover, GUI actions executed when a particular event occurs are represented using red boxes like *Open "MainForm" window* box.

This model is less verbose than our initial informal description and easier to understand. For example, the action performed when the `ok` button is pressed defines a transition in the machine (c.f. Figure 1 and the Java code fragment): to the same state if the username/password is not valid, or into a different state, otherwise.

It should be noted that this machine is a deterministic finite state machine. From state 1, for example, the transition labeled with *Press "Ok" button and valid user/pass* moves to a different state (state2 from *MainForm* window) and execute two GUI actions: close the *Login* window and open the *MainForm* window.

Thus, we may use well-known finite state machine techniques to refactor this model. For example, by computing the equivalent smallest machine (i.e., a machine with the minimum number of states). Furthermore, we can also use techniques to detect properties of the interface. For example, we can use graph-based algorithms to compute if all the states are accessible from the initial one, in order to detect whether a particular window of the application will ever be displayed or not. We can also produce valid or invalid *sentences* of the language defined by the machine to use as test cases. These test cases can be used to prove more advanced properties of the interface. For example: "If there are no contacts in the agenda then users must not have access to *ContactEditor* window".

In this section we have shown an instance of applying our generic method to a Java/Swing application. Indeed, the GUI model in Figure 2 (apart from some beautifying) was automatically generated by our tool from the Java/Swing source code of the application. In the next sections, we will describe the techniques, models and tools that we have defined to achieve this result.

3. PROBLEM DOMAIN

In order to achieve our goal of developing a reverse engineering approach for user interfaces testing, we must first define both what type of user interfaces to test, and what type of model to generate [14, 18]. In this section, we present an overview of the graphical user interface we will focus on, and we describe related work.

The recent past has seen a lot of innovation in terms of interaction technology, with use of computers moving away from classical

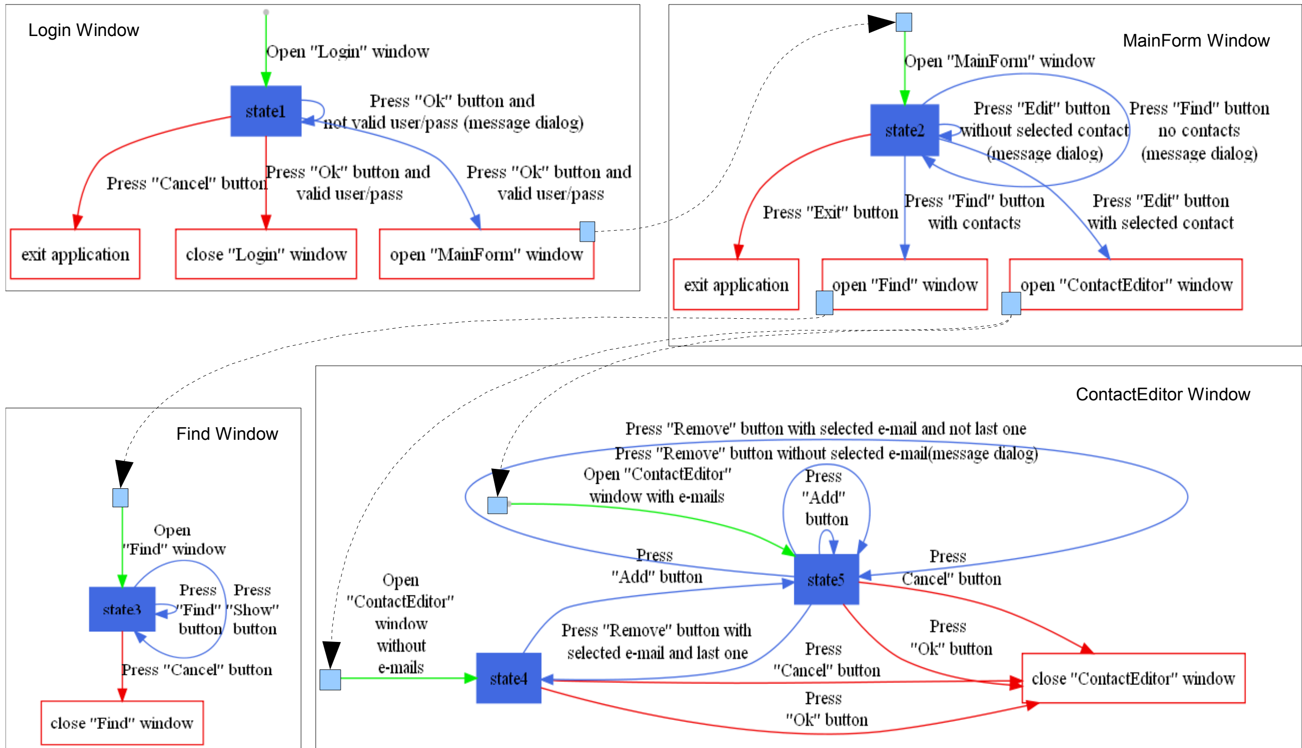

Figure 2: GUI behavioral model

WIMP[2] interfaces. The most usual class of user interface, however, are still WIMP-style interfaces, whose presentation structure consists of a hierarchy of objects creating a front-end to software systems, and use a event-based programming model to link the graphical objects to the rest of the system's implementation. These user interfaces produce deterministic output from user input, and system status and events.

A GUI contains graphical widgets (buttons, menus, textfields, etc). Each widget has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI. The software user interacts with the objects by performing actions that manipulate the graphical user interface widgets, thus generating events at the software level. Events cause deterministic changes to the state of the software. Briefly, from a user's perspective graphical user interfaces accept as input user-generated and system-generated events from a fixed set of events and produces a graphical output.

This paper focuses on techniques to reverse engineer this class of user interfaces. Our assumptions are the following:

1. An interactive system allows a dialogue between the computer and one or more users.

2. Interactive systems are event driven systems since the computer only responds after the user provides input, or some internal event has happened.

3. The gap between the computer and users is implemented by a graphical user interface.

4. User interfaces are composed of widgets which can be windows, buttons, textfields, etc.

5. The user interacts by performing operations on the widgets, and these operations originate events at the widget level.

[2]Windows, Icon, Mouse, and Pointer

4. MODEL-BASED GRAPHICAL USER INTERFACE TESTING

In order to achieve our goal of developing an approach for user interface manipulation and testing we introduce several techniques and demonstrate that source code may be transformed to a specification which captures their graphical user interface functionalities. A model-based testing approach is applied to reason about GUI behavior correct.

4.1 GUI Reverse Engineering

In [21] we presented techniques to reverse engineer Java interactive applications composed by one single window. In this paper, we extend our work and present a generic module for the reverse engineering of any GUI.

Using a parser, an Abstract Syntax Tree (AST) is obtained from the source code of the system for which the user interface related code is to be analyzed. Then we identify all fragments in the abstract syntax tree that are members of the GUI layer. We use the GUI constructors to focus the slicing in the subtrees that represent the GUI. Slicing is based on the program dependency graph. Figure 3 describes our approach to model-based GUI testing through a reverse engineering process. The module *GUI code slicing* extracts graphical user interface AST fragments through code slicing. This is a generic module to extract GUI fragments from any AST, i.e. Java/Swing, wxHaskell, C#, etc. This allows us to identify all of the program fragments that interacts with the graphical user interface. We do a traversal of the tree (based on the program dependency graph) and detect all GUI nodes. This process allows us

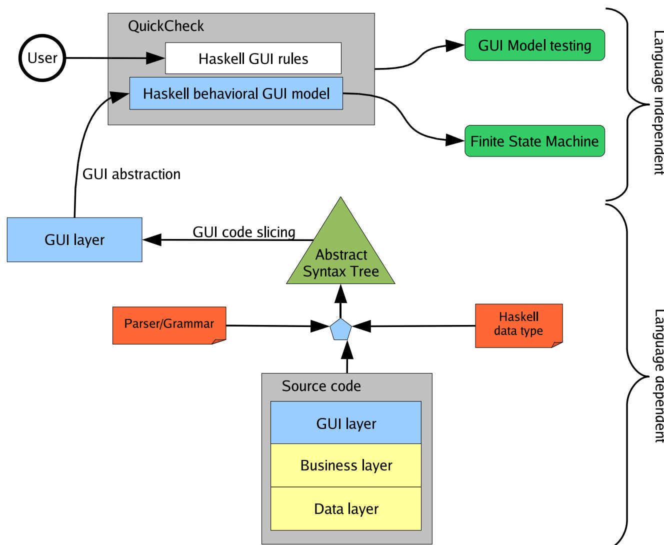

Figure 3: Model-based GUI testing process

to ignore unnecessary details and see just the user interface layer. Next, we describe our GUI code slicing methodology.

In order to extract the user interface behavior from the source code of the interactive applications, we need to construct a slicing function that isolates a sub-program from the entire program. The straightforward approach is to define a recursive function that traverses the AST of the program under consideration and returns the subtrees. Because we want to reuse our approach across different programming languages and paradigms, we need to use generic techniques that work for *any* AST and not for a particular language only. Thus, our re-enginnering combines two language-independent techniques, namely Strategic Programming (ST) [26, 27] and program slicing [25].

Strategic programming is a form of generic programming, allowing programmers to concisely define generic tree-traversal functions. Such strategic functions can work on different data types (*e.g.*, lists, binary trees, etc), and ASTs. Thus, ST provides the right setting to express our generic tree-traversal functions. Programs slicing is a technique for simplifying programs by focusing on selected aspects of semantics. Program slicing is also a generic technique that is able to slice programs of any programming language based on their program dependency graph [10].

Using strategic programming we make use of a pre-defined set of (strategic) generic traversal functions that traverse any AST using different traversal strategies (e.g. top-down,left-to-right, etc). Thus, the programmer has to focus in the nodes of interest, only. In fact, the programmer does not need to have a knowledge of the entire grammars/AST, but only of those parts he is interested in (the Swing sub-language in our case). As a result, the programmer does not need full knowledge of the grammar to write recursive functions that isolate the graphical user interface sub-program from the entire program.

To implement our approach we use the *Uminho Haskell Software* [1]: a set of libraries and tools developed in the *Haskell* [11] purely functional programming language. The software includes generic libraries for strategic programming and slicing. As a consequence, we use *Haskell* to develop a GUI code slicing library which contains a generic set of traversal functions that traverse any AST.

The following *Haskell* prototype function is a member of the *GUI code slicing* library:

```
slice :: AST -> Const -> InitPos -> SliceType ->
          [(Const,[AST],InitAST,EndAST)]
```

This function allows us to extract a list of fragments in a particular abstract syntax tree. Basically, the *slice* function receives four parameters:

- *AST*: The abstract syntax tree from any source code;
- *Const*: The constructor to be used to extract fragments by pattern matching;
- *InitPos*: The initial position in the AST for the code slicing process;
- *SliceType*: The slice type which can equal *1* or *2*. The first one, makes code slicing until any extraction, the last one continues code slicing within extracted sub trees.

The result is composed by all fragments in the abstract syntax tree that match the constructor.

As an example, to extract all button definitions from a Java/Swing source code's AST we could call the following instruction:

```
slice javaAST “JButton” 1 1
```

From a WxHaskell source code's AST, the same action could be executed as:

```
slice wxHaskellAST “button” 1 1
```

This is a generic function which is configured with the AST and the constructor pattern to be extracted.

At this point, we have a set of abstract syntax tree fragment that just consists of instructions that affect the user interface. Anchor points for these fragments are detected by syntactic pattern matching.

Figure 3 shows also that module *GUI abstraction* uses these fragments to produce a behavioral user interface description (*Haskell behavioral GUI model*). The fragments relevant to the GUI reverse engineering are limited to graphical user interface instructions, control flow information and methods invocation.

Thus, the problem of understanding the interface has been reduced to the problem of understanding the slice with respect to a certain user interface component.

To explain the developed GUI code slicing module in more detail, let us consider the following Java/Swing code fragment, which defines a new button through the *JButton* class:

```
addButton = new javax.swing.JButton();
```

After parsing this code fragment we obtain the following fragment of the AST:

```
Statem( Exps( Eassign( Evar [Ident add]), Assign(
Enewalloc( Anewclass( ClassType [Ident javax, Ident
swing, Ident JButton]) (Args [])))))
```

Having the knowledge of this particular fragment of the Java grammar/AST, we are able to define a function that given the complete AST extracts all *JButton* object assignments. First, we need to collect the list of assignments in a Java program. We define this function in *Haskell* as follows:

- We make use of the *slice* function in order to define a function that traverses the AST in a top-down fashion.

- Next, we need to define the slicing parameter to use while traversing the AST. This parameter identifies the tree nodes where work has to be done. In the complete Java AST the nodes of interest correspond to the constructor `Eassign` (see AST above). Thus, our slice function simply returns a singleton list with the left-hand side of the assignment and the respective expression. All the other nodes are simply ignored!

This function, named *statementsAssignment*, looks as follows:

```
statementsAssignment ast = slice ast "Eassign" 1 1
```

Having collected the list of assignments we can now filter that list also by code slicing in order to produce the list containing all *JButtons* objects in the Java/Swing code.

The code slicing library works for any abstract syntax tree and not only for the Java AST under consideration in this paper. As a result, the function we define not only extracts the Swing fragment from a Java program, but may also be re-used to slice another GUI toolkit for other languages/ASTs.

Obviously, we can easily configure these functions language-specific constructors.

Another important task is the extraction of the instructions list executed from a particular source code anchor point. In other words, to implement this reverse engineering process from source code we must extract the instructions sequence executed when a particular event occurs. This information is obtained through code slicing within a particular instruction block and considering all external invocations. As an example the function implemented to extract all external invocations from an AST is:

```
statementsExternalMethod ast = slice ast "Emth" 1 2
```

Applying several generic functions the methodology returns a transitive instructions closure (a partial program dependency graph).

From these fragments of the original AST it's finally possible to extract the GUI layer and reason about it.

4.2 Models for GUI reasoning

The reverse engineering approach of Figure 3 is based on three steps: GUI layer extraction, GUI behavioral abstraction and finally model-based GUI testing. The first one, as described in the above section, allows us to extract the graphical user interface layer. Understanding the code related to the user interface consists of recovering a plausible specification by abstracting the user interface structure and behavior. So, the overall model-based GUI testing process will extract only the GUI code layer.

In order to define the GUI slicing code mentioned above, we defined a small set of abstractions for the interactions between the user and the system. The abstractions that we look for in the source code are *user inputs*, *user selections*, *user actions* and finally *output to User*.

Thus, we look for any widget that enables users to input data (*user input*), any widget that enables users to choose between several different options such as a command menu (*user selection*), any action that is performed as the result of user input or user selection (*user action*) and any widget that enables communication from application to users such as a user dialogue (*output to user*).

Given the user interface code of an interactive system and this set of abstractions, we can generate its graphical user interface abstraction. To execute this step we combine the GUI code slicing library with techniques and tools for specifying and verifying systems.

From the sliced source code, we extract a GUI behavioral model of the interface. Next we present the developed GUI Haskell model (*GuiModel*):

```
type EventRef = String
type CondRef = String
type WindowName = String
type ExpRef = Int
type GuiModel = Map (EventRef,CondRef) [ExpRef]
type Pres = Map ExpRef (EventRef,Bool)
type End = [ExpRef]
type Close = [ExpRef]
type NewWindow = Map ExpRef WindowName
```

Basically this model specifies a window behavior through a partial finite mapping (*GuiModel*) from pairs of events and condition references to a sequence of actions references. Each $EventRef$ and $CondRef$ data links to a particular event subtree from the original abstract syntax tree. $EventRef$ links to events and $CondRef$ links to conditional Java code information. In other hand End meta-model contains a list of all exit actions and *Close* contains a list of all window close actions. *NewWindow* contains all actions which open a new window. Finally, $Pres$ allows us to recognize actions which change events status. As an example a button press event could be disabled by an particular action[3].

To describes the $GuiModel$ we will consider the login window from Figure 1. Basically, the *login* window allows to validate users through username and password. The window contains several widgets allowing user authentication.

Applying the prototype to the *Login* window's code, enables us to extract information about all widgets presented at the interface, such as *JButton*, *JLabel*, *JTextField*, etc. Once the AST for the application code is built we can apply different slicing operations as needed. These are filters that allow us to analyse particular GUI components.

In this case study, we will consider for simplicity only the *JButton* widget. Applied to the code of the *login* window and considering this GUI filter, the GUISURFER tool automatically generates a *Haskell* GUI behavior specification (*GuiModel*) including the initial application state (*init*), events (press *Ok* button, press *Cancel* button, etc) and respective conditions:

```
guimodel::GuiModel
guimodel=fromList
 [(("Cancel","cond1"),[1]),(("Ok","cond2"),[2,3]),
 (("Ok","cond3"),[4]),
 (("init","condInit1"),[5,6,7,8,9])]
pres::Pres
pres=fromList [(8,("Cancel",True)),(9,("Ok",True))]
end::End
end=[1]
newWindow::NewWindow
newWindow=fromList [(2,"MainForm"),(5,"Login")]
```

In this case study, events reference *Ok* and *Cancel* describe respectively the press *Ok* button and press *Cancel* button actions.

The *Pres* function specifies actions which change events status. These actions are related with all button states:

```
[(8,("Cancel",True)),(9,("Ok",True))]
```

As an example, action reference 8 enables the *Cancel* button. *End* function contains the exit action reference.

Considering the *Ok* event representation $(("Ok","cond2"),[2,3])$, $(("Ok","cond3"),[4])$, we can see that there are two sequences

[3] In the Java language programming we use the *setEnabled* method.

of action references associated. These are "$[2,3]$" and "$[4]$". The execution of one sequence of actions depends on its condition's logical value (*cond2* or *cond3*). These conditional expressions are automatically extracted from source code. To explain this extraction process, let us study the following associated Java source code (executed when *Ok* button is pressed):

```
private void OkActionPerformed(...)
{if (isValid(user.getText(),pass.getText())==true)
    {new MainForm().setVisible(true);
     this.dispose();}
 else javax.swing.JOptionPane.showMessageDialog
          (this,"User/Pass not valid","Login",0);
}
```

In *GuiModel*, the *OkActionPerformed* method's content is reduced to the expression $if\ (cond2)\ [2,3]\ else\ [4]$ where:

- $cond2$ represent the following condition: $(isValid(user.getText(), pass.getText()) == true)$
- $cond3$ represent the negation of the above condition;
- Expression reference [2] correspond to the instruction: $newMainForm().setVisible(true)$;
- Expression reference [3] correspond to the instruction: $this.dispose()$;
- Expression reference [4] correspond to the message dialog instruction within else statement.

Thus *OkActionPerformed* method execution causes two distinct cases in *GuiModel*: The first one occurs when condition $cond2$ is true (("Ok","cond2"),[2,3]) and the second one when $cond2$ is false (("Ok","cond3"),[4])

Each $GuiModel$ data (events, conditions and actions) are linked to their tree representation in the original abstract syntax tree. So, at any time it's possible to consult their internal structure and visualize the associated source code.

As we can see each state, event or condition in the behavioral model is related to it abstract syntax tree fragment. Therefore, this will allow us in the future to make a refactoring or transformation process to the source code.

Currently, the tool is also capable of automatically generating a finite state machine (FSM) model of the interface [20]. Next we describe an example of the FSM that we can automatically generate from the application source code (considering the particular GUI filter above described).

In the finite state machine of Figure 4, each state defines a representation of a GUI window in one particular period of time. The arrow specifies an event moving from one state to another. All events are associated to a particular condition and sequence of actions such as in the *GuiModel*. To move from one state to another, the associated condition (*cond*) must be validated. Conditions are extracted directly from conditional instructions in the source code. As an example, an event's *if condition then action1 else action2* conditional instruction is abstracted to two different states. A first one with the *actions1*'s graphical user interface instructions set that occurs when the condition is true, and a second one with the *action2*'s graphical user interface instruction set.

As we can see, the GUISURFER prototype extracts automatically a GUI model like the model described in Figure 2 from section 2. The prototype achieves our main objective. We generate automatically a GUI behavioral model directly from an application's source code. For each application window, we generate a behavioral model which describes its window states, events, conditions and actions.

We now want to be able to reason about this GUI model. We will show how we can use testing techniques to analyse aspects of the original application's usability characteristics.

4.3 GUI Testing

The reverse engineering approach described in this paper allows us to extract an abstract GUI behavior specification. Our next goal is to perform model-based GUI testing [4]. To this end, we make use of the *QuickCheck Haskell* library tool. QuickCheck [6] is a tool for testing *Haskell* programs automatically. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in *Haskell*, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

Considering the application described in the previous section and its abstract GUI model-based we could now write some rules and test them through the *QuickCheck* tool.

To illustrate the approach, we will test if the application satisfies the following rule: users can only access the following windows *Login*, *MainForm*, *Find*, *ContactEditor*. We consider a *GuiModel* representation for each application's window.

The rule is specified in the *Haskell* language on top of the *GuiModel* windows cases. From the *GuiModel* windows set we automatically generate randomly cases. We extract valid GUI sentences from a GUI behavioral model. Then the rule is tested in a large number of cases (10000 in this GUI testing process!). The number of random cases and event lengths are specified by the GUISURFER user. Each random case is a sequence of valid events associated with their conditions, actions and the respective window. In other words, each case is a sequence of possible events, so all respective conditions are true in this context. As example:

```
[("Login","Ok","cond2",[2,3]),
 ("MainForm","Find","cond3",[3]),
 ("Find","Search","cond1",[]),
 ("Find","Cancel","cond2",[1]),
 ("MainForm","Exit","cond1",[1])]
```

In this particular sentence, the user presses *Ok* button in the *Login* window with a valid username/password (condition reference *cond2* represent the *isValid* method, cf. section 2). Associate actions references 2 and 3 represent respectively *Login* window closing and *MainForm* window opening. Then from *MainForm* window, the user presses the *Find* button (condition reference *cond3* test if there are contacts in the agenda). In the *MainForm* window action reference 3 corresponds to *Find* window opening action. From *Find* window, the user makes a search and then cancel the window. Finally with the last action, the user exits the application pressing the *Exit* button ("MainForm","Exit","cond1",[1]).

Considering *vs* a valid sentence, we can specify the above rule as:

```
rule vs = [a|(a,b,c)<-vs] ==
  ["Login","MainForm","Find","ContactEditor"]
```

Testing through *QuickCheck* the application's *GuiModel* with this rule , we obtain the following result:

```
OK, passed 10000 tests.
87% events sequence length: 5.
11% events sequence length: 4.
1.5% events sequence length: 3.
0.5% events sequence length: 2.
0% events sequence length: 1.
```

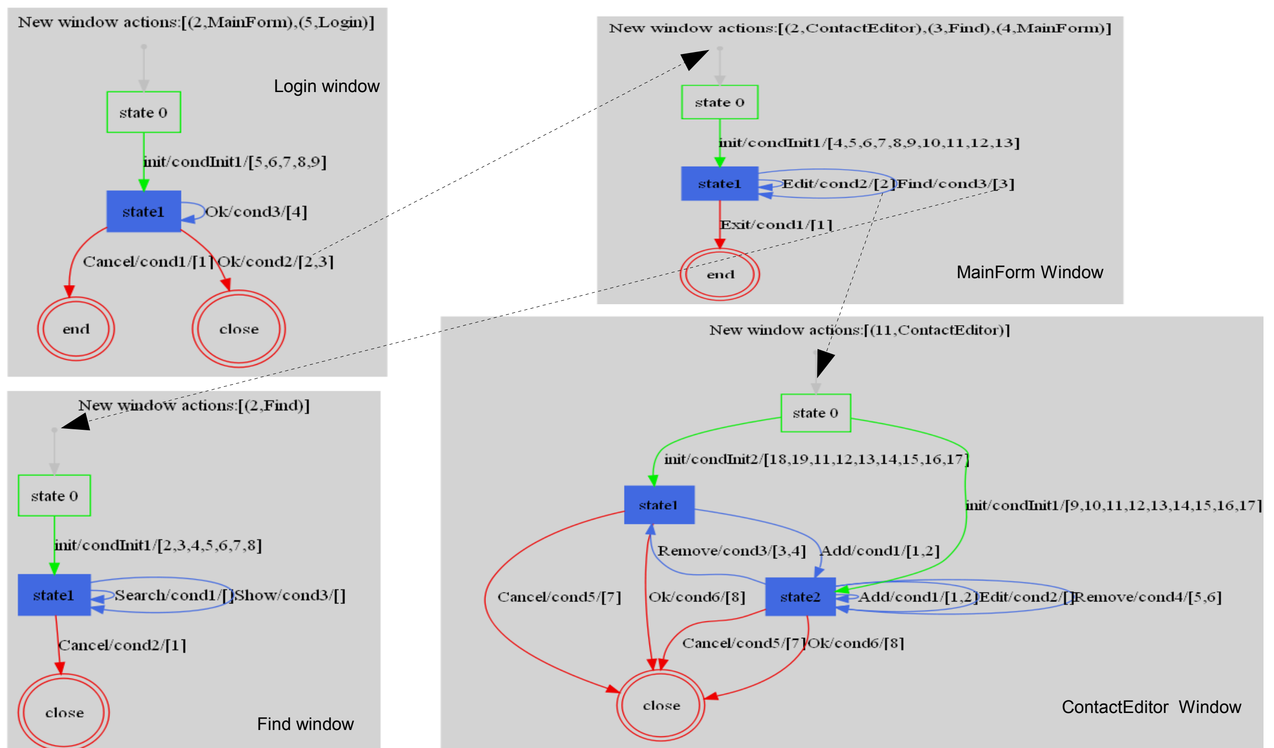

Figure 4: Application's GUI state machine

The rule was tested in 10000 randomly generated cases. All of them satisfies the rule.

Five maximum sequence length events correspond to 116650 different cases. We check the rules with 10000 of them. Obviously we can check the rule with a wide range of cases. We demonstrate here that it is possible to analyse a GUI model using a model-based testing technique. Though our approach is non-exhaustive, this is a technique which allows us to test the quality of models at a lower cost than other exhaustive techniques such as model checking.

This paper's focus is on GUI testing. Coverage criteria for GUIs are important rules that provide an objective measure of test quality. We plan to include coverage criteria to help determine whether a GUI has been adequately tested. These coverage criteria use event sequences to specify a measure of test adequacy. Since the total number of permutations of event and condition sequences in any GUI is extremely large, the GUI's hierarchical structure must be exploited to identify the important event sequences to be tested.

5. RELATED WORK

In the Software Engineering area, the use of reverse engineering approaches has been explored in order to derive models directly from the existing interactive system.

Different approaches have been used. One possibility is to use dynamic analysis. A typical approach is to run the interactive system and automatically record its state and events. Memon et al. [15] describe a tool which automatically transverses a user interface in order to extract information about its widgets, properties and values. Chen et al. [5] propose a specification-based technique to test user interfaces. Users graphically manipulate test specifications represented by finite state machines which are obtained from running the system. Systa studies the run-time behavior of Java software trough a reverse engineering process [24]. Running the target software under a debugger allows for the generation of state diagrams. The state diagrams can be used to examine the overall behavior of a component such as a class, an object, or a method. Paiva proposes a process to reverse engineer structural and behavioral formal models of a GUI application by a dynamic technique. A skeleton of a state machine model of the GUI is generated automatically and represented in a specification language. From this model abstract test cases are generated and executed over the GUI application [17].

Another alternative is the use of statical analysis. The reverse engineering process is based on analysis of the application's code, instead of its execution, as in the previous approaches. One such approach is the work by d'Ausbourg et al. [7] in reverse engineering UIL code (User Interface Language – a language to describe user interfaces for the X11 Windowing System, see [9]). In this case, models are created at the level of the events that can happen in the components of the user interface (e.g., pressing a button).

Moore [16] describes a technique to partially automate reverse engineering character based user interfaces of legacy applications. The result of this process is a model for user interface understanding and migration. The work shows that a language-independent set of rules can be used to detect interactive components from legacy code. Merlo [8] proposes a similar approach. In both cases static analysis is used.

In this work, we are using static analysis as in [16]. When compared to their work our challenges are reverse engineering code for graphical user interfaces, as opposed to character based user interfaces in [16].

At the moment we are working with Java/Swing. However, our approach is generic and can be applied to any programming lan-

guage. We are interested in models that reflect the design of the user interface and the interaction that it creates. Hence, we need models that are more abstract than those produced in, for example, [15].

6. CONCLUSIONS AND FUTURE WORK

The major contribution of this work is the development of the GUISURFER prototype, an approach for improving model-based Java/Swing applications testing through reverse engineering. The model-based testing of interactive systems is a complex task. Therefore, many contributions have been achieved. These contributions are the identification of common characteristics of user interfaces, the abstraction of interactive systems, the development of a model-based testing prototype and the description of a case study testing an abstract user interface model through the reverse engineering process.

We demonstrated how the user interface layer can be extracted from any source code. We identified a set of widgets (graphical objects) that can be modeled. We identified also a set of user interface actions. Finally, we presented a methodology to extract a behavioral and test user interface model.

The work presents an approach to the reverse engineering of GUI applications. The approach is very flexible, indeed we have already applied the same techniques to extract similar models from *Haskell/WxHaskell* interactive applications. From the abstract syntax tree representation we are able to derive a GUI meta model and state machine. Theses models enables us to reason about both usability properties of the design, and the quality of the implementation of that design. Our objective has been to investigate the feasibility of the approach.

Coverage criteria and the refactoring/transformation of users interface code from the user interface aspects modeled in *GuiModel* are important features to implement as future work.

Acknowledgments

This work was partially supported by the Portuguese Research Foundation (FCT) and FEDER (European Union) under contracts: `POSC/EIA/56646/2004`, `SFRH/BD/30729/2006` and `SFRH/BSAD/782/2008`.

7. REFERENCES

[1] UMinho Haskell Software – Libraries and Tools - http://wiki.di.uminho.pt/twiki/bin/view/research/pure/puresoftware.

[2] G. Abowd, J. Bowen, Alan Dix, M. Harrison, and R. Took. User interface languages: a survey of existing methods. Technical report, Oxford University, 1989.

[3] B. Berard. *Systems and Software Verification*. Springer edition, 2001.

[4] Peter Bumbulis. *Combining Formal Techniques and Prototyping in User Interface Construction and Verification*. PhD thesis, University of Waterloo, 1996.

[5] J. Chen and S. Subramaniam. A gui environment for testing gui-based applications in java. *Proceedings of the 34th Hawaii International Conferences on System Sciences*, 2001.

[6] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP, ACM SIGPLAN, 2000*, 2000.

[7] Bruno d'Ausbourg, Guy Durrieu, and Pierre Roché. Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour. In *DSV-IS 96*. 1996.

[8] Merlo E., Gagne P. Y., Girard J.F., Kontogiannis K., Hendren L.J., Panangaden P., and De Mori R. Reengineering user interfaces. *IEEE Software, 12(1), 64-73*, 1995.

[9] Dan Heller and Paula M. Ferguson. *Motif Programming Manual*, volume 6A of *X Window System Seris*. O'Reilly & Associates, Inc., second edition, 1994.

[10] Susan Horwits and Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. In *14th International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, may 1992. ACM.

[11] Simon Peyton Jones, John Hughes, Lennart Augustsson, et al. Report on the programming language haskell 98. Technical report, Yale University, February 1999.

[12] Clark Edmund M. and Jeannette Wing M. *Formal Methods: State of the Art and Future Directions*. Carnegie mellon university edition, 1996.

[13] Moore Melody. A survey of representations for recovering user interface specifications for reengineering. Technical report, Institute of Technology, Atlanta, 1996.

[14] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. PhD thesis, Department of Computer Science, University of PittsBurgh, july 2001.

[15] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. Technical report, University of Maryland,USA, 2003.

[16] M. M. Moore. Rule-based detection for reverse engineering user interfaces. *Proceedings of the Third Working Conference on Reverse Engineering, pages 42-8, Monterey, CA*, november 1996.

[17] Ana C. R. Paiva, João C. P. Faria, and Pedro M. C. Mendes, editors. *Reverse Engineered Formal Models for GUI Testing, 10th International Worksho on Formal Methods for Industrial Critical Systems Berlin, Germany, July 1-2*, 2007.

[18] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, 2000.

[19] ISO/TC159 Sub-Commitee SC4. Draft International ISO DIS 9241-11 Standard. International Organization for Standardization, September 1994.

[20] R.K. Shehady and D.P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceeding of the 27th International Symposium on Fault-Tolerant Computing*, 1997.

[21] J.C. Silva, José Creissac Campos, and João Saraiva. Combining formal methods and functional strategies regarding the reverse engineering of interactive applications. In *DSV-IS 2006, Dublin, Irland*. Springer, 2006.

[22] J.C. Silva, José Creissac Campos, and João Saraiva. Models for the reverse engineering of java/swing applications. *ATEM 2006, Genova, Italy*, October 2006.

[23] A. G. Sutcliffe. *Human-Computer Interface Design*. MacMillan, 2nd edition, 1995.

[24] T. Systa. Dynamic reverse engineering of java software. Technical report, University of Tampere, Finland, 2001.

[25] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, september 1995.

[26] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. 2003.

[27] Joost Visser and João Saraiva. Tutorial on strategic programming across programming paradigms. In *8th Brazilian Symposium on Programming Languages*, Niteroi, Brazil, May 2004.