The GUISurfer tool: towards a language independent approach to reverse engineering GUI code

João Carlos Silva
jcsilva@ipca.pt

Carlos Silva
carlosebms@gmail.com

Rui Gonçalo
uch.073@gmail.com

João Saraiva
jas@di.uminho.pt

José Creissac Campos
jose.campos@di.uminho.pt

Departamento de Informática/CCTC
Universidade do Minho, Campus de Gualtar
4710-057 Braga, Portugal

ABSTRACT
Graphical user interfaces (GUIs) are critical components of today's software. Developers are dedicating a larger portion of code to implementing them. Given their increased importance, correctness of GUIs code is becoming essential. This paper describes the latest results in the development of GUISurfer, a tool to reverse engineer the GUI layer of interactive computing systems. The ultimate goal of the tool is to enable analysis of interactive system from source code.

Keywords
Graphical User Interfaces, Source code, Analysis

ACM Classification Keywords
H.5.2 User Interfaces: Graphical user interfaces (GUI). D.2.7 Distribution, Maintenance, and Enhancement: Restructuring, reverse engineering, and reengineering.

General Terms
Languages, Reliability, Human Factors

INTRODUCTION
Model-based development of software systems, and of interactive computing systems in particular [6], promotes a development life cycle in which models guide the development process, and are iteratively refined until the source code of the system is obtained. Models can be used to capture, not only the envisaged design, but also its rational, thus documenting the decision process undertook during development. Hence, they provide valuable information for the maintenance and evolution of the systems.

It is not always the case, however, that models exist for a given piece of software. Indeed, not all software development approaches promote the use of models as highlighted above (e.g. Agile Development methods place relatively little emphasis on documentation). In the specific case of interactive computing systems, the *de facto* standard for modeling software (UML [2]) do not cater for the adequate modeling of user interfaces. Even so, it is still the case that models would help in the maintenance and evolution of such software.

With the above in mind, we are working on the development of tools to automatically extract models from the user interface layer of interactive computing systems' source code. To make the project manageable we focus on event-based programming toolkits for graphical user interfaces (GUI) development (Java/Swing being a typical example).

The goal is that extracted models will enable the analysis of existing interactive applications. This might be required to ascertain the quality of a given implementation, when an existing application must be ported, or simply updated [5].

A particular emphasis is being placed on developing tools that are, as much as possible, language independent. Through the use of generic programming techniques, the developed tool aims at being targetable to different user interface programming languages and toolkits (possibly from different programming paradigms – e.g. object oriented or functional), from professional to end-user programmed interactive systems (e.g. spreadsheets). At this time, our tool is able to reverse-engineer Java (either with Swing or GWT) and Haskell application's source code.

Our goal is to be able to extract a range of models from the source code. In the present context we focus on finite state models that represent the behavior of GUI. That is, when can a particular GUI event occur, which are the related conditions, which system actions are executed, or which GUI state is generated next. We choose this type of model because we want to be able to reason and test this GUI model in order to analyze aspects of the original application's usability, and the quality of the implementation.

In previous papers [7, 8] we have explored the applicability of slicing techniques to our reverse engineering needs, and developed the building blocks for the approach. In this paper we explore both the integration of testing techniques

EICS'10, June 19–23, 2010, Berlin, Germany.

allowing us to reason about GUI models, and the feasibility of targeting the tool to different programming languages.

The paper uses a simple agenda application as running example.

THE AGENDA EXAMPLE

Throughout the paper we will use an interactive agenda of contacts as a running example. The system allows users to perform the usual actions of adding, removing and editing contacts. Furthermore, it also allows users to search for a contact by its name. This particular example was chosen because it has enough features to allow us to demonstrate the approach in the scope of this paper.

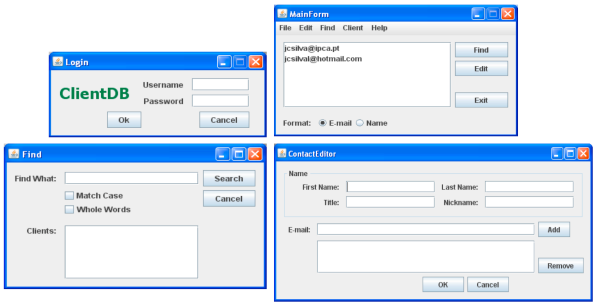

Figure 1 – The agenda application

Each agenda consists of four windows. Namely, `Login`, `MainForm`, `Find`, and `ContactEditor`. Figure 1 presents the windows of a Java/Swing implementation. At application start, the Login window (Figure 1, top-left window) is used to control users' access to the agenda. A login and password have to be provided by the user. If the user introduces a valid login/password pair, and presses the `Ok` button, then the login window closes and the main window of the application is displayed. If the user introduces an invalid login/password pair, then the input fields are cleared, a warning message is produced, and the login window continues to be displayed. 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, the user opens the Find window (Figure 1, bottom-left window). This window is used to search for a particular contact's data by name. By pressing the `Edit` button, 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.

A first implementation was done in Java/Swing [4]. The implementation has 821 lines of code, and was developed with the NetBeans IDE. GUI code was generated automatically by the IDE. The only change introduced in the generated code was the explicit setting of the visibility attribute of the buttons in the interface. Currently GUISurfer (see below) does not assume the initial state of buttons to default to visible, but this can be easily fixed.

GUISURFER

In order to achieve our goal of developing an approach for reverse engineering of GUI source code, we have started the development of the GUISurfer tool. GUISurfer resorts to a number of techniques to make it easier to achieve easily re-targetable reverse engineering of GUI source code. Figure 2 describes our approach. Using a parser for the relevant programming language, an Abstract Syntax Tree (AST) is obtained from source code. In order to subsequently extract the user interface from the AST, we need to construct a function that isolates a sub-program from the entire program. Because we want to reuse our approach across different programming languages and paradigms, we need to use generic techniques that work with *any* AST and not with the AST of a particular programming language only. Thus, our reverse engineering approach combines two language-independent techniques: strategic programming (ST) [11, 12], and program slicing [10].

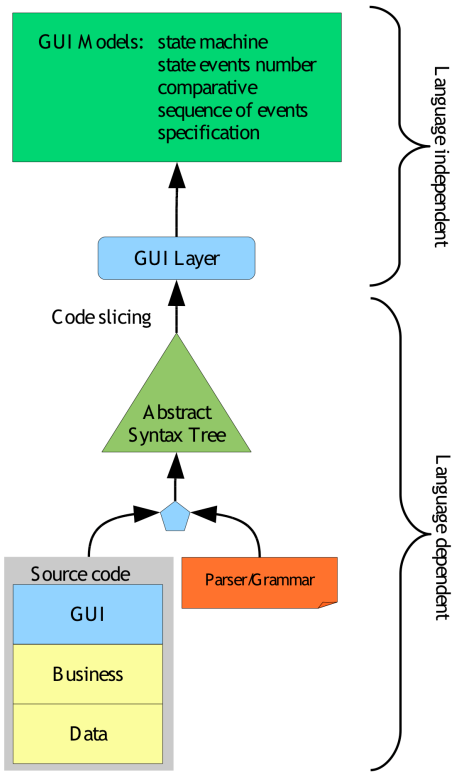

Figure 2 - Model-based GUI reasoning process

GUI components' constructors are used to focus the slicing in the subtrees that represent the GUI. The *GUI code slicing* module performs this code slicing of relevant GUI AST fragments. It performs a traversal of the tree (based on the program dependency graph) in order to detect all GUI nodes. This is a generic module to extract GUI fragments from any AST, i.e. Java/Swing, WxHaskell, C#, etc.

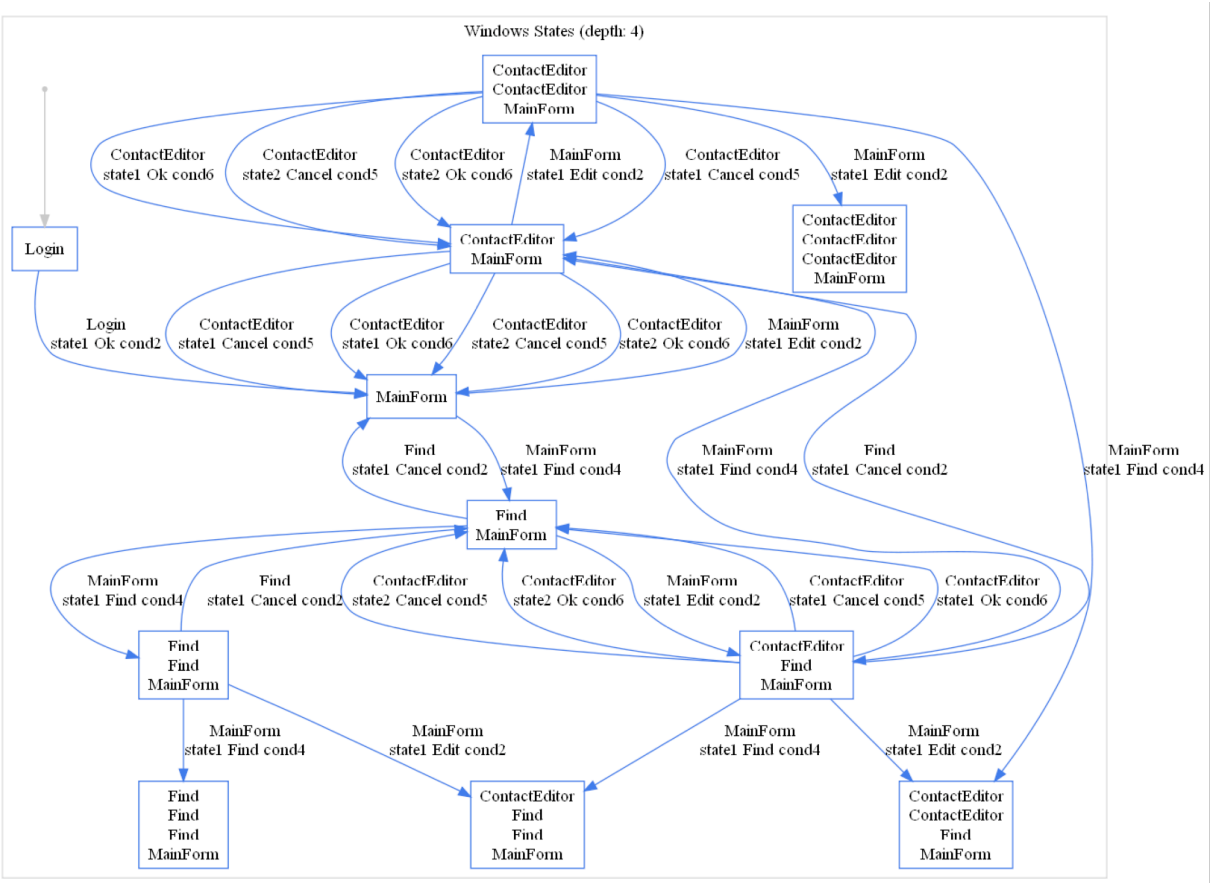

Figure 3 - Agenda GUI state machine

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). These functions enable us to focus on nodes of interest, only. In fact, we do not need to have knowledge of the entire grammar/AST, only of those parts that are of interest (the GUI toolkit sub-language in our case). As a result, we do not need full knowledge of the grammar to write recursive functions that isolate the graphical user interface sub-program from the entire program.

Using the described approach we aim to make the manipulation of the AST easily re-targetable to different programming languages and GUI toolkits. While building the AST is clearly language dependent, regarding code slicing, only the GUI components constructors need to be specified.

The components that we look for in the source code are widgets that enable users to input data (*user input*), widgets that enable 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 the application to users (*output to user*).

Given the user interface code of the interactive system and a list of relevant GUI components, we can generate its graphical user interface abstraction. GUISurfer receives the list of components to look for as a parameter, meaning it is possible to extract models at different levels of detail. From models focused on specific types of components, up to models of the complete interface.

GUISurfer is composed by three tools: `FileParser`, `AstAnalyser`, and `Graph`. These tools are configurable through command line parameters. Below we outline some of the more important parameters for each tool. For more details on the techniques behind these tools see [7, 8].

The `FileParser` tool is used to parse a particular source code file. For example, the command "`FileParser Login.java`" allows us to parse the Login class from the Agenda application. As a result, we obtain its AST.

The `AstAnalyser` tool is used to slice an abstract syntax tree, considering only it graphical user interface layer. It consists of a slicing library, containing a generic set of traversal functions, and has been implemented in *Haskell*. This tool must be used with three arguments, i.e. the abstract syntax tree, the entry point in source code, and a list with all widgets to consider during the GUI slicing process. The command "`AstAnalyser Login.java.ast main JButton`" let us extract the GUI layer from Login.java abstract syntax tree, starting the slice process from main method, and extracting only "JButton" related data. The tool generates two files "`initState.gui`" and "`eventsFromInitState.gui`" which contain initial states and events from initial states, respectively.

Finally, the `Graph` tool receives as arguments the "`initState.gui`" and "`eventsFromInitState.gui`" files, and generates several metadata files with events, conditions, actions, and states extracted form source code. Each of these types of data is related to a particular fragment from the AST. Another important outputs generated by the `Graph` tool are the "GuiModel.hs" and "GuiModelFull.hs" files. These are GUI specifications written in the Haskell programming language. These specifications define the GUI layer mapping events/conditions to actions. Finally, this last tool allows us also to generate several visual models through the GraphViz tool, such as state machines, behavioral graph, etc.

GUI MODELS

Interactive systems can be represented as directed graphs [5]. User actions are mapped into arcs and states are application GUI idle time. Figure 3 presents a directed graph describing the agenda application GUI behavior. This graph was generated from the Java/Swing implementation. User actions are mapped into arcs and states describe sets of active windows. Each transition has the generic form <window><state><action><condition>. The meaning of each element is the following: <window> identifies the window in which the action occurred; <state> identifies a specific internal state of the window; <action> identifies a user action; <condition> identifies a condition that must hold for the transition to occur.

In Figure 3, window states and conditions are identified by name. These names reference AST fragments that are identified in the metadata files generated by GUISurfer. Hence, the top left corner of the figure shows that, initially, only the login window is presented to users. If the user presses OK, and condition cond1 (the login/password pair is valid) is verified then, the interface will present the MainForm window. From their other windows can be opened.

One aspect that can be immediately noticed is that application does not constrain the number of Find and ContactEditor windows that can be open at the same time (the left bottom state corresponds to three Find windows and the MainForm window being displayed – the model was generated considering a depth of up to four user actions). While it might make sense to edit several contacts at the same time, performing several searches simultaneously might be problematic and would deserve some investigation. This illustrates how simple visual inspection of the graph already provides valuable information.

As referred to above, information about the internal states of each window is also extracted. Figure 4 illustrates a type of model identifying the internal states for each application window, together with the total number of events associated with each state. This is useful as a metric to detect windows complexity.

A number of approaches can be used to reason about the system from the generated models. 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.

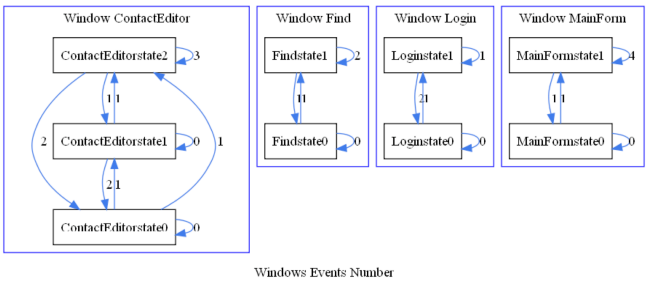

Figure 4 - GUI state events number

ONGOING WORK

GUISurfer is already able to reverse engineer Java/Swing applications (limitations are analyzed in the Discussion section). This section puts forward a number of improvements we are currently working on.

Support for Analysis

As already stated, using graphs to model Graphical User Interfaces opens up a number of venues for analysis [9]. These range from using graph theoretic concepts such as the shortest path between two vertices (which can be seen as defining the most efficient way a user can achieve a particular change of state), through testing relevant properties of the behavior of the system (as described by the graph), up to formal verification of such properties (c.f. using model checking [3]).

Graph operators

In order to support graph based analysis, we are extending GUISurfer with graph operators. At this time we have implemented intersection, union and difference of graphs. This has proven particularly useful in comparing versions of an application, allowing analysis of whether different versions have the same behavior.

Consider a new version of the Agenda application, where the Contact Editor form was for some reason left out. To simulate this we simply remove the following Java/Swing instruction from the code:

```
new ContactEditor().setVisible(true);
```

Using the newly introduced graph difference operator we are able to obtain the behavioral differences between the two versions of the application. In this case we obtain the graph of the Figure 5. The graph shows explicitly all behavioral differences between these two Agenda versions (bold transitions). In this case it can be seen that accessing the contact editor can be done in the first version, but not in the second. Either the change was intentional, and the analysis confirms that it was successfully accomplished, or it was unintentional, and the analysis detects a problem in the new version of the software.

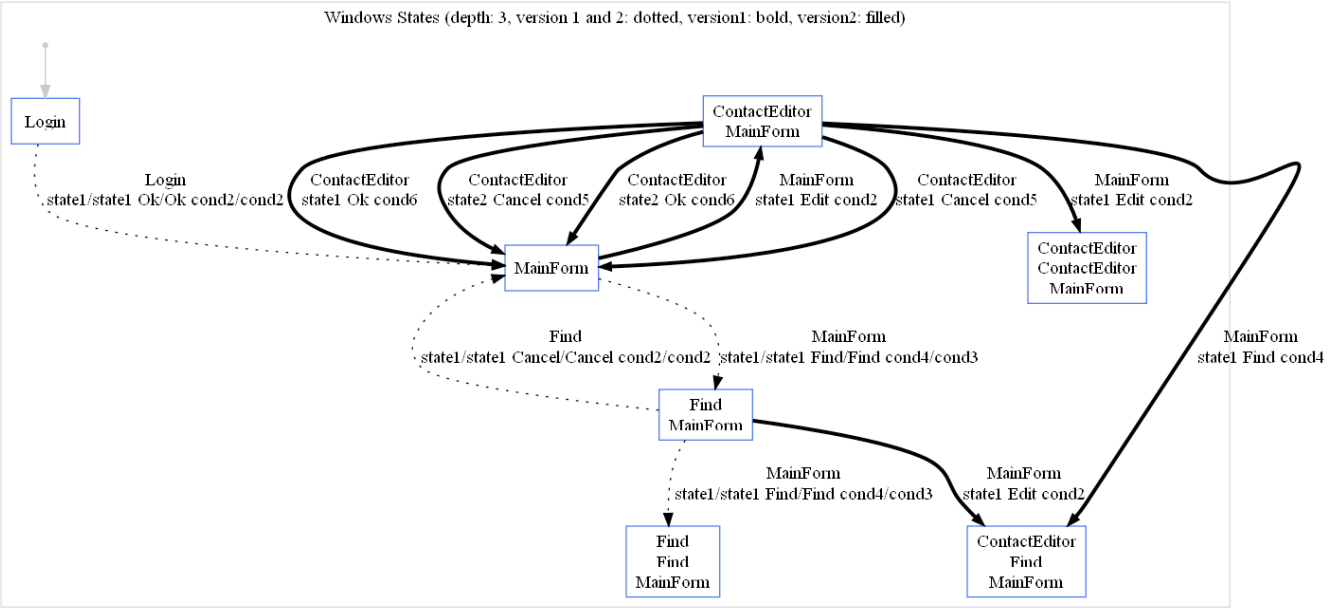

Figure 5 - Agenda GUI behavioral graph (using difference operator)

Testing

In order to support testing of user interface properties over the extracted behavioral models, we make use of the QuickCheck Haskell library tool. QuickCheck [1] is a tool for the automated testing of Haskell programs. The programmer provides a specification of the program, in the form of properties that functions should satisfy, and QuickCheck then tests whether 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.

Consider again the version of the agenda without the ContactEditor window. If we run a set of tests in QuickCheck to determine whether the different windows in the application are reachable, we get as results that after running 10,000 randomly generated tests it was still impossible to reach the ContactEditor window.

Re-targeting the tool

As stated from the start, the goal when developing GUISurfer has been to make it a easily re-targetable tool. The tool was originally designed to work with code written in the Java/Swing. Recently we have started work on adding capabilities to analyze code written in the Haskell functional programming language (with the WxHaskell toolkit), and Java code using the Google Web Toolkit (GWT), a Java based toolkit to develop Rich Internet Applications.

For WxHaskell, an Haskell parser was added to the tool. Next, the specific set of WxHaskell GUI components had to be supported. This amounted to configuring the slicing process with the WxHaskell components. A more fundamental issue related to event handling. In WxHaskell, events handler in the interface, are registered through the "on command" property, not Java's "addActionListener". To solve this problem, the GUISurfer tool again considers a different set of operators and properties. The main adaptation work was due to the different programming paradigm being used, i.e functional paradigm, where programs are executed by evaluating expressions. Thus, the GUISurfer tool must consider this different types of expression evaluation. As example, Haskell control structures can be defined by multiple equations as an alternative to if or case expressions.

This first version of WxHaskell support is already able to reverse engineer an application such as the agenda used above, and our goal is to extend it to a larger set of Haskell code constructions.

Regarding GWT, since it is a Java toolkit, the same parser already used by GUISurfer for Java/Swing code could be used. Ideally then there would only be the need to perform the slicing step with a different set of GUI components (those of GWT instead of those from Swing). However a few issues arose. The first related to the genericity of the tool and was due to GUISurfer's original implementation using the "addActionListener" method of Swing components to identify actions. In GWT however methods are registered though the "addClickHandler" method. Solving this problem meant parameterizing GUISurfer on the method used to register event handler in the interface.

A second issue related to differences in the functionality of both toolkits (Swing and GWT). Since a GWT application is a web application, the closing window (in GWT, panels) actions available in Java Swing are not present. Closing a web application is an unusual action, and thus there is no direct support in GWT for doing it, though it can be achieved by invoking native JavaScript. Another issue occurred in detecting a change from a window/panel to an-

other. In Swing this is achieved by invoking the "dispose" method on a class. In GWT this is achieved by making the visibility attribute of the panels. Again, changes were introduced to address this situation.

In this first version of GWT support, an assumption is also made that the GWT code is structured as similar as possible to Java Swing code. Work is currently ongoing and our goal is to loosen these restrictions as much as possible, and generally improve support for panel handling.

DISCUSSION

Using GUISurfer, programmers are able to reason at a higher level of abstraction than that of code. GUISurfer makes possible high-level graphical representation of thousand of lines of code. The process is almost automatic and enables reasoning over the interactive layer of computing systems. Examples were provided of some of the analysis that can be performed.

While results show this type of approach is useful, it must be acknowledged that there are limitations. One relates to the focus on event listeners for discrete events. This means the approach is not able to deal with continuous media and synchronization/timing constraints among objects. Another has to due with layout management issues. GUISurfer cannot extract, for example, information about overlapping windows since this must be determined at run time. Thus, we cannot find out in a static way whether important information for the user might be obscured by other parts of the interface. A third issue relates to the fact that generated models reflect what was *programmed* as opposed to what was *designed*. Hence, if the source code does the wrong thing, static analysis alone is unlikely to help because it is unable to know what the intended outcome was. For example, if an action is intended to insert a result into a text box, but input is sent to another instead. However, if the design model is available, GUISurfer can be used to extract a model of the implemented system, and a comparison between the two can be carried out.

CONCLUSIONS AND FUTURE WORK

In this paper we have present the latest results of work on investigating the extraction of different GUI models from application's source code. Our goal is to produce a fully functional reverse engineering tool. The tool is not only useful to enable the analysis of existing interactive applications, it can also be helpful when an existing application must be ported or simply updated.

The GUISurfer tool is already able to derive user interface models of interactive computing systems written in Java/Swing. Currently the main assumption made about the code is that the NetBeans IDE generated it. Initial support for WxHaskell and GWT is already available. We plan to extend our implementation to handle more complex user interfaces. We plan to continue with others programming languages/toolkits, in order to make the approach as generic as possible. Support for reasoning about the generated models has also been developed and was illustrated.

In order to make the tool available for an audience as wide as possible, we have started on the development of a web portal where the tool will be made available. The goal is that users will be able to upload their own software and use the tool to analyze it online

Open research problems related with this work are the coverage criteria definitions used for testing models, and implementation of refactoring/transformation rules for user interface source code.

ACKNOWLEDGMENTS

GUISurfer development is being carried out in the context of the CROSS and SSaaPPP projects, supported by the Portuguese Research Foundation (FCT) under contracts: PTDC/EIA-CCO/108995/2008 and PTDC/EIA-CCO/1086-13/2008. J.C. Silva is supported by a FCT PhD grant (SFRH/BSAD/782/ 2008).

REFERENCES

1. Claessen, K., and Hughes, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In ICFP, ACM SIGPLAN, 2000, 2000.
2. Fowler, M., and Scott, K. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 2003.
3. Harrison, M.D., Campos, J.C., and Loer K. Formal analysis of interactive systems: opportunities and weaknesses. In Research Methods in Human Computer Interaction, pp 88-111. CUP, 2008.
4. Loy, M., Eckstein, R., Wood, D., Elliott, J., and Cole, B. Java Swing, 2nd Edition. O Reilly, 2002.
5. Moore, M. A survey of representations for recovering user interface specifications for reengineering. Technical report, Institute of Technology, Atlanta, 1996.
6. Paternò, F. Model-based Design and Evaluation of Interactive Applications. Springer, November 1999.
7. Silva, J.C., Campos, J.C., and Saraiva, J. Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications. In vol. 4323 of LNCS, pp 137-150. Springer, 2007.
8. Silva, J.C., Saraiva J., and Campos, J.C. A Generic Library for GUI Reasoning and Testing. In ACM Symposium on Applied Computing, pp 121-128. ACM, 2009.
9. Thimbleby, H., and Gow, J.. Applying graph theory to interaction design. pp 501–519, 2008.
10. Tip, F. A survey of program slicing techniques. Journal of Programming Languages, September 1995.
11. Visser, E. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. 2003.
12. Visser, J., and Saraiva, J.. Tutorial on strategic programming across programming paradigms. In 8th Brazilian Symposium on Programming Languages, Niteroi, Brazil, May 2004.