Models for the Reverse Engineering of Java/Swing Applications

João Carlos Silva[1,2], José Creissac Campos[1], and João Saraiva[1]

[1] Departamento de Informática/CCTC, Universidade do Minho, Braga, Portugal
{jose.campos,jas}@di.uminho.pt
[2] Grupo de Sistemas e Tecnologias de Informação, IPCA, Barcelos, Portugal
jcsilva@ipca.pt

Abstract. Interest in design and development of graphical user interface (GUIs) is growing in the last few years. However, 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 describes our approach to reverse engineering abstract GUI models directly from the Java/Swing code.

1 Introduction

The correctness of the user interface is essential to the correct execution of the overall software [1]. Regarding user interfaces, correctness is expressed as usability: the effectiveness, efficiency, and satisfaction with which users can use the system to achieve their goals [7]. In order for a user interface to have good usability characteristics it must both be adequately designed and adequately implemented.

Tools are currently available to developers that allow for fast development of user interfaces 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. Moreover, the code produced by such tools is dificult to understand and maintain. In many cases users have problems in identifying all the supported tasks of a system, or in understanding how to reach them.

Model-based design helps to identify high-level models which allow designers to specify and analyse systems. Different types of models can been used in the design and development of interactive systems, from user and task models to software engineering models of the implementation. The authors are currently engaged in a R&D project (IVY – A model-based usability analysis environment[3]) which aims at developing a model-based tool for the analysis of interactive systems designs. In the context of the project we are investigating the applicability of reverse engineering approaches to the derivation of user interface's abstract models amenable for verification of usability related properties.

In this paper we present the initial results of work on investigating the application of strategic programming and slicing to the reverse engineering of Java/Swing [5] user interfaces. Our goal is to produce a fully functional reverse engineering prototype tool.

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

The tool will be capable of deriving user interface abstract models of interactive applications.

Section 2 explains the technique applied in the reverse engineering of graphical user interfaces. Section 3 describe the diferent kinds of models extracted by the prototype. Finally, in sections 4 and 5 we present some limitations, conclusions and our plans for future work.

2 A Technique for Reverse Engineering Graphical User Interfaces

The technique explained in this section aids in identifying a graphical user interface abstraction from legacy code. The goal is to detect components in the user interface through functional strategies and formal methods. These components include user interface objects and actions.

In order to extract the user interface model from a Java/Swing program we need to construct a slicing function [8, 6] that isolates the Swing sub-program from the entire Java program. The straightforward approach is to define a explicit recursive function that traverses the Abstract Syntax Tree (AST) of the Java program and returns the Swing sub-tree. We use strategic programming wich contains 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).

Strategic programming is a form of generic programming that combines the notions of *one-step traversal* and *dynamic nominal type case* into a powerful combinatorial style of traversal construction. Strategic programming has been defined in different programming paradigms. In this paper we will use the STRAFUNSKI library [4]: a Haskell [3] library for generic programming and language processing.

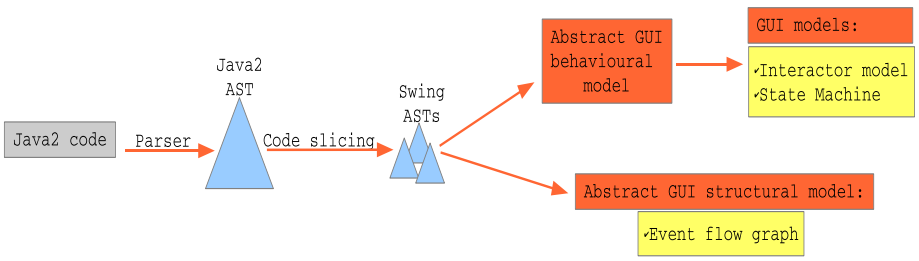

Fig. 1. The reverse engineering process

3 Models for Reverse Engineering Graphical User Interfaces

In order to define the slicing functions mentioned above, we defined a small set of abstractions for the interactions between the user and the system. These are the abstractions that we look for in the legacy code:

- User input: Any data inserted by the user;
- User selection: Any choice that the user can make between several different options, such as a command menu;
- User action: An action that is performed as the result of user input or user selection;
- Output to User: Any communication from application to user, such as a user dialogue;

Through 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 STRAFUNSKI library with formal and semi-formal methods, which are mathematically-based languages, techniques, and tools for specifying and verifying systems.

This section shows the application of the prototype to a small example: the *JBank* transfers system. Basically, the *JBank* system is a simple JAVA/SWING "toy" example allowing for account transfers (see figure 2).

Fig. 2. *JBank* system

Applying the prototype to the *JBank*'s code, enables us to extract information about all widgets presented at the interface, such as *JButton*, *JLabel*, *JComboBox*, *JTextField*, *JPanel*, etc. Once the AST for the application code is built we can apply different slicing operations as needed.

Currently the prototype enables the extraction of diferent kinds of models wich are described in the following sections.

3.1 Event-Flow Graph Model

The prototype is capable of generating the *JBank*'s partial event-flow graph (see figure 3). All widgets and their relationship are abstracted to this graph. As an example, blue nodes specify *JButtons* abstractions, arrows specify methods calls from one widget to another.

In this graph, we can see all graphical user interface widgets and their relationships.

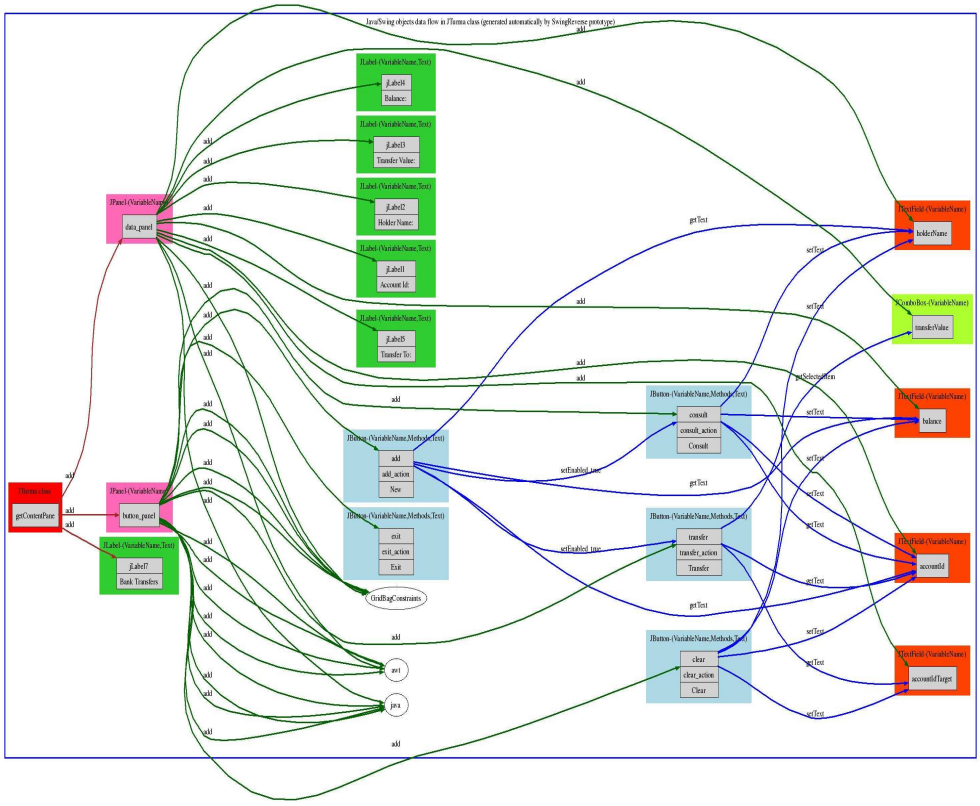

Fig. 3. *JBank* system's partial GUI event-flow graph

3.2 GUI abstract behaviour models

GUI metamodel - From the swing slice of the source java code, we extract an abstract GUI behavioural model of the interface. Next we present the mathematical model *Gui*

that allows us to abstract from any GUI's behaviour.

$$
\begin{aligned}
Gui &\equiv 2^{(AtributesValues \times ActionName \times Parameters \times Conditions \times AtributesValues)} \\
AtributesValues &\equiv AtributeName \hookrightarrow Value \\
Parameters &\equiv ParameterName \hookrightarrow ParameterType \\
Conditions &\equiv String \\
Value &\equiv String \\
ParameterType &\equiv String \\
ParameterName &\equiv String \\
ActionName &\equiv String \\
AtributeType &\equiv String \\
AtributeName &\equiv String
\end{aligned}
$$

Basically this metamodel specify a set of transition states:

$$2^{(AtributesValues \times ActionName \times Parameters \times Conditions \times AtributesValues)}$$

Each state is abstracted by $AtributesValues$ wich is a parcial finite mapping from atributes names to values. In other hand, the relation between diferent state is abstracted by $ActionName$, $Parameters$ and $Conditions$ attributes. These atributes allows us to represent all actions that can be executed from a particular state.

The Modal Action Logic Interactors - The Modal Action Logic (MAL) interactors is a domain specific language for (??????????????????) structuring the use of standard specification techniques in the context of interactive systems specification. In IVY the MAL interactors language from [2] is used.

The definition of a MAL interactor contains a state, actions, axioms and presentation information. This language allows us to abstract both static and dynamic perspectives of interactive systems. The static perspective is achieved with **attributes** and **actions** abstractions which aggregate the state and all visible components in a particular instant. The **axioms** abstraction formalizes the dynamic perspective from an interactive state to another.

Applied to the code of the *JBank* application, the tool automatically generates an interactor specification including the initial application state and dynamic actions. This interactor contains a set of attributes (cf. figure 4) - one for each information input widget, and one for each button's enabled status.

The interactor also contains a set of actions (cf. figure 5) - one for each button, and one for each input widget (representing user input).
And, finally axioms like the following wich define the effect of the add button in the interface. Similar axioms are generated for all other `set` actions, for brevity we include only one here.

```
[add]
newEnabled'=newEnabled & consultEnabled'=true &
transferEnabled'=transferEnabled &
```

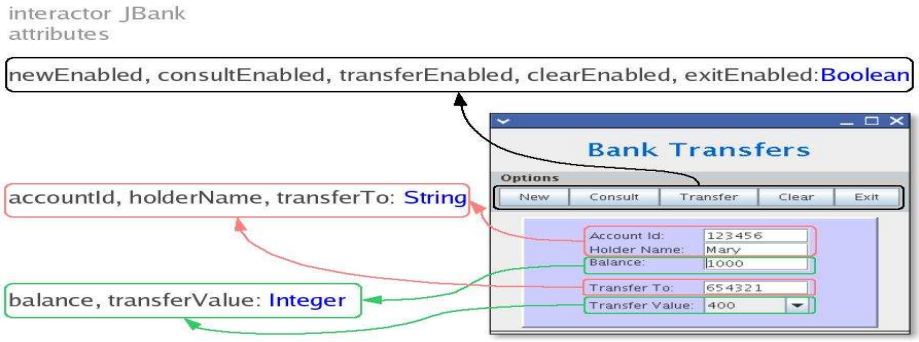

Fig. 4. Interactor's attributes abstraction

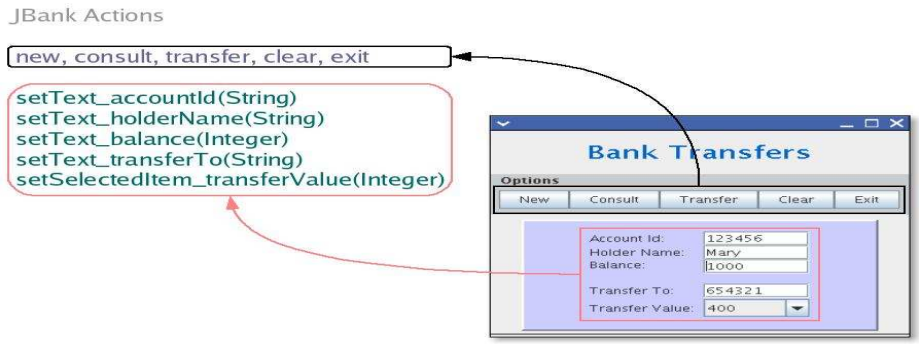

Fig. 5. Interactor's actions abstraction

```
clearEnabled'=clearEnabled & exitEnabled'=exitEnabled &
accountId'=accountId & holderName'=holderName &
transferTo'=transferTo & balance'=balance &
transferValue'=transferValue
```

Finite State Machine - Currently, we are working on the extraction of a finite state machine (FSM) model of the interface. Next we present an example of the FSM that we can automatically induce from the GUI model.

Figure 6 contains part of the FSM obtained from the *JBank* application. In this state machine each state defines an abstraction of the GUI window in one particular period of time. The arrow specifies an action moving from one state to another.

Fig. 6. Part of the *JBank* state machine

The action displayed in this machine defines that a bank transfer can only occur if the balance of the source account is greater or equal than the value to be transfered.

balance = 0 ?????

4 Limitations

We have developped a tool, named Java Swing Reverse (JSR), that extracts the three models presented from a java/swing application. The current version of the tool has several limitations: First, it does not handle all the large set of Swing graphical objects. Second, we consider only the java applications produced by NETBEANS, wich produces a particular Java/Swing structured code. However, we can easily update JSR to handle any Java GUI code. Third, coputanional level ??????

Finally, we do not consider GUIs such as web-user interfaces that have synchronization/timing constraints among objects, movie players that show a continuous stream of video rather than a sequence of discrete frames, and non-deterministic GUIs in which it is not possible to model the state of the software in its entirety.

5 Conclusions and Future Work

Currently the tool automatically extracts the software's windows, and a subset of their widgets, properties, and values. The execution model of the user interface is obtained by using a classification of its events.

The approach has also proven very flexible. From the Abstract Syntax Tree representation we are already able to derive both GUI metamodel, interactor based model, event flow graphs and state machines. 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. In the future, we will extend our implementation to handle more complex user interfaces.

Acknowledgments

This work is partially supported by FCT (Portugal) and FEDER (European Union) under contract POSC/EIA/56646/2004.

References

1. Shneiderman B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction (3nd Edition)*. Addison wesley edition, 1997.
2. José C. Campos and Michael D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3-4):275–310, August 2001.
3. Simon Peyton Jones, John Hughes, Lennart Augustsson, et al. Report on the Programming Language Haskell 98. Technical report, February 1999.
4. R. Lammel and J. Visser. A STRAFUNSKI application letter. Technical report, CWI, Vrije Universiteit, Software Improvement Group, Kruislaan, Amsterdam, 2003.
5. Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java Swing, 2nd Edition*. OReilly, 2002.
6. Andrea De Lucia. Program slicing: Methods and applications. *IEEE workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001.
7. ISO/TC159 Sub-Commitee SC4. Draft International ISO DIS 9241-11 Standard. International Organization for Standardization, September 1994.
8. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, september 1995.