# Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications

J. C. 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.** 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 describes our approach to reverse engineer an abstract model of a user interface directly from the GUI's legacy code. We also present results from a case study. These results are encouraging and give evidence that the goal of reverse engineering user interfaces can be met with more work on this technique.

## 1 Introduction

Enterprise competitiveness in the information age is very much dependent on the quality of the graphical user interfaces (GUIs) being used [10]. However, the quality of large and complex user interfaces is hard to maintain. These very rapidly originate failures, a problem nowadays identified under the *usability* heading. A very large proportion of failures in interactive systems takes place due to erroneous human actions [12]. As pointed out by Leveson [15], human error in computer systems use is often due to errors in their user interface design, and not the sole result of errors performed by the direct users of the systems.

The correctness of the user interface is essential to the correct execution of the overall software. Regarding user interfaces, correctness is expressed as usability: the effectiveness, efficiency, and satisfaction with which users can use the system to achieve their goals [24]. In order for a user interface to have good usability characteristics it must both be adequately designed and adequately implemented, having its target users, their goals, and the operating environment in mind.

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. In many cases users have problems in identifying all the supported tasks of a system, or in understanding how to reach them. The problem seems to lie more on the design of the systems, than in their actual implementation.

> *Problems could be largely solved if designers had methods and tools to provide indications about the most effective interaction and presentation techniques to support the possible user activities* [22].

Traditionally design aspects of user interfaces have been the concern of Human-Computer Interaction, while software engineers have been mainly concerned with implementation aspects. Clearly there is a need to enable software engineers to consider aspects of design when developing interactive systems.

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. The tool will act as a front end to the SMV model checker, creating an abstraction layer where models of interactive systems can be developed and analysed. The models used are expressed in the MAL interactors language [3], and express both the information present at the user interface and the behaviour of the system in response to user input. 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 user interfaces. Our goal is to produce a fully functional reverse engineering prototype tool. The tool will be capable of deriving abstract models of interactive applications' user interfaces. This will enable reasoning about the design of the system. In particular we are interested in applying automated reasoning techniques to ensure a thorough analysis of all possible behaviours of a given system.

In section 2 we briefly introduce the IVY project. Then in section 3 we describe some related work. Section 4 explains the technique applied in the reverse engineering of graphical user interfaces. Thus, we describe the model-based technique used for reverse engineering interactive systems. In section 5 we shows the application of the actual prototype to a simple system. Finally, in section 6 we present some conclusions and put forward our plans for future work.

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

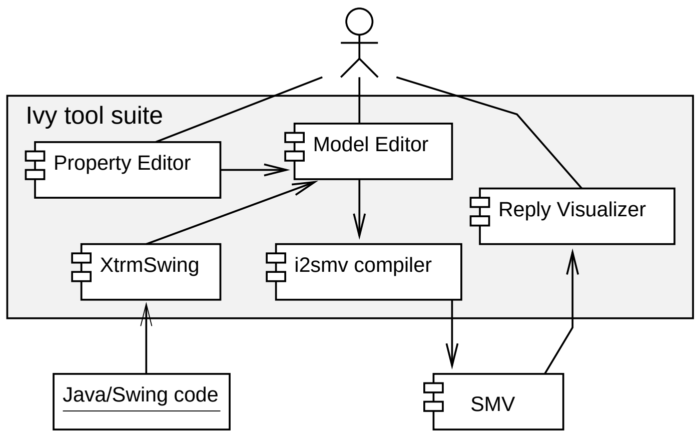

**Fig. 1.** IVY architecture

## 2 About the IVY project

### 2.1 The project

IVY follows from the development of i2smv [3], a compiler enabling the verification of interactive systems' models using the SMV model checker [18]. The objective now is to develop, as a front end to SMV, a model based tool for the analysis of behavioural issues of interactive systems' designs. This tool will not only translate the models into the SMV input language, but fully support the process of modelling and analysis by providing editors, for models and properties, and a reply visualizer for the analysis of the verification process results (see figure 1).

A considerable number of tools for model checking have been proposed. Interactive systems, however, have specificities that make it difficult to use typical model checking tools [3]. Two major types of problem can be identified:

- the richness of the interaction between user and system affects the models;
- the assumptions that must be made about the user's capabilities affects the analysis of the verification results;

Tools are needed that facilitate modelling, and reasoning about the results of the verification, from an interactive systems perspective. In IVY we aim at creating an abstraction layer where models of interactive systems can more easily be developed and analysed.

Being modular, the tool will also act as a test-bed for different styles of modelling/analysis of interactive systems. One approach we are exploring is the use of reverse engineering techniques to enable the generation of models from user interface code. Our goal is to support the verification of existing user interfaces in a semi-automated manner. This will not only be useful to enable the analysis of existing interactive applications, but can also be helpful in a re-engineering process when a existing application must be ported or simply updated. 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.

### 2.2 The language

Interactors act as a mechanism for structuring the use of standard specification techniques in the context of interactive systems specification [7]. In IVY the MAL interactors language from [3] is used.

The definition of a MAL interactor contains a state, actions, axioms and presentation information:

- The state of an interactor is defined by a collection of attributes.
- Actions correspond to events the system can respond to.
- Axioms allow the expression of what effect actions have on the state. In order to describe behaviour, a deontic logic is used:
  - deontic operator $obl(ac)$: $ac$ is obliged to happen some time in the future;
  - deontic operator $per(ac)$: $ac$ is permitted to happen next;
  - model operator $[ac]exp$: $expr$ is true after action $ac$ takes place;
  - $[]expr$: $expr$ is true in the initial state;
  - $per(ac) \rightarrow exp$: $ac$ is permitted only if $exp$ is true;
  - $exp \rightarrow obl(ac)$: if $exp$ is true then action $ac$ becomes obligatory.
- Presentation information allows us to assert that a particular attribute/action is visible. This is done with a *vis* annotation on relevant attributes/actions.

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.

## 3 Related work

In the Human-Computer Interaction area, quality is typically addressed by the use of empirical methods that involve testing (a prototype of) the system. These methods work by placing users in front of a system in order to assess its usability. Analytic methods have also been proposed as a means of reducing the effort of analysis. These approaches work by inspection of the system (or a model thereof) and range from less structured approaches such as Heuristic Evaluation [21] to more structured ones such as Cognitive Walkthroughs [16]. In all cases, these approaches are geared towards the analysis of the design of the interactive system, and in particular aspects related to its usability.

The use of mathematically rigorous (formal) models of the interactive systems, as a means of reasoning about usability issues, has also been proposed (see, for example, [3, 23]). One advantage of formal approaches is that they enable the thorough verification of the validity of the properties/system under scrutiny. One of their drawbacks is the difficulty in incorporating human considerations in the analysis process. Approaches such as Syndectic Modelling [8] attempt to formalize the user but become too complex to be used in practice. Other approaches have been proposed were specific aspects of human behaviour are included in the models (see for example, [23, 4, 2]).

In Software Engineering concerns are more geared towards testing the quality of the code produced (absence of bugs) and its correctness vis-a-vis the system's specification. Testing of user interface implementations has also attracted attention.

Testing typically progresses by having the code execute pre-defined test cases and compare the result of the execution with the result of some test oracle. In the case of interactive systems, models of the user interface are needed both to aid the generation of the test cases, and for the test oracle. In this area, the use of reverse engineering approaches has been explored in order to derive such models directly from the existing interactive system.

A typical approach is to run the interactive system and automatically record its state and events. Memon et al. [19] 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 and analyses the run-time behaviour of Java software trough a reverse engineering process [25]. Running the target software under a debugger allows for the generation of state diagrams. The state diagrams can be used to examine the overall behaviour of a component such as a class, a object, or a method.

Another alternative is the use of statical analysis. The reengineering process is based on analysis of the application's code, instead of its execution, as in previous approaches. One such approach is the work by d'Ausbourg et al. [6] in reverse engineering UIL code (User Interface Language – a language to describe user interfaces for the X11 Windowing System, see [11]). In this case models are created at the level of the events that can happen in the components of the user interface. For example, pressing or releasing a button.

In the last decade the reengineering of interactive systems has also been investigated by several authors. Moore [20] 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 [9] proposes a similar approach. In both cases static analysis is used.

We are using static analysis as in [9, 20, 6]. When compared to their work our challenges are twofold:

– We are reverse engineering code for graphical user interfaces, as opposed to character based user interfaces in [9, 20]. At the moment we are working

with Java/Swing (however, our long term goal is to develop a more generic approach).
– We are more interested in models that reflect the design of the user interface and the interaction that it creates, than the actual architecture of the underlying software implementing it. Hence, we need models that are more abstract than those produced in, for example, [19] or [6].

## 4 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. This includes identifying data entities and actions that are involved in the graphical user interface, as well as relationships between user interface components. The goal is to detect components in the user interface through functional strategies and formal methods. These components include user interface objects and actions.

### 4.1 Graphical User Interface definition

The most usual class of user interfaces are hierarchical graphical front-ends to software systems. These user interfaces produce deterministic graphical output from user input and system events. A graphical user interface (GUI) contains graphical widgets. 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.

This paper focuses on techniques to reverse engineer this first class of user interfaces. Another class of user interfaces are 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.

### 4.2 GUI Slicing Through Strategic Programming

In order to extract the user interface model from a Java/Swing program we need to construct a slicing function [27, 17] 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. A typical Java grammar/AST, however, has 105 non-terminal symbols and 239 productions [1]. As a result, writing such a function forces the programmer to have a full knowledge of the grammar and to write a complex and long mutually recursive function. We use a different approach by using strategic programming. In this style of programming, there is 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

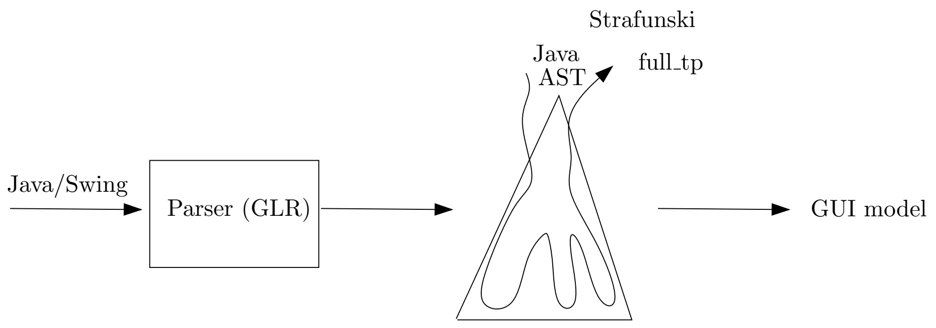

**Fig. 2.** The reverse engineering process

programmer needs 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).

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 allows novel forms of abstraction and modularization that are useful for program construction in general. In particular when large heterogeneous data structures are involved (e.g. the abstract syntax tree representing a Java program), strategic programming techniques enable a high level of conciseness, composability, and traversal control [29, 28]. Strategic programming has been defined in different programming paradigms. In this paper we will use the STRAFUNSKI library [14]: a Haskell [13] library for generic programming and language processing. STRAFUNSKI not only contains the strategic library, but also a powerful GLR parser generator. It contains also a set of grammars for most existing programming languages (for example, a full Java grammar).

In order to explain strategic programming and the STRAFUNSKI library in more detail, let us consider the following JAVA/SWING code fragment:

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

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

```
...
Statement(
 StatementWithoutTrailingSubstatement(
  ExpressionStatement(
   semicolon2(
    Assignment(
     AssignmentOp(
      Name2(Identifier-p(["addButton"])),
            equal1,
      StatementExpression(
         ClassInstanceCreationExpression(
         new-comma(ClassOrInterfaceType1(
         Name(Identifier-p(["javax","swing","JButton"]))),
         []))))))))
...
```

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

– We begin by identifying the type of strategy needed to collect the desired information. We make use of the traversal combinator `full_tdTU` in order to define a function that traverses the AST in a top-down fashion (although, in this particular example, we could use a `full_bu` strategy).
– Next, we need to define the *worker* function that will do the work while traversing the AST. This worker function identifies the tree nodes where work has to be done. In the complete Java AST the nodes of interest correspond to the constructor `AssignmentOp` (see AST above). Thus, our worker function simply returns a singleton list with the left-hand side of the assignment and the expression. All the other nodes are simply ignored! The functions *applyTU*, *full_tdTU*, *constTU*, and *adhocTU* are library functions used to construct the results and apply the traversal combinators. Because they are not relevant to understand our techniques, we omit their definitions here.

This function, named `getAssignmentIdentifiers`, looks as follows:

```
getAssignmentIdentifiers :: (Term t) => t -> [([Id],[Id])]
getAssignmentIdentifiers ast =
 runIdentity (applyTU (full_tdTU worker) ast)]
 where
 worker = constTU [] ‘adhocTU‘ getexp
 getexp (AssignmentOp left op exp) = return [(left,exp)]
```

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

```
getJButtons :: (Term t) => t -> [[Id]]
getJButtons ast = jButton
 where assignments = getAssignmentIdentifiers e
       jButton = [a | (a,b) <- assignments
                    , (b==["javax","swing","JButton"])]
```

Functional strategic combinators allow us to construct programs that only mention those constructors that are strictly relevant to the problem. Furthermore, they work for *any* abstract syntax tree and not only for the Java AST under consideration in this paper. As a result, the strategic 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. Observe that in the Haskell/Strafunski code presented above a small part of it is specific of the Java language/AST. Obviously, we can easily parameterize these functions with that language specific constructors. It also should be noticed that the basic concepts of strategic programming are independent of the programming paradigm.

### 4.3 User Interface Abstraction

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. The use of formal methods does not guarantee correctness. However, they aid in making specifications more concise and less ambiguous, making it easier to reason about them.

## 5 An example

This section shows the application of the prototype to a small example. Basically, the *JClass* system is a simple JAVA/SWING ”toy” example allowing for marks management (see figure 3).

Applying the prototype to the application's code, enables us to extract information about all widgets presented at the interface, such as *JButton*, *JLabel*, *JComboBox*, *JTextField*, *JSlider*, *JProgressBar*, *JPanel*, etc. To reverse engineer the graphical user interface of an interactive system it is not necessary to analyse all of the application's functionality. Therefore, irrelevant information from the *JClass* system is discarded by the tool during the slicing phase in order to make the representations much more clear.

Once the AST for the application code is built we can apply different slicing operations as needed. This means we can easily tailor the information (models) we want to extract from the AST (and, thus, from the code).

Currently the prototype enables the extraction of two types of models:

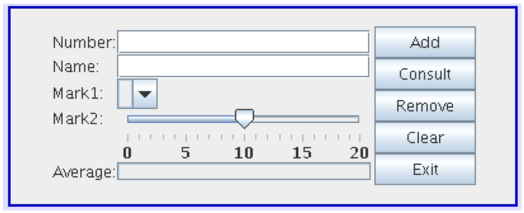

**Fig. 3.** *JClass* system

– Interactors models, which capture a more Human Computer Interaction perspective of the system. These models are more oriented towards usability analysis.
– Event-flow graphs which allow the analysis of the code's quality from a software engineering perspective.

In the first case, applied to the code of the *JClass* application, the tool automatically generates an interactor specification including the initial application state and dynamic actions. This interactor contains a set of attributes:

```
interactor JClass
attributes
number, name: String
mark1, mark2, average: Integer
addEnabled, consultEnabled, removeEnabled, clearEnabled,
                                                    exitEnabled: Boolean
```

one for each information input widget, and one for each button's enabled status. The names of the attributes are derived from the names of the widget variables in the code. Note that the `String` and `Integer` types must later be defined in the IVY editor.

The interactor also contains a set of actions:

```
actions
  add, open, close, consult, remove, clear, exit,
  setText_name(String), setSelectedItem_mark2(Integer),
  setValue_mark1(Integer), setValue_average(Integer),
  setText_number(Integer)
```

one for each button, and one for each input widget (representing user input).

And, finally, a set of axioms:

```
[] number="" & name="" & mark1=10 & mark2=10 & average=0
[] addEnabled=true & clearEnabled=true & exitEnabled=true &
 consultEnabled=false & removeEnabled=false & number="" &
 name="" & mark1=10 & mark2=10 & average=0
[add] number'=number & name'=name & mark1'=mark1 &
 mark2'=mark2 & average'=average & consultEnabled'=true &
 removeEnabled'=true & addEnabled'=addEnabled &
 clearEnabled'=clearEnabled & exitEnabled'=exitEnabled
[consult] number'=number & name'=?ref1? & mark1'=?ref2? &
 mark2'=?ref3? & average'=?ref4? & addEnabled'=addEnabled &
 consultEnabled'=consultEnabled & removeEnabled'=removeEnabled &
 clearEnabled'=clearEnabled & exitEnabled'=exitEnabled
[remove] number'=number name'=name & mark1'=mark1 & mark2'=mark2 &
 average'=average & addEnabled'=addEnabled &
 clearEnabled'=clearEnabled & exitEnabled'=exitEnabled
[clear] number'=?ref5? & name'=?ref6? & mark1'=?ref7? & mark2'=?ref8? &
 average'=?ref9? & addEnabled'=addEnabled &
 consultEnabled'=consultEnabled & removeEnabled'=removeEnabled &
 clearEnabled'=clearEnabled & exitEnabled'=exitEnabled
[setText_name(a)] name'=a & number'=number & mark1'=mark1 & mark2'=mark2 &
 average'=average & consultEnabled'=consultEnabled &
 removeEnabled'=removeEnabled & addEnabled'=addEnabled &
 clearEnabled'=clearEnabled & exitEnabled'=exitEnabled
...
```

The first two axioms define the initial state of the system. The next four define the effect of the buttons in the interface. The `?refX?` expressions represent values that must be filled in using the IVY editor. To help complete the model, each expression is a pointer to the Java code which constructs the value to be assign. Remember that this is a semi-automated process. At least at this stage, we do not want to go into the semantics of the application's functional layer. The final axiom defines the effect of user input in the name text field. Similar axioms are generated for all other `set` actions, for brevity we include only one here. We have not included the rendering annotations in the interactor since all attributes and actions are *visible* (i.e. they are all available to users).

Even incomplete, this interactor already includes considerable information regarding the application's interactive behaviour. For example, the fourth axiom expresses the interactive state after executing the *consult* action. We can see that attributes *number*, *addEnabled*, *consultEnabled*, *removeEnabled*, *clearEnabled* and *exitEnabled* remain unchanged, and that attributes *name*, *mark1*, *mark2* and *average* receive new data. Once fully instantiated the model can be used in the IVY tool for verification of its behaviour.

Alternatively the prototype is also capable of generating the *JClass*'s partial event-flow graph (see figure 4). 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. Through this particular example, we are able to detect all JCLASS's widgets (*JButtons*, *TextFields*, *ComboBoxs*, etc) and interactive methods called from these widgets (*setText*, *getText*, *getSelectedItem*, *setEnabled*, etc).

At the same time, the event-flow graph allows us to detect irrelevant widgets in the JCLASS system. In figure 4 these widgets are represented through two disconnected nodes. Basically the JCLASS code used to generate the graph contains two widgets which are not visualized nor manipulated by the system.

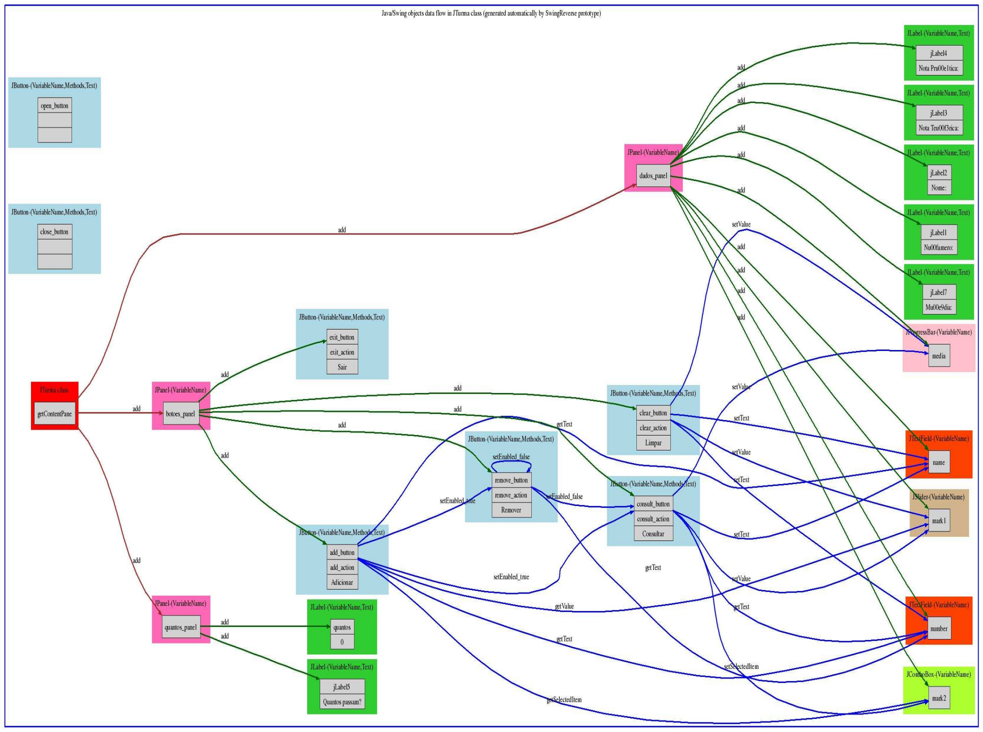

**Fig. 4.** *JClass* system's partial GUI event-flow graph

These are the *open* and *close* nodes in the event-flow graph, which are related to the *open* and *close* actions in the interactor specification actions set.

## 6 Conclusions and Current Work

In this paper we have shown how strategic programming and slicing techniques can be combined to reverse engineer user interfaces from application code. The results of this work are encouraging and give evidence that the goal of reverse engineering user interfaces can be met. A prototype has been developed that allows us to obtain models of the user interface's structure and (partially) behaviour in an automatic manner.

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 interactor based models, and event flow graphs. In the first case the models capture a user oriented view of the interface. In the second case the models capture the internal structure of the code. This enables us to reason about both usability properties of the design, and the quality of the implementation of that design.

At the moment only a subset of all Swing widgets are being processed by the tool. 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.

We will also explore broadening the scope of the approach, both at the input and output of the tool. In the first case we plan to experiment with different programming languages/toolkits, in order to make the approach as generic as possible. In the second case we want to further investigate the possibility of generating different types of models for analysis. For example, we envisage that generating Event Matrixes in the style of [26] will be possible.

## Acknowledgments

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

## References

1. Tiago Alves and Joost Visser. Metrication of sdf grammars. Technical Report DI-PURe-05.05.01, Departamento de Informática, Universidade do Minho, 2005.
2. Ann Blandford, Richard Butterworth, and Paul Curzon. Models of interactive systems: a case study on programmable user modelling. *International Journal of Human-Computer Studies International Journal of Human-Computer Studies*, 60:149–200, 2004.
3. José C. Campos and Michael D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3-4):275–310, August 2001.
4. José Creissac Campos. Using task knowledge to guide interactor specifications analysis. In J. A. Jorge, N. J. Nunes, and J. F. Cunha, editors, *Interactive Systems: Design, Specification and Verification — 10th International Workshop, DSV-IS 2003*, volume 2844 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2003.
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*, january 2001.
6. 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 F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, Springer Computer Science, pages 105–122. Springer-Verlag/Wien, June 1996.
7. D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum 12(3), 25-36*, 1993.
8. D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Syndetic modelling. *Human-Computer Interaction*, 13(4):337–393, 1998.

9. Merlo E., Gagne P. Y., Girard J.F., Kontogiannis K., Hendren L.J., Panangaden P., and De Mori R. Reverse engineering and reengineering of user interfaces. *IEEE Software, 12(1), 64-73*, 1995.
10. B. Lientz e E. Swanson. *Software Maintenance Management*. Addison-wesley edition, 1980.
11. Dan Heller and Paula M. Ferguson. *Motif Programming Manual*, volume 6A of *X Window System Seris*. O'Reilly & Associates, Inc., second edition, 1994.
12. E. Hollnagel. *Human Reliability Analysis: Context and Control*. Academic press edition, 1993.
13. Simon Peyton Jones, John Hughes, Lennart Augustsson, et al. Report on the Programming Language Haskell 98. Technical report, February 1999.
14. R. Lammel and J. Visser. A STRAFUNSKI application letter. Technical report, CWI, Vrije Universiteit, Software Improvement Group, Kruislaan, Amsterdam, 2003.
15. Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, Inc., 1995.
16. Clayton Lewis, Peter Polson, Cathleen Wharton, and John Rieman. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *CHI '90 Proceedings*, pages 235–242, New York, April 1990. ACM Press.
17. Andrea De Lucia. Program slicing: Methods and applications. *IEEE workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001.
18. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
19. Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. Technical report, Department of Computer Science and Fraunhofer Center for Experimental Software Engineering, Department of Computer Science University of Maryland,USA, 2003.
20. M. M. Moore. Rule-based detection for reverse engineering user interfces. *Proceedings of the Third Working Conference on Reverse Engineering, pages 42-8, Monterey, CA*, november 1996.
21. Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90 Proceedings*, pages 249–256, New York, April 1990. ACM Press.
22. Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, 2000.
23. John Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, February 2002.
24. ISO/TC159 Sub-Commitee SC4. Draft International ISO DIS 9241-11 Standard. International Organization for Standardization, September 1994.
25. T. Systa. Dynamic reverse engineering of java software. Technical report, University of Tampere, Finland, 2001.
26. Harold Thimbleby. User interface design with matrix algebra. *ACM Transactions on HUman-Computer Interaction*, 11(2):181–236, June 2004.
27. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, september 1995.
28. Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science. Spinger-Verlag, November 2003. (Draft; Accepted for publication).
29. Joost Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, February 2003.