

Universidade do Minho

Escola de Engenharia

JOÃO CARLOS CARDOSO DA SILVA

**GUI SURFER: A Generic Framework for Reverse Engineering of Graphical
User Interfaces**

Tese de Doutoramento em Informática

Ramo de Fundamentos da Computação

Trabalho efectuado sob a orientação de

Professor Doutor José Creissac Campos

Professor Doutor João Alexandre Saraiva

Dezembro de 2010

É autorizada a reprodução integral desta tese apenas para efeitos de investigação,
mediante declaração escrita do interessado, que a tal se compromete.

Universidade do Minho, / /

Assinatura:

Acknowledgments

Many people contributed to this thesis. I would like to use these lines to acknowledge every person and institution that contributed to its elaboration.

First of all, I would like to thank my supervisors, Professor José Creissac Campos and Professor João Alexandre Saraiva, both from the Department of Informatics of *Universidade do Minho*, for giving me the opportunity to do my Ph.D and for their guidance and encouragement that made this thesis possible. I feel very lucky to have been their student. Collaborating with them was an enriching professional experience, and an opportunity to work with two persons whose human side I highly appreciate.

I would like to thank also my colleagues from *Universidade do Minho* and *Instituto Politécnico do Cávado e do Ave* for the continuous encouragement and friendship.

I would like to thank the *Fundação para a Ciência e a Tecnologia* (FCT) for supporting this thesis under contract SFRH/BD/30729/2006.

I wish to express also my gratitude to my parents, Abílio and Glória, who always believed and supported me in every moment of my life, to my brother, José Luís, for all his support and for being my biggest friend.

Finally, I would like to thank Susana for all her encouragement and comprehension.

This work was supported by projects CROSS (PTDC/EIA-CCO/108995/2008) and SSAAPP (PTDC/EIA-CCO/108613/2008) funded by the Portuguese Foundation for Science and Technology

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR



Abstract

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 by end users. 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 effort to develop tools to support the automated analysis of interactive system designs, this research investigates the applicability of reverse engineering approaches to graphical user interface (GUI) analysis from source code. Our objective consists in developing tools to automatically extract models containing GUI behaviours, from its source code. The models should specify which widgets are present in the interface, when can a particular GUI event occurs, which are the conditions, which system actions are executed, and which GUI state is generated next. Subsequently, this research aims to reasoning over GUI models in order to analyse aspects of the original application's behaviour, and the implementation's quality.

GUI SURFER, a tool developed in the context of this doctoral thesis, is capable of automatically deriving and reason about graphical user interface behavioural models of applications written in *Java/Swing*, *WxHaskell* and *GWT*. This work is useful to enable the analysis of existing interactive applications, and also when an existing application must be ported or simply updated.

Resumo

Os programadores têm já ao seu dispor diversas ferramentas que permitem o rápido desenvolvimento de interfaces gráficas com o utilizador (GUI). Todavia, o desenho dos sistemas interactivos não parece tirar partido destas ferramentas. Em muitos casos, os utilizadores têm problemas em identificar todas as tarefas suportadas pelo sistema, e têm dificuldades em perceber como atingir determinados objectivos. Por outro lado, o código gerado pelas ferramentas é difícil de analisar e manipular.

No contexto do desenvolvimento de ferramentas de suporte à análise automatizada de sistemas interactivos, foram realizados estudos baseados em engenharia reversa para a extracção de modelos comportamentais de interfaces com o utilizador. O nosso objectivo consiste em desenvolver ferramentas para extrair automaticamente modelos descrevendo o comportamento da GUI. Os modelos descrevem quando um evento pode ocorrer, sob que condições, quais são as acções executadas e qual é o estado da GUI gerado a seguir. Consequentemente, é possível raciocinar e testar os modelos da GUI de modo a analisar aspectos relacionados com a usabilidade da aplicação e a qualidade da sua implementação.

A ferramenta, com nome GUI SURFER, desenvolvida no âmbito deste trabalho de doutoramento, permite extrair e testar modelos de comportamentos de interfaces gráficas com o utilizador escritas nas linguagens *Java/Swing*, *WxHaskell* e *GWT*. Este trabalho é útil para analisar aplicações existentes bem como para dar apoio na manutenção e migração de aplicações.

to Susana and Rodrigo

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	User Interface Development	2
1.1.2	An Illustrative Example	4
1.2	Thesis Genesis	10
1.3	Objectives	11
1.4	Research Questions	13
1.5	Structure of the Thesis	14
2	Reverse Engineering Applied to GUI Modelling	17
2.1	Types of Engineering	18
2.2	Reverse Engineering	19
2.3	Graphical User Interfaces	20
2.4	Types of GUI relevant Models	22
2.5	GUI Representations	26
2.5.1	Grammars	26
2.5.2	Finite State Machines	29
2.5.3	Other Models	30

2.5.4	Models for Quality Evaluation	33
2.6	GUI Reverse Engineering	34
2.6.1	Dynamic Analysis	35
2.6.2	Static Analysis	36
2.7	Conclusions	37
3	An Approach to GUI Reverse Engineering	39
3.1	An Interactive Application as Running Example	40
3.2	GUI Reverse Engineering Approach	44
3.3	GUI Source Code Extraction	48
3.4	Retargetable Methodology	49
3.4.1	Program Dependency Graph	49
3.4.2	Code Slicing	52
3.4.3	Strategic Programming	53
3.4.4	Case Study: Regular Expressions Processing with <i>TOM</i>	56
3.4.5	Retargetable Methodology	58
3.5	Behavioural Models Generation	59
3.6	Conclusions	60
4	GUI SURFER: A Reverse Engineering Tool	63
4.1	The Architecture of GUI SURFER	63
4.1.1	Source Code Slicing	65
4.1.2	GUI Behavioural Modelling	65
4.1.3	GUI Reasoning	65
4.2	The GUI SURFER Implementation	66

<i>CONTENTS</i>	ix
4.2.1 Parsing the Source Code	67
4.2.2 Extracting the GUI Layer	68
4.2.3 Generating of GUI Behavioural Models	74
4.2.4 Evaluating GUI Behavioural Models	75
4.3 Models for GUI Reverse Engineering	75
4.3.1 GUI Meta-Model	75
4.3.2 MAL Interactors	82
4.3.3 Event Flow Graphs	84
4.3.4 Finite State Machines	86
4.3.5 Others Models	87
4.4 A Language Independent Tool	89
4.4.1 Retargetability	91
4.4.2 <i>WxHaskell</i> example	93
4.4.3 <i>GWT</i> example	96
4.5 Conclusions	98
5 GUI Reasoning from Reverse Engineering	101
5.1 Testing with the <i>QuickCheck</i> Tool	103
5.2 GUI Inspection Through Graph Theory	106
5.2.1 <i>Agenda's</i> Behavioural Graph	106
5.2.2 Graph Events Count	106
5.2.3 Operations on Graphs	108
5.2.4 GUI Metrics	109
5.2.5 <i>Graph-Tool</i>	112

5.2.6	GUI Test Cases Generation	120
5.3	Conclusions	125
6	HMS Case Study: A Larger Interactive System	127
6.1	Login Window	128
6.2	Main Window	131
6.3	Patient Management	132
6.4	Doctors Management	136
6.5	Bills Management	136
6.6	Overall Behaviour	137
6.7	GUI Reasoning	140
6.8	Conclusions	145
7	Conclusions and Future Work	149
7.1	Answers to Research Questions	149
7.2	Summary of Contributions	152
7.3	Discussion	153
7.4	Future Work	155
7.4.1	GUISURFER Extension	155
7.4.2	GUI Reengineering	156
7.4.3	Patterns for GUI transformation	156
	Bibliography	159
A	GUISURFER GUI Meta-Model Specification	179

<i>CONTENTS</i>	xi
B Agenda’s GUI SURFER Script Analysis	181
B.1 Meta-model, Interactor and State Machine Extraction	181
C Agenda’s Windows Behaviour Specification	183
C.1 <i>Login</i> Window	183
C.2 <i>MainForm</i> Window	184
C.3 <i>Find</i> Window	185
C.4 <i>ContactEditor</i> Window	186
D Agenda’s Windows States Extraction	189
D.1 <i>Login</i> Window	189
D.2 <i>MainForm</i> Window	190
D.3 <i>Find</i> Window	190
D.4 <i>ContactEditor</i> Window	190
E Agenda’s Windows Events Sequences Extraction	193
E.1 <i>Login</i> Window	193
E.2 <i>MainForm</i> Window	193
E.3 <i>Find</i> Window	194
E.4 <i>ContactEditor</i> Window	195
F Agenda Script Reasoning through <i>Graph-Tool</i>	197

List of Figures

1.1	A login window	4
1.2	<i>Java/Swing</i> source code for a login window	5
1.3	The login window behaviour	9
2.1	System evolution	19
2.2	A grammar for a login user interface	28
2.3	Finite state machine specification of a login user interface	31
3.1	A GUI application	40
3.2	<i>Agenda's</i> GUI behavioural model automatically produced from its source code	43
3.3	The reverse engineering process	45
3.4	The <i>Agenda's Login</i> window source code	45
3.5	<i>Login</i> window's program dependency graph	51
3.6	Example of code slicing	53
3.7	Regular expression data type in <i>TOM</i>	57
3.8	<i>TOM</i> strategy that normalizes a regular expression	58
4.1	GUI SURFER architecture and retargetability	64

4.2	GUI SURFER applied to a <i>Login</i> window	67
4.3	<i>Java/Swing Login</i> class	71
4.4	Fragment of the <i>Login</i> 's AST	71
4.5	<i>JButton</i> fragment of the <i>Login</i> 's AST	73
4.6	<i>JBank</i> system	82
4.7	Interactor's attributes abstraction	83
4.8	Interactor's actions abstraction	83
4.9	<i>JClass</i> system	85
4.10	<i>JClass</i> system's partial GUI event-flow graph	86
4.11	<i>Agenda</i> 's finite state machine	88
4.12	<i>Agenda</i> application's windows states	90
4.13	Two <i>Agenda</i> applications	94
4.14	<i>Login WxHaskell</i> partial source code implementation	95
4.15	<i>Login</i> behavioural state machine	95
4.16	<i>GWT FlexTable</i> application	96
4.17	<i>GWT FlexTable</i> 's FSM behavioural model	97
5.1	<i>Agenda</i> 's behaviour graph	107
5.2	<i>Agenda</i> application's events Count	108
5.3	Comparing <i>Agenda</i> application's windows states	109
5.4	<i>Agenda</i> 's behaviour graph	112
5.5	<i>Python</i> command for Pagerank algorithm	116
5.6	<i>Agenda</i> 's pagerank results	117
5.7	<i>Python</i> command for Betweenness algorithm	117

5.8	<i>Agenda's</i> betweenness values	118
6.1	<i>HMS</i> : Login window	128
6.2	<i>HMS</i> : Login state machine	129
6.3	<i>HMS</i> : Main window	131
6.4	<i>HMS</i> : Main window state machine	132
6.5	<i>HMS</i> : Main patient form	133
6.6	<i>HMS</i> : Main patient state machine	134
6.7	<i>HMS</i> : View patient information form	134
6.8	<i>HSM</i> : View patient information state machine	135
6.9	<i>HSM</i> : Add doctor form	137
6.10	<i>HSM</i> : Add doctor behavioural state machine	138
6.11	<i>HSM</i> : Billing form	139
6.12	<i>HSM</i> : Billing form behaviour state machine	139
6.13	<i>HSM</i> : The overall behaviour (left part)	140
6.14	<i>HSM</i> : The overall behaviour (center part)	141
6.15	<i>HSM</i> : The overall behaviour (right part)	141
6.16	<i>HSM</i> : The overall behaviour	142
6.17	<i>HSM's</i> pagerank results	144
6.18	<i>HSM's</i> betweenness values	146
7.1	Re-engineering of interactive applications from source code	157

Chapter 1

Introduction

In the context of developing tool support to the automated analysis of interactive systems implementations, this thesis investigates the applicability of reverse engineering approaches to the derivation of user interfaces behavioural models. The ultimate goal is that these models might be used to reason about the quality of the system, both from an usability and an implementation perspective, as well as being used to help systems' maintenance, evolution and redesign.

This Chapter provides an introduction to this research. Section 1.1 presents the motivation of the thesis. Section 1.2 describes the genesis of the thesis. Section 1.3 contains the objectives. Section 1.4 introduces the research questions. Finally, Section 1.5 presents the structure of the thesis.

1.1 Motivation

Developers of interactive systems are faced with a fast changing technological landscape, where a growing multitude of technologies (consider, for example, the case of web applications) can be used to develop user interfaces for a multitude of form factors, using a growing number of input/output techniques. Additionally,

they have to take into consideration non-functional requirements such as the usability and the maintainability of the system. This means considering the quality of the system both from the user's (i.e. external) perspective, and from the implementation's (i.e. internal) perspective. A system that is poorly designed from a usability perspective will most probably fail to be accepted by its end users. A poorly implemented system will be hard to maintain and evolve, and might fail to fulfill all intended requirements. Furthermore, when subsystems are subcontracted, the problem is faced of how to guarantee the quality of the implemented system during acceptance testing. The generation of user interface models from source code has the potential to mitigate these problems. The analysis of these models enables some degree of reasoning about the usability of the system, reducing the need to resort to costly user testing (cf. [DFAB03]), and can support acceptance testing processes. Moreover, the manipulation of the models supports the evolution, redesign and comparison of systems.

1.1.1 User Interface Development

Many types of user interfaces can be identified in interactive systems. Dix et al. [ASD04] identify a number of interaction styles: command line, menus, natural language, question/answer and query dialog, form filling and spreadsheets, WIMP (Windows, Icons, Mouse and Pointer), point and click, and three dimensional. Other types could be included such as haptic, or sketch based.

From a reverse engineering perspective, the focus of interest is on the technology used to implement the interface, whatever interaction strategy is used. The more pervasive approach is to use toolkits consisting of collections of interaction

objects (widgets) to define the user interface contents, and an event-based approach to define the behaviour of the interface. Widgets have a predefined behaviour that is readily available without further programming effort [ASD04]. Widgets are configured via attributes. For example, a button can be configured regarding its size, position, label, which routine is executed when the button is clicked, and so on. Particularly relevant here is the fact that objects react to user actions by executing call-back routines which can change the interface and/or access underlying functionality. From an implementation perspective, user interfaces consist of collections of interaction objects glued together by call-back routines which are executed in response to user actions. This means that understanding the logic of the user interface becomes a non-trivial task.

Nowadays, there are tools which enable developers to see the layout of each screen without having to execute the application. These are user interface development tools. Such tools enable automatic generation of part of the user interface source code and consequently enable creating user interfaces much more quickly. These tools are composed of a screen designer, an object navigator, a property sheet, a toolbox of user interface classes, and a text editor. The screen designer window enables the developer to see and directly manipulate the layout of each screen. The object navigator provides a way of navigating through objects and their associated source code. The property sheet is basically used to have access to all properties names and values of each user interface object. The values of properties can be changed, and developers can immediately see the effect on related objects without having to compile and execute the code. The toolbox of user interface classes is a window containing a set of classes which are used to create user

interface objects. Finally, a text editor is used by developers to enter source code for the application.

User interface development tools enable the construction of large and complex user interfaces. Using these tools, part of the user interface source code is automatically generated and consequently developers need to write a small amount of code only. However, the generated source code can be difficult to understand and maintain.

1.1.2 An Illustrative Example

In this Subsection a simple interactive application is presented and discussed. Figure 1.1 presents a sample Graphical User Interface (GUI) which will be used to illustrate the objectives of the work.



Figure 1.1: A login window

The GUI provides a single window enabling users to login into the system through an username and password pair. The window is composed of several widgets, i.e. two textfield enabling users to introduce their username (textfield with *Username* label) and password (textfield with *Password* label), and two buttons enabling to cancel the window (*Cancel* button) or to confirm input data (*Ok* button).

The source code for programming this particular example is presented in Figure 1.2. The code is written in the *Java* programming language making use of the Swing class library to develop the graphical user interface [LEW⁺02].

```
public class Login extends JFrame {
public Login() {
JButton Cancel = new JButton();
JButton Ok = new JButton();
JLabel jLabel1 = new JLabel();
JLabel jLabel2 = new JLabel();
JLabel jLabel3 = new JLabel();
JTextField login = new JTextField();
JTextField password = new JTextField();
Cancel.setEnabled(true);
Ok.setEnabled(true);
Ok.setText("Ok");
Cancel.setText("Cancel");
jLabel1.setText("Password");
jLabel2.setText("Username");
jLabel3.setText("ClientDB");

Cancel.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent evt)
{System.exit(0);});

Ok.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent evt)
{if (valid(login.getText(),password.getText()))
    {new MainForm().setVisible(true);
    this.dispose();
    }
else showMessageDialog(this,"not valid","Login",0);
});}

public static void main(String args[])
{new Login().setVisible(true);} }
```

Figure 1.2: *Java/Swing* source code for a login window

This *Java* program uses several constructors/methods from the Swing library

to define GUI objects, namely, *JLabel*, *JTextField*, *JButton*, *setText*, *getText* and *setEnabled*.

The constructors used define object types. For example, in the case of the textfields the constructor used is *JTextField*:

```
JTextField login = new JTextField();
JTextField password = new JTextField();
```

Labels are created with the *JLabel* constructor, and the text to be displayed is given as an argument to the *setText* method. In this login window three labels are created, namely *ClientDB*, *Username* and *Password*. As stated, this is done by using the *JLabel* constructor, and the *setText* method to assign text to the labels:

```
JLabel jLabel1 = new JLabel();
JLabel jLabel2 = new JLabel();
JLabel jLabel3 = new JLabel();
jLabel1.setText("Password");
jLabel2.setText("Username");
jLabel3.setText("ClientDB");
```

Finally, buttons are defined via the *JButton* constructor, and the text to be assigned is the argument of the *setText* method. For example, in Figure 1.1 the two buttons are created through the following code:

```
JButton Ok = new JButton();
JButton Cancel = new JButton();
```

The *setText* method is used to assign *Ok* and *Cancel* labels to the previous two buttons, as follows:

```
Ok.setText("Ok");
Cancel.setText("Cancel");
```

Buttons may be enabled, or not, using the *setEnabled* method, i.e. click events can be active or not. In this source code, the buttons are both enabled:

```
Cancel.setEnabled(true);  
Ok.setEnabled(true);
```

Finally, the behaviour of the system when buttons are clicked is defined through *addActionListener* methods. Using this method enables the GUI programmer to define a sequence of instructions to execute when a particular button is clicked by the user. For example, the code executed when the user clicks the *Ok* button is:

```
Ok.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent evt)  
    {if (valid(login.getText(),password.getText()))  
        {new MainForm().setVisible(true);  
        this.dispose();  
        }  
    else showMessageDialog(this,"not valid","Login",0);  
    });}
```

This code extracts textfields contents through the *getText* method. Then an auxiliary function (*valid*) is executed to test whether this input data is valid or not:

```
valid(login.getText(),password.getText())
```

If the user introduces a valid username and password, then the system opens a new window (i.e. executing the *MainForm* window constructor) and closes the *Login* window (i.e. executing the instruction *this.dispose()*):

```
new MainForm().setVisible(true);  
this.dispose();
```

Otherwise, if the user introduces a non-valid username and password, the respective action is implemented by calling the *showMessageDialog* method that sends a message box to the user as a modal window (in this case the action sends the message *not valid*):

```
showMessageDialog(this, "not valid", "Login", 0);
```

The code executed when the user clicks the *Cancel* button is also defined by making use of the *addActionListener* method, as follows:

```
Cancel.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent evt)  
    {System.exit(0);}});
```

This code enables the user to exit the system through the execution of the instruction: *System.exit(0)*.

So far, the login interactive application has been presented/developed by writing its source code (in *Java*, in this case), and GUI functionality has been reused as provided by the Swing library. As can be seen, from the source code it is not possible to graphically visualize the behavior of the system.

Nowadays, user interface development tools can be used. Such tools enable to the quick creation of user interfaces. These tools enable programmers to develop GUIs without the need to write all the source code related to widgets creation and manipulation. Most of the user interface code is automatically generated and these tools enable programmers to visualize the screen layout without having to compile and execute the code. However, with user interface development tools there is no support for GUI behaviour analysis. Developers can only see the behaviour of the system when executing its code. Consequently, new tools are needed for the automatic generation of GUI behaviour from source code and subsequent GUI behavioural reasoning.

Next, a graphical model that (abstractly) specifies the behaviour of the login application is presented. Considering the above source code and its analysis, a

kind of model defined in Figure 1.3 could be useful to understand and test the system. With this model, the system's behaviour can be graphically visualized.

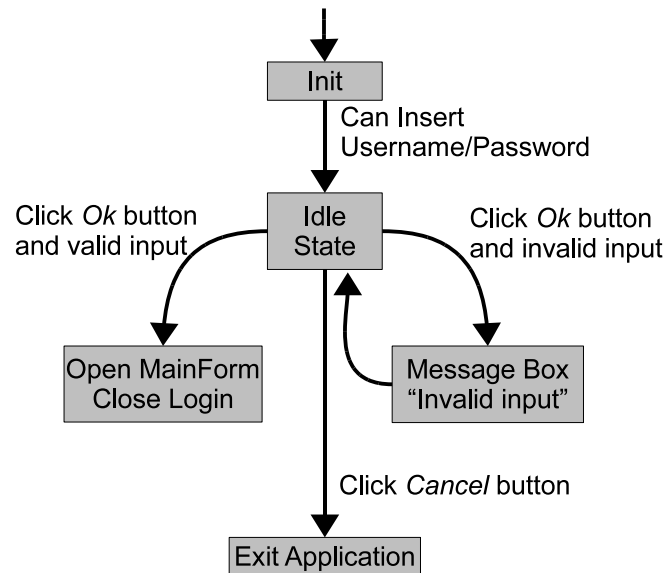


Figure 1.3: The login window behaviour

The model represents user events as arrows and system actions/states as nodes. By using such a graphical model it becomes easier to understand the behaviour of the graphical user interface of the login window. Hence, the system starts from an initial state where users can introduce their usernames and passwords. The model then specifies then another state from which users can execute one of three events. They can click the *Ok* button after entering a valid input. In this case the system responds by opening the *MainForm* window and closing the *Login* window. Users can also click the same button after entering an invalid input. This originates an *invalid input* message box. Afterward, the model defines that the system returns to the previous state allowing users to introduce new username and password. Finally

users can click on the *Cancel* button which causes the application to exit.

In this Section, the source code of a (simple) interactive application (Figure 1.2) has been presented. A behavioural model of that application (Figure 1.3) has been defined. In fact, this is the kind of behavioural model we will automatically generate from source code.

1.2 Thesis Genesis

This thesis has its genesis in a R&D project named IVY, a model-based usability analysis environment¹ which aimed at developing a model-based tool for the analysis of interactive systems designs [SCS06a, SCS06c, SCS06b]. The reverse engineering results obtained during the IVY project are being explored in two others R&D projects, namely, an Infrastructure for Certification and Re-engineering of Open Source Software² (CROSS) and SpreadSheets as a Programming Paradigm³ (SSaAPP).

In the context of the IVY project this research aimed to investigate the applicability of reverse engineering approaches to the derivation of user interface's models amenable for verification of usability related properties. IVY follows from the development of I2SMV [CH01], a compiler enabling the verification of interactive system's models using the SMV model checker [McM93]. The objective was to develop, as a front end to SMV, a model based tool for the analysis of behavioural issues of interactive systems' designs. This tool translates the models into the SMV

¹A model-based usability analysis environment - <http://www.di.uminho.pt/ivy>, last accessed November 22, 2010

²<http://wiki.di.uminho.pt/twiki/bin/view/Research/CROSS/>, last accessed November 22, 2010.

³<http://ssaapp.di.uminho.pt>, last accessed November 22, 2010i.

input language, and fully supports the process of modelling and analysis by providing editors, for models and properties, and a reply visualizer for the analysis of the verification results.

Being modular, the IVY tool acts as a test-bed for different styles of modelling/analysis of interactive systems. One approach that needed to be explored was the use of reverse engineering techniques to enable the generation of models from user interface code. The goal being to support the verification of existing user interfaces in a semi-automated manner.

Next Chapters describe the techniques, models and tools defined to achieve these results and describe also techniques used to reason about extracted models.

1.3 Objectives

Human-Computer interaction is an important and evolving area. Therefore, it is very important to reason about GUIs. In several situations (for example the mobile industry) it is the quality of the GUI that influences the adoption of certain software.

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 the 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 by end users. In many cases users have problems in identifying all the supported tasks of a system, or in understanding how to achieve their goals [LH05].

Moreover, these tools produce *spaghetti* code which is difficult to understand

and maintain. The generated code is composed by call-back procedures for most widgets like buttons, scroll bars, menu items, and other widgets in the interface. These procedures are called by the system when the user interacts with the system through widget's event. Graphical user interfaces may contains hundreds of widgets, and therefore many call-back procedures which makes difficult to understand and maintain the source code [Mye91].

At the same time it is important to ensure that GUI based applications behave as expected [Mem01]. The correctness of the GUI is essential to the correct execution of the software [Ber01]. 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 [SC494, ABD⁺89].

The main objective of this thesis consists in developing tools to automatically extract models containing the GUI behaviour. Models allow designers to analyse systems and could be used to validate system requirements at reasonable cost [MTWH04]. Different types of models can be used for interactive systems, like user and task models. Models must specify which GUI components are present in the interface and their relationship, when a particular GUI event may occur and the associated conditions, which system actions are executed and which GUI state is generated next. Another goal of this thesis is to be able to reason about, and test these GUI models, in order to analyse aspects of the original application's usability, and the implementation quality.

This work will be useful to enable the analysis of existing interactive applications and to evolve/update existing applications [Me196]. In this case, being able to reason at a higher level of abstraction than that of code will help in guarantee-

ing that the new/updated user interface has the same characteristics of the previous one.

As can be seen, the main objective of this thesis is not to develop a new methodology, but indeed to combine a set of methodologies from several area of software engineering in the construction of a tool to solve the described problem.

1.4 Research Questions

Given the objectives above, this research is essentially a study guided by an overarching goal which is to investigate whether:

Interactive application's source code can be used for the automatic generation of GUI behavioural models and subsequent GUI behavioural reasoning through a retargetable approach.

This goal raises a number of issues that need to be addressed. Firstly, there is a need to identify the characteristics of realistic behavioural models of a GUI. Secondly, the research goal is based on a retargetable approach. Here, there is a need to generate GUI behavioural models from several programming languages and paradigms. Thirdly, there is the implication that methods can be used to reason about GUI behavioural models.

Therefore the three primary questions the research needs to address are:

- **Question One:** *Can we infer realistic behavioural models of a GUI from its application's source code?* The source code is the most detailed specification of a software implementation. Through this research question, a methodology needs to be defined allowing to infer behavioural models from source code.

- **Question Two:** *Can we define a language independent technique for GUI modelling and reasoning?* To answer this question, a retargetable technique needs to be investigated, implemented, and applied to different programming languages and paradigms.
- **Question Three:** *Can we use well-known algorithms and metrics to reason about GUI behavioural models?* Considering this last question, algorithms and metrics are required to reason about GUI behavioural models. This needs to be investigated and results need to be evaluated.

1.5 Structure of the Thesis

This thesis is structured into three main logical parts. The first one presents the reverse engineering area relating it to the GUI modelling area. It consists of Chapter 2. Reverse engineering techniques' state of the art, related work and additional methodologies used within this research are firstly described. Then, the Chapter follows with a review of the approaches to model GUIs. A graphical user interface representation is exposed, and the aspects usually specified by graphical user interfaces are described.

The second part presents the approach proposed in this thesis in Chapters 3, 4, 5 and 6. Chapter 3 presents methodologies to retargetable GUI reverse engineering. This contribution has been described in the following papers presented at international and national conferences:

- *Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications*, J.C.Silva, J.C. Campos, J.

Saraiva, presented at the XIII International Workshop on Design, Specification and Verification of Interactive System (DS-VIS 2006), Dublin, Ireland, 2006;

- *Models for the Reverse Engineering of Java/Swing Applications*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering (ATEM 2006), Genova, Italy, 2006;
- *Engenharia Reversa de Sistemas Interactivos Desenvolvidos em Java/Swing*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the “Segunda Conferência Nacional em Interacção Pessoa-Máquina” Universidade do Minho (Interacção 2006), Braga, Portugal, 2006.

Chapter 4 presents the GUISURFER: the developed reverse engineering tool. It describes the GUISURFER architecture, the techniques applied for GUI reverse engineering and respective generated models. These contributions were described in the following papers presented at international conferences:

- *A Generic Library for GUI Reasoning and Testing*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the 24th Annual ACM Symposium on Applied Computing (SAC 2009), USA, 2009;
- *The GUISURFER tool: towards a language independent approach to reverse engineering GUI code*, J.C.Silva, C. Silva, J.C. Campos, J. Saraiva, in proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems (EICS 2010), pages 181-186. ACM, Berlin, Germany, 2010.

Then, Chapter 5, describe the research about GUI reasoning through behavioural models of interactive applications. This contribution was described in the following papers presented in international and national conferences:

- *GUI Behaviour from Source Code Analysis*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the 4th *Conferência Nacional Interação Humano-Computador* (Interação 2010), Aveiro, Portugal, 2010;
- *GUI Inspection from Source Code Analysis*, J. C. Silva, J. C. Campos, J. Saraiva. In proceedings of the 4th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010). Electronic Communications of the EASST, Pisa, Italy, 2010.

Chapter 6 describes the application of GUISURFER to a realistic third-party application.

Finally, the last part (Chapter 7) presents conclusions, discussing the contributions achieved with this research, and indicating possible directions for future work.

Chapter 2

Reverse Engineering Applied to GUI Modelling

In the Software Engineering area, the use of reverse engineering approaches has been explored in order to derive models directly from existing systems. Reverse engineering is a process that helps understand a computer system. Similarly, user interface modelling helps designers and software engineers understand an interactive application from a user interface perspective. This includes identifying data entities and actions that are present in the user interface, as well as relationships between user interface objects.

In this Chapter, reverse engineering and user interface modelling aspects are described [Cam04, DH93]. Section 2.1 describes different types of engineering. Section 2.2 provides details about the reverse engineering area. Section 2.3 identifies the specific type of user interface we will study. Then, the type of GUIs models to be used is discussed in Section 2.4. GUI representations are provided in Section 2.5. Methodologies and tools to reverse engineer interactive systems are described in Section 2.6. Finally, the last Section summarizes the Chapter presenting some

conclusions.

2.1 Types of Engineering

According to Chikofsky [Chi93] the following four terms characterize software systems evolution (Figure 2.1):

- forward engineering: This process implies moving from high-level abstractions to a system's physical implementation. The process moves first from the requirements phase to the design phase. Then from this one to the implementation phase. Therefore, forward engineering corresponds to one or more transitions to a lower abstraction level;
- reverse engineering: Reverse engineering aims to extract information from already existing application systems. It can be described as analysing a software system in order to obtain representations of the system at a higher level of abstraction. This process is the inverse of forward engineering. It consists on a process of analysing a system to discover its components and their interrelationships.
- restructuring: This process is used to change a representation to a new one at the same abstraction level. The system should maintain the same level of functionality as well as semantics. Restructuring transforms the system but functionality remains the same. Code refactoring is an example of restructuring at source code level enabling, for example, to change source code from an unstructured form to a structured one;

- reengineering: This is a process of reverse engineering followed by forward engineering in order to change the system. The main difference between reengineering and restructuring is that while restructuring is made at the same level of abstraction, reengineering involves moving to a higher abstraction level. The reengineering process enables the creation of a specification at a higher abstraction level, new functionalities can then be added to this specification and a new implementation can be developed using forward engineering techniques.

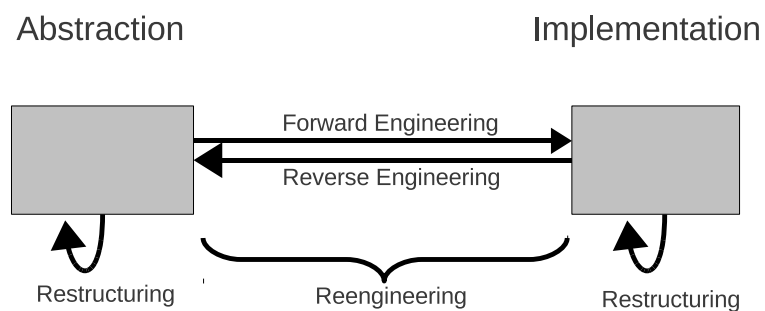


Figure 2.1: System evolution

2.2 Reverse Engineering

Reverse engineering is useful in several tasks like documentation, maintenance, and re-engineering [ESS03].

In the software engineering area, the use of reverse engineering approaches has been explored in order to derive models directly from existing interactive system using both static and dynamics analysis [PFM07, CS01, Sys01]. Static analysis is

performed on the source code without executing the application. Static approaches are well suited for extracting information about the internal structure of the system, and about dependencies among structural elements. Classes, methods, and variables information can be obtained from the analysis of the source code. On the contrary, dynamic analysis extracts information from the application by executing it [Moo96]. Within a dynamic approach the system is executed and its external behaviour is analysed.

Program analysis, plan recognition and redocumentation are applications for reverse engineering [MJS⁺00]. Source code program analysis is an important goal of reverse engineering. It enables the creation of a model of the analysed program from its source code. The analysis can be performed at several different levels of abstraction. Plan recognition aims to recognize structural or behavioural patterns in the source code. Pattern-matching heuristics are used on source code to detect patterns of higher abstraction levels in the lower level code. Redocumentation enables one to change or create documentation for an existing system from source code. The generation of the documentation can be considered as the generation of a higher abstraction level representation of the system.

2.3 Graphical User Interfaces

A Graphical User Interface (GUI) is a graphical front-end to an application that accepts events as input and produces graphical output. A GUI contains graphical widgets. Each widget has a set of properties and respective values which constitutes the state of the GUI. Users interact with the system by performing actions on the GUI. In brief, and from a user's perspective, graphical user interfaces accept as

input a pre-defined set of user-generated events, and produce graphical output.

The most common class of graphical user interfaces are those whose presentation structure consists of a hierarchy of graphical widgets (buttons, menus, textfields, etc) creating a front-end to software systems [Pat00]. An event-based programming model is used to link the graphical objects to the rest of the system's implementation. Users interact with the system by performing actions on the graphical user interface's widgets. These, in turn, generate events at the software level, which are handled by appropriate listener methods. Events cause changes to the state of the interactive application that may be reflected by a change in the presentation layer.

This thesis focuses on techniques to reverse engineer this class of user interfaces. Interactive applications with synchronization constraints or non-deterministic behaviour are not considered in this thesis [JNeC03, dS02].

Our assumptions are the following:

1. An interactive system allows a dialogue between the system and one or more users;
2. Interactive systems only respond after the user provides input, or some internal event has happened;
3. User interfaces contains a set of widgets (windows, buttons, textfields, etc).

A GUI may have multiple windows on screen with interactive objects. Typically, the user makes use of the mouse as a pointing device to select a command from the menu, click on a button, select an item, etc.

There are several types of GUIs such as web-based, form-based, and virtual reality. Web-based user interfaces allow the access remote computers through the Internet or an intranet. They accept input and provide output by generating web pages which can be viewed by the user using a web browser application. Form-based interface are composed by independent graphical windows. Finally, virtual reality user interfaces simulate a real environment with three dimensions.

Like other systems, user interfaces can be sequential or concurrent, synchronous or asynchronous, and timed, timeless or real time. GUIs present a complex structure and a complex event-driven behaviour. Based on the literature, the next Section describes which methodologies are used to model GUIs.

2.4 Types of GUI relevant Models

Model-based development of software systems, and of interactive computing systems in particular, 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 rationale, thus documenting the decision process undertaken during development. Hence, they provide valuable information for the maintenance and evolution of the systems.

User interface models can describe the domain over which the user interface acts, the tasks that the user interface supports, and others aspects of the graphical view presented to the user [MGG⁺95]. The use of interface models gives an abstract description of the user interface, potentially allowing to:

- express the user interfaces at different levels of abstraction, thus enabling

choice of the most appropriate abstraction level;

- perform incremental refinement of the models, thus increasing the guarantee of quality of the final product;
- re-use user interface specifications between projects, thus decreasing the cost of development;
- reason about the properties of the models, thus allowing validation of the user interface within its design, implementation and maintenance processes.

One possible disadvantage of a model based approach is the cost incurred in developing the models. The complexity of today's systems, however, means that controlling their development becomes very difficult without some degree of abstraction and automation. In this context, modelling has become an integral part of development.

However, there is currently no agreement as to which model is more complete for describing user interfaces [Bum96]. In fact, it is commonly argued that a number of models is required, each addressing specific aspects of the design [ACRPM07, FMD97, Doh98, ABD⁺89]. This is also the view that this thesis will take.

Two questions must be considered when thinking of modelling an interactive system:

- which aspects of the system are programmers interested in modelling;
- which modelling approach should programmers use.

These two issues will now be discussed.

In order to build any kind of model of a system, the boundaries of such system must be identified. Therefore the following kinds of models may be considered of interest for user interface modelling:

- *Domain models* are useful to define the domain of discourse of a given interactive system. Domain models are able to describe object relationships in a specific domain but do not express the semantic functions associated with the domain's objects. Hence they define the objects that a user can view, access and manipulate through the user interface.
- *User models* are a first type of model. In its simplest form, they can represent the different characteristics of end users and the roles they are playing. Such user models can provide a way to model preferences for users and group of users, and are supported in some model-based user interface development environments [BY93]. In their more ambitious form, user models attempt to mimic user cognitive capabilities, in order to enable prediction of how the interaction between the user and the device will progress [DBDM98, YGS89];
- *Task models* express the tasks a user performs in order to achieve goals. Task models describe the activities users should complete to accomplish their objectives. The goal of a task model is not to express how the user interface behaves, but rather how a user will use it. Task models are important in the application domain's analysis and comprehension phase because they capture the main application activities and their relationships. Another of

task models applications is as a tool to measure the complexity of how users will reach their goals. Task models can also be used as a documentation method in order to support the users in using the system, and developers in developing it since they are an abstract model of the interaction the system should support;

- *Dialogue models* describe the behaviour of the user interface. Unlike task models, where the main emphasis is the users, dialogue model focus on the device, defining which actions are made available to users via the user interface, and how it responds to them. These models capture all possible dialogues between users and the user interface. Dialog models express the interaction between human and computer. To this end, they stipulate all the widgets the user can interact with (e.g. buttons, commands, etc.) and the results of those interactions on the system;
- *Presentation models* represent the application appearance. They describe the graphical objects in the user interface. Presentation models represent the materialization of widgets in the various dialog states. They define the visual appearance of the interface;
- *Navigation models* defines how objects can be navigated through the user interface from a user view perspective. These models represent basically a objects flow graph with all possible objects's navigation. This view allows to specify which objects are made available to users. Contrary to *dialogue models*, the user interface response is not represented;
- And, finally, *Platform models* define the physical devices that are intended

to host the application and how they interact with each other.

This thesis will focus in generating dialogue models. On the one hand they are one of the more useful type of models to design or analyse the behaviour of the system. On the other hand, they are one of type of models that is closest to the implementation, thus reducing the gap to be filled by reverse engineering.

2.5 GUI Representations

Different type of notations are being used to model graphical user interfaces behaviour. The next Sections describes some of these notations.

2.5.1 Grammars

A grammar defines precisely a formal language by a set of rules which can be used to generate all possible strings in the language [Shn82a]. Grammars are used to analyse if an input string is a member of the language. A grammar can be defined by a quad-tuple (N, T, P, S) , where:

- N is a non-empty finite set of non-terminals;
- T is a finite set of terminal symbols, disjoint from N ;
- P is a finite set of production rules;
- S is the start symbol (a non-terminal from N).

One of the first formalisms used to model user interfaces has been formal grammars [Jac83, HT90, PP98]. They provide a intuitive way to describe the dialogue control of an interactive system, and were a natural choice when user interface were

predominantly command based [Tau90, JBS78]. Several grammars have been used to model user interface aspects. The VEG (Visual Event Grammar) is an example of a grammar based approach for modelling GUIs. The language is supported by a visual editor called Dialog Control Editor [CMP04]. *Multiparty Grammars* is another example [Shn82b]. Considering the human-computer dialogue, this grammar describes both the user and computer sub-languages. Each grammar non-terminal is associated with a label. Non-terminals describing the user interaction are labeled with *H*, nonterminals related to the computer are labeled with *C*. Therefore, this methodology enables specifying, with a single grammar, both user and computer languages and their relationship.

To illustrate the use of a grammar applied to user interface modelling, an example is specified here for a login user interface. The login prompts the user to enter its name. If the name is not valid, then the system asks the user to re-enter it, until he enters a valid name. Then, the system requests a password and if the password entered is incorrect, the user gets one more try to enter a correct one and continue.

A grammar defining this user interface interaction is described [Jac83]. The grammar in Figure 2.2 was extracted from this paper. The grammar is composed by 10 rules (e.g. *getu*, *getpw*, *bappw*, *getsl*, etc). Lower case names denote nonterminal symbols, which are subsequently defined in terms of terminal symbols. Upper case names are terminal symbols. Some definition rules are also annotated with boolean conditions (*cond*), system responses (*resp*), or actions (*act*), all placed in braces. If a rule contains a condition, that condition must be true for the rule to be matched. When a rule is matched, the system will display the response and perform the

actions, if any are given.

```

start: LOGIN
  resp: "Enter name" -> getu
getu: USER
  cond: not EXISTS_USER($USER)
  resp: "Incorrect user name--reenter it" -> getu
getu: USER
  cond: EXISTS_USER($USER)
  resp: "Enter password" -> getpw
getpw: PASSWORD
  cond: $PASSWORD=GETPASSWD_USER($USER)
  resp: "Enter security level" -> getsl
getpw: PASSWORD
  cond: $PASSWORD=GETPASSWD_USER($USER)
  resp: "Incorrect password--reenter it" -> badpw
badpw: PASSWORD
  cond: $PASSWORD=GETPASSWD_USER($USER)
  resp: "Enter security level" -> getsl
badpw: PASSWORD
  cond: $PASSWORD=GETPASSWD_USER($USER)
  resp: "Incorrect password--start again" -> start
getsl: SECLEVEL
  cond: $SECLEVEL>GETCLEARANCE_USER($USER)
  resp: "Security level too high--reenter it" -> getsl
getsl: SECLEVEL
  cond: $SECLEVEL<GETCLEARANCE_USER($USER)
  act: CREATE_SESSION($USER,$PASSWORD,$SECLEVEL) -> end
getsl: ANY
  resp: "Your security level is Unclassified"
  act: CREATE_SESSION($USER,$PASSWORD,Unclassified) -> end

```

Figure 2.2: A grammar for a login user interface, from [Jac83]

As mentioned above, grammars are more oriented towards textual user interfaces than GUIs. Grammar based techniques are difficult to use for describing a number of GUI aspects, for example concurrency. Grammar based specifications also become difficult to use as user interfaces become more complex. In particular when describing complex graphical user interfaces, such as windowed interfaces.

This was particularly the case for direct user interfaces manipulation. In this context, finite state machines, which were also being explored, quickly become a popular alternative for expressing the control logic of user interfaces [Jac83].

2.5.2 Finite State Machines

Finite state machines (FSM) are widely used in modelling system behaviour, in particular, interactive systems behaviour [SS97]. A finite state machine is composed of states, actions and transitions, and can be represented using a state diagram [Jac83]. There are two kinds of state machines: deterministic finite state automaton, i.e. from a state and an input symbol there is only one target state, and nondeterministic finite state automaton, i.e. there are several possible target states from a state and an input symbol [Was85]. Finite state machines are defined as a tuple (I, E, S, X, Y, T) , where:

- I defines the initial state;
- E defines end states;
- S is a finite set of possible states;
- X is a finite set of inputs;
- Y is a finite set of outputs;
- T defines all transitions from a state to another through an input and output.

When modelling a GUI through a state machine, each transition is triggered by a user input. In response to the user input, the system performs an action that can change the state and produces outputs to the user.

As an example, in [Har80] statecharts are used to describe interactive systems. VFMS (Variable Finite State Machine) is an extension to finite state machines for interactive systems modelling. This approach reduces model complexity and focus the attention on more relevant aspects of the states [SS97].

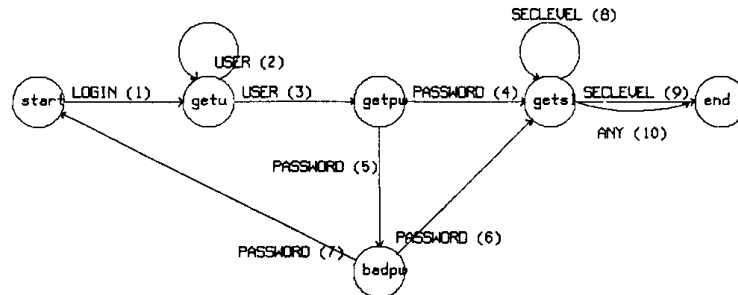
To illustrate the use of a finite state machines applied to user interface modelling, an example is specified in Figure 2.3. This example is retrieved from Jacob's paper [Jac83] and models the behaviour of a login window as described in the previous Section.

The notation follows the usual conventions. Each state is represented by a circle. The start state is represented by a *start* named circle. The end state is represented by a *end* named circle. Transitions between states are defined through arcs. Each arc provides also the name of the input, in capital letters, and, in some cases, a footnote containing boolean conditions, system responses, and actions. A state transition will occur if the input is received and the condition is satisfied. When the transition occurs, the system displays the response and performs the action.

2.5.3 Other Models

GUI models that are based on other formal mathematical notations can also be found in the literature (cf. [TH90]). One advantage of mathematical 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.

A number of different formal approaches have been applied in the specifica-

**Login**

- (1) resp: "Enter name"
- (2) cond: not EXISTS_USER(\$USER)
resp: "Incorrect user name--reenter it"
- (3) cond: EXISTS_USER(\$USER)
resp: "Enter password"
- (4) cond: \$PASSWORD=GETPASSWD_USER(\$USER)
resp: "Enter security level"
- (5) cond: \$PASSWORD≠GETPASSWD_USER(\$USER)
resp: "Incorrect password--reenter it"
- (6) cond: \$PASSWORD=GETPASSWD_USER(\$USER)
resp: "Enter security level!"
- (7) cond: \$PASSWORD≠GETPASSWD_USER(\$USER)
resp: "Incorrect password--start again"
- (8) cond: \$SECLEVEL>GETCLEARANCE_USER(\$USER)
resp: "Security level too high--reenter it"
- (9) cond: \$SECLEVEL≤GETCLEARANCE_USER(\$USER)
act: CREATE_SESSION(\$USER,\$PASSWORD,\$SECLEVEL)
- (10) resp: "Your security level is Unclassified"
act: CREATE_SESSION(\$USER,\$PASSWORD,Unclassified)

Figure 2.3: Finite state machine specification of a login user interface, from [Jac83]

tion of user interfaces. As an example, a window interface has been specified by Clement in [Cle98]. In this work Clement makes use of the VDM-SL language [Pre98] to abstract graphical user interface aspects.

Another set of notations is used to describe graphical user interface behaviour. Examples of these notations are Petri nets and temporal logic. A Petri net is composed by places and transitions. Each place can contain a set of tokens. Transitions move tokens from source places to target places. A transition can only occur if its source places contain all needed tokens, and they can be accepted in the target places. Several works about user interfaces modelling with Petri nets have been published. In [Pal94], Palanque presents a formalism based on Petri nets for modelling event-driven interfaces. In [SRF⁺10] they are used to model ubiquitous environments.

Looking at models from another perspective there are temporal models [BC96]. These models use temporal operators to describe what should or not happen in the system. Another approach is modal action logic (MAL). MAL is a domain specific language for describing interactive systems [CH01]. It uses the notion of interactors as a mechanism for structuring the use of standard specification techniques in the context of interactive systems specification [DH93, RFM91]. MAL is used to specify the behavioural parts of the models.

In recent years, several flavors of extensions to UML have been proposed for user interface modelling. UMLi is basically an extension to UML and adds support for representations commonly occurring in user interfaces [dS02]. DiaMODL allows to model the data flow as well as the behaviour of interaction objects, by combining a data flow oriented language with UML state charts [MT06].

A growing trend in GUI modelling is the use of markup languages. These are typically used in the context of model-based development approaches. An example of one such language is UsiXML (USer Interface eXtensible Markup Language). UsiXML is a XML-compliant markup language that describes a user interface independently of a particular programming language, computing platform and working environment [LVM⁺04]. UsiXML allows for user interfaces to be modelled at several levels of abstraction, enabling analysts, designers, programmers and end-users, to use it during the development life cycle. The language is inspired by the Cameleon framework (Context Aware Modelling for Enabling and Leveraging Effective interactiON), which defines development stages for interactive applications with multiple contexts [CCT⁺03].

2.5.4 Models for Quality Evaluation

In the Human-Computer Interaction area, quality of the GUI 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 empirically assess its usability. Analytic methods have also been proposed as a means of reducing the effort of the analysis. These approaches work by inspection of the system and range from less structured approaches such as Heuristic Evaluation [NM90] to more structured ones such as Cognitive Walkthroughs [LPWR90]. 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.

In Software Engineering concerns are more oriented towards testing the quality of the produced code (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 program execute pre-defined test cases, comparing the results of the execution with the results of some test oracle. In the case of interactive systems, models of the user interface are needed both to help 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 systems.

A typical approach is to run the interactive system and automatically record its states and events. Memon et al. [MBN03] describe a tool which automatically analyses a user interface in order to extract information about its widgets, properties and values. Chen et al. [CS01] 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 studied and analysed the run-time behaviour of *Java* software through a reverse engineering process [Sys01]. Running the target software under a debugger allows for the generation of state diagrams. Then, the state diagrams can be used to examine the overall behaviour of a component such as a class, an object, or a method.

2.6 GUI Reverse Engineering

From the programmers perspective, as user interfaces grow in size and complexity, they become a tangle of object and listener methods, usually all having access to a common global state [Mye91]. Considering that the user interface layer of interactive systems is typically the one most prone to suffer changes, due to changing requirements and request for additional features, maintenance can become a time

consuming and error prone task. Integrated development environments (IDEs), while helpful in enabling the graphical definition of the interface, are limited when it comes to the correctness of the behaviour of the interface. In this case a reverse engineering process is helpful. This Section describes methodologies and tools to reverse engineer interactive systems both through dynamic analysis and static analysis.

2.6.1 Dynamic Analysis

A typical approach using dynamic analysis is to run the interactive system and automatically record its state and events. Several works are described in the literature.

As already mentioned in Section 2.5.4, Chen et al. [CS01] propose a tool to test user interfaces. The solution provides a visual environment for manipulating test specifications of GUI-based applications in *Java*. This visual environment uses dynamic analysis techniques to obtain information about the GUI under test in order to generate concrete test cases. Users can then graphically manipulate these test specifications. The prototype runs with conditions and limitations, i.e. GUI components must be visible, initialized and defined as public variables, the *Java* version should be 1.1 or higher and all GUI objects must be serializable.

Systa studies the run-time behaviour of *Java* software through a reverse engineering process [Sys01]. Systa's paper discusses an experimental environment that has been built to reverse engineer *Java* software. The static information is extracted from the byte code and it is then analysed (see Section 2.6.2 for a discussion on static analysis). The dynamic event trace information is generated automatically as a result of running the target system against predefined execution scenarios under

a debugger. Running the target software under a debugger allows for the generation of state diagrams. These state diagrams can be used to examine the overall behaviour of a component such as a an object, or a single method, disconnected from the rest of the system.

Paiva et al. [PFM07] proposes a tool to reverse engineer structural and behavioural formal models of a GUI application, again by means of a dynamic technique. The application under test is automatically explored through its GUI to discover as much as possible of the GUI behaviour and to generate a corresponding GUI model. The tool provides a front-end for automatic and manual exploration. Manual exploration mode is used to overcome situations when the automatic exploration process cannot progress because of dependencies that it cannot discover or because of functionalities that might be unaccessible (e.g. because they are protected by a password). The model, automatically generated by the reverse engineering process, has to be validated and completed manually so that it can be used as a test oracle in a model-based testing setting. From this revised model, abstract test cases are generated and executed over the GUI to check the conformity between the model and the implementation with the help of the Spec Explorer tool [VCG⁺08].

2.6.2 Static Analysis

As an alternative to the dynamic analysis, some researches apply static analysis. The reverse engineering process is based on analysis of the application's source code, instead of its execution. One such approach is the work by d'Ausbourg et al. [dDR96] in reverse engineering UIL code (User Interface Language – a language to

describe user interfaces for the X11 Windowing System, see [HF94]). 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). The work consists of generating and analysing models of behaviour from the UIL specification of the system. The specification is produced by a designer using generator tools. The objective is to design and implement a model builder in order to analyse and verify the derived model. This model is used also to generate test cases in order to test the application against its specification.

Moore [Moo96] describes another 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. The first implemented step is to identify the User Interface Subset (UIS). Essentially, the UIS includes all routines and data structures that are affected by the user interface. The next step enables the identification of data structures that are related to input/output. Finally, rules are used to identify statically user interface components from legacy code. Following these rules and procedures, the approach helps in detecting the functionality of the existing user interface.

2.7 Conclusions

This Chapter introduced Reverse Engineering, a technique which is useful in several software engineering tasks like documentation, maintenance and reengineering. Two kinds of reverse engineering processes were described: static and dynamic analysis. Several approaches exist, each aiming at particular systems and

objectives. One common trend, however, is that the approaches are not retargetable, i.e. in all cases it is not possible to apply the approach to a different language than that it was developed for. Considering the plethora of technological solutions currently available to the development of GUIs, retargetability is an helpful/important feature. As a solution, this research proposes that static analysis can be used to develop a retargetable tool for GUI analysis from source code.

Several models may be considered for user interface modelling. *Task models* describe the tasks that an end user can performs. *Dialogue models* represent all possible dialogues between users and the user interface. *Domain models* define the objects that a user can view, access and manipulate through the user interface. *Presentation models* represent the application appearance. *Platform models* define the physical system used to host the application. The goal of the approach will be the generation of *dialogue models*.

As described in this Chapter, grammars were very common to specify command-based user interfaces. However they are so not well adapted to model concurrency of the modern windowed systems. A grammar-based specification does not represent state explicitly. Without an explicit representation of state, it is hard to represent the state viewed by the user. State machines are another used interface models. State-based specifications are better adapted to model GUIs application.

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

Chapter 3

An Approach to GUI Reverse Engineering

This thesis presents work on interactive systems analysis through reverse engineering of GUI (see [SCS06a, SCS06c, SCS06b, SCS09, SCS, SCS10a, SCS10b]). The goal is to produce a fully functional reverse engineering prototype tool. The tool must be able to derive user interface models of interactive applications. Therefore, the research revolves around the three following identified areas:

1. GUI reverse engineering;
2. GUI modelling;
3. Model-based GUI reasoning.

This Chapter describes an approach to GUI reverse engineering. The approach makes use of static analysis as in [Moo96]. When compared to their work, the current reverse engineering approach aims to be retargetable to different programming languages and not to be limited to a particular implementation technology.

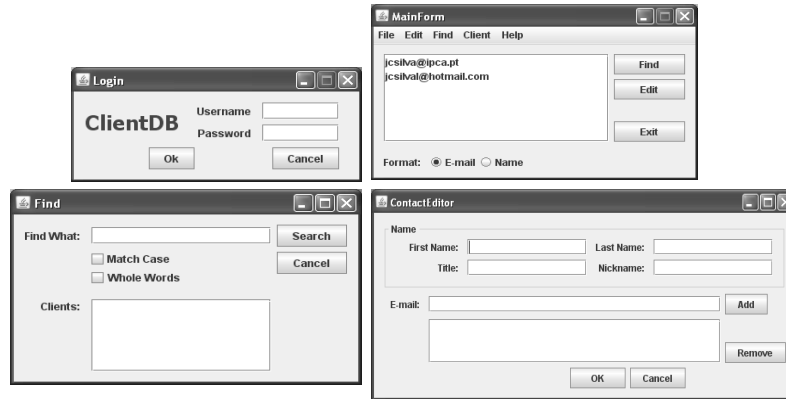


Figure 3.1: A GUI application

Section 3.1 describes an example of an interactive application to be analysed. Section 3.2 describes an approach for GUI abstraction. Section 3.3 provides details about a methodology for GUI extraction from source code. Section 3.4 presents a methodology for a retargetable process. The behavioural models' generation approach is defined in Section 3.5. Finally, the last Section presents some conclusions.

3.1 An Interactive Application as Running Example

Throughout this document we will make use of interactive applications as running examples. The first application, named *Agenda*, 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 by giving its name. The application consists of four windows, named *Login*, *MainForm*, *Find* and *ContactEditor*, as shown in Figure 3.1.

We will use this example to present our approach to GUI reverse engineering.

Let us discuss it in detail. The initial *Login* window (Figure 3.1, top left window) is used to control users' access to the agenda. Thus, a login and password have to be introduced 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. On the contrary, 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 3.1, 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 3.1, bottom left window). This window is used to search and obtain a particular contact's data given its name. By pressing the *Edit* button in the main window, the user opens the *ContactEditor* window (Figure 3.1, 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 list of e-mail addresses of the contact. If there are no e-mails in the list then the *Remove* button is automatically disabled.

Until now, the structure and behaviour of this interactive application has been informally described. Such descriptions, however, can be ambiguous and often lead to different interpretations of what the application should do. In order to unambiguously and rigorously define an application, a model may be used. Moreover, by using a model to define the interactive application, techniques can be used to refactor, manipulate and test such applications. Figure 3.2 shows a possible model to specify the behaviour of the running example: a finite state machine as presented

in Section 2.5.2. In this machine, states represent the GUI idle periods, i.e. when there are no relevant events or actions being executed (filled boxes such as *state1*, *state2*, etc), and the transitions between states are defined by the events associated with the GUI objects. These are modeled in figure 3.2 by arrows (e.g. the labeled arrow *Press “Ok” button and valid user/pass*). Moreover, the GUI actions executed when a specific event occurs are represented using white boxes (e.g. the *Open “MainForm” window* box).

This model is less verbose than our initial informal description and easier to understand. For example, it can be seen that the action performed when the *Ok* button is pressed defines a transition in the machine to the same state if the username/password pair is not valid, or into a different state, otherwise.

Considering the case where a valid pair username/password was given, then the transition labeled with *Press “Ok” button and valid user/pass* moves the system to a different state (*state2* of the *MainForm* window) and two GUI actions are executed: the close of the *Login* window, and the opening of the *MainForm* window.

As an illustration, the GUI model in Figure 3.2 (apart from some beautifying) was automatically generated from the source code of the *Agenda* application by the tool developed in this thesis.

Well-known techniques can be used to detect properties of the interface. For example, graph-based algorithms may be applied to compute if all states are accessible from the initial one, in order to detect whether a particular window of the application will ever be displayed or not. Valid or invalid *sentences* of the language may be also defined by the machine to be used as test cases. These test cases can

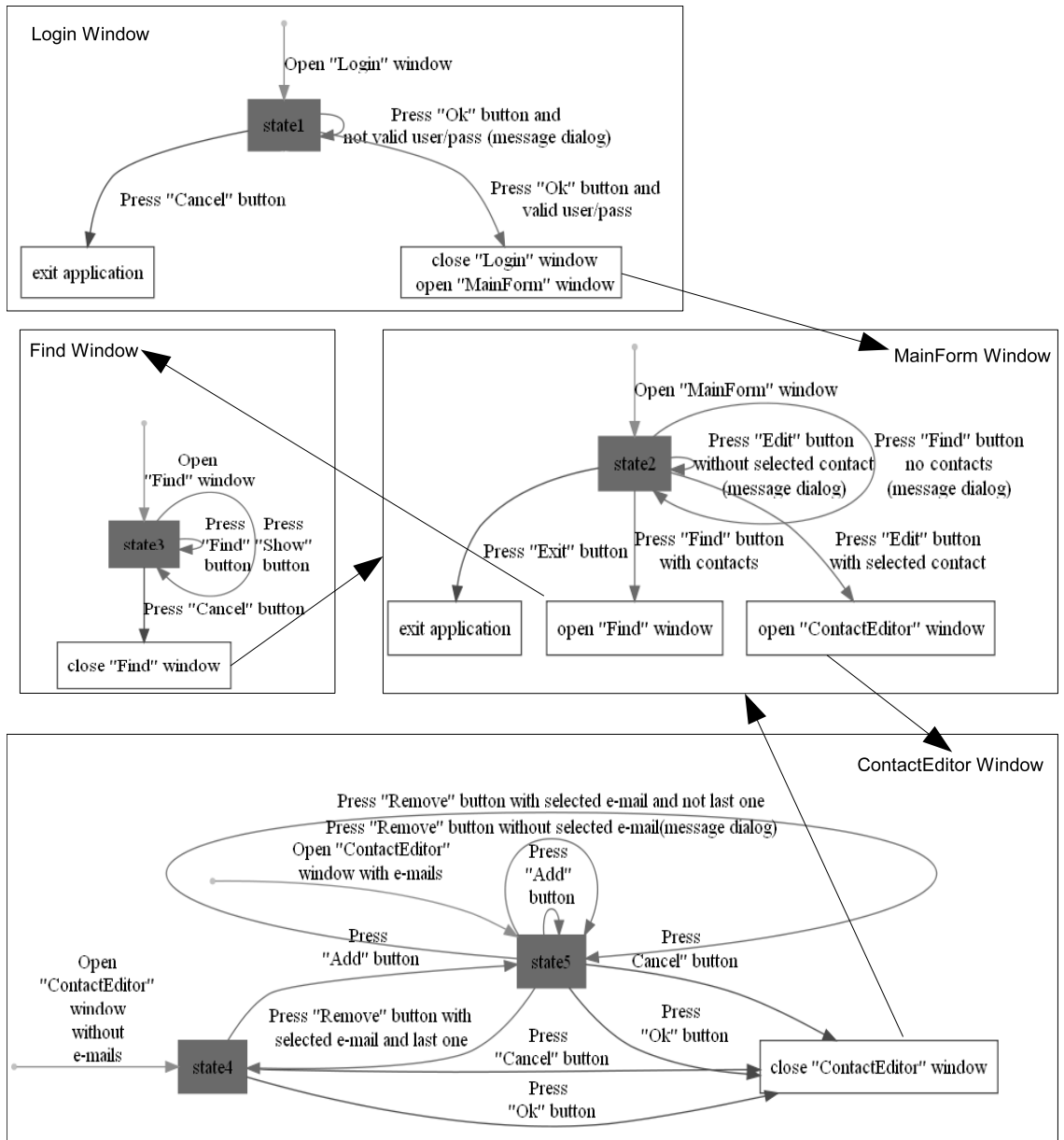


Figure 3.2: Agenda's GUI behavioural model automatically produced from its source code

be used to prove more advanced properties of the interface, as will be discussed in Chapter 5.

3.2 GUI Reverse Engineering Approach

In order to extract a behavioural model (for example, the one described in Figure 3.2) from source code, we follow the reverse engineering approach in Figure 3.3.

The goal is to be able to extract a range of models from the source code of interactive systems, focusing on models that represent the behaviour of the GUI. That is, models defining which are the graphical components of a GUI and their relationship, when can a particular GUI event occurs, which are the related conditions, which system actions are executed, and which GUI state is generated next. This type of models has been chosen to enable reasoning about GUI models in order to analyse aspects of the original application's behaviour.

To define such GUI models, a small set of abstractions is used for the interactions between the user and the system. To illustrate this set of abstractions, the source code of the *Login* window of the *Agenda* application described in Section 3.1, will be used. Figure 3.4 contains the *Java* source code of the *Login* window. This is essentially the same code already described in Section 1.1 (see Figure 1.2).

As explained, this code was written in the *Java* programming language making use of the Swing class library which allow programmers to easily develop the graphical user interface. A detailed description of the code has already been provided. In brief, constructors are used to create widgets, methods are used to set attribute values, and event listeners are attached to widget events. Event listeners are the methods that implement the behaviour of the user interface.

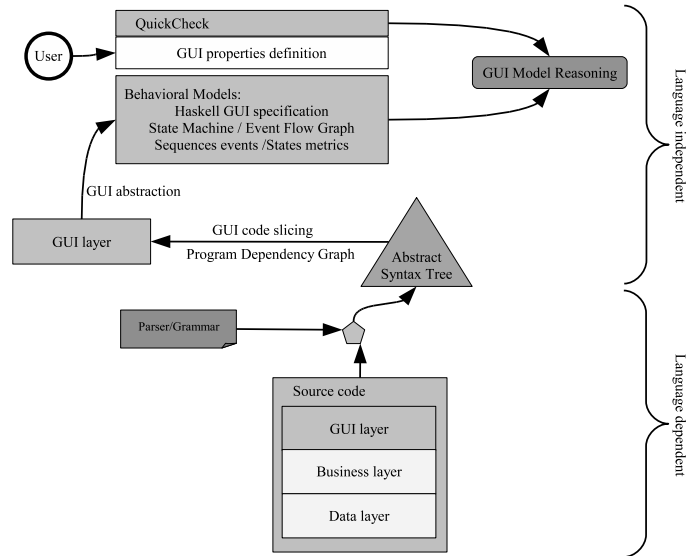


Figure 3.3: The reverse engineering process

```

JButton Ok = new JButton();
JButton Cancel = new JButton();
Ok.setText("Ok");
Cancel.setText("Cancel");
JTextField login = new JTextField();
JTextField password = new JTextField();
Ok.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent evt) {
if (validLogin(login.getText(), password.getText()))
{ new MainForm().setVisible(true);
this.dispose();
}
}
else showMessageDialog(this, "not valid", "Login", 0);
}});

```

Figure 3.4: The Agenda's Login window source code

Now, a set of abstractions will be defined over the above code relating it to the four types of interactions between the user and the system, as follows:

- *User input*: Any data inserted by the user.

In this particular case, abstractions must be defined for values introduced by users through textfields. Considering the code fragment, this can be done by extracting the instructions related to textfield input (the method *getText()* returns the textfield content):

```
JTextField login = new JTextField();
login.getText();
```

and

```
JTextField password = new JTextField();
password.getText();
```

- *User selection*: Any choice that the user can make between different options, such as buttons instructions.

```
JButton Ok = new JButton();
JButton Cancel = new JButton();
```

- *User action*: Any GUI action that is performed as the result of user input or user selection, such as the listener related to the *Ok* button in the example.

```
public void actionPerformed(ActionEvent evt) {
    if (validLogin(login.getText(), password.getText()))
    { new MainForm().setVisible(true);
      this.dispose();
    }
    else showMessageDialog(this, "not valid", "Login", 0);
    ...
}}
```

In this case, the code contains two GUI actions that are executed when users press the *Ok* button. First, the *Login* window is closed and, then, a new window form is opened when the pair username/password is valid:

```
new MainForm().setVisible(true);  
this.dispose();
```

and if the pair is not valid, a message dialog is displayed:

```
showMessageDialog(this, "not valid", "Login", 0);
```

- *Output to User*: Any communication from the application to the user, such as a user dialogue. As an example, if the user does not provide a valid login and password then a message dialog is displayed.

```
showMessageDialog(this, "not valid", "Login", 0);
```

- *GUI control flow*: The control structure needs also to be identified, i.e. user input, user selection, user action, or user output may be related to particular conditions. As an example, the actions performed when the user presses the *Ok* button are only executed if the following condition is true:

```
validLogin(login.getText(), password.getText())
```

From the user interface source code of an interactive system, and this set of abstractions, our research aims to generate its GUI behavioural models. The methodology explained in this Section helps to identify models for an interactive application. This includes identifying data entities and actions that are involved in the graphical user interface, as well as relationships between user interface components.

3.3 GUI Source Code Extraction

The GUI source code extraction process starts by defining/reusing a front-end for the programming language of the interactive application's source code. Modern parser generators automatically produce a parser and the construction of the Abstract Syntax Tree (AST) given the context-free grammar defining the programming language of the source code. Using this front-end, an AST is obtained from the source code of the system for which the user interface related code is to be analysed. Then, the process needs to identify all fragments in the AST that are members of the GUI layer. To achieve this the set of abstractions described in the previous Section is used.

In order to extract user interface relevant data from the AST, a slicing function is proposed [Tip95, Luc01] which isolates the GUI sub-program from the entire program. The straightforward approach would be to define an explicit recursive function that traverses the AST of the program and returns the GUI sub-trees.

However, a typical grammar/AST for a real programming language (like *Java*) has more than one hundred of non-terminal symbols and productions [AV05]. As a result, writing a function to traverse the AST forces the programmer to have full knowledge of the grammar and to write a complex and long mutually recursive function.

We propose the use of an alternative approach by using the strategic programming. In this style of programming, there is a pre-defined set of (strategic) generic traversal functions that traverse any AST using different strategies. Thus, the programmer is able to focus in the nodes of interest only. In fact, the programmer

does not need to have a deep knowledge of the entire grammar/AST, but only of those parts he is interested in (the GUI sub-language in this case). The goal for strategic programming is to be reused across different programming languages and paradigms. Generic techniques can be used to work with *any* AST and not with a particular one only.

Thus, for our reverse engineering approach, we heavily rely on two language-independent techniques, namely code slicing [Tip95] and strategic programming [Vis03a, VS04].

3.4 Retargetable Methodology Through Generic Programming

Generic programming aims at the definition of algorithms and data structures at an abstract or generic level [DJ05]. Code slicing and strategic programming are two forms of generic programming and are both based on program dependency graphs. This Section describes these techniques in more detail.

3.4.1 Program Dependency Graph

A program dependency graph for a program P is a mathematical abstraction, namely a graph, and consists of a set of vertices, and a set of edges. Each edge connects two vertices in the graph [HR92]. In other words, a graph is a pair (V, E) , where V is a finite set and E is a binary relation on V . V is called a vertex set whose elements are called vertices. E is a collection of edges, where an edge is a pair (u, v) with u, v in V . Graphs are directed or undirected. In a directed graph, edges are ordered pairs, connecting a source vertex to a target vertex. In an

undirected graph edges are unordered pairs of two vertices.

A program dependency graph is a directed graph G whose vertices represent the assignment statements and predicates of P . In addition, G includes a special entry vertex which is the source of the dependency graph. The edges of the graph represent control and data dependence. The intuitive meaning of a dependence edge from vertice u to vertice v is the following:

- if the program component represented by vertice u is evaluated/validated during program execution, then, assuming that the program terminates normally, the component represented by v will eventually execute;
- if the vertice u is not evaluated/validated, then the component represented by v will never execute.

Figure 3.5 contains the program dependency graph of the *Login* source code shown in Figure 3.4. For example, the statement `new MainForm().setVisible(true);` will be executed only if expression `validLogin(login.getText(), password.getText())` is valid. Furthermore, this last statement will be executed only if the following statements have been previously executed: `JTextFieldlogin = newJTextField();` `JTextFieldpassword = newJTextField();` and `Ok.addActionListener`. These dependencies can be identified in the graph.

Program dependency graphs are the basis for code slicing which is discussed in the next Section.

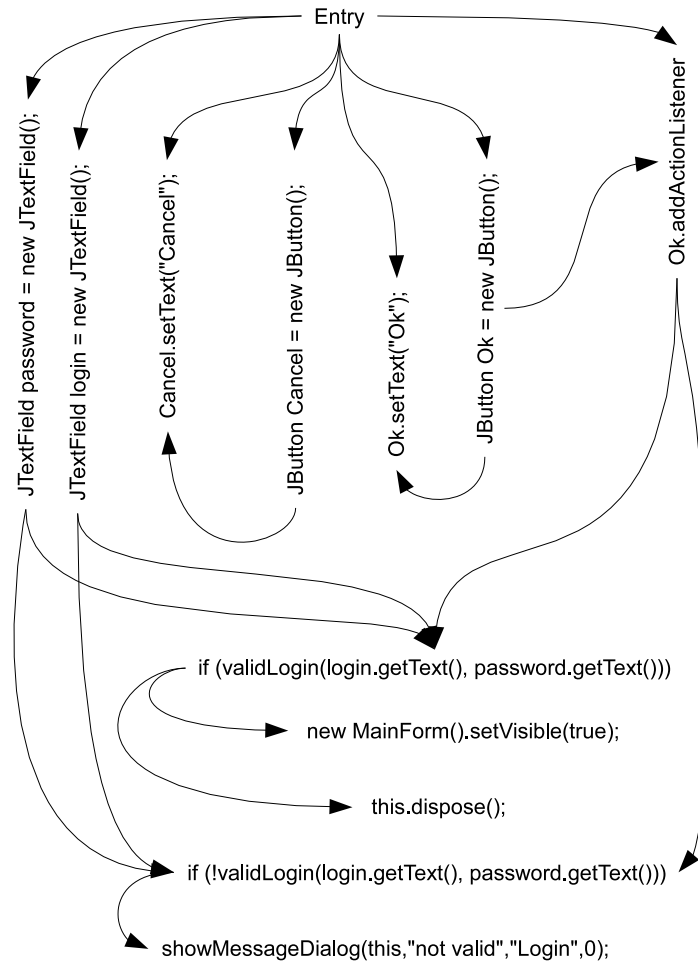


Figure 3.5: *Login* window's program dependency graph

3.4.2 Code Slicing

Code slicing is a form of generic programming. Basically, a slicing process discards those parts of the program which can be determined to have no effect upon the semantics of interest. Hence, a slice is a subset of program statements that preserves a predefined subset of the original behaviour of the program. Applications of program slicing include software testing, program debugging, measurement, reverse engineering, program restructuring, etc. Most of the applications for slicing are related to software testing and debugging, and to software maintenance tasks [Tip95, Luc01].

In practical terms, the objective of program slicing is to remove statements from a program that do not affect the values of variables at a point of interest. Hence, the first step in code slicing is to identify the point of interest (i.e. a statement in the program to be sliced) and a set of variables of interest. This is called the slicing criterion. A program slice is then executed by discarding statements that can not affect (or can not be affected by, depending on the type of slicing) the values of the specified variables at the given point of interest.

There are two main types of code slicing, backward and forward slicing [HR92]. Backward slicing is executed by traversing backwards the code from the point of interest finding all statements that are related to the specified variables at the point of interest and removing the other statements. Conversely, forward slicing is executed forward from the point of interest finding all statements that can be affected by changes to the specified variables at the point of interest.

Figure 3.6(a) highlights in bold the fragments of the login source code that

<pre> JButton Ok = new JButton(); JButton Cancel = new JButton(); Ok.setText("Ok"); Cancel.setText("Cancel"); JTextField login = new JTextField(); JTextField password = new JTextField(); Ok.addActionListener(new ActionListener() { public void actionPerformed(ActionEvent evt) { if (validLogin(login.getText(), password.getText())) { new MainForm().setVisible(true); this.dispose(); } else showMessageDialog(this,"not valid","Login",0); }); </pre> <p>(a) Backward slice from "Ok.addActionListener"</p>	<pre> JButton Ok = new JButton(); JButton Cancel = new JButton(); Ok.setText("Ok"); Cancel.setText("Cancel"); JTextField login = new JTextField(); JTextField password = new JTextField(); Ok.addActionListener(new ActionListener() { public void actionPerformed(ActionEvent evt) { if (validLogin(login.getText(), password.getText())) { new MainForm().setVisible(true); this.dispose(); } else showMessageDialog(this,"not valid","Login",0); }); </pre> <p>(b) Forward slice from "JButton Ok = new JButton();"</p>
---	--

Figure 3.6: Example of code slicing (backward (a) and forward (b) slicing)

are in the backward slice with respect to the statement: *Ok.addActionListener()*. And Figure 3.6(b) highlights in bold the fragments of the same program that are in the forward slice with respect to statement: *JButton.Ok = newJButton()*.

3.4.3 Strategic Programming

In the two previous sections we have discussed two software engineering techniques that will help us to automatically extract the GUI model from the source code of an application. Now, we present a generic technique to traverse *any* AST representing the source code expressed in *any* programming language.

Strategic programming is another form of generic programming and is useful for program construction, allowing a high level of composability and traversal control through strategic functions [VS04]. For example, strategic programming can be used to manipulate large heterogeneous data structures like an AST representing a *Java* program [Vis03b, Vis03a]. Strategic programming has been defined in different programming paradigms [LV03], and can work on different data types (*e.g.*, lists, binary trees, ASTs) providing the right setting to express generic traversal

functions.

Strategic programming follows two programming concepts: *dynamic type-case*, and *one-step traversal*:

- *Dynamic type-case* allows for the computation of both generic behaviour and type-specific behaviour: depending on the type of input data, either the type-specific behaviour is executed, or the generic default behaviour is;
- *One-step traversal* makes generic traversals through the use of combinators.

To implement the above two concepts in strategic programming, several combinators are offered. The mathematical representation of some of the basics combinators proposed by strategic programming are:

- *id* - return input term unchanged;
- *sequence(f,g)* - apply *f* to the input term, and *g* to the result of that;
- *all(f)* - apply *f* to all immediate subterms of the input term;
- *fail* - react to any input term with failure;
- *choice(f,g)* - dynamic type-case combinator used to apply *f* to the input term. If it fails, apply *g* instead;
- *one(f)* - apply *f* to a single immediate subterm of the input term;
- *adhoc(f,g)* - dynamic type-case combinator used to apply *g* to the input term if its type matches, otherwise *f* is applied;
- *apply(f,t)* - apply strategy *f* to input term *t*.

Making use of these combinators, strategic programming enables the definition of different types of generic traversals. Examples of generic traversals are:

- $bottomup(f) = sequence(all(bottomup(f)),f)$
- $topdown(f) = sequence(f,all(topdown(f)))$

The composed combinators *bottomup* and *topdown* model full bottom-up, or top-down, recursive traversal schemes, respectively. They apply their argument strategy at the root of the incoming data, and at all its immediate and non-immediate components.

These combinators can be used to traverse particular structures passing as parameter a working function. For example, the *adhoc* combinator can be used to create the *nodeAction* working function, and pass it as argument to the *bottomup* combinator:

- $nodeAction = adhoc(id, \lambda x \cdot .x > 0)$
- $searchPositives = bottomup(nodeAction)$

The *nodeAction* function can be executed with a float number as parameter, returning the true boolean value if the parameter is positive and false otherwise. When applied to any other input term, the function returns the input term unchanged. Thus, the traversal *searchPositives* works on input terms of any type. This composed combinator follows a bottom-up traversal scheme and, through the *nodeAction* working function, transform numerical values into boolean values, i.e. true value if positive and false value otherwise.

The above example serves to illustrate dynamic type-case and one step traversal concrete concepts. In this particular case, the type-case concept is specified through the *ad hoc* combinator, and the one step traversal concept is defined with the use of the *all* and *sequence* combinators.

Strategic programming can be easily used with the object-oriented or functional programming paradigms. Two examples of application of strategic programming within these programming paradigms are the *TOM* [Mac10] or *Strafunski* libraries [LV03], respectively. *TOM* is a framework for programming rule-based systems, allowing the manipulation of any structure. *TOM* incarnates the strategic paradigm through an extension to the *Java* programming language. All strategies defined in this Section are implemented in *TOM*. *Strafunski* is a functional software bundle that also aims to provide generic programming and language processing capabilities, in this case for the *Haskell* programming language.

3.4.4 Case Study: Regular Expressions Processing with *TOM*

As an illustrative case study the implementation, using *TOM*, of an interpreter to process and normalize regular expressions will now be described.

Regular Expressions are used in the recognition of character patterns and are composed by the following constructors:

- Sequence of expressions: ab, abc ;
- Alternative combinator (i.e. $|$), which defines the occurrence of one of two expressions: $a|bc, ab|c$;
- Optional combinator (i.e. $?$), which defines the occurrence of the expression

between zero or one times: $a?$, $(abc)?$;

- Star combinator (i.e. $*$) defining that there are zero or more occurrences of an expression: a^* , $a|b^*$, $(abc)^*$;
- Plus combinator (i.e. $+$) defining that there are one or more occurrences of an expression: a^+ , $a(bc)^+$;
- Terminal symbols: $(,), *, +, ?, |, 'a', 'b', 'c', \dots, 'z'$ and ϵ .

Examples of regular expressions are: $a?b+c^*$, $(abc)?(def)^*g^+$, $(aaa)^*b+c^*$, etc.

To normalize a regular expression, all occurrences of optional combinators $a?$ must be transposed to the expression $a|\epsilon$, where ϵ is the empty expression.

To implement an interpreter for regular expressions processing and normalization in *TOM*, a data type must be defined for the language to be used. Figure 3.7 provides a data type defined in *TOM* for regular expressions. The list of constructors from the above definition of regular expressions is transposed in the same order to the data type list in Figure 3.7. For example, the optional and star combinators are implemented with the *Opt* and *Star* data types.

```

RegExp =
  Seq(s1:RegExp , s2:RegExp)
  | Alt(a1:RegExp , a2:RegExp)
  | Opt(o:RegExp)
  | Star(s:RegExp)
  | Plus(p:RegExp)
  | Literal(c:String)
  | Empty()

```

Figure 3.7: Regular expression data type in *TOM*, adapted from [Mac10]

Finally, *TOM* strategies can be used to implement a solution for regular expression processing and normalization. To normalize any regular expressions, the solution needs to consider all occurrence of optional combinator (*Opt*) replacing them by an (*Alt*). Thus, a full traversal is needed. Figure 3.8 provides a possible solution. This solution transposes all occurrences of the *Opt(o)* combinator to *Alt(Empty(),o)*. This is done through the following instruction:

```
visit RegExp {Opt(o) -> {return `Alt(Empty(),o)}};
```

A full traversal bottom-up approach making use of the *BottomUp* strategy, applies *visit* to all nodes in the expression:

```
`BottomUp(Norm()).visit(rg);
```

However, the top-down scheme can also be applied, because both provide a full traverse over regular expressions specification.

```
%strategy Norm() {
visit RegExp{
  Opt(o) -> {return `Alt(Empty(),o); }
}
}
`BottomUp(Norm()).visit(rg);
```

Figure 3.8: *TOM* strategy that normalizes a regular expression, adapted from [Mac10]

3.4.5 Retargetable Methodology

In this thesis we investigate the use of the presented (generic) programming methodologies in order to reason about interactive applications implemented in different programming languages (i.e, *Java*, *GWT*, and *Haskell*). This approach proposes to

make use of generic programming and to apply it to different programming languages, i.e. *Java* (*Swing* or *GWT* toolkit) and *Haskell* programming languages [LEW⁺02, JHA⁺99, HT07]. The methodology will enable us to extract graphical user interface AST fragments through code slicing and strategic programming. Following a retargetable methodology, we will be able to extract GUI fragments from any AST, i.e. *Java/Swing*, *WxHaskell*, *C#*, etc. This will allow us to identify all of the program fragments that interact with the graphical user interface.

Strategic programming and code slicing are techniques easily retargetable to different programming languages. The next Chapter, the application of code slicing and strategic programming processes will be described in more detail.

3.5 Behavioural Models Generation

After defining a methodology to extract GUI related data, it is important to generate behavioural models. The approach proposes the generation of several kinds of models. In this Section, different models are enumerated. A detailed description of the different models, with examples of their application is provided in Chapter 4, where the GUI SURFER tool will be discussed.

First, by using the *Haskell* programming language, the approach aims to model GUI behaviour through a specification which maps events and related conditions to a list of GUI actions references. This list of actions needs also to contain other relevant information, such as the window name and initial state, the window close and end application actions references, as well as new windows action references.

Another notation used for describing interactive systems is the language of MAL interactors. It is a domain specific language and includes the notion of inter-

actors as a mechanism for structuring the use of standard specification techniques in the context of interactive systems specification [DH93].

For a visual experience and easier reasoning, event-flow graphs and finite state machines will be used. Event-flow graph will enable the abstraction of all the interface widgets and their relationships, and the definition of all possible interactions by specifying the events in a GUI system [MBN03]. For every event e , an event flow graph identifies all the events that can follow event e . Finite state machine may be used to model GUI behaviour considering GUI events, related conditions, system actions execution, etc. This type of model has been chosen in order to be able to reason about, and test, the dialogue supported by a given GUI implementation.

3.6 Conclusions

This Chapter described the main building blocks of our retargetable approach to graphical user interface reverse engineering. Behavioural models may be generated which capture graphical user interface behaviour by detecting components in the user interface through source code analysis. These components include user interface objects, events, actions and respective control flow. The technique explained will help in identifying graphical user interface abstractions from source code.

The contributions related to this Chapter were described in the following papers presented at international and national conferences:

- *Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications*, J.C.Silva, J.C. Campos, J.

Saraiva, presented at the XIII International Workshop on Design, Specification and Verification of Interactive System (DS-VIS 2006), Dublin, Ireland, 2006;

- *Models for the Reverse Engineering of Java/Swing Applications*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering (ATEM 2006), Genova, Italy, 2006;
- *Engenharia Reversa de Sistemas Interactivos Desenvolvidos em Java/Swing*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the “Segunda Conferência Nacional em Interacção Pessoa-Máquina” Universidade do Minho (Interacção 2006), Braga, Portugal, 2006.

Chapter 4

GUISURFER: A Reverse Engineering Tool

This Chapter describes GUISURFER, a tool developed as a testbed for the reverse engineering approach proposed in the previous Chapter. The tool automatically extracts GUI behavioural models from the applications source code, and automates some of the activities involved in the analysis of these models.

This Chapter is organized as follows: Section 4.1 describes the architecture of the GUISURFER tool. Section 4.2 provides details about its implementation. Section 4.3 presents models used within the tool for GUI reverse engineering. A description about the retargetability of the tool is provided in Section 4.4. Finally, Section 4.5 presents some conclusions.

4.1 The Architecture of GUISURFER

One of GUISURFER's development objectives is making it as easily retargetable as possible to new implementation languages. This is achieved by dividing the process in two phases: a language dependent phase and a language independent phase,

as shown in Figure 4.1. Hence, if there is the need of retargeting GUI SURFER into another language, ideally only the language dependent phase should be affected.

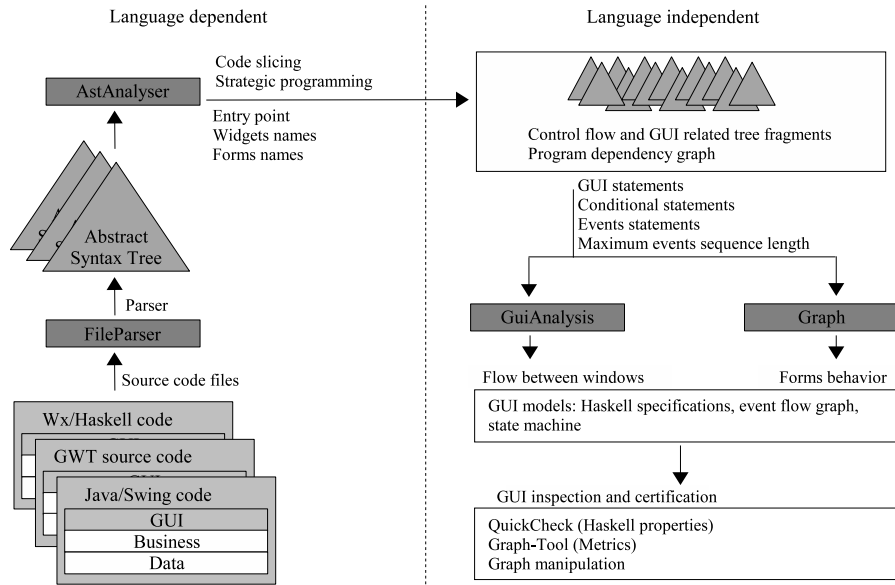


Figure 4.1: GUI SURFER architecture and retargetability

To support these two phases process, the GUI SURFER architecture is composed of four modules:

- *FileParser*, which enables parsing the source code;
- *AstAnalyser*, which performs code slicing;
- *Graph*, which support GUI behavioural modelling;
- *GUIAnalysis*, which also support also GUI behavioural modelling;

The *FileParser* and *AstAnalyser* modules are implementation language dependent. They are the front-end of the system. The *Graph* and *GUIAnalysis* modules are independent of the implementation language.

4.1.1 Source Code Slicing

The first step GUISURFER performs is the parsing of the source code. This is achieved by executing a parser and generating an abstract syntax tree. An AST is a formal representation of the abstract syntactical structure of the source code. Moreover, the AST represents the entire code of the application. However, the tool's objective is to process the GUI layer of interactive systems, not the entire source code. To this end, GUISURFER was built using two generic techniques (see Chapter 3): strategic programming and code slicing. On the one hand, the use of strategic programming enables transversing heterogeneous data structures while aggregating uniform and type specific behaviours. On the other hand, code slicing allows extraction of relevant information from a program source code, based on the program dependency graph and a slicing criteria.

4.1.2 GUI Behavioural Modelling

Once the AST has been created and the GUI layer has been extracted, GUI behavioural modelling can be processed. It consists in generating the user interface behaviour. The relevant abstractions are user inputs, user selections, user actions and output to user. In this phase, behavioural GUI models are created. Therefore, a GUI intermediate representation is created in this phase (see Section 4.3.1).

4.1.3 GUI Reasoning

It is important to perform reasoning over the generated models. For example, GUISURFER models can be tested by using the *Haskell QuickCheck* tool [CH00], a tool that tests *Haskell* programs automatically. Thereby, the programmer defines

certain properties functions, and afterwards tests those properties through the generation of random values.

GUISURFER is also capable of creating event-flow graph models. Models that abstract all the interface widgets and their relationships. Moreover, it also features the automatic generation of finite state machine models of the interface. These models are illustrated through state diagrams in order to make them visually appealing. The different diagrams GUISURFER produces are a form of representation of dialog models.

GUISURFER's graphical models are created through the usage of *GraphViz*, an open source set of tools that allows the visualization and manipulation of abstract graphs [EGK⁺01]. GUI reasoning is also performed through the use of *GraphTool*¹ for the manipulation and statistical analysis of graphs. In this particular case an analogy is considered between state machines and graphs.

4.2 The GUISURFER Implementation

The four architectural modules/components identified in Section 4.1 (see Figure 4.1), give rise to GUISURFER's four software components (cf. filled boxes in Figure 4.1). GUISURFER's executables are *FileParser*, *AstAnalyser*, *GuiAnalysis* and *Graph*.

In this Section, we use the *Login's Agenda* window example (cf. Figure 4.2) to outline some of the more important features of each executable. Appendix B contains the complete script to generate and analyse all available behavioural models for the *Agenda* application. The behavioural model on the right side of

¹see, <http://projects.skewed.de/graph-tool/>, last accessed 27 November, 2010

Figure 4.2 is one result obtained through the execution of GUI SURFER, giving the source code shown in Figure 1.3 as input.

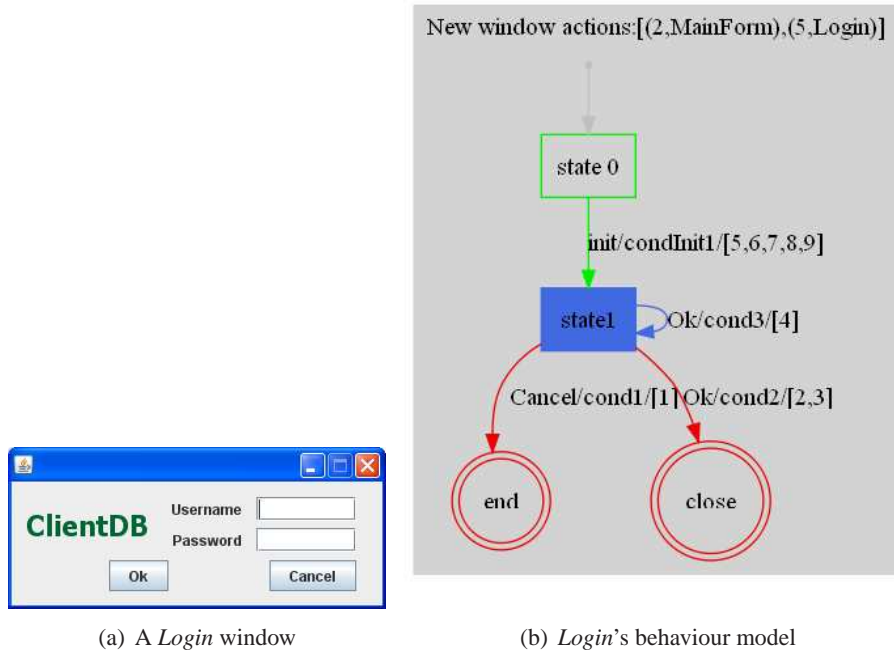


Figure 4.2: GUI SURFER applied to a *Login* window

4.2.1 Parsing the Source Code

The *FileParser* module is used to parse source code. This tool is language dependent. To implement this first tool, a parser for the programming language being considered is used.

GUI SURFER has been used to reverse engineer *Java* and *Haskell* programs written using the (*Java*) *Swing*, *GWT*, and (*Haskell*) *WxHaskell* GUI toolkits. For the *Java/Swing* and *GWT* toolkits, the SGLR parser has been applied whose implementation can be accessible via the Strafunski tool [LV03]. For the *WxHaskell*

toolkit the *Haskell* parser that is included on the *Haskell* standard libraries was used.

4.2.2 Extracting the GUI Layer

The *AstAnalyser* component implements a GUI code slicing process using strategic programming. This module is used to extract the GUI layer from the AST produced by the compiler. The *AstAnalyser* is a language dependent tool used to slice an AST, considering only its graphical user interface layer. Part of this tool is easily retargetable, however most of the tool needs to be rewritten to consider another particular programming language. This happens because programming languages follow different programming paradigms.

The *AstAnalyser* tool is composed of a slicing library, containing a generic set of traversal functions that traverse any AST. This library is composed by the files *SlicingX.hs* and *GuiX.hs*.

- *SlicingX.hs* contains the generic slicing functions (which are language independent). For example, a function to slice an AST to find all elements that match a specific constructor given as a parameter;
- *GuiX.hs* uses the general slicing functions from *SlicingX.hs* and contains more specific language dependent slicing functions.

This tool must be used with three arguments, i.e. the AST, the entry point in the source code (e.g., the main method for *Java* source code), and a list with all widgets to consider during the GUI slicing process.

As an illustrative example, the following function call enables us to extract the

GUI layer from *Login.java*'s AST (*Login.java.ast* file), starting the slice process at the *main* method, extracting *JButton* related data (*Java/Swing* constructor for button creation), and four other *Java/Swing* instructions: *exit*, *showMessageDialog*, *dispose*, and *setEnabled*.

```
>AstAnalyser
  "Login.java.ast"
  "main"
  "JButton,setEnabled,exit,showMessageDialog,dispose"
```

By executing this command, the function generates two files *initState.gui* and *eventsFromInitState.gui* which contain the GUI's initial state and all possible GUI events from the initial state, respectively.

GUI SURFER has been implemented in the *Haskell* programming language. The following *Haskell* prototype function is one of the main functions of the *GUI code slicing* library:

$$\begin{aligned} \text{slice} &:: \text{AST} \rightarrow \text{Const} \rightarrow \text{InitPos} \rightarrow \text{SliceType} \rightarrow \\ &(\text{Const} \times \text{AST} \times \text{InitAST} \times \text{EndAST})^* \end{aligned} \quad (4.1)$$

This function extracts a list of fragments in a particular AST. The *slice* function is a generic function which is configured with the AST and the constructor pattern to be extracted. Basically, the *slice* function receives four parameters:

- *AST*: the abstract syntax tree;
- *Const*: the AST 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 be 1 or 2. The type 1 code slicing extracts the first pattern matching. Type 2 slicing continues within extracted subtrees.

The *slice* function returns all fragments in the AST that match the giving constructor.

As an example, to extract all buttons's definitions from a *Java/Swing* source code's AST, the following instruction could be executed:

```
slice javaAST ``JButton`` 1 1
```

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

```
slice wxHaskellAST ``button`` 1 1
```

where *JButton* and *button* are constructors which create buttons in the *Java* and *Haskell* programming languages, respectively.

At this point, a set of AST fragments can be obtained that just consists of instructions that affect the user interface as seen above. Anchor points for these fragments are detected by syntactic pattern matching.

To explain the developed GUI code slicing module in more detail, let us consider the *Java/Swing Login* class fragment described in Figure 4.3, which defines a new button through the *JButton* class:

Figure 4.4 describes the fragment of the AST obtained after parsing the *Login* source code.

The AST described in Figure 4.4 uses several constructors. For example, the *ClassDecE* constructor is used to declare the *Login* class. The *Statem* constructor is used to define a statement. Methods are specified through the *Dmth* constructor.

```

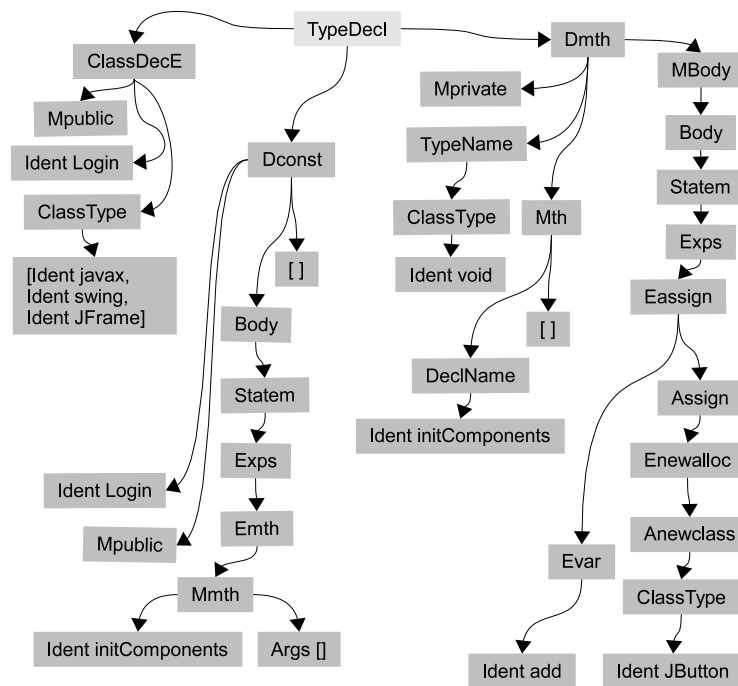
public class Login extends javax.swing.JFrame {

public Login() {
    initComponents();
}

private void initComponents() {
    add = new JButton();
    ...
}
...
}

```

Figure 4.3: Java/Swing Login class

Figure 4.4: Fragment of the *Login*'s AST

In this case, the use of the *Dmth* constructor is used to define the *initComponents* method.

The knowledge of this particular fragment of the *Java* AST, enables us to define a function that, given the complete AST, extracts all *JButton* object assignments:

- First, we need to collect the list of assignments in the source code. This function is defined in *Haskell* making use of the *slice* function (cf. definition 4.1) in order to traverse the AST. Next, the slicing parameter to be used while traversing the AST needs to be defined. 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 in Figure 4.4). Thus, our *slice* function simply returns a singleton list with the left-hand side of the assignment and the respective expression.

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

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

- Having collected the list of assignments the process can now filter that list again in order to produce the list containing all *JButtons* assignments in the *Java/Swing* code. The function to extract *JButton* assignments becomes:

```
statementsButtons subast = slice subast "JButton" 1 1
```

As result, the fragment of the AST, provided in Figure 4.5, is obtained which contain the *JButton* object assignment. This fragment correspond to the following *Java* code `add = newJButton();`

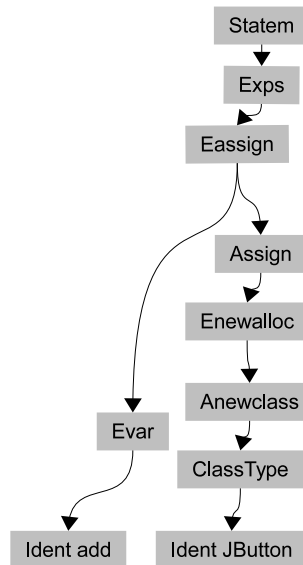


Figure 4.5: *JButton* fragment of the *Login*'s AST

Another important task is the extraction of the list of instructions executed from a particular source code anchor point. In others words, to implement this reverse engineering process from source code the tool must extract the sequence of instructions executed when a particular event occurs. This information is obtained through code slicing within a particular instructions block and considering all external invocations. With this particular *Java* parser, external invocations are identified with the *Emth* constructor. Hence, *Java* external invocations are extracted by pattern matching against the *Emth* constructor. The function implemented to extract all external invocations from a *Java* AST is:

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

The GUISURFER tool uses these fragments to produce GUI models, like be-

havioural user interface descriptions. The fragments relevant to the GUI reverse engineering are widget related instructions, control flow information and methods invocation. Thus, the problem of understanding the interface has been reduced to the problem of understanding the slice done with respect to certain user interface components.

GUI SURFER considers also GUI applications with several windows each containing widgets which, in turn, may invoke one or more windows. To obtain a list of all GUI windows that can be invoked from a GUI window w , the tool queries each of w 's widgets. The GUI SURFER procedure performs a search of the GUI tree rooted at GUI window w , creating a list l of the widgets in w , and searches the list of widgets for those which invoke other GUI windows.

4.2.3 Generating of GUI Behavioural Models

The *Graph* module implements the *GUI abstraction* step described in Figure 4.1. The *Graph* tool is language independent and receives five arguments: the *initState.gui* and *eventsFromInitState.gui* files (both generated by the *AstAnalyser* module), the names of windows in the application, the name of the window to be analysed, and finally the project's name. An example of *Graph* tool invocation can be:

```
Graph eventsFromInitState.gui initState.gui  
      "ContactEditor,Find,Login,MainForm"  
      "Login" "ClientDBjava"
```

This invocation generates behavioural models of the login window. The project (*ClientDBjava*) contains three others windows, namely *ContactEditor*, *Find*, and *MainForm*.

As a result of the invocation, the *Graph tool* generates GUI-related metadata files with events, conditions, actions, and states. Each of these types of data are related to a particular fragment from the AST. Results are stored in files *actions.txt*, *events.txt*, and *conds.txt*.

Another important output generated by the *Graph tool* are the *GuiModel.hs* and *GuiModelFull.hs* files. These are GUI specifications written in the *Haskell* programming language [JHA⁺99]. These specifications define the GUI layer by mapping pairs of event/condition to actions.

The *GuiAnalysis* tool supports the generation of visual models, such as state machines and event flow graphs through the *GraphViz* tool [EGK⁺01].

4.2.4 Evaluating GUI Behavioural Models

Finally, external tools are used to enable reasoning over the graphical user interface models produced in the previous step. To implement this task, *QuickCheck*, *Graph-Tool* and graph theory are used. GUI reasoning is discussed in Chapter 5.

4.3 Models for GUI Reverse Engineering

GUI SURFER extracts different kinds of models. These are described in the following Sections.

4.3.1 GUI Meta-Model

One GUI representation obtained with the reverse engineering process is defined by a meta-model (*guimodel*) which represents the behaviour of the GUI's windows as a mapping of events and related conditions to a list of GUI actions references. The

guimodel meta-data is defined in the *Haskell* programming language as follows²:

```

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

```

The *guimodel* models a window at a particular time t in terms of the widgets within the window, the events enabled by the widgets, and the conditions associated to events. The model describes also the window's initial state (*Pres* type), end application actions references (*End* type), close window actions references (*Close* type), and finally open window actions references (*NewWindow* type). In this model, each data is related to a particular sub tree of the AST. So, it is always possible to access concrete source code fragments.

As an example, and considering the *Agenda*'s login source code (cf. Section 3.1), the respective *guimodel* specification is³:

```

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)) ]

```

²see Appendix A for complete definition

³see Appendix A for the *Agenda*'s windows specifications


```

end :: End
end = [1]

newWindow :: NewWindow
newWindow = fromList
    [(2,"MainForm"),
     (5,"Login")]

close :: Close
close = [3]

```

This specification represents user events with associated conditions and sequences of GUI actions. Therefore, an event is only performed when the related condition is verified. In this case, the associated sequence of actions is then executed.

This model enables analysis of the dialogue supported by each window. Although this is a very simple example, several conclusions can be reached from the analysis of this particular source code abstraction.

For instance, the expression $((\text{"Ok"}, \text{"cond2"}), [2,3])$ means that the *Ok* event has a condition, which GUI SURFER automatically named as *cond2*, and if the condition is verified executes the actions referenced by the number, 2 and 3. The AST slices that corresponds to condition *cond2*, and actions 2 and 3 are available in the *conds.txt* and *actions.txt* files, respectively. In this particular case, the *Java* source code for *cond2* condition is:

```
valid(login.getText(), password.getText())
```

The *Java* source code related to action reference 2 is:

```
newMainForm().setVisible(true);
```

The *Java* source code related to action reference 3 is:

```
this.dispose();
```

Therefore, all the visual information available in right side of Figure 4.2 originates from the textual meta-model representation defined in this *GuiModel.hs* file.

This model gives also access to data related with the initial state of the *Login* window:

```
pres :: Pres
pres = fromList
      [(8, ("Cancel", True)),
       (9, ("Ok", True))]
```

Boolean values indicate whether events are initially enabled or not. In this case both *Cancel* and *Ok* events are initially enabled. Identifiers are assigned to each event. In this case *Cancel* and *Ok* events are identified through reference 8 and 9, respectively.

Another important aspect regards actions references which end the application or close the window. These actions are specified through *end* and *close* functions:

```
end :: End
end = [1]

close :: Close
close = [3]
```

In this case numerical values refer to GUI actions in the source code (fragment of the AST) which enable ending the application (i.e. identifier 1 in *end* function) and closing the *Login* window (i.e. identifier 3 in *close* function). These actions are executed by *(("Cancel", "cond1"), [1])* and *(("Ok", "cond2"), [2,3])* events respectively. So the first event (*Cancel* event verifying *cond1* condition) terminates the

Agenda application, and the last one (*Ok* event verifying *cond2* condition) closes the *Login* window.

Finally, the *newWindow* function, specifies all actions references which enable opening new windows:

```
newWindow :: NewWindow
newWindow = fromList
            [(2,"MainForm"),
             (5,"Login")]
```

This function maps actions references to the appropriate window identifiers. These are action references 2 and 5, enabling the opening of the *MainForm* and *Login* windows respectively.

As can be seen in the source code in Section 3.1, the *Login* window enables the opening of the *MainForm* window and the closing of itself. This knowledge is captured in the *(("Ok","cond2"),[2,3]) guimodel* expression. In this case, if the user presses the *Ok* button, and the related *cond2* condition is true then action reference 2 opens the *MainForm* window, and action reference 3 closes the *Login* window.

Let us consider another example and present also the *guimodel* extracted from the *MainForm Agenda's* window:

```
guimodel :: GuiModel
guimodel = fromList
            [((("Exit","cond1"),[1]),
             (("Edit","cond2"),[2]),
             (("Edit","cond3"),[3]),
             (("Find","cond4"),[4]),
             (("Find","cond5"),[5]),
             (("init","condInit1"),[6,7,8,9,10,11,12,13,14,15]))]

pres :: Pres
pres = fromList
```

```

        [(10, ("Exit", True)),
         (11, ("Edit", True)),
         (12, ("Find", True))]

end :: End
end = [1]

newWindow :: NewWindow
newWindow = fromList
            [(2, "ContactEditor"),
             (4, "Find"),
             (6, "MainForm")]

close :: Close
close = []

```

In this model, the window behaviour includes three events (*Exit*, *Edit* and *Find*). All of them are enabled at the initial state window (cf. true values assigned to each event in the *pres* function).

The *newWindow* function defines three action references which open different windows. These are action references 2, 4 and 6 which open windows *ContactEditor*, *Find* and *MainForm*, respectively. Hence, from the *MainForm* window, it is possible to open the *ContactEditor* and *Find* windows (action references 2 and 3 respectively) through the code associated with events *Edit* and *Find* (cf. $((\text{"Edit"}, \text{"cond2"}), [2])$ and $((\text{"Find"}, \text{"cond4"}), [4])$ expressions in the *guimodel* function), which execute new window actions 2 and 4 (cf. function *newWindow*). However to access these windows *cond2* and *cond4* must be valid, respectively. Finally to close the application the source code related to action reference 1 must be executed (cf. *end* function), which is triggered within the $((\text{"Exit"}, \text{"cond1"}), [1])$ event.

Besides the *guiModel Haskell* specification, other meta-models have been defined to express other kinds of knowledge (see Appendix D and E). All of them are also automatically generated. The following model relates pairs of states with all possible event/condition pairs representing transitions between them, in the *Agenda Login* window:

```
statesLogin :: (Map (StateRef,EventRef,CondRef,[ExpRef])
StateRef, Map StateRef State)
statesLogin = (fromList [
(("state0","init","condInit1",[5,6,7,8,9]),"state1"),
(("state1","Cancel","cond1",[1]),"state0"),
(("state1","Ok","cond2",[2,3]),"state0"),
(("state1","Ok","cond3",[4]),"state1")],
fromList [("state0",fromList []),
("state1",fromList [("Cancel",True),("Ok",True)])])
```

The next model, specifies all possible sequences of events for the *Login* window from its initialization. As an example, if we consider an event sequence length lower than 3, then 7 different sequences of events are obtained:

```
waysLogin :: [M]
waysLogin = [N(["condInit1"],["init"]),
N(["condInit1","cond1"],["init","Cancel"]),
N(["condInit1","cond2"],["init","Ok"]),
N(["condInit1","cond3"],["init","Ok"]),
N(["condInit1","cond3","cond1"],["init","Ok","Cancel"]),
N(["condInit1","cond3","cond2"],["init","Ok","Ok"]),
N(["condInit1","cond3","cond3"],["init","Ok","Ok"])]
```

It can be seen that *["init","Ok","Cancel"]* is one possible sequence of events. To execute this sequence, the conditions *condInit1*, *cond3*, and *cond1* must be verified. Each reference corresponds to a particular event condition within the sequence of events.

4.3.2 MAL Interactors

Another notation for describing interactive systems is the language of MAL interactors [CH01]. It is a domain specific language and includes the notion of interactors as a mechanism for structuring the use of standard specification techniques in the context of interactive systems specification [DH93]. Modal Action Logic [RFM91] is used to specify the behavioural parts of the models.

The definition of a MAL interactor contains a state, actions, axioms and presentation information. This language allows 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.

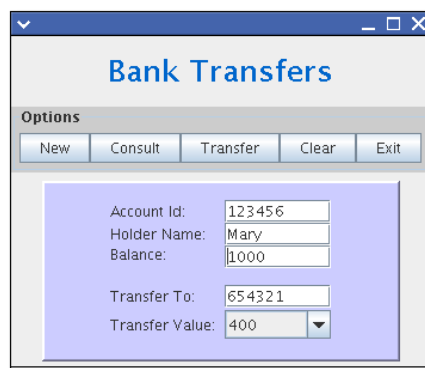


Figure 4.6: *JBank* system

Now to describe an example of MAL interactors, let us consider another interactive application: the *JBank* transfers system. Basically, the *JBank* system is a simple *Java/Swing* example allowing for account transfers (see Figure 4.6).

Through a single window composed of several widgets, users have access to five buttons, allowing the creation of new accounts, consulting the data for each account, execution of transfers from one account to another one, and finally to clear all input widget's values.

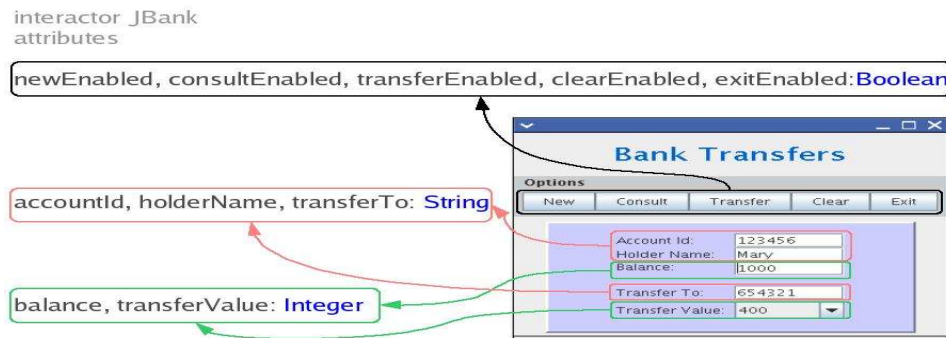


Figure 4.7: Interactor's attributes abstraction

Applied to the code of the *JBank* application, GUI SURFER automatically generates an interactor specification including the initial application state and dynamic actions. This interactor contains a set of attributes (cf. Figure 4.7) - one for each information input widget, and one for each button's enabled status.

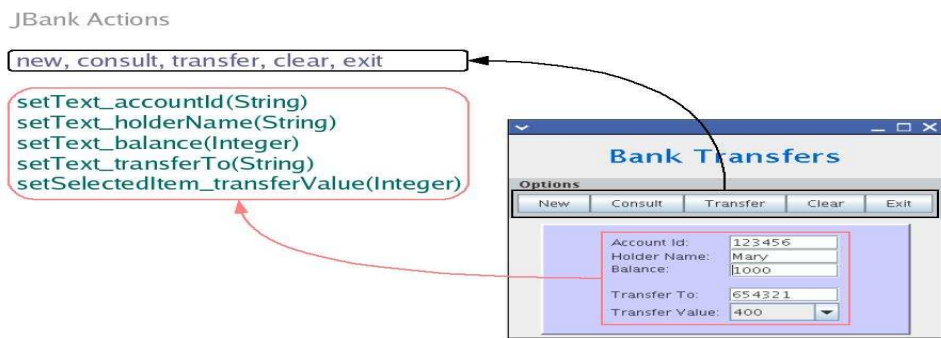


Figure 4.8: Interactor's actions abstraction

The interactor also contains a set of actions (cf. Figure 4.8) - one for each button, and one for each input widget (representing user input). Finally, it also contains a set of MAL axioms like the presented below which define the effect of the *new* button in the interface. In this particular case, the effect of the *new* button is to enable the *consult* button. All others widgets status remains unchanged.

```
[New]
newEnabled'=newEnabled & consultEnabled'=true &
transferEnabled'=transferEnabled &
clearEnabled'=clearEnabled & exitEnabled'=exitEnabled &
accountId'=accountId & holderName'=holderName &
transferTo'=transferTo & balance'=balance &
transferValue'=transferValue
```

Similar axioms are generated for all other actions.

4.3.3 Event Flow Graphs

At all times during interaction with a GUI, the user interacts through events. In the literature a GUI component's flow of events may be represented as an event flow graph [MBN03].

An event flow graph defines all possible interactions by specifying the events in a GUI system. For every event e , an event flow graph identify all the events that can follow e .

Formally, an event-flow graph is a 4-tuple (V, E, B, I) where:

1. V is a set of vertices representing all the events in the component;
2. $E = V \times V$ is a set of directed edges between vertices. Event e_j follows e_i if e_j may be performed immediately after e_i . An edge $(vx, vy) \in E$ if the event represented by vy follows the event represented by vx ;

3. $B \subseteq V$ is a set of vertices representing those events of C that are available to the user when the component is first invoked;
4. $I \subseteq V$ is the set of events that explicitly terminates the application.

This Section shows the extraction of an event flow graph from another small example: the *JClass* application. Basically, the *JClass* system is a *Java/Swing* example allowing for student marks management (see Figure 4.9). Users can add student's data (*Add* button), i.e. student's number, name and two marks. The system enables also finding (*Consult* button) and removing (*Remove* button) students. The *Clear* button empties all system's widgets.

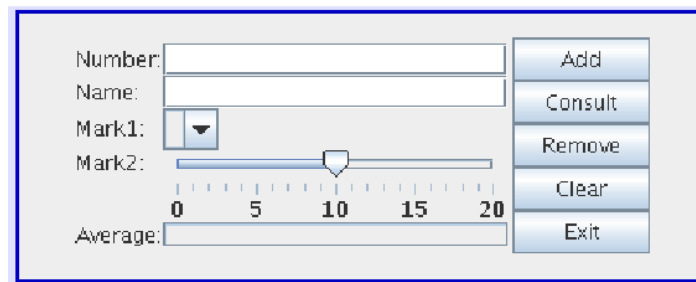
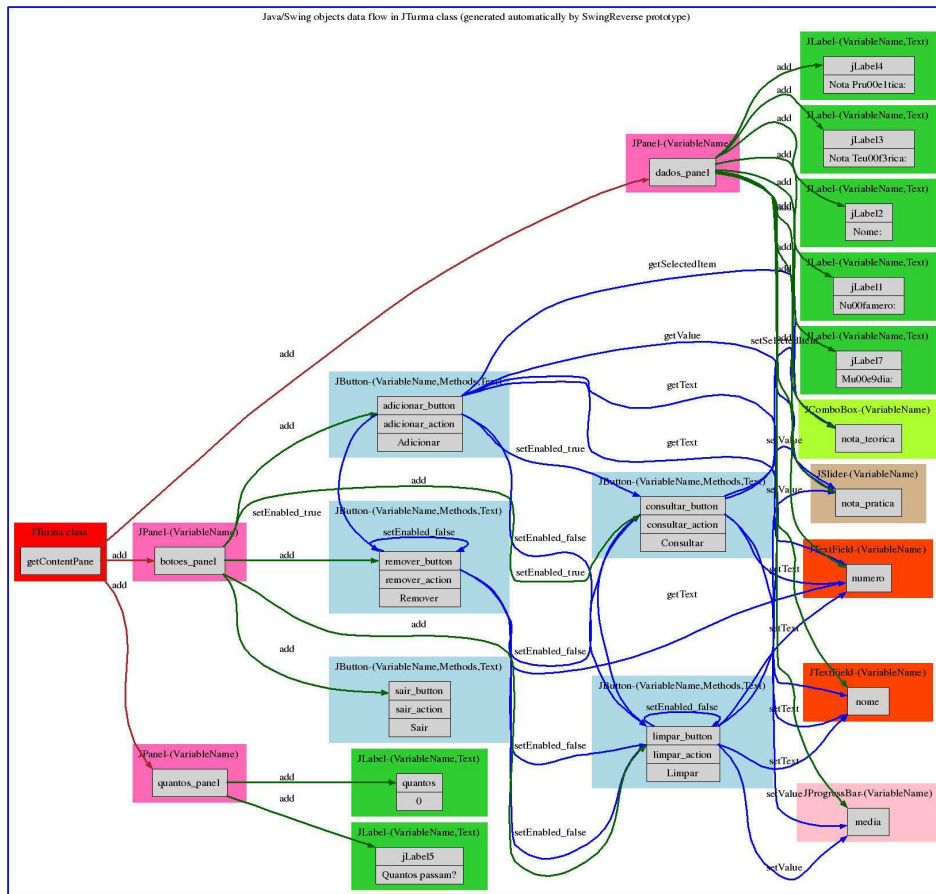


Figure 4.9: *JClass* system

In this particular case, GUI SURFER enables the extraction of an event-flow graph which allows the analysis of the code's quality from a software engineering perspective. Figure 4.10 provides the obtained event-flow graph. As with other models, the graph is generated by the *Graph* module. All widgets and their relationship are abstracted to this graph and arrows specify methods calls from one widget to another. As an example, nodes with three internal fields specify *JButtons* abstractions.

Figure 4.10: *JClass* system's partial GUI event-flow graph

4.3.4 Finite State Machines

As has been discussed in Section 2.5.2, finite state machine may be used to model GUI behaviour. This type of model has been chosen in order to support reasoning about and testing the dialogue supported by a given GUI implementation.

An interactive system can be represented as a FSM considering that user events are mapped into arcs, and GUI states are mapped into vertices. When the user

performs an event, the current state A is changed to the next state B where there is an arc from A to B labeled with that event.

GUISURFER automatically generates a finite state machine model of the interface. Next, an example of FSM generation will be presented, considering the *Agenda* interactive system (cf. Figure 3.1).

The state machine in Figure 4.11 is obtained from the *Agenda*'s source code. Arrows specify a user interface event changing from one state to another. In this particular case each arrow abstracts a particular button press action (*event*). For each action there is a associated condition (*cond*) which must be validated to move from one state to the next. Conditions are extracted directly from conditional instructions in the source code. For example an *if condition then action1 else action2* conditional instruction is abstracted to two different transitions. A first one, with *actions1*'s graphical user interface instructions, that occurs when the condition is true, and a second one with *action2*'s graphical user interface instruction.

This model is composed by four finite state machine, one for each *Agenda*'s window. As can be see, some events allow opening a new window, so these are related to other finite state machine.

4.3.5 Others Models

In this Section, a further model is presented. The model is useful to visualize an application's behaviour in terms of active windows (see. Figure 4.12). Each state of this model contains all the open windows in a particular period of time. Transitions between states correspond to events that open or close windows. Each transition refers the source window's name, it state status, the event and respec-

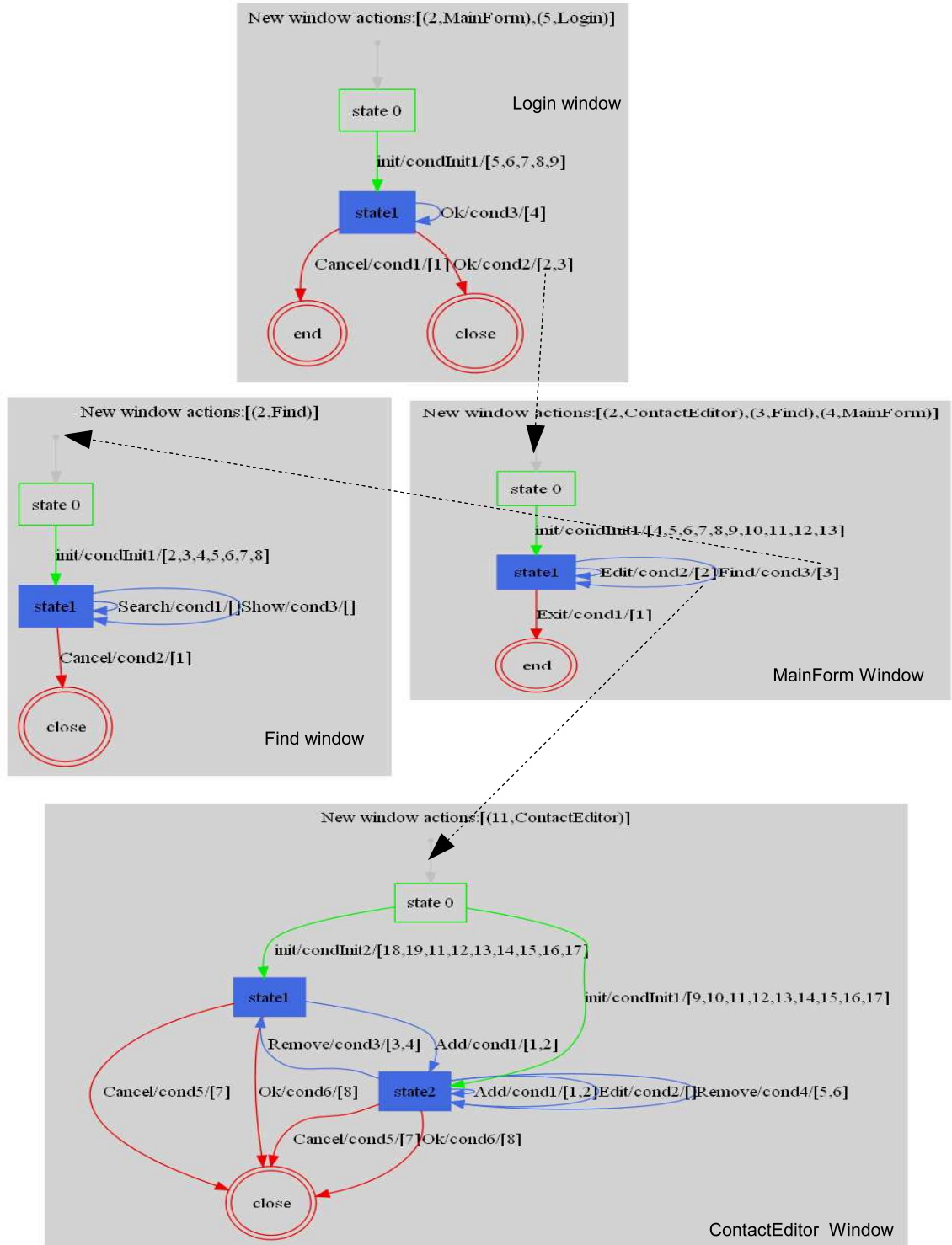


Figure 4.11: Agenda's finite state machine

tive condition. In this case, one can reason about which windows can be opened along a session, which are the related events and conditions. At the top left corner, this model specifies the *Login* window as an entry point for the application. Then, from the *Login* window there is one transition to *MainForm* window. This transition happens from *state1* in reaction of the *Ok* event if condition *cond2* is verified. From *MainForm*'s *state1*, it is possible to open the *ContactEditor* window through the *Edit* event if condition *cond2* holds. The referred transition is: *MainForm state1 Edit cond2*

The model in Figure 4.12 was generated from the *Agenda*'s source code considering non-modal *MainForm*, *ContactEditor* and *Find* windows. Thus, an *Agenda* session may be composed by several instances of the same windows as represented in this model.

4.4 A Language Independent Tool

A particular emphasis has been placed on developing tools that are, as much as possible, language independent. Although *Java/Swing* was used as the target language during initial development, through the use of generic programming techniques, the developed tool aims at being retargetable to different user interface toolkits, and different programming languages. Indeed, the *GUISURFER* tool has already been extended to enable *GWT* and *WxHaskell* based applications analysis.

Google Web Toolkit (*GWT*) is a Google technology [HT07]. *GWT* provides a *Java*-based environment which allows for the development of *JavaScript* applications using the *Java* programming language. *GWT* enables the user to create rich Internet applications. The fact that applications are developed in the *Java* language

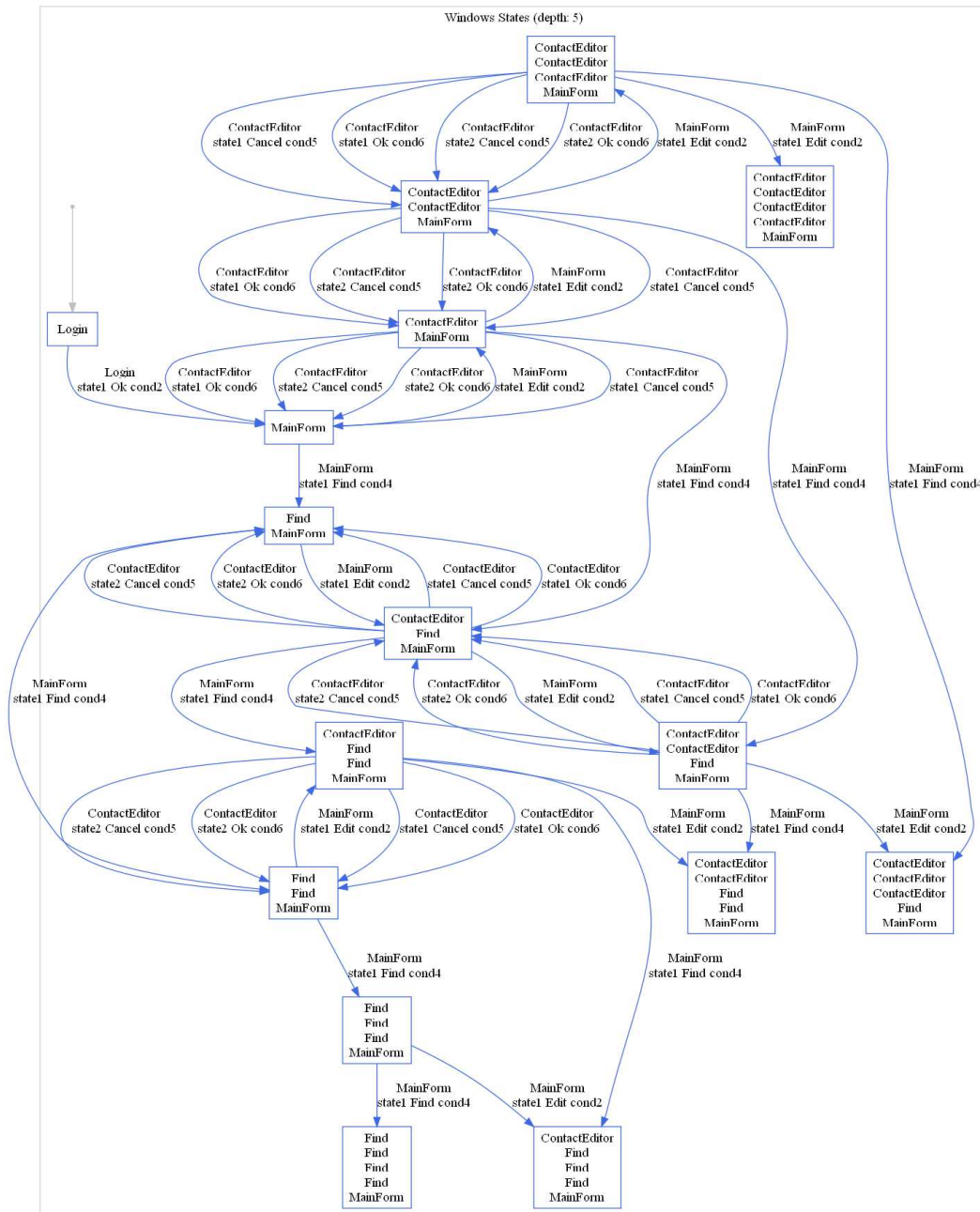


Figure 4.12: Agenda application's windows states

allows *GWT* to bring all of *Java*'s benefits to web applications development. *GWT* provides a set of user interface widgets that can be used to create new applications. Since *GWT* produced a *JavaScript* application, it does not require browser plug-ins additions.

WxHaskell is a portable and native GUI library for *Haskell*. The library is used to develop a GUI application in a functional programming setting.

4.4.1 Retargetability

In this Section the applicability of *GUI SURFER* to *GWT* and *WxHaskell* code is discussed. Our retargets to *GWT* and *WxHaskell* highlight successes and problems with the initial approach. The size of the adaptations and the time it took to code them are distinct. They were also carried out under different conditions. The latter was performed by the author, the former was performed by another researcher under the author's guidance [Sil10].

The adaptation to *GWT* was easier because it reuses the same *Java* parser. The adaptation to *WxHaskell* was more complex as the programming paradigm is different, i.e. *Haskell* is a lazy functional programming language.

The application of *GUI SURFER* to *WxHaskell* has been implemented by defining the slicing step for functional programming, more specifically for the *WxHaskell* syntax. This task was made complex by the fact that the *WxHaskell* toolkit has a different structure to define GUI components like windows, event actions, etc. The first *GUI SURFER*'s phase, *FileParser*, needed to be defined to consider another programming language. Although a parser for the *Haskell* programming language has been reused to generate abstract syntax trees, we needed to imple-

ment the program dependency graph. Finally the *AstAnalyser* import file named *GuiX* contains several changes, making use of the *WxHaskell* specific constructors.

Considering the applicability of GUI SURFER to *GWT*, as *GWT* is written in the *Java* programming language, the first GUI SURFER's phase, *FileParser*, remained unchanged. The slicing was the same but applied to the set of GUI components from *GWT*, which are different from those of *Swing*. During the adaptation to *GWT* some aspects of the slicing function that were relying on *Swing* specificities were identified and generalised. For example, *Swing*'s *addActionListener* method was being used to identify actions. In *GWT* the corresponding method is the *addClickListener* method.

Table 4.4.1 (from [Sil10]) shows the total number of language dependent lines of code for the five GUI SURFER modules. The column *Java/Swing* contains the number of lines of *Haskell* code each *Java/Swing* module has. As *Java/Swing* was the first approach made with GUI SURFER, the other languages were made by re-targeting this initial approach. Hence, other columns, specifically, *WxHaskell* and *GWT*, besides containing the number of lines each GUI SURFER module has, also indicate (between parentheses) information regarding how many lines were changed⁴.

As an example, *FileParser* made for *Java/Swing* has 54 source lines of code. *FileParser* for *WxHaskell* has 26 lines, and 41 lines from the *Java/Swing* source code were changed. *GWT*'s *FileParser* has 54 lines where no line was changed, thus it is identical to the *Java/Swing* file.

⁴Lines changed include code lines erased and source code lines changed. The visual file comparison tool named ExamDiff was used to compare files.

GUI SURFER as a retargetable tool incurred in few source code updates for the new programming languages. Focusing on the approach to *GWT*, just a few line codes were changed. Moreover, changes performed to extend GUI SURFER to a new programming language, specifically *GWT* or *WxHaskell*, did not reflect on architectural changes. Hence, GUI SURFER's objective of being a retargetable tool was accomplished. It should be stressed that changes made to the *GuiX.hs*, *SlicingX.hs*, and *Graph.hs* files represent improvements to for the modules which are backwards compatibles.

Module/Toolkit	<i>Java/Swing</i>	<i>WxHaskell</i>	<i>GWT</i>
FileParser.hs	54	26(41)	54(0)
AstAnalyser.hs	88	85(5)	87(9)
GuiX.hs	218	140(179)	219(5)
SlicingX.hs	135	135(0)	135(0)
Graph.hs	665	467(245)	669(9)

Table 4.1: Total language dependent lines of code [Sil10]

4.4.2 *WxHaskell* example

Throughout this Section another *Agenda* interactive application implementation is used to illustrate the applicability of the tool to different languages (cf. Figure 4.13).

The new *Agenda* was implemented in *WxHaskell* with the same functionalities of the *Java/Swing* implementation. Users can perform the usual actions of adding, removing, finding and editing contacts.

Figure 4.14 provides the *WxHaskell* source code of the *Login* implementation (window at the top left corner of the bottom half of Figure 4.13). As can be seen,

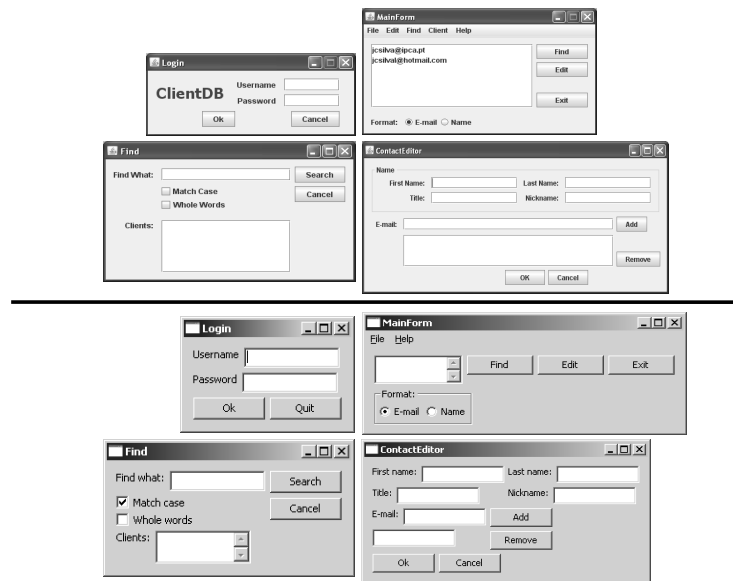


Figure 4.13: Two *Agenda* applications - *Java/Swing* (top) and *WxHaskell* (bottom)

widget constructors are specific to the *WxHaskell* language. These are *staticText*, *textEntry* and *button* to define labels, text boxes and buttons, respectively. The constructor *on command* is used to define the behaviour when a button is clicked.

For example the instruction:

```
quit = button pn [text := "Quit", on command := close fr]
```

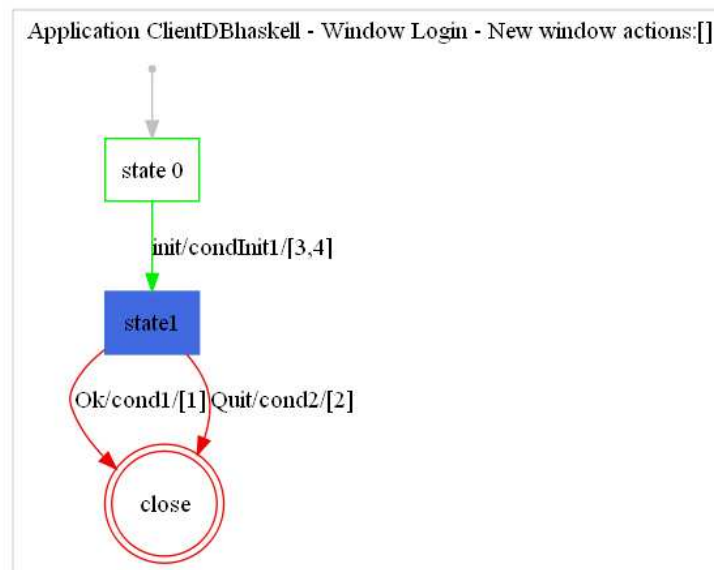
creates the *Quit* button which closes the login window when clicked (*on command* := *close fr*).

From the source code of this new implementation, the *GUIsurfer* tool has generated the same kind of behavioural models as those obtained from the *Java/Swing* implementation. Figure 4.15 provides the behavioural state machine for the *Login WxHaskell* implementation. As can be seen, the extracted model reflects the behaviour of the implemented source code.

```

login :: IO ()
login =
  fr = frame [text := "Login"]
  pn = panel fr []
  lb1 = staticText pn [text := "Username"]
  tb1 = textEntry pn [enabled := True,
                    wrap := WrapNone]
  lb2 = staticText pn [text := "Password"]
  tb2 = textEntry pn [enabled := True,
                    wrap := WrapNone]
  ok = button pn [text := "Ok",
                on command := {close fr;
                              start mainForm}]
  quit = button pn [text := "Quit",
                  on command := close fr]

```

Figure 4.14: *Login WxHaskell* partial source code implementationFigure 4.15: *Login* behavioural state machine (*WxHaskell* implementation)

4.4.3 GWT example

FlexTable is an example of a *GWT* application. The application⁵ simulates a dynamic table as presented in Figure 4.16. It starts, as shown on the left side of the image with an empty table, and a single button visible, the button *New Row*. After the button *New Row* is pressed, the table has a new row, and the application enables two others buttons: *New Cell* and *Clear*. The button *New Cell* adds a new cell in the last created row. Moreover each cells has the values of its respective coordinate on the table. Finally, the button *Clear*, clears the table, removing all created cells.

Applying GUI SURFER into the source code of the application, it produced the finite state machine depicted on Figure 4.17.

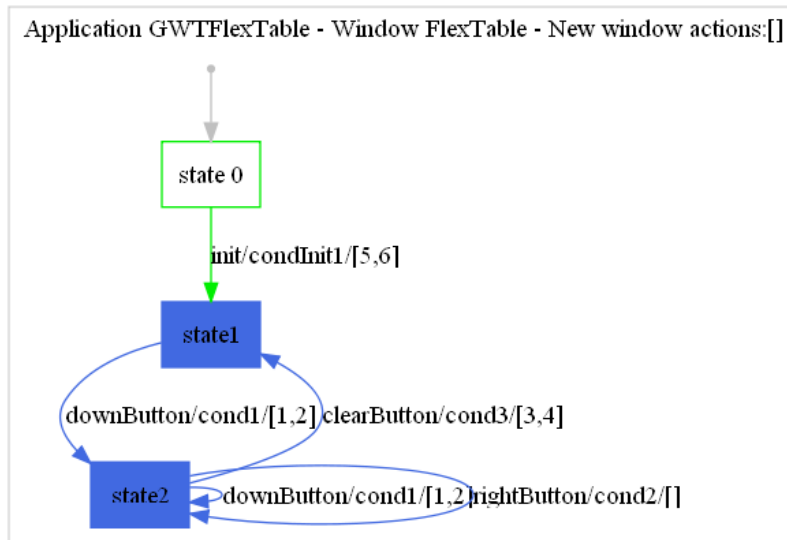


Figure 4.16: GWT FlexTable application

The FSM from Figure 4.17 is composed of three states, the initial state (*state0*), the state with a single button, namely the button *New Row* (*state1*) and the state with the three buttons visible (*state2*).

Because the application always has either one or three buttons, GUI SURFER has correctly identified two states. Moreover, GUI SURFER properly defined two different events after pressing the button *New Row* (event *downButton* in the state machine model), if the table is empty, it will perform a transition to the following

⁵cf. <http://examples.roughian.com/index.htm#WidgetsFlexTable> for available source code, last accessed November 22, 2010

Figure 4.17: *GWT FlexTable's* FSM behavioural model

state, `state2`. Otherwise it will remain in the same state. The example shows that GUI SURFER was able to correctly deduce the behaviour of the *New Row* button. Creating two transitions to model it. This is relevant because the handler associated to the event has no conditional guards. The decision to create the two transitions was not based on the code alone, but on an inference about the state of the interface, and the fact that the button will always make the *Clear* and *New Cell* buttons visible. Figure 4.17 also shows that there is no final states, that is, there are no *close* or *cancel* states. This is because the FlexTable application does not end (WEB applications can always end by navigating to a different web page) and there is just a single window in that application.

4.5 Conclusions

In this Chapter a reverse engineering tool was described. The GUI SURFER tool enables extraction of different behavioural models from application's source code. The tool is flexible, indeed the same techniques has already been applied to extract similar models from different programming paradigm.

The GUI SURFER architecture was presented and important parameters for each GUI SURFER's executable file were outlined. A particular emphasis was placed on developing a tool that is, as much as possible, language independent.

A technique was presented to help identifying a graphical user interface abstraction from legacy code. Then different kinds of GUI models extracted by the tool were described. These are *Haskell* GUI behavioural specification, MAL interactors, event flow graphs, finite state machines, GUI internal states and GUI's windows states. For each model, a particular example was presented.

This work will not only be useful to enable the analysis of existing interactive applications, but can also be helpful in a re-engineering process when an existing application must be ported or simply updated [Mel96]. 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.

The contributions related with this Chapter were described in the following papers presented at international conferences:

- *A Generic Library for GUI Reasoning and Testing*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the 24th Annual ACM Symposium on Applied Computing (SAC 2009), USA, 2009;

- *The GUISURFER tool: towards a language independent approach to reverse engineering GUI code*, J.C.Silva, C. Silva, J.C. Campos, J. Saraiva, in proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems (EICS 2010), pages 181-186. ACM, Berlin, Germany, 2010.

Chapter 5

GUI Reasoning from Reverse Engineering

The term GUI reasoning refers to the process of validating and verifying if interactive applications behave as expected [Ber01, dSGD98, Cam99]. Verification is the process of checking whether an application is correct, i.e. if it meets its specification. Validation is the process of checking if an application meets the requirements of its users [BA95]. Hence, a verification and validation process is used to evaluate the quality of an application. For example, to check if a given requirement is implemented (Validation), or to detect the presence of bugs (Verification) [Bel01].

GUI quality is a multifaceted problem. Two main aspects can be identified. For the Human-Computer Interaction (HCI) practitioner the focus of analysis is on Usability, how the system supports users in achieving their goals. For the Software Engineer, the focus of analysis is on the quality of the implementation. Clearly, there is an interplay between these two dimensions. Usability will be a (non-functional) requirement to take into consideration during development, and problems with the implementation will create problems to the user.

In a survey of usability evaluation methods, Ivory and Hearst [IH01] identified 132 methods for usability evaluation, classifying them into five different classes: (User) Testing; Inspection; Inquiry; Analytical Modelling; and Simulation. They concluded that automation of the evaluation process is greatly unexplored. Automating evaluation is a relevant issue since it will help reduce analysis costs by enabling a more systematic approach.

Another possible division of evaluation methods is between those that require users to use the system, and those that rely on models or simulations of the system for the analysis. In the first case, the costs remain high due to the need for testing sessions with real users of the system to be carried out. Moreover, and given the high costs of user testing, the analysis will not be exhaustive in terms of all the possible interactions between the users and the system. This means that problems with the implementation might remain unnoticed during the analysis. In the second case, an assumption is being made that the implementation will be faithful to the model. This begs the question of how to evaluate the implementation (ideally, without resorting to human users).

The reverse engineering approach described in this thesis allows for the extraction of GUI behavioural models from source code. This Chapter describes an approach to GUI reasoning from these models. To this end, the *QuickCheck Haskell* library [CH00], graph theory, and the *Graph-Tool*¹ are used.

A description about the techniques implemented for GUI reasoning is provided in the following Sections. Section 5.1 illustrates an approach for GUI reasoning making use of the *QuickCheck* tool. Section 5.2 describes others approaches mak-

¹see, <http://projects.skewed.de/graph-tool/>, last accessed 27 November, 2010.

ing use of graph theory and the *Graph-Tool*. Finally, Section 5.3 presents conclusions.

5.1 Testing with the *QuickCheck* Tool

This Section illustrates an approach, based on the *QuickCheck* tool, that enables the automatic generation and validation of test cases. This approach uses the behavioural model of the interactive application under test.

QuickCheck is a tool for testing *Haskell* programs automatically. The programmer provides a specification of the program, in the form of properties that functions in the program should satisfy, and *QuickCheck* then tests that the properties hold in a large number of randomly generated test 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.

Using *QuickCheck*, test cases may be generated automatically from a model of the GUI. This Section present methodologies to perform model-based GUI reasoning by generating, executing and verifying GUI test cases [Bel01, Bum96, Pat95]. The *guimodel* specification described in Section 4.3.1 is used as the basis for the approach.

To illustrate the approach, the *Agenda* application's *guimodel* (cf. Figure 3.1) will be considered. The goal will be to test if the application satisfies the following rule: users can only access the following windows *Login*, *MainForm*, *Find*, and *ContactEditor*. The objective is to test whether there is any hidden code in the application, providing access to extraneous windows. This would be useful, for

example, in the maintenance or migration of large and complex interactive systems.

The rule is specified in the *Haskell* language on top of the *guimodel* test cases. From the *guimodel* specification, *QuickCheck* automatically generates random test cases. Then the rule can be tested on them. Each randomly generated test case is a sequence of valid events associated with their conditions (which will be true, since the event is valid), system actions and target windows. The following *Haskell* code express a particular test case for the *Agenda guimodel*.

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

The above expression contains a sequence of events that can be executed within the *Agenda* application. This is a list of tuples (w, e, c, la) where w is the window name, e express the event executed, c is the related condition and finally la contains a list of GUI actions references.

In this particular sentence, the user starts by pressing the *Ok* button in the *Login* window with a valid username/password (*"Login", "Ok", "cond2", [2,3]*). Condition reference *cond2* represent the invocation of the *isValid* boolean function, cf. Section 1.1. Associated actions references 2 and 3 represents respectively *Login* window closing and *MainForm* window opening. Then, from the *MainForm* window, the user presses the *Find* button, (*"MainForm", "Find", "cond3", [3]*). Condition reference *cond3* test if there are contacts in the agenda. In the *MainForm* window action reference 3 correspond to the *Find* window opening action. Next, from the *Find* window, the user makes a search (*"Find", "Search", "cond1", []*), and then cancels the window, (*"Find", "Cancel", "cond2", [1]*). Finally with the last action,

the user exits the application pressing the *Exit* button from the *MainForm* window (*"MainForm"*, *"Exit"*, *"cond1"*, *[1]*).

Now, considering *vtc* a valid test case, the rule can be specified in *Haskell* as:

```
rule :: [(Window, Event, Condition, Actions)] → Bool
rule vtc =
  all ('elem' ["Login", "MainForm", "Find", "ContactEditor"]) w1
      w1 = map (λ(w,_,_,_) → w) vtc
```

where

- *'elem'* returns true if a list contains a particular item;
- *all* returns True if all items in a list fulfill a condition.

This rule receives a test case as parameter (list of tuples) and verifies (through the *all* function) if each of the events gives access to one, and only one, of the following windows: *Login*, *MainForm*, *Find* or *ContactEditor*.

Testing the application's *guimodel* with this rule through *QuickCheck*, and making use of 10000 randomly generated test cases, the following results are obtained:

```
*GuiModelAnalysis> deepCheck rule1
OK, passed 10000 tests.
87% events sequence length: 5.
10% events sequence length: 4.
1% events sequence length: 3.
0% events sequence length: 2.
0% events sequence length: 1.
```

The rule hold in 10000 randomly generated test cases with up to 5 events length. All of test cases satisfies the rule. This approach is non-exhaustive but could be applied to test a GUI model with a wider range of test cases.

5.2 GUI Inspection Through Graph Theory

This Section describes additional behavioural analysis performed through graph theory. To illustrate the approach, the *Agenda* application's behavioural model will be considered (cf. Figure 5.1). *Graph-Tool* will be used to manipulate and analyse the graph.

5.2.1 *Agenda's* Behavioural Graph

Until now, several behavioural models of the interactive application have been described. Such models unambiguously and rigorously define the behaviour of an application. Moreover, by using models to define the interactive application, several techniques can be used to implement GUI reasoning.

One particular modeling approach that has been considered to enable GUI reasoning from GUI models is the use of finite state machine model. In this Section, graph theory is used to explore these models.

Within this approach, an analogy will be considered between state machines and graphs. State machines states will be represented as vertices, and transitions as edges. For example, Figure 5.1 shows a graphs expressing the behaviour of the *Agenda* application. Now GUI reasoning will be executed through this representation.

5.2.2 Graph Events Count

A first useful model that can be automatically extracted from Figure 5.1 is related to GUI events counts between states (i.e. edges count between vertices). This model helps to visualize the behavioural complexity of a particular window. The

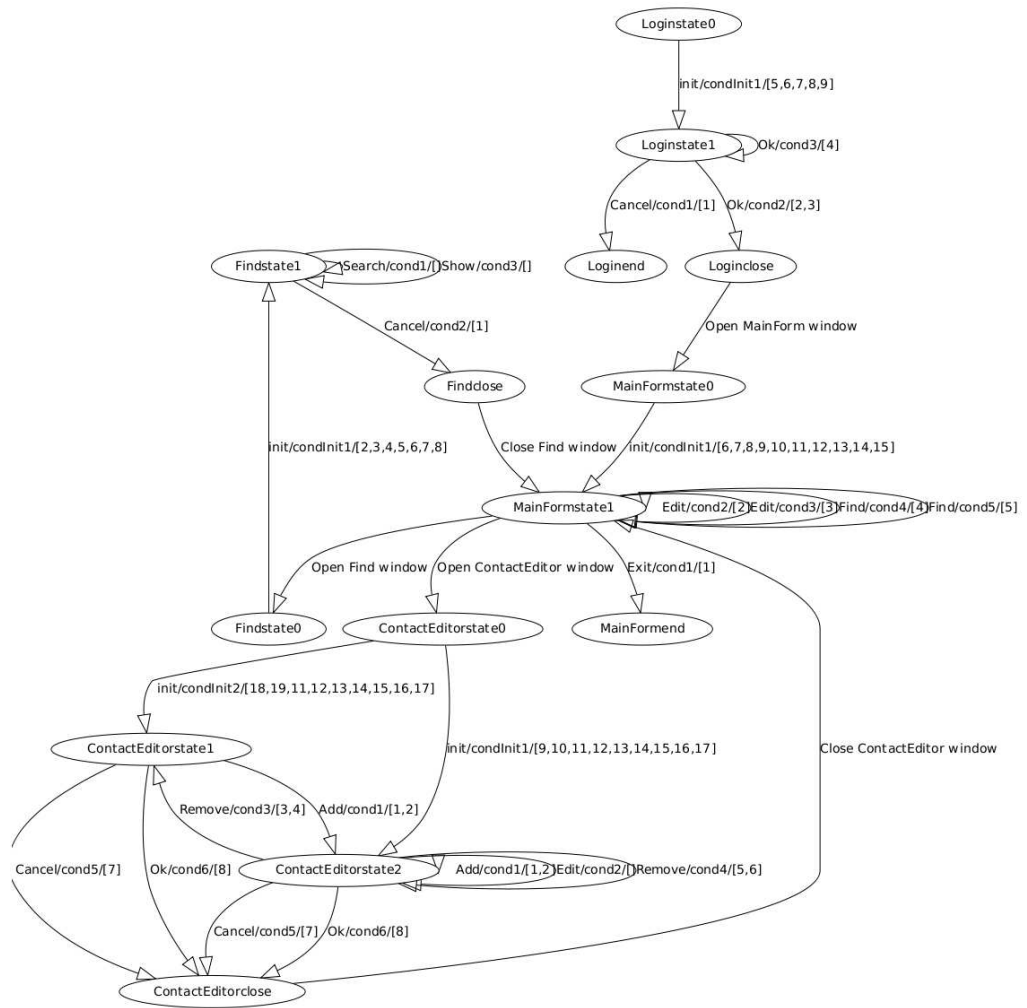


Figure 5.1: Agenda's behaviour graph

total number of events from each state to another one can be visualized (cf. Figure 5.2). This model can be used to measure aspects related to the distribution of events considering the overall application behaviour.

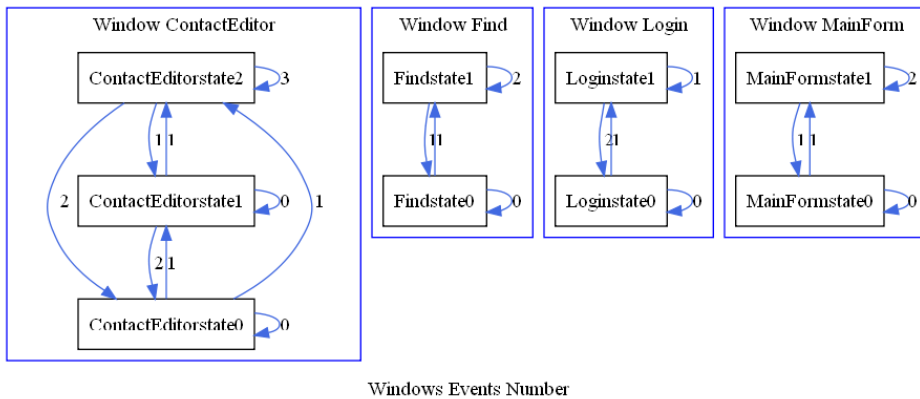


Figure 5.2: Agenda application's events Count

5.2.3 Operations on Graphs

Graphs may be easily manipulated through intersection, union, and difference operators. These operations allow us to compare, for example, the GUI model of two versions of an interactive application. This is particularly relevant in the presence of evolving applications.

An example of such a result is shown in Figure 5.3. The figure corresponds to the union of two graphs, generated by GUI SURFER, using two distinct versions of the Agenda application (one version makes use of a Find window to search contacts while the other does not).

As indicated in the legend, bold edges correspond to events from the first version of the application only, filled edges correspond to events from the second

version, and dotted edges are events that can be executed on both versions. As it can be seen, the first version enables access to the *ContactEditor* window, while the second edition does not. In fact, as demonstrated by the absence of filled edges, the second version implements a subset of the first version's behaviour only.

This approach is helpful to reason about different versions of an application. These can be versions implemented in the same programming language or not. GUI SURFER functionalities also include graph intersection and difference.

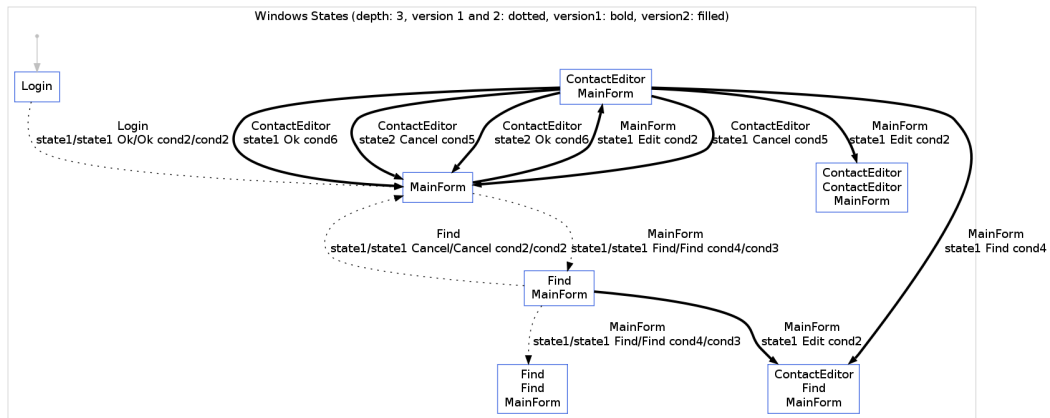


Figure 5.3: Comparing *Agenda* application's windows states

5.2.4 GUI Metrics

The analysis of source code can provide a means to guide the development of the application and to certify software. Software metrics aim to address software quality by measuring software aspects, such as lines of code, functions' invocations, etc. For that purpose, adequate metrics must be specified and calculated. Metrics can be divided into two groups: internal and external [ISO99].

External metrics are defined in relation to running software. In what concerns GUIs, external metrics can be used as usability indicators. They are often associated with the following attributes [Nie93]:

- Easy to learn: The user can carry out the desired tasks easily without previous knowledge;
- Efficient to use: The user reaches a high level of productivity;
- Easy to remember: The re-utilization of the system is possible without a high level of effort;
- Few errors: The system prevents users from making errors, and recovery from them when they happen;
- Pleasant to use: The users are satisfied with the use of the system.

However, the values for these metrics are not obtainable from source code analysis, rather through users' feedback.

In contrast, internal metrics are obtained from the source code, and provide information to improve software development. A number of authors have looked at the relation between internal metrics and GUI quality.

Stamelos et al. [SAOB02] used the Logiscope² tool to calculate values of selected metrics in order to study the quality of open source code. Ten different metrics were used. The results enable evaluation of each function against four basic criteria: testability, simplicity, readability and self-descriptiveness. While the GUI

²<http://www-01.ibm.com/software/awdtools/logiscope/>, last accessed November 22, 2010

layer was not specifically targeted in the analysis, the results indicated a negative correlation between component size and user satisfaction with the software.

Yoon and Yoon [YY07] developed quantitative metrics to support decision making during the GUI design process. Their goal was to quantify the usability attributes of interaction design. Three internal metrics were proposed and defined as numerical values: complexity, inefficiency and incongruity. The authors expect that these metrics can be used to reduce the development costs of user interaction.

While the above approaches focus on calculating metrics over the code, Thimbleby and Gow [TG08] calculate them over a model capturing the behaviour of the application. Using graph theory they analyse metrics related to the user's ability to use the interface (e.g., strong connectedness ensure no part of the interface ever becomes unreachable), the cost of erroneous actions (e.g., calculating the cost of undoing an action), or the knowledge needed to use the system. In a sense, by calculating the metrics over a model capturing GUI relevant information instead of over the code, the knowledge gained becomes closer to the type of knowledge obtained from external metrics.

While Thimbleby and Gow manually develop their models from inspections of the running software/devices, an analogous approach can be carried out analysing the models generated by GUI SURFER. Indeed, by calculating metrics over the behavioural models produced by GUI SURFER, relevant knowledge may be acquired about the dialogue induced by the interface, and, as a consequence, about how users might react to it.

5.2.5 Graph-Tool

Graph-Tool is an efficient python module for manipulation and statistical analysis of graphs³. It allows for the easy creation and manipulation of both directed or undirected graphs. Arbitrary information can be associated with the vertices, edges or even the graph itself, by means of property maps.

Furthermore, *Graph-Tool* implements all sorts of algorithms, statistics and metrics over graphs, such as shortest distance, isomorphism, connected components, and centrality measures.

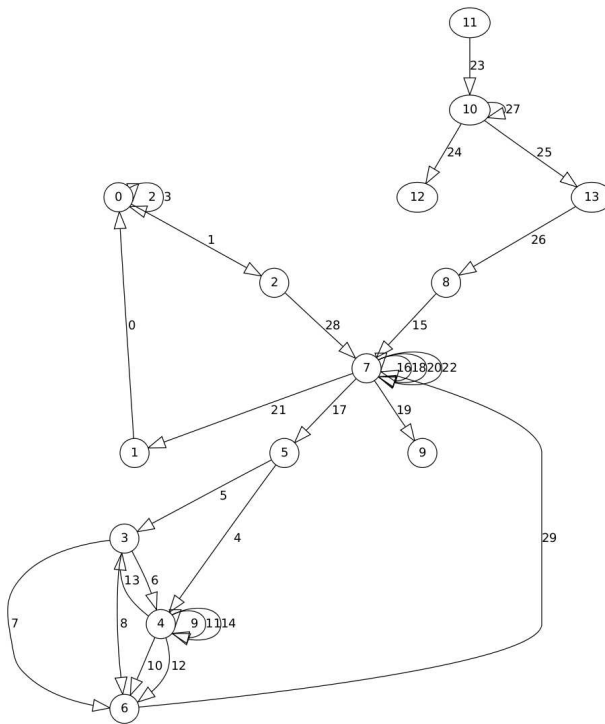


Figure 5.4: *Agenda*'s behaviour graph (numbered)

³see, <http://projects.skewed.de/graph-tool/>, last accessed 27 November, 2010.

Now, for brevity, the graph described in Figure 5.4 (automatically obtained from model of Figure 5.1) will be considered. All vertices and edges are labeled with unique identifiers.

To illustrate the analysis performed with *Graph-Tool*, three metrics will be considered: Shortest distance between vertices, Pagerank and Betweenness. Appendix F contains the complete *Python* script for generating the results of these metrics.

Shortest Distance

Graph-Tool enables the calculation of the shortest path between two vertices. A path is a sequence of edges in a graph such that the target vertex of each edge is the source vertex of the next edge in the sequence. If there is a path starting at vertex *u* and ending at vertex *v* then *v* is reachable from *u*.

For example, the following *Python* command calculate the shortest path between vertices 11 and 6 (i.e. between the *Login* window and a particular *ContactEditor* window state), cf. Figure 5.4.

```
vlist, elist = shortest_path(g, g.vertex(11), g.vertex(6))
print "shortest path vertices", [str(v) for v in vlist]
print "shortest path edges", [str(e) for e in elist]
```

The results for the shortest path between vertices 11 and 6 are:

```
shortest path vertices:
  ['11', '10', '13', '8', '7', '5', '4', '6']
shortest path edges:
  ['(11,10)', '(10,13)', '(13,8)', '(8,7)',
   '(7,5)', '(5,4)', '(4,6)']
]
```

Two representations of the path are provided, one focusing on vertices, the another on edges. This is useful to calculate the number of steps a user needs to

perform in order a particular task.

Now let us consider another inspection. The next result gives the shortest distance (minimum number of edges) from the *Login* window (vertice 11) to all other vertices. The *Python* command is defined as follows:

```
dist = shortest_distance(g, source=g.vertex(11))
print "shortest_distance from Login"
print dist.get_array()
```

The obtained result is a sequence of values:

```
shortest distance from Login
[6 5 7 6 6 5 7 4 3 5 1 0 2 2]
```

Each value gives the distance from vertice 11 to a particular target vertice. The index of the value in the sequence corresponds to the vertice's identifier. For example the first value is the shortest distance from vertice 11 to vertice 0, which is 6 edges long.

Another similar example makes use of *MainForm* window (vertice 7) as starting point:

```
dist = shortest_distance(g, source=g.vertex(7))
print "shortest_distance from MainForm"
print dist.get_array()
```

The result list may contains negative values: they indicate that there are no paths from Mainform to those vertices.

```
shortest distance from MainForm
[2 1 3 2 2 1 3 0 -1 1 -1 -1 -1 -1]
```

This second kind of metric is useful to analyse the complexity of an interactive application's user interface. Higher values represent complex tasks while

lower values express behaviour composed by more simple tasks. This example also shows that its possible to detect parts of the interface that can become unavailable. In this case, there is no way to go back to the login window once the Main window is displayed (the value at indexes 10-13 are equal to -1).

This metric can also be used to calculate the center of a graph. The center of a graph is the set of all vertices where the greatest distance to other vertices is minimal. The vertices in the center are called central points. Thus vertices in the center minimize the maximal distance from other points in the graph.

Finding the center of a graph is useful in GUI applications where the goal is to minimize the steps to execute tasks (i.e. edges between two points). For example, placing the main window of an interactive system at a central point reduces the number of steps a user has to execute to accomplish tasks.

Pagerank

Pagerank is a distribution used to represent the probability that a person randomly clicking on links will arrive at any particular page [Ber05]. That probability is expressed as a numeric value between 0 and 1. A 0.5 probability is commonly expressed as a "50% chance" of something happening.

Pagerank is a link analysis algorithm, used by the Google Internet search engine, that assigns a numerical weighting to each element of a hyperlinked set of documents. The main objective is to measure their relative importance.

This same algorithm can be applied to our GUI's behavioural graphs. Figure 5.5 provides the *Python* command when applying this algorithm to the *Agenda*' graph model.

```

pr = pagerank(g)
graph_draw(g, size=(70,70),
           layout="dot",
           vsize = pr,
           vcolor="gray",
           ecolor="black",
           output="graphTool-Pagerank.pdf",
           vprops=dict([('label', "")]),
           eprops=dict([('label', ""),
                       ('arrowsize', 2.0),
                       ('arrowhead', "empty")]))

```

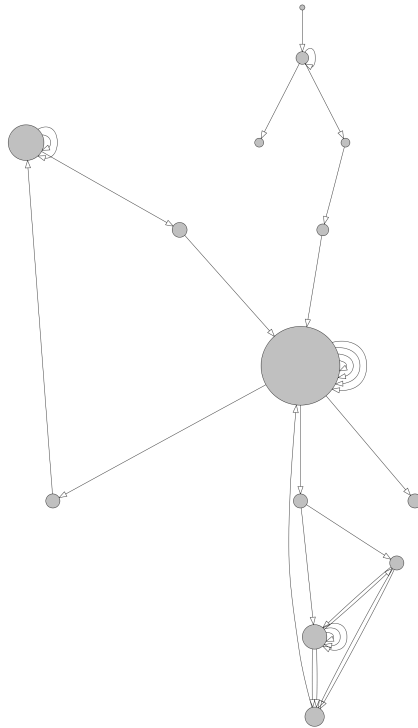
Figure 5.5: *Python* command for Pagerank algorithm

Figure 5.6 shows the result of the Pagerank algorithm giving the *Agenda's* model/graph as input. The size of a vertex corresponds to its importance within the overall application behaviour. This metric is useful, for example, to analyse whether complexity is well distributed along the application behaviour. In this case, the Main window is clearly a central point in the interaction (cf. Figure in 5.1 to see vertices and edges description).

Betweenness

Betweenness is a centrality measure of a vertex or an edge within a graph [Sea09]. Vertices that occur on many shortest paths between other vertices have higher betweenness than those that do not. Similar to vertices betweenness centrality, edge betweenness centrality is related to shortest path between two vertices. Edges that occur on many shortest paths between vertices have higher edge betweenness.

Figure 5.7 provides the *Python* command for applying this algorithm to the *Agenda's* graph model. Figure 5.8 displays the result. Betweenness values for vertices and edges are expressed visually. Highest betweenness edges values are

Figure 5.6: *Agenda's* pagerank results

```

bv, be = betweenness(g)
be1 = be
be1.get_array()[:] = be1.get_array()[:]*120+1
graph_draw(g, size=(70,70),
           layout="dot",
           vcolor="white",
           ecolor="gray",
           output="graphTool-Betweenness.pdf",
           vprops=dict([('label', bv)]),
           eprops=dict([('label', be),
                        ('arrowsize', 1.2),
                        ('arrowhead', "normal"),
                        ('penwidth', be1)]))

```

Figure 5.7: *Python* command for Betweenness algorithm

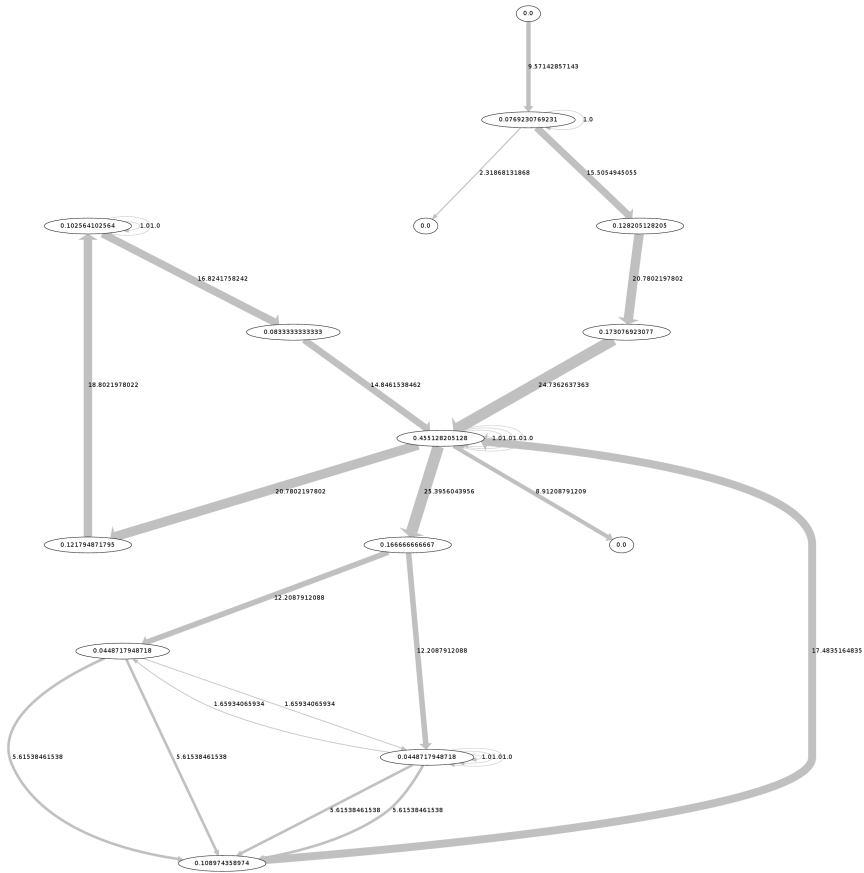


Figure 5.8: Agenda's betweenness values

represented with thicker edges. The Main window has the highest (vertices and edges values) betweenness, meaning it acts as a hub from where different parts of the interface can be reached. Clearly it will be a central point in the interaction.

Cyclomatic Complexity

Another important metric is cyclomatic complexity which aims to measure the total number of decision points in an application [Tho76]. It is used to give the number of tests for software and to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow graph and is defined as $M = E - V + 2P$ (considering a single exit statement) where E is the number of edges, V is the number of vertices and P is the number of connected components.

Considering Figure 5.6 where edges represent decision logic in the *Agenda* GUI layer, the GUI's overall cyclomatic complexity is 18 and each *Agenda*'s window has a cyclomatic complexity less or equal than 10. In applications there are many good reasons to limit cyclomatic complexity. Complex structures are more prone to error, are harder to analyse, to test, and to maintain. The same reasons could be applied to user interfaces. McCabe proposed a limit of 10 for functions's code, but limits as high as 15 have been used successfully as well [Tho76]. McCabe suggest limits greater than 10 for projects that have operational advantages over typical projects, for example formal design. User interfaces can apply the same limits of complexity, i.e. each window behaviour complexity could be limited to a particular cyclomatic complexity. Defining appropriate values is an interesting topic for further research, but one that is out of the scope of the present thesis.

5.2.6 GUI Test Cases Generation

Test case generation is very important since it enables the evaluation of a system by manual or automatic means, and the verification that it satisfies specified properties or identification of differences between expected and actual results.

Section 5.1 has already presented an approach to test case generation and execution using *QuickCheck*. This Section use the graph models generated by GUI SURFER in an approach to GUI test cases generation.

Coverage Criteria

Ideally test cases should contain event sequences to test the totality of an application. Typically this is not possible due the applications' size. Coverage criteria help to determine if a GUI has been sufficiently tested. These coverage criteria use events sequences to specify a measure of test adequacy. More code coverage means higher testing quality. Since the total number of permutations of event and condition sequences in any GUI is extremely large, the GUI's data will be exploited to identify the important event sequences to be tested.

Usually, testers make thousands of paths to cover the most likely user operations. However simulating user behaviour is not enough to prove that the model-based testing process covers all user actions. Considering test cases generation, some user behaviour will be more likely than other. Consequently, if test cases are generated randomly then there is no guarantee that *interesting* behaviours will be tested.

Because our GUI's model representation can be viewed as a graph, the Memon approach to coverage criteria for GUI reasoning has been applied [MSP01]. This

Section defines several coverage criteria following Memon's approach. First an event sequence is formally defined, which is used to describe all the coverage criteria.

An event-sequence is a tuple $\langle e_1, e_2, e_3, \dots, e_n \rangle$ where e_i is a particular event which can be executed after event e_{i-1} , $2 \leq i \leq n$.

Next three coverage criterion are presented. These are applied to GUI behaviour graph-based models.

- **Event Coverage:** The event coverage criterion enables to capture a set of event-sequences considering all possible events. The event coverage criterion is satisfied if and only if for any event e , there is at least one event sequence es such that es contains e ;
- **State Coverage:** State coverage requires that each state is reached at least once, i.e. for any state s there is at least one event-sequence es such that state s is reached in es ;
- **Length-n-Event-sequence Coverage:** Within GUI systems, the behaviour of events may change when executed in different contexts. The length-n-event-sequence coverage criterion defines the set of event-sequences which contains all event-sequences of length equal to n . For example the length-n-event-sequence coverage criterion applied to the *Agenda*'s behavioural model in Figure 5.1 returns the number of test cases provided in the following table:

The result of this last criterion shows that the total number of event sequences grows with increasing length. The large number of event sequences makes it difficult to test a GUI for all possible event sequences.

Length-n	Event-sequences total number
1	1
2	3
3	4
4	10
5	40
6	190
7	940
8	4690
9	23440
10	117190

Table 5.1: *Agenda*'s total number of event-sequences for n event-sequence length

Consequently, Memon proposes to assign priorities to each event-sequence and first test event-sequences with higher priorities. For example, event-sequences related with the main window could have a higher priority since they may be used more times.

The considerations in this Section enable construction of test suites. A test suite is a set of event sequences starting from the initial vertice of the graph. Intuitively, if a test suite satisfies event coverage, it also satisfy state coverage. In other hand event coverage and state coverage are special cases of length- n -event-sequence coverage.

Moreover, in some cases it could be interesting to consider the overall behaviour of the GUI. This perspective can be achieved trough a unique path reaching all possible states (or all possible events) between a start state and a final state. These particular test cases are generated through the Chinese Postman Tour and Travelling Salesman Problem algorithms, described next.

Chinese Postman Tour

The background of the Chinese Postman Problem is about a Chinese postman who wishes to travel along every road in a city in order to deliver letters, while traveling the least possible distance. Solving the problem corresponds to finding the shortest route in a graph in which each edge is traversed at least once [Thi03, PC05, Ski90]. The Chinese postman problem was defined by a Chinese mathematician, Meigu Guan. If the path must get back to the starting point, the problem is said to be closed. If it does not need to go back, it is called an open problem.

The algorithm to solve the open problem is used to generate minimal sequences of user actions between pairs of states, each sequence including all possible users actions in the interface. These sequences are used as test cases for testing the interface against defined properties.

The length of the optimal path for the closed problem acts as a measure of the user interface's complexity [Thi03]. Considering weighted graphs, and assign weights to the transitions that correspond to the time users are expected to take performing the corresponding actions, the optimal path for closed problem is used to calculate how long a user takes to explore an entire application.

In order to optimize our model-based reasoning approach, we apply the Chinese postman algorithm to generate a test case that uses all possible events / conditions and states.

Travelling Salesman Problem

The Travelling Salesman Problem (TSP) considers a salesman whose task is to find a shortest possible tour that visits each city in a region exactly once.

While in the Chinese Postman Problem the goal is to traverse every edge at least once, in the Travelling Salesman Problem the goal is to visit every node. There is no need to use all edges in the graph. Paths produced as a solution to this problem specify that all window states will be visited by the user, while keeping user actions to a minimum.

Both algorithms are implemented and used in GUISURFER. These algorithms generate test cases which are used to explore an entire application considering all possible events and states.

Related Work

Several alternatives to generate test cases are proposed in the literature. Finite state machines can be used to model system and to generate test cases [SL89, Ura92].

Memon's approach to coverage criteria for GUI testing makes use of an event flow graph for GUI's behavioural representation [MSP01]. His work presents a methodology for generating test cases from GUI behaviour graph-based specifications. Coverage criteria is presented to help determine whether a GUI has been adequately tested.

Ping Li describes another approach to testing GUI systems in [LHRM07]. In the proposed approach, GUI systems are divided into two abstract tiers: the component tier and the system tier. On the component tier, a flow graph is created for each GUI component, describing relationships between the pre-conditions, event sequences and post-conditions. On the system tier, the components are integrated resulting in a view of the entire system. Finally, tests on the system tier analyse the interactions between the components.

5.3 Conclusions

In this Chapter a GUISURFER based GUI analysis process has been illustrated. The process uses GUISURFER's reverse engineering capabilities to enable a range of model-based analysis being carried out. Different analysis methodologies are described. The methodologies automate the activities involved in GUI reasoning, such as, test case generation, or verification. GUI behavioural metrics are also described as a way to analyse GUI quality.

The contributions related with this Chapter were described in the following papers presented at international and national conferences:

- *GUI Behaviour from Source Code Analysis*, J.C.Silva, J.C. Campos, J. Saraiva, presented at the 4th *Conferência Nacional Interação Humano-Computador (Interação 2010)* Universidade de Aveiro, Aveiro, Portugal, 2010;
- *GUI Inspection from Source Code Analysis*, J. C. Silva, J. C. Campos, J. Saraiva. In proceedings of the Fourth International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010). Electronic Communications of the EASST, Pisa, Italy, 2010.

Chapter 6

HMS Case Study: A Larger Interactive System

In previous Chapters, we have presented the GUISURFER tool and all the different techniques involved in the analysis and the reasoning of interactive applications. We have used several simple/small examples in order to motivate and explain our approach. In this Chapter, we present the application of GUISURFER to a complex/large real interactive system: a Healthcare management system available from *Planet-source-code*. The goal of this Chapter is twofold: Firstly, it is a proof of concept for the GUISURFER. Secondly, we wish to analyse the interactive parts of a real application.

The chosen interactive system is related to a Healthcare Management System (HMS), and can be downloaded from *Planet-source-code* website¹. *Planet-source-code* is one of the largest public source code database on the Internet.

The HMS system is implemented in *Java/Swing* and supports patients, doctors and bills management. The implementation contains 66 classes, 29 windows

¹<http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=6401&lngWId=2>, last accessed November 22, 2010

forms (message box included) and 3588 lines of code. The following Subsections provide a description of the main *HMS* windows and the results generated by the application of GUISURFER to its source code.

6.1 Login Window

The window in Figure 6.1 is the first *HMS* window that appears to users. This window gives authorized users access to the system, and the *HMS* main form by introducing a username and password. This window is very similar to the *Agenda*'s login window we have presented in the thesis (see Section 3.1). It is composed of two text box (i.e. username and password input) and two buttons (i.e. *Login* and *Exit* buttons).

If the user introduces a valid username/password and presses the *Login* button, then the window closes and the main window of the application is displayed. On the contrary, if the user introduces invalid data, then a warning message is produced and the login window continues to be displayed. By pressing the *Exit* button, the user exits the application.

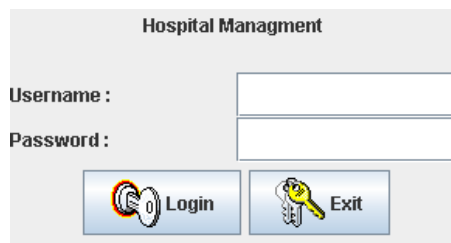


Figure 6.1: *HMS*: Login window

Applying the GUISURFER to the source code of the application, and focusing

on the login window, enables the generation of several models. Figure 6.2, for example, shows the state machine generated to capture the login window's behaviour.

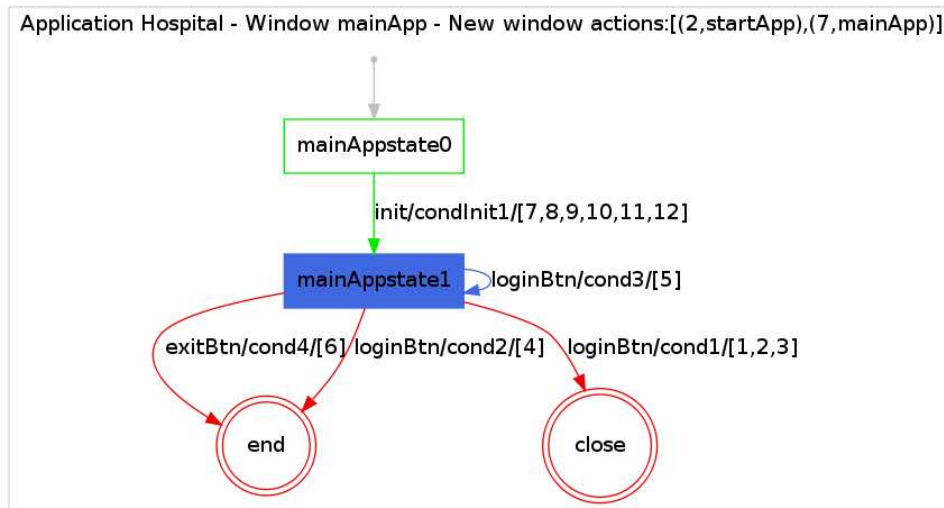


Figure 6.2: HMS: Login state machine

Analysing this model, one can infer that there is a pair event/condition (edge *loginBtn/cond1* with action list [1,2,3]) which closes the window (cf. edge moving to *close* node) and subsequently opens another window (identified as *startApp* through action reference 2). Furthermore, one can also infer that there are two event/condition pairs (edge *exitBtn/cond4* with action list [6] and edge *loginBtn/cond2* with to action list [4]) which exit the system. These events can be executed by clicking the *Exit* or *Login* buttons, respectively.

The informal description of login window behaviour provided at the start of the Section did not included the possibility of exiting the system by pressing the *Login* button. The extracted state machine however defines that possibility, which can

occur if condition *cond2* is verified (cf. pair *loginBtn/cond2* with action list [4]).

Analysing condition *cond2* (*source.equals(exitBtn)*), dead code was encountered.

The source code executed when pressing the *Login* button is the following:

```
public void actionPerformed(ActionEvent event)
{ Object source = event.getSource();
  if (source.equals(loginBtn))
  { String loginname, loginpass;
    loginname = userTxt.getText().trim();
    loginpass = passwordTxt.getText().trim();
    if(valid(loginname, loginpass))
    { new start();
      setVisible(false);
      frame.dispose();
    }
    else showMessageDialog("Invalid User name and password");
  }
  else if(source.equals(exitBtn))
  { System.exit(0);
  }
}
```

This code uses a condition to test whether the clicked button is the *Login* button or not. This is done through the boolean expression *source.equals(loginBtn)*. However, the above action source code is only performed when pressing the *Login* button. Thus, the condition will always be verified and the following *else* component of the conditional statement will never be executed.

```
else if(source.equals(exitBtn))
{ System.exit(0);
}
```

Summarizing the results obtained for the login window, one can say that the generated state machine contains an event/condition/actions representation which can not be executed despite being defined on the behavioural model. This example demonstrates how comparing expected application behaviour against the models

generated by GUISURFER can help to understand and detect dead code in applications.

6.2 Main Window

The window displayed by *HMS* system after login is presented in Figure 6.3. From this window, users can have access to four other windows through buttons *Patient*, *Doctor*, *Billing* and *Report*. Users can also exit the applications through the *EXIT* button, and finally the *BACK* button ends the session and the system goes back to the login window.

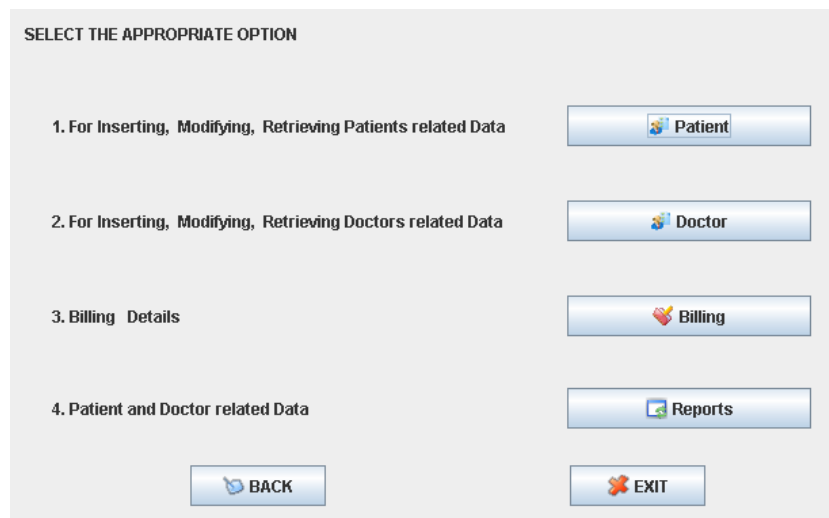


Figure 6.3: *HMS*: Main window

Figure 6.4 shows the state machine generated for *HMS*'s main window. As expected, GUISURFER inferred six events, one for each button. Five events close the main window when executed (edges moving to the *close* node). These events are identified as the following *bback*, *bbill*, *bdoc*, *bpat*, and *breport*. At the top

of the Figure 6.4, the first line defines action references to open others windows (*New window actions*). These are 1, 3, 5, 8 and 10 which open windows *patStart*, *docStart*, *Billing*, *mainApp* and *Report*, respectively. Thus, events *bback*, *bbill*, *bdoc*, *bpat* and *breport* open windows *mainApp*, *Billing*, *docStart*, *patStart*, and *Report*, respectively. Finally, the *bexit* event ends the application (edge moving to the *end* node).

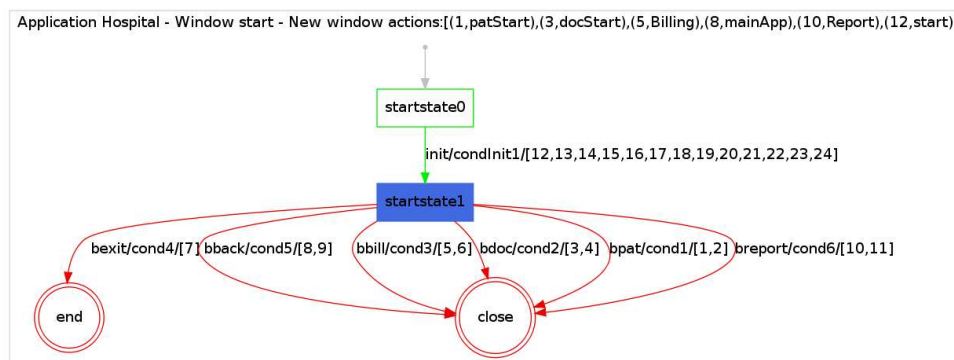


Figure 6.4: HMS: Main window state machine

6.3 Patient Management

In this Section, Figures 6.5 and 6.7 present two forms related to the patient management. The first Figure contains the patient's main form. Through four buttons, users can open forms to add new patient information (clicking on button *Add Data*), to modify patient information (clicking on button *Modify Data*) or to view patient information (clicking on button *View Data*). Finally the *BACK* button takes the interface back to the previous form (cf. Main window in Figure 6.3). Clicking on each of these buttons will also close the patient's main form.

The second Figure (6.7) provides the form that users can access when clicking on button *View Data* from the patient's main form (see Figure 6.5). Basically, this form enables viewing information regarding a particular patient identified through the patient's number. This form is composed of several text and list boxes. Three buttons are also present allowing to search of new patient information by clicking on button *SEARCH*, the clearing of all widget data is done by clicking on button *CLEAR*, and going back to the patient's main form is done by clicking on button *BACK*.

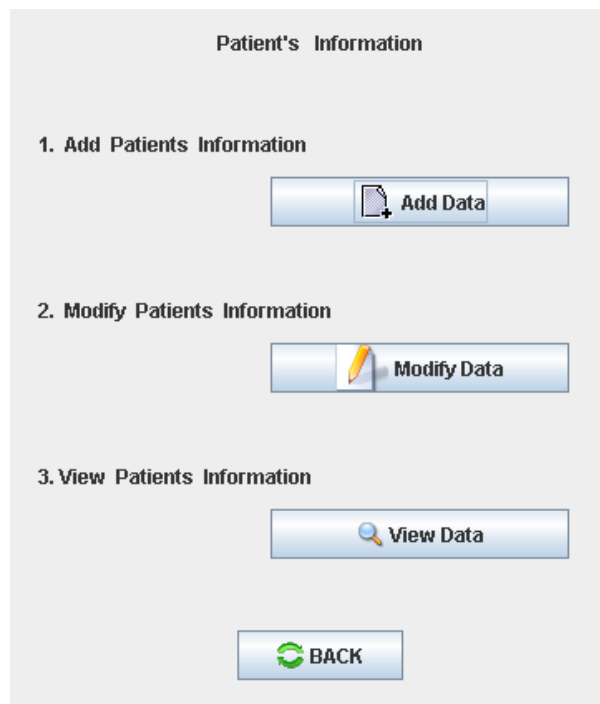


Figure 6.5: *HMS*: Main patient form

Applying GUISURFER to the source code of these two forms (cf. Figure 6.5 and 6.7) enables the generation of two states machines. The state machine in Figure

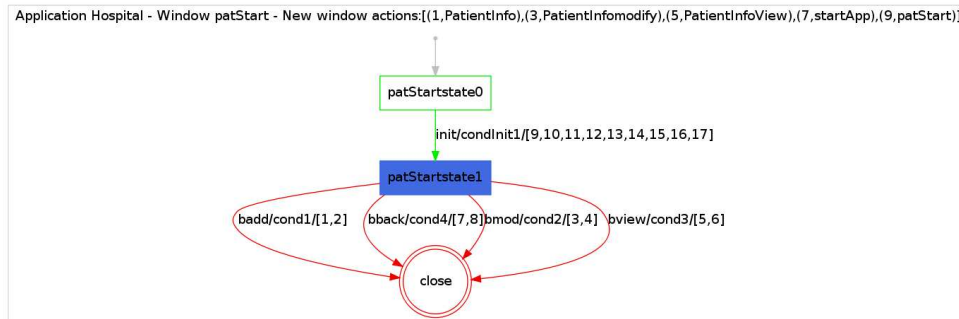


Figure 6.6: HMS: Main patient state machine

View Patient Information

Personal Information		Insert Patient Number	
Name :	<input type="text"/>	Patient No.:	<input type="text"/> Room No.: <input type="text"/>
Address :	<input type="text"/>	Contact :	<input type="text"/>
		Date Of Admission :	<input type="text"/> (dd-mm-yyyy)
		Gender :	<input type="text"/>
Medical Information			
Blood Group :	<input type="text"/>	Date of Birth :	<input type="text"/> (dd-mm-yyyy)
History :	<input type="text"/>	Current Problem :	<input type="text"/>
Type Of Room :	<input type="text"/>	Attending Doctor :	<input type="text"/>

Figure 6.7: HMS: View patient information form

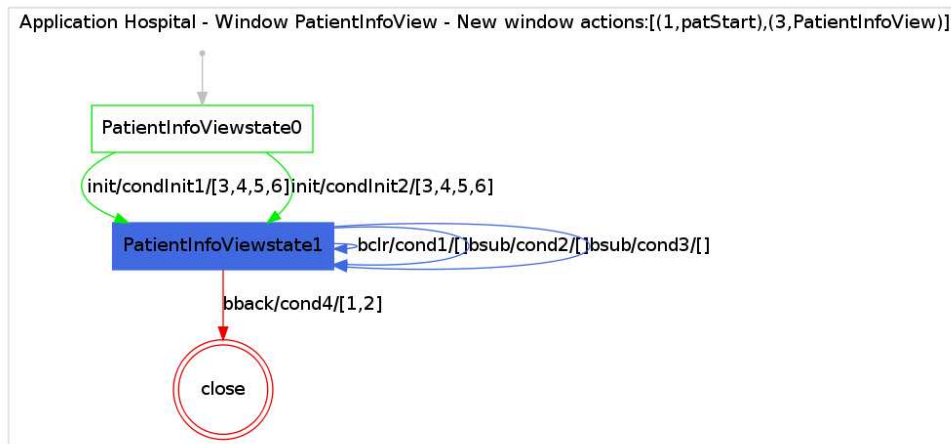


Figure 6.8: *HSM*: View patient information state machine

6.6 represents the first form's behaviour. From state *patStartstate1*, there are four possible pairs of event/condition. As described above, users can add, modify, view patient information or go back to the previous form (i.e. to the *startApp* form). This state machine patient's information also shows that each pair of event/condition when executed will close the patient's information form (each pair moves to the *close* node).

Finally, the state machine in Figure 6.8 describes the behaviour model extracted from the form shown in Figure 6.7. As expected, the obtained results show that users can execute several searches from the same form state (state *PatientInfoViewstate1* and pairs of event/condition *bclr/cond1*, *bsub/cond2*, *bsub/cond3*). Users can also exit the form and go back to the main patient's form (*patStart*) through the *bback/cond4* event/condition pair. This pair is associated with action reference 1 which opens the main patient's form (see at the top of the Figure 6.8, *New window*

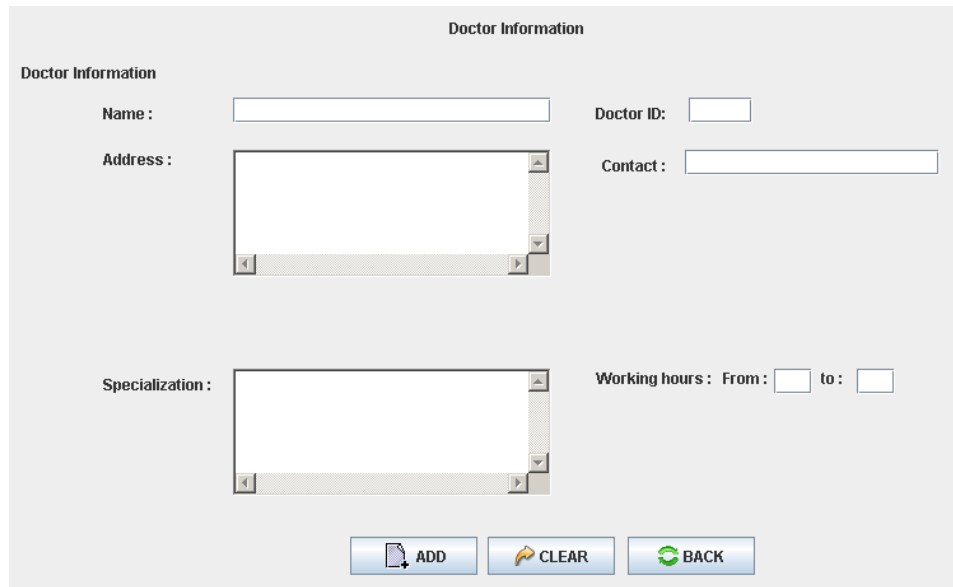
actions: 1, *patStart*) where *patStart* specifies the main patient form described in Figure 6.5).

6.4 Doctors Management

More results are listed in this Section considering doctor's information management forms. Figure 6.9 contains a form (namely, *DoctorInfoAdd*) which can be used to insert new doctor's data in the system. The form uses the usual widgets, i.e. textfields, labels, list boxes, buttons, etc. As with previous forms GUI SURFER automatically generates a state machine specifying its behaviour. This state machine is shown in Figure 6.10. The generated state machine specifies several event/condition pairs manipulating *DoctorInfoAdd* form's state *DoctorInfoAddstate1*. Furthermore, there is only one pair event/condition to close the form, i.e. pair *bback/cond1/[5,6]* moving to the *close* node, where action reference 5 opens the *docStart* form.

6.5 Bills Management

This Section presents results obtained when working with the billing form provided in Figure 6.11. Using this form, users can search bills (by clicking on the *SEARCH* button), clear all widget's assigned values (by clicking on the *CLEAR* button) or go back to the previous form (i.e. *startApp* form provided in Figure 6.3). Figure 6.12 presents the generated state machine. There is only one way to close the form *Billing*. Users must select the *bback* event, verifying the *cond9* condition (cf. pair *bback/cond9/[1,2]*). This event enables moving to the *close* node, thus closing the *Billing* form, and opening the *startApp* form through action reference 1.



Doctor Information

Doctor Information

Name :

Address :

Specialization :

Doctor ID:

Contact :

Working hours : From : to :

ADD CLEAR BACK

Figure 6.9: HSM: Add doctor form

6.6 Overall Behaviour

GUI SURFER extracts the overall behavioural model of an interactive system considering its windows. Figures 6.13, 6.14 and 6.15 provides three fragments of a same model that constitutes the behavioural model: left, right and center parts of a state machine where each node specifies a window. Edges between nodes define tuples of *state / event / condition / actions* showing which *event* allows opening a new window from a particular *state* of the source window. Each of these transitions between nodes opens a new window and closes the initial one. For example, in Figure 6.14, from the internal state of the *startApp* window *state1*, users have access to the *patStart* window through the *bpat* event if *cond1* is verified. This information is specified by an edge with the label *startApp state1 bpat cond1*,

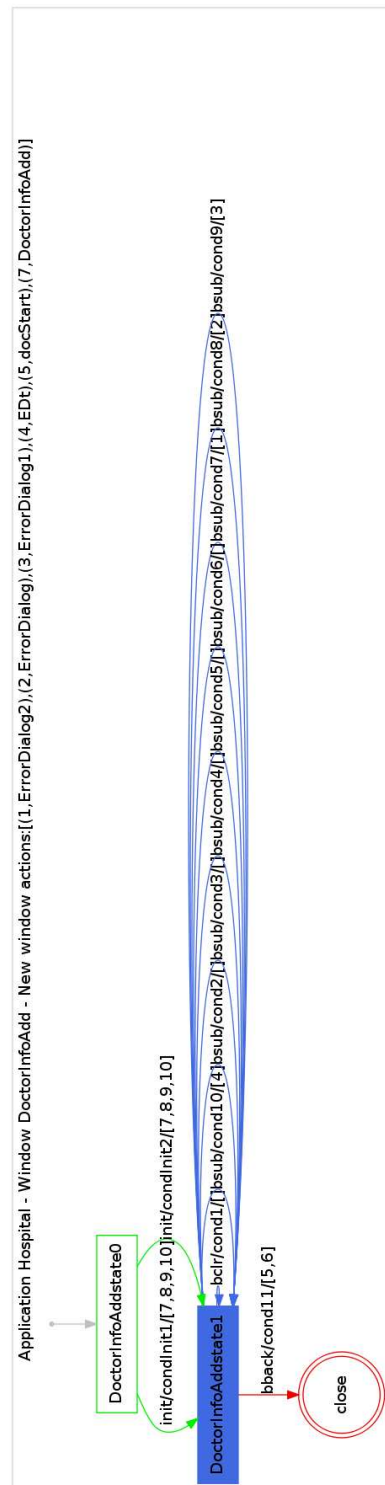


Figure 6.10: HSM: Add doctor behavioural state machine

Billing Information

Patient Name : Patient No. :

Date of Admission : Date of Discharge : 13-09-2010

Room Type :

Total Amount :

Figure 6.11: *HSM*: Billing form

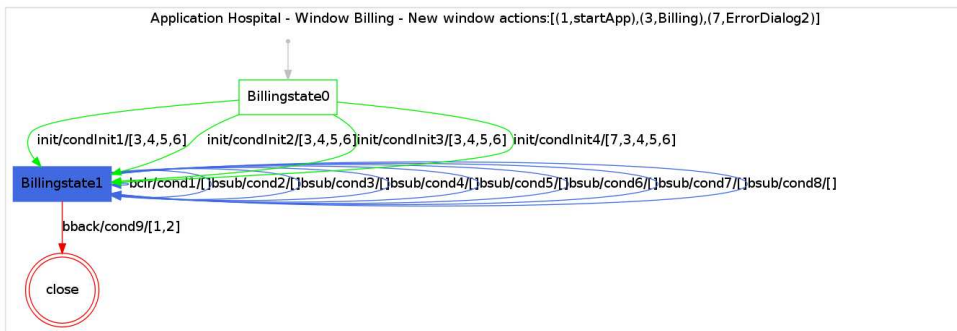


Figure 6.12: *HSM*: Billing form behaviour state machine

from the *startApp* node to *patStart* node

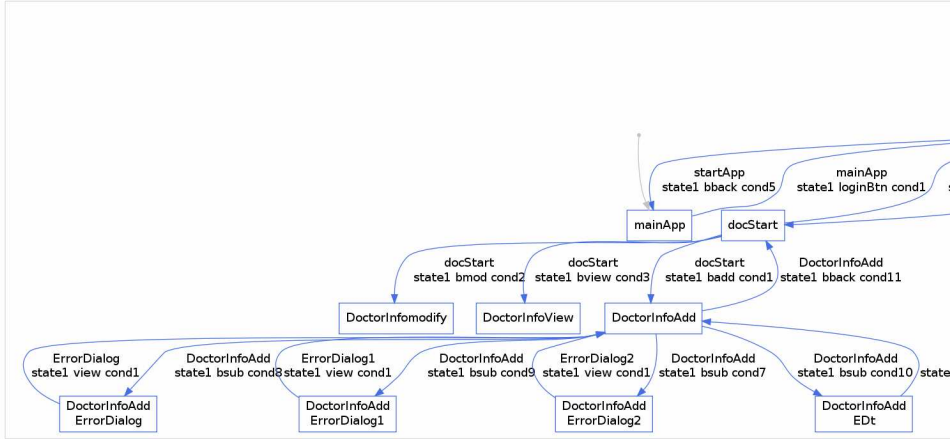


Figure 6.13: *HSM*: The overall behaviour (left part)

6.7 GUI Reasoning

Section 5.2 described GUI analysis performed on the *Agenda* application’s behavioural graph (cf. Figure 5.1), using the *Graph-Tool* for the manipulation and statistical analysis of graphs.

In this Section, two metrics will be applied in order to illustrate the same kind of analysis: PageRank and Betweenness.

Figure 6.16 provides a graph with the overall behaviour of the HMS system. This model can be seen in more detail in the electronic version of this thesis. Basically, this model aggregates the state machines of all HMS forms. The right top corner node specifies the HMS entry point, i.e. the *mainAppstate0* creation state from the login’s state machine (cf. Figure 6.2).

PageRank is a link analysis algorithm, that assigns a numerical weighting to

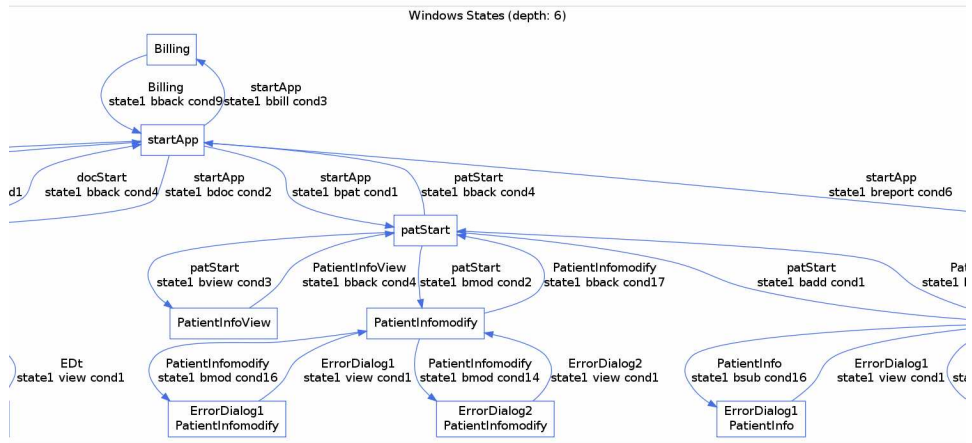


Figure 6.14: HSM: The overall behaviour (center part)

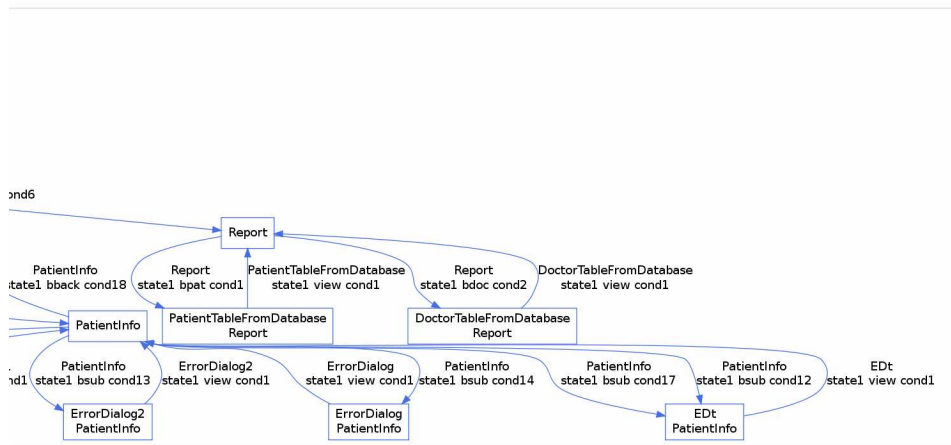


Figure 6.15: HSM: The overall behaviour (right part)

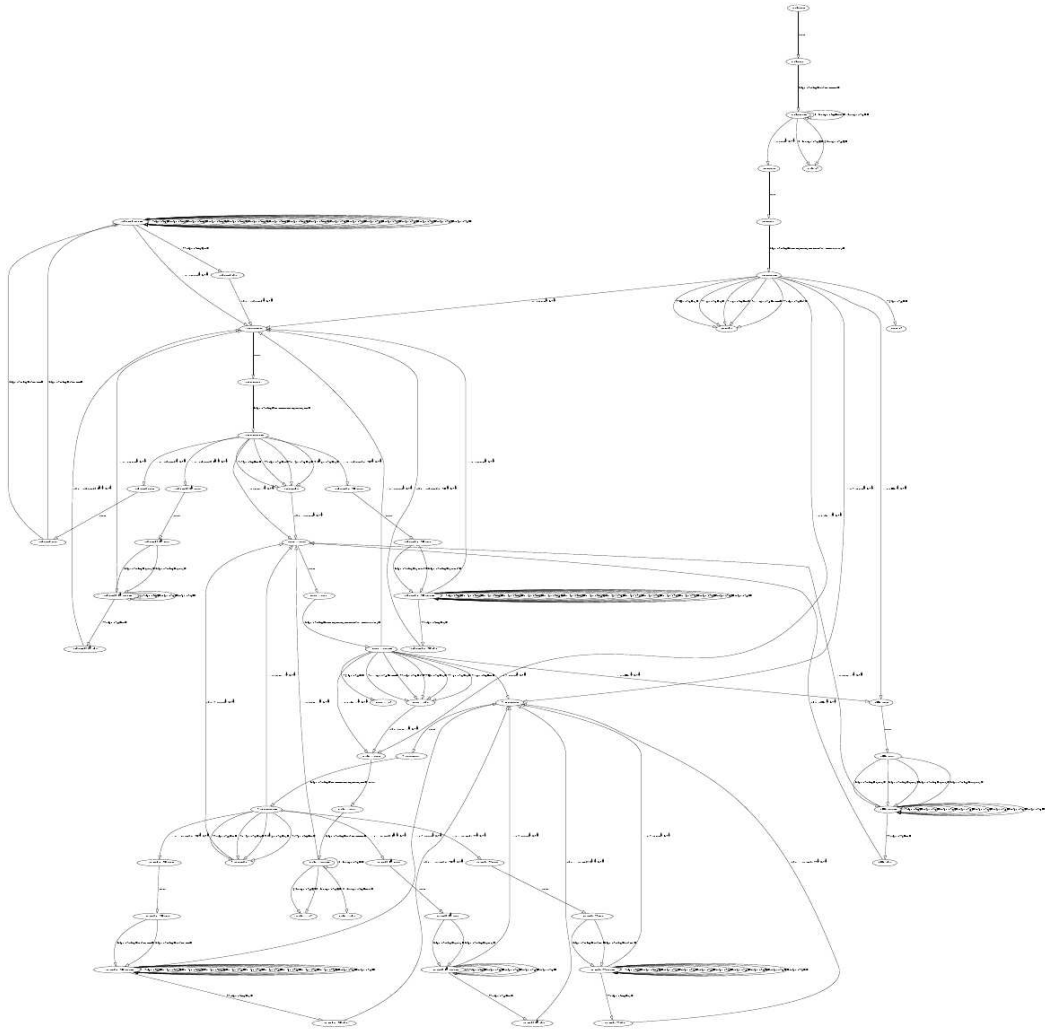


Figure 6.16: *HSM*: The overall behaviour

each node. The main objective is to measure the relative importance of the states. Larger nodes specifies window internal states with higher importance within the overall application behaviour.

Figure 6.17 provides the result obtained when applying the pagerank algorithm to graph of Figure 6.16. This metric can have several applications, for example, to analyse whether complexity is well distributed along the application behaviour. In this case, there are several points with higher importance. The interaction complexity is well distributed considering the overall application.

Betweenness is a centrality measure of a vertex or an edge within a graph. Vertices that occur on many shortest paths between other vertices have higher betweenness than those that do not. Similar to vertices betweenness centrality, edge betweenness centrality is related to shortest path between two vertices. Edges that occur on many shortest paths between vertices have higher edge betweenness. Figure 6.18 provides the obtained result when applying the betweenness algorithm (c.f. Section 5.2.4). Betweenness values are expressed numerically for each vertices and edges. Highest betweenness edges values are represented by larger edges. Some states and edges have the highest betweenness, meaning they act as a hub from where different parts of the interface can be reached. Clearly they represent a central axis in the interaction between users and the system. In a top down order, this axis traverses the following states *patStartstate0*, *patStartstate1*, *startAppstate0*, *startAppstate1*, *docStartstate0* and *docStartstate1*. States *startAppstate0* and *startAppstate1* are the main states of the *startApp* window's state machine (cf. Figure 6.3 and Figure 6.4). States *patStartstate0*, *patStartstate1* are the main states of the *patStart* window's state machine (cf. Figure 6.5 and Figure 6.6). Finally, *docStart-*

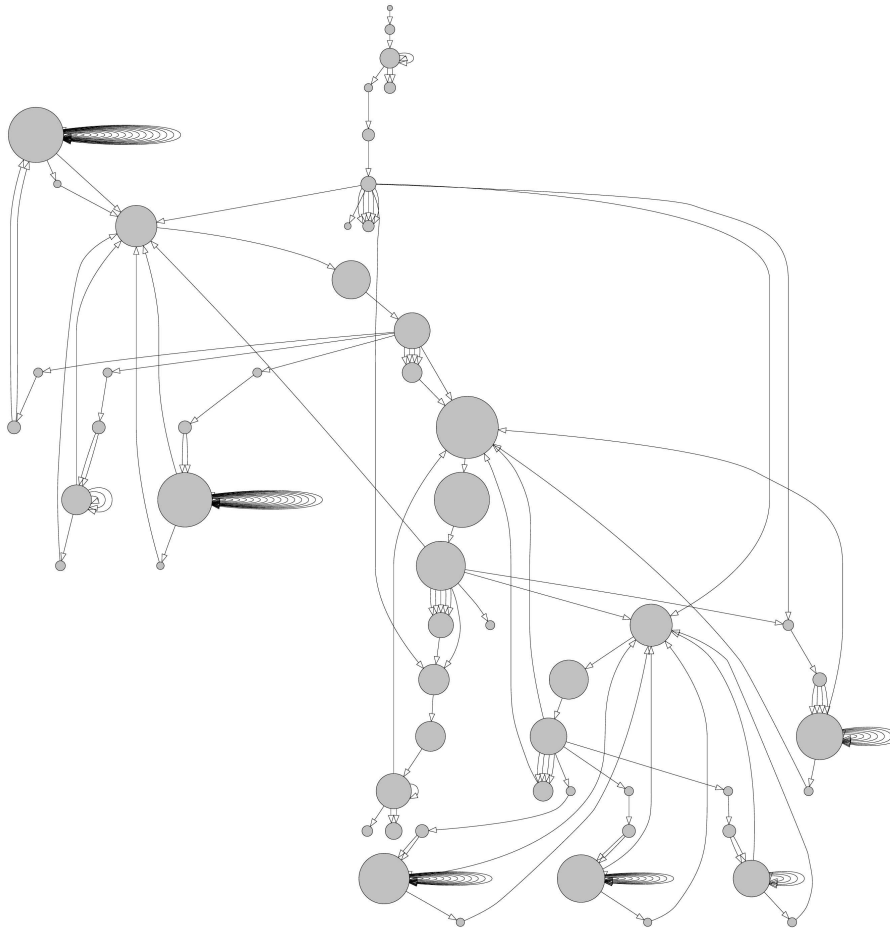


Figure 6.17: *HSM's* pagerank results

state0 and *docStartstate1* belong to *docStart* window's state machine (*docStart* is the main doctor window).

As another perspective, the event/condition/actions tuples present in this central axis are:

- *bback/cond4/[7,8]*: This tuple specifies a *patStart* form's event opening the *startApp* window from *patStartstate1* state. See Figures 6.5 and 6.6 providing *patStart* form and *patStart*'s state machine;
- *bdoc/cond2/[3,4]*: This tuple defines a *startApp* form's event opening *docStart* window from *startAppstate1* state. See Figures 6.3 and 6.4 providing *startApp* form and *startApp*'s state machine;
- *bpac/cond1/[1,2]*: This tuple defines a *startApp* form's event opening *patStart* window from *startAppstate1* state. See Figures 6.3 and 6.4 providing *startApp* form and *startApp*'s state machine;
- *bback/cond4/[7,8]*: This tuple defines a *docStart* window event. This is the *bback* event which opens the *starApp* window from *docStartstate1* state.

It is expectable that the dialogue between users and *HMS* system is essentially around these events, states and windows.

6.8 Conclusions

This Chapter described the results obtained with *GUISURFER* when applying it to a larger interactive system. The chosen interactive system case study is related to a healthcare management system (*HMS*). The *HMS* system is implemented in

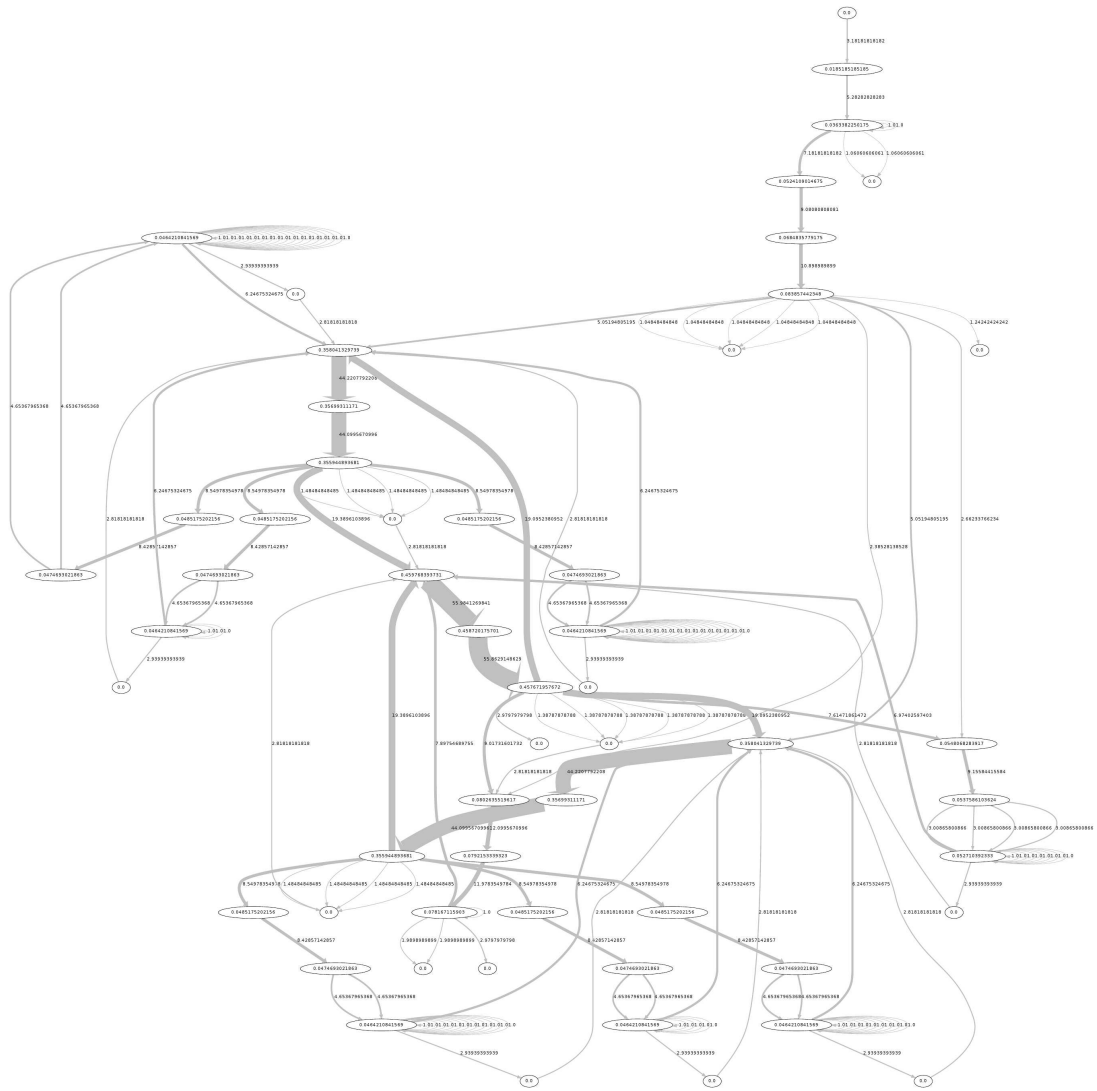


Figure 6.18: *HSM's* betweenness values

Java/Swing programming language and implement operations to allow for patients, doctors and bills management. A description of main *HMS* windows has been provided, and *GUIsurfer* results have been described. The *GUI SURFER* tool enabled the extraction of different behavioural models. Methodologies have been also applied automating the activities involved in GUI model-based reasoning, such as, pagerank and betweenness algorithms. GUI behavioural metrics have been used as a way to analyse GUI quality. This case study demonstrated that *GUI SURFER* enables the analysis of real interactive applications written by third parties.

Chapter 7

Conclusions and Future Work

This thesis presented an approach to GUI reasoning using reverse engineering techniques. This document concludes with a review of the work developed. The resulting research contributions are presented and directions for future work are suggested.

The first Section presents the answers to research questions defined in the first Chapter. The second Section describes the contributions of the thesis. A discussion about GUISURFER limitations is provided in Section 3. Finally, the last Section presents some future work.

7.1 Answers to Research Questions

In the beginning of this Phd project several aspects of interactive applications have been identified. In order to guide our study three research questions have been raised. These are defined in the introduction of this document and will answered in this Section.

The research goal for this thesis was to demonstrate that:

Interactive application's source code can be used for automatic generation of

GUI behavioural models and subsequent GUI behavioural reasoning through a retargetable approach.

The three research questions raised by the research goal provided guidance for the research. The findings related both directly and indirectly to answering each question have been discussed in depth along appropriate Chapters. Issues related to answering Question One about inferring realistic behavioural models of a GUI from its application's source code are discussed in Chapter 3 (Sections 3.2, 3.3 and 3.5) and Chapter 4 (Section 4.3). Results related to Question Two, about language independent techniques for GUI modelling and reasoning are discussed in Chapter 3 (Section 3.4) and Chapter 4 (Section 4.4). The third research question is directed at re-using well-known algorithms and metrics for GUI reasoning through GUI behavioural models. Results related to this question are discussed along Chapter 5 (Sections 5.1 and 5.2). The following Sections summarize these discussions and reflect more widely on relevant issues.

- **Question One:** *Can we infer realistic behavioural models of a GUI from its application's source code?* This research has demonstrated that the behaviour of a GUI can be automatically extracted considering the software's windows, their widgets, properties, values and control flow. The approach follows a reverse engineering methodology and starts by making use of a parser to generate an AST. Therefore, traversals of the AST may be executed enabling extraction of the GUI layer. The execution model of the user interface is obtained by using a classification of its events. The approach has proven very flexible. From application's source code the developed tool is able to derive both interactor based models, event flow graphs and state

machines. In the first case the models capture a user oriented view of the interface. In other cases models capture the internal behaviour of the application. Several examples of case studies have been described, showing realistic behavioural models automatically extracted from source code.

- **Question Two:** *Can we define a language independent technique for GUI modelling and reasoning?* Our study demonstrated that strategic programming and slicing techniques can be combined to reverse engineer user interfaces from the source code of its application written in different languages (cf. Section 4.4). A first prototype has been developed allowing analysis of *Java/Swing* source code. Afterwards, the prototype was extended to consider other programming languages, namely *GWT* and *WxHaskell*. Hence, the work has proven the retargetability of the proposed techniques and the developed prototype.
- **Question Three:** *Can we use well-known algorithms and metrics to reason about GUI behavioural models?* This research has demonstrated that GUI reasoning can be done making use of GUI behavioural models. Two tools have been applied. The first one enables generation and validation of test cases from a model of the GUI using *QuickCheck*, a tool for testing *Haskell* programs automatically. *Graph-Tool* is another tool used for manipulation and statistical analysis of graphs. The tool implements all sorts of algorithms, statistics and metrics over graphs, such as shortest distance, isomorphism, and centrality measures. Using *Graph-Tool* and an analogy between state machines and graphs, several algorithms and metrics have been used to

reason about GUI behavioural models.

7.2 Summary of Contributions

The major contribution of this work is the development of the GUI SURFER prototype, an approach for improving GUI analysis through reverse engineering. This research has demonstrated how user interface layer can be extracted from different source codes, identifying a set of widgets (graphical objects) that can be modeled, and identifying also a set of user interface actions. Finally this thesis has presented a methodology to generate behavioural user interface models from the extracted information and to reason about it.

The approach is very flexible, indeed the same techniques have been applied to extract similar models from *Java/Swing*, *GWT* and *WxHaskell* interactive applications.

In what concerns user interface development, two perspectives on quality can be considered. Users, on one hand, are typically interested on what can be called external quality: the quality of the interaction between users and system. Programmers, on the other hand, are typically more focused on the quality attributes of the code being produced.

This work is an approach to bridging the gap between users and programmers by allowing the reasoning about GUI models from source code. This thesis described GUI models extracted automatically from the code, and presented a methodology to reason about the user interface model. A number of metrics over the graphs representing the user interface were investigated. Some initial thoughts on testing the graph against desirable properties of the interface were also put for-

ward. We believe this style of approach can feel a gap between the analysis of code quality via the use of metrics or other techniques, and usability analysis performed on a running system with actual users.

This thesis has shown that reasoning from models provides an easy way to implement interactive systems analysis. Models provide a tool to explore GUI properties. This thesis provides a variety of models and discuss their importance to GUI analysis. These models have been from different case studies, demonstrating how the approach enables to reason about GUI models.

7.3 Discussion

Using GUISURFER, programmers are able to reason about the interaction between users and a given system at a higher level of abstraction than that of code. The generated models are amenable to analysis via model checking (c.f. [CH09]). In this work, alternative lighter weight approaches have been explored .

Considering that the models generated by the reverse engineering process are representations of the interaction between users and system, this research explored how metrics defined over those models can be used to obtain relevant information about the interaction. This means that the approach enable to analyse the quality of the user interface, from the users perspective, without having to resort to external metrics which would imply testing the system with real users, with all the costs that the process carries.

It must be noted that, while the approach enables to analyse aspects of user interface quality without resorting to human test subjects, the goal is not to replace user testing. Ultimately, only user testing will provide factual evidence of the us-

ability of a user interface. The possibility of performing the type of analysis will help in gaining a deeper understanding of a given user interface. This will promote the identification of potential problems in the interface, and support the comparison of different interfaces, complementing and minimizing the need to resort to costly user testing.

Results show the reverse engineering approach adopted is useful but there are still some 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 limitation has to do with layout management issues. GUI SURFER cannot extract, for example, information about overlapping windows since this must be determined at run time. Thus, it can not be found 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, GUI SURFER can be used to extract a model of the implemented system, and a comparison between the two can be carried out.

A number of other issues still needs addressing. In the examples used throughout the thesis, only one window could be active at any given time (i.e., windows were modal). When non-modal windows are considered (i.e., when users are able to freely move between open application windows), nodes in the graph come to

represents sets of open windows instead of a single active window. This creates problems with the interpretation of metrics that need further consideration. The problem is exacerbated when multiple windows of a given type are allowed (e.g., multiple editing windows).

7.4 Future Work

The work developed in this thesis open a new set of interesting problems that need research. This Section provides some pointers for future work.

7.4.1 GUISURFER Extension

In the future, the implementation can be extended to handle more complex widgets. Others programming languages/toolkits can be considered, in order to make the approach as generic as possible.

GUISURFER may be also extended to other kinds of interactive applications. There are categories of user interfaces that cannot be modeled in GUISURFER, for example, system incorporating continuous media or synchronization/timing constraints among objects. Thus, the identification of the problems that GUISURFER may present when modelling these user interfaces would be the first step towards a version of GUISURFER suitable for use with other kinds of interactive applications. Finally, the tool and the approach must be validated externally. Although the approach has already been applied by another researcher, it is fundamental to apply this methodology with designers and programmers. Empirical studies need also to be executed to compare the GUI models users would define by hand with the automatic one produced by GUISURFER.

7.4.2 GUI Reengineering

As another area of future work, techniques can be explored for restructuring GUI source code in order to make it more reusable, reliable and maintainable. The GUI SURFER tool is capable of deriving GUI models of applications written in different programming paradigms. Now, the goal is to go one step further and re-engineer GUI applications, extending the existing approach, exploring model transformation and analysis, and applying the approach to large scale industrial systems. Figure 7.1 models the proposed re-engineering process of interactive applications from source code. The approach is basically a process of reverse engineering followed by a forward engineering process in order to change a system. Hence, the re-engineering process involves moving to a higher abstraction level, enabling to create GUI specifications, adding new functionality to this specification and developing a new implementation by using forward engineering techniques. The higher GUI abstraction level could be used as the basis for a transformation/refactoring process. Then, through forward engineering, a new source code for the GUI could be generated.

7.4.3 Patterns for GUI transformation

Patterns may be used to obtain better systems through the re-engineering of GUI source code across paradigms and architectures. The architect Christopher Alexander has introduced design patterns in early 1970. He defines a pattern as a relation between a context, a problem, and a solution. Each pattern describes a recurrent problem, and then describes the solution to that problem. Design patterns gained popularity in computer science, cf. [GHJV95]. In software engineering, a design

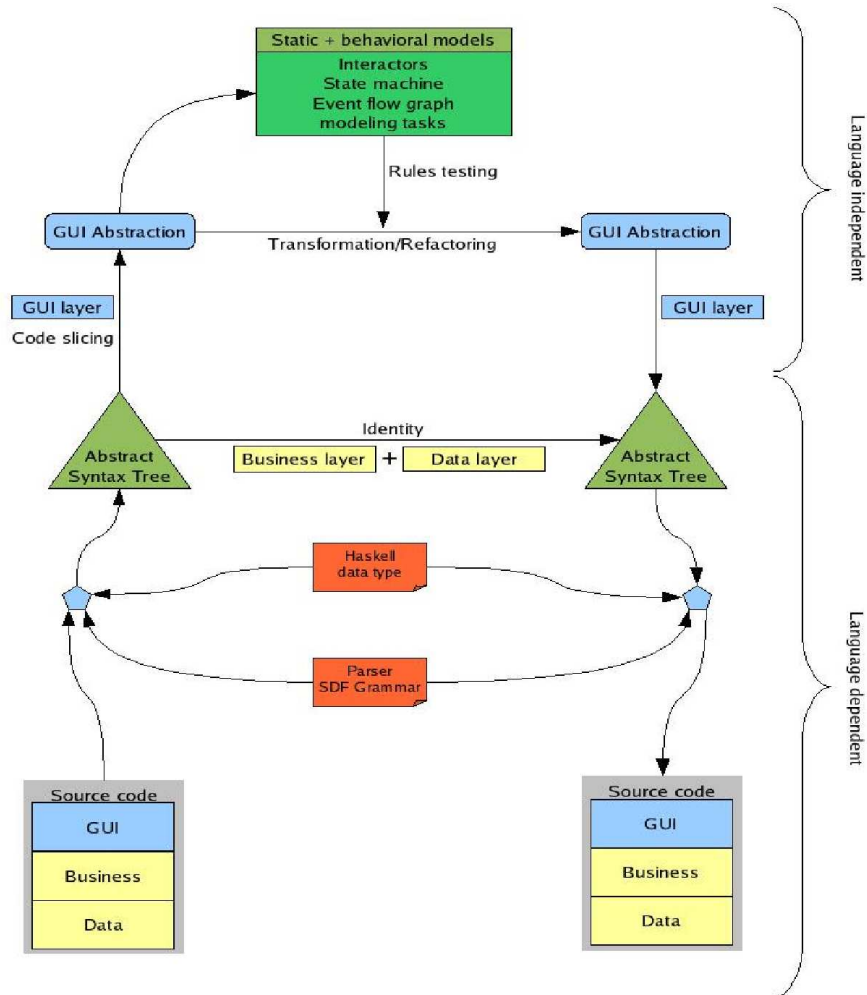


Figure 7.1: Re-engineering of interactive applications from source code

pattern is a general reusable solution to a commonly occurring problem in software design. Patterns are used in different areas including software architecture, requirements and analysis. The human computer interaction (HCI) community has also adopted patterns as a user interface design tool. In the HCI community, patterns are used to create solutions which help user interfaces designers to resolve GUI development problems. Patterns have been used in two different contexts: [SJBG08] proposes usability supporting software architectural patterns (USAPs) that provide developers with useful guidance for producing a software architecture design that supports usability (called these architectural patterns). Tidwell [Tid05] uses patterns from a user interface design perspective, defining solutions to common user interface design problems, without explicit consideration of the software architecture (called these interaction patterns). Harrison makes use of interaction styles to describe design and architectural patterns to characterize the properties of user interfaces [GH04]. In any case these patterns have typically been used in a forward engineering context.

The use of re-engineering approaches has been explored by several authors in order to derive new interactive systems. The re-engineering approach includes three phases: reverse engineering, transformation and forward engineering.

Application of patterns-based re-engineering techniques could be used to implement the interactive systems adaptation process. One of the most important features of patterns, which justifies its use here, is that they are platform and implementation independent solutions. Pattern-based approach may support user interface plasticity [CC08] and generally help the maintenance and migration of GUI code.

The main goal could be to develop patterns-based techniques and tools to demonstrate that source code may be automatically transformed, in a process of restructuring interactive systems, to make it more reusable, reliable and maintainable [Sut95]. The use of patterns may be an interesting technique for re-engineering because the same pattern can be implemented in several platforms.

Bibliography

- [ABD⁺89] G. Abowd, J. Bowen, A. Dix, M. Harrison, and R. Took. User interface languages: a survey of existing methods. Technical report, Programming Research Group, Oxford University, 1989.
- [ACRPM07] João C. P. Faria Ana C. R. Paiva and Pedro M. C. Mendes. Reverse engineered formal models for GUI testing. *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, July 2007.
- [ASD04] O. Al-Shara and A. Dix. Graphical user interface development enhancer (guide). 2004.
- [AV05] Tiago Alves and Joost Visser. Metrication of sdf grammars. Technical Report DI-PURE-05.05.01, Departamento de Informática, Universidade do Minho, 2005.
- [BA95] Peter Bumbulis and P.S C. Alencar. A framework for prototyping and mechanically verifying a class of user interfaces. *IEEE*, 1995.
- [BC96] R.J. Butterworth and D.J. Cooke. Using temporal logic in the specification of reactive and interactive systems. In *Proceedings of the*

- BCS-FACS WorkShop on Formal Aspects of the Human Computer Interface*, Sheffield Hallam University, September 1996.
- [Bel01] Fevzi Belli. Finite state testing and analysis of graphical user interfaces. In *Proceedings of the 12th International Symposium on Software Reliability Engineering, ISSRE 2001*, pages 34–42. IEEE, November 2001.
- [Ber01] B. Berard. *Systems and Software Verification*. Springer edition, 2001.
- [Ber05] Pavel Berkhin. A survey on pagerank computing. *Internet Mathematics*, 2:73–120, 2005.
- [Bum96] Peter Bumbulis. *Combining Formal Techniques and Prototyping in User Interface Construction and Verification*. PhD thesis, University of Waterloo, 1996.
- [BY93] Ann E. Blandford and Richard M. Young. Developing runnable user models: Separating the problem solving techniques from the domain knowledge. In J. Alty, D. Diaper, and S. Guest, editors, *People and Computers VIII — Proceedings of HCI'93*, pages 111–122, Cambridge, 1993. Cambridge University Press.
- [Cam99] J. C. Campos. *Automated Deduction and Usability Reasoning*. PhD thesis, Department of Computer Science, University of York, 1999.
- [Cam04] José C. Campos. The modelling gap between software engineering and human-computer interaction. In Rick Kazman, Len Bass, and

- Bonnie John, editors, *ICSE 2004 Workshop: Bridging the Gaps II*, pages 54–61. The IEE, 2004.
- [CC08] Joëlle Coutaz and Gaëlle Calvary. HCI and software engineering: Designing for user interface plasticity. In *The Human Computer Interaction Handbook*, chapter 56, pages 1107–1125. 2008.
- [CCT⁺03] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15:289–308, 2003.
- [CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP, ACM SIGPLAN, 2000*, 2000.
- [CH01] José C. Campos and Michael D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3-4):275–310, August 2001.
- [CH09] J. C. Campos and M. D. Harrison. Interaction engineering using the IVY tool. In *ACM Symposium on Engineering Interactive Computing Systems (EICS 2009)*, pages 35–44, New York, NY, USA, 2009. ACM.
- [Chi93] Elliot J. Chikofsky. Business reengineering and software maintenance. In *ICSM*, page 100, 1993.

- [Cle98] T. Clement. The formal development of a windows interface. In *Proceeding of the 3rd BCS-FACS Northern Formal Methods Workshop*, 1998.
- [CMP04] A. Campi, E. Martinez, and P.S. Pietro. Experiences with a formal method for design and automatic checking of user interfaces. In *Proceedings of the Position paper in IUI/CADUI'2004 Workshop on Making Model-Based UI Design Practical: usable and open methods and tools*, January 2004.
- [CS01] 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.
- [DBDM98] D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Syndetic modelling. *Human-Computer Interaction*, 13(4):337–393, 1998.
- [dDR96] 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.
- [DFAB03] Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

- [DH93] David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.
- [DJ05] Gabriel Dos Reis and Jaakko Järvi. What is generic programming? In *Proceedings of the First International Workshop of Library-Centric Software Design (LCSD '05). An OOPSLA '05 workshop*, October 2005.
- [Doh98] Gavin John Doherty. *A Pragmatic Approach to the Formal Specification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1998.
- [dS02] Paulo Pinheiro da Silva. *Object Modelling of Interactive Systems: The UMLi Approach*. PhD thesis, Department of Computer Science, University of Manchester, United Kingdom, 2002.
- [dSGD98] Bruno d'Ausbourg, Christel Seguin, and Pierre Rochk Guy Durrieu. Helping the automated validation process of user interfaces systems. *IEEE*, 1998.
- [EGK⁺01] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz - an open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.
- [ESS03] P. Iglinski E. Stroulia, M. El-ramly and P. Sorenson. User interface reverse engineering in support of interface migration to the web. *Automated Software Engineering*, 2003.

- [FMD97] Bob Fields, Nick Merriam, and Andy Dearden. DMVIS: Design, modelling and validation of interactive systems. In M. D. Harrison and J. C. Torres, editors, *Design, Specification and Verification of Interactive Systems '97*, Springer Computer Science, pages 29–44. Springer-Verlag/Wien, June 1997.
- [GH04] Stephen W. Gilroy and Michael D. Harrison. Using interaction style to match the ubiquitous user interface to the device-to-hand. In *EHCI/DS-VIS*, pages 325–345, 2004.
- [GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [Har80] D. Harel. Statecharts: a visual formalism for complex systems. In *Science of Computer Programming, vol. 8(Eds.)*, pages 231 – 274, 1980.
- [HF94] Dan Heller and Paula M. Ferguson. *Motif Programming Manual*, volume 6A of *X Window System Series*. O'Reilly & Associates, Inc., second edition, 1994.
- [HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering, ICSE '92*, pages 392–411, New York, NY, USA, 1992. ACM.

- [HT90] M. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction*. Cambridge Series on Human-Computer Interaction. Cambridge University Press, 1990.
- [HT07] Robert Hanson and Adam Tacy. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA, 2007.
- [IH01] Melody Y. Ivory and Marti A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM COMPUTING SURVEYS*, 33:470–516, 2001.
- [ISO99] ISO/IEC. Software products evaluation, 1999. DIS 14598-1.
- [Jac83] Robert J. K. Jacob. Using formal specifications in the design of a human-computer interface. *Communications of the ACM*, 26(4):259–264, 1983.
- [JBS78] Harold W. Lawson Jr., Miquel Bertran, and Javier Sanagustin. The formal definition of human/machine communications. *Softw., Pract. Exper.*, 8(1):51–58, 1978.
- [JHA⁺99] S. P. Jones, J. Hughes, L. Augustsson, et al. Report on the programming language haskell 98. Technical report, Yale University, February 1999.
- [JNeC03] Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors. *Interactive Systems. Design, Specification, and Verification*,

- 10th International Workshop, DSV-IS 2003, Funchal, Madeira Island, Portugal, June 11-13, 2003, Revised Papers*, volume 2844 of *Lecture Notes in Computer Science*. Springer, 2003.
- [LEW⁺02] Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java Swing, 2nd Edition*. O Reilly, 2002.
- [LH05] K. Loer and M.D. Harrison. Analysing user confusion in context aware mobile applications. In M.F. Constabile and F. Paternò, editors, *PINTERACT 2005*, volume 3585 of *Lecture Notes in Computer Science*, pages 184–197, New York, NY, USA, 2005. Springer.
- [LHRM07] Ping Li, Toan Huynh, Marek Reformat, and James Miller. A practical approach to testing GUI systems. *Empirical Softw. Eng.*, 12(4):331–357, 2007.
- [LPWR90] Clayton Lewis, Peter Polson, Cathleen Wharton, and John Rie-
man. 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.
- [Luc01] Andrea De Lucia. Program slicing: Methods and applications. *IEEE workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001.
- [LV03] R. Lammel and J. Visser. A STRAFUNSKI application letter. Technical report, CWI, Vrije Universiteit, Software Improvement Group, Kruislaan, Amsterdam, 2003.

- [LVM⁺04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. Usixml: A user interface description language for context-sensitive user interfaces. In *Workshop Developing User Interfaces with XML: Advances on User Interface Description Languages, Advanced Visual Interfaces 2004, Gallipoli, Italy*, 2004.
- [Mac10] Helder Nuno Ribeiro Macedo. A strategic-based weaver for aspect-matlab design and implementation, 2010. Departamento de Informática, Universidade do Minho.
- [MBN03] 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.
- [McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [Mel96] Moore Melody. A survey of representations for recovering user interface specifications for reengineering. Technical report, Institute of Technology, Atlanta GA 30332-0280, june 1996.
- [Mem01] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. PhD thesis, Department of Computer Science, University of PittsBurgh, july 2001.

- [MGG⁺95] E. Merlo, P. Gagne, J. F. Girard, K. Kontogiannis, L.J. Hendren, P. Panangaden, and R. Mori. Reengineering user interfaces. *IEEE Software*, 12(1), 64-73, 1995.
- [MJS⁺00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA, 2000. ACM.
- [Moo96] 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.
- [MSP01] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267, New York, NY, USA, 2001. ACM Press.
- [MT06] Pedro J. Molina and Hallvard Traetteberg. An approach to refining specifications towards implementation. In *Computer-Aided Design of User Interfaces IV*, pages 211–222. Springer Netherlands, 2006.
- [MTWH04] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P.E. Heimdahl. Proving the shalls early validation of requirements through formal methods. 2004.

- [Mye91] Brad A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. 1991.
- [Nie93] J. Nielsen. *Usability Engineering*. Academic Press, San Diego, CA, 1993.
- [NM90] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90 Proceedings*, pages 249–256, New York, April 1990. ACM Press.
- [Pal94] P. Palanque. Petri net design of user-driven interfaces using interactive cooperative objects formalism. In *Proceeding of the Design, Specification and Verification of Interactive Systems - DSV-IS'94*, 1994.
- [Pat95] Fabio D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995. Available as Technical Report YCST 96/03.
- [Pat00] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, 2000.
- [PC05] W. L. Pearn and W. C. Chiu. Approximate solutions for the maximum benefit chinese postman problem. *Intern. J. Syst. Sci.*, 36(13):815–822, 2005.
- [PFM07] Ana C. R. Paiva, João C. P. Faria, and Pedro M. C. Mendes, editors. *Reverse Engineered Formal Models for GUI Testing, 10th Interna-*

- tional Workshop on Formal Methods for Industrial Critical Systems*, 2007. Berlin, Germany.
- [PP98] Philippe Palanque and Fabio Paternò, editors. *Formal Methods in Human-Computer Interaction*. Formal Approaches to Computing and Information Technology series. Springer-Verlag, London, 1998.
- [Pre98] Cambridge University Press, editor. *J. Fitzgerald and P.G. Larsen. Modelling Systems: Practical Tools and Techniques in Software Development*, 1998.
- [RFM91] Mark Ryan, José Fiadeiro, and Tom Maibaum. Sharing actions and attributes in modal action logic. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 569–593. Springer-Verlag, 1991.
- [SAOB02] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12:43–60, 2002.
- [SC494] ISO/TC159 Sub-Committee SC4. Draft International ISO DIS 9241-11 Standard. International Organization for Standardization, September 1994.
- [SCS] João Carlos Silva, José Creissac Campos, and João Saraiva. The GUISurfer tool: towards a language independent approach to reverse engineering gui code. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems, Berlin*.

- [SCS06a] João Carlos Silva, José Creissac Campos, and João Saraiva. Combining formal methods and functional strategies regarding the reverse engineering of interactive applications. In *Interactive Systems, Design, Specifications and Verification, Lecture Notes in Computer Science. DSV-IS 2006, the XIII International Workshop on Design, Specification and Verification of Interactive System, Dublin, Ireland*, pages 137–150. Springer Berlin / Heidelberg, July 2006.
- [SCS06b] João Carlos Silva, José Creissac Campos, and João Saraiva. Engenharia reversa de sistemas interactivos desenvolvidos em Java/Swing. *Interacção 2006, Segunda Conferência Nacional em Interacção Pessoa-Máquina, Universidade do Minho*, October 2006.
- [SCS06c] João Carlos Silva, José Creissac Campos, and João Saraiva. Models for the reverse engineering of Java/Swing applications. *ATEM 2006, 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering, Genova, Italy*, October 2006.
- [SCS09] João Carlos Silva, José Creissac Campos, and João Saraiva. A generic library for GUI reasoning and testing. *24th Annual ACM Symposium on Applied Computing, USA*, March 2009.
- [SCS10a] João Carlos Silva, José Creissac Campos, and João Saraiva. GUI behavior from source code analysis. *Interacção 2010, Quarta Conferência Nacional em Interacção Humano-Computador, Universidade de Aveiro*, October 2010.

- [SCS10b] João Carlos Silva, José Creissac Campos, and João Saraiva. GUI inspection from source code analysis. *OpenCert-2010, 4th International Workshop on Foundations and Techniques for Open Source Software Certification, Pisa, Italy*, September 2010.
- [Sea09] Shu Yan Shan and et al. Fast centrality approximation in modular networks, 2009.
- [Shn82a] Ben Shneiderman. Multi-party grammars. *IEEE Transactions on Systems, Man, and Cybernetics*, 1982.
- [Shn82b] Ben Shneiderman. Multi-party grammars and related features for defining interactive systems. *IEEE Systems, Man, and Cybernetics SMC-12*, pages 148–154, 1982.
- [Sil10] Carlos Eduardo Silva. Reverse engineering of rich internet applications, 2010. Master thesis, Universidade do Minho.
- [SJBG08] Pia Stoll, Bonnie E. John, Len Bass, and Elspeth Golden. Preparing usability supporting architectural patterns for industrial use, 2008. Computer science, Datavetenskap, Malardalen University, School of Innovation, Design and Engineering.
- [Ski90] S. Skiena. Implementing discrete mathematics: Combinatorics and graph theory with Mathematica. *Addison-Wesley*, 1990.
- [SL89] Deepinder P. Sidhu and Ting-kau Leung. Formal methods for protocol testing: A detailed study. *IEEE Trans. Softw. Eng.*, 15(4):413–426, 1989.

- [SRF⁺10] J. L. Silva, O. R. Ribeiro, J. M. Fernandes, J. C. Campos, and M. D. Harrison. The apex framework: prototyping of ubiquitous environments based on petri nets. In J. Gulliksen R. Bernhaupt, P. Forbrig and M.K. Lárusdóttir, editors, *Human-Centred Software Engineering*, volume 6409 of *Lecture Notes in Computer Science*, pages 6–21. Springer, 2010.
- [SS97] 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.
- [Sut95] A. G. Sutcliffe. *Human-Computer Interface Design*. 1995. MacMillan, 2nd edition.
- [Sys01] T. Systa. Dynamic reverse engineering of Java software. Technical report, University of Tampere, Finland, 2001.
- [Tau90] M. J. Tauber. Etag : Extended task action grammar, a language for the description of the user’s task language. *3rd IFIP TC 13 Conference On Human-Computer Interaction Interact*, 1990.
- [TG08] Harold Thimbleby and Jeremy Gow. Applying graph theory to interaction design. pages 501–519, 2008.
- [TH90] Harold. Thimbleby and M. D. Harrison. *Formal methods in human-computer interaction / edited by Michael Harrison and Harold Thimbleby*. Cambridge University Press, Cambridge ; New York :, 1990.

- [Thi03] Harold Thimbleby. The directed chinese postman problem. *In journal of Software - Practice and Experience*, 2003.
- [Tho76] J. McCabe Thomas. A complexity measure. *Intern. J. Syst. Sci.*, 2(4):308, 1976.
- [Tid05] Jenifer Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. 2005. O'Reilly Media, Inc.
- [Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, september 1995.
- [Ura92] H. Ural. Formal methods for test sequence generation. In *Computer Comm.*, pages 311–325, 1992.
- [VCG⁺08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Formal methods and testing. chapter Model-based testing of object-oriented reactive systems with spec explorer, pages 39–76. Springer-Verlag, Berlin, Heidelberg, 2008.
- [Vis03a] 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).
- [Vis03b] Joost Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, February 2003.

- [VS04] Joost Visser and João Saraiva. Tutorial on strategic programming across programming paradigms. In *8th Brazilian Symposium on Programming Languages*, Niteroi, Brazil, May 2004.
- [Was85] Anthony I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Trans. Softw. Eng.*, 11(8):699–713, 1985.
- [YGS89] Richard M. Young, T. R. G. Green, and Tony Simon. Programmable user models for predictive evaluation of interface designs. In K. Bice and C. Lewis, editors, *CHI'89 Proceedings*, pages 15–19. ACM Press, NY, May 1989.
- [YY07] Young Sik Yoon and Wan Chul Yoon. Development of quantitative metrics to support UI designer decision-making in the design process. In *Human-Computer Interaction. Interaction Design and Usability*, pages 316–324. Springer Berlin / Heidelberg, 2007.

Appendix A

GUISURFER GUI Meta-Model Specification

This Appendix shows the complete *guimodel* abstract data type.

```
GuiTypes
  Data.Map
  EventRef = String
  CondRef = String
  WindowName = String
  ExpRef = Int
  GuiModel = Map (EventRef,CondRef) [ExpRef]
  Pres = Map ExpRef (EventRef,Bool)
  End = [ExpRef]
  Close = [ExpRef]
  Window = WindowName
  NewWindow = Map ExpRef WindowName
  State = Map EventRef Bool
  StateRef = String
  N = N ([CondRef], [EventRef])
      (Eq,Show)
  Type = String
  AstList = [[String]]
  InitPos = Int
  EndPos = Int
  SourcePosList = [(InitPos,EndPos)]
  Exp = (Type,AstList,InitPos,EndPos,SourcePosList)
  Exps = Map ExpRef Exp
  Events = Map EventRef Exp
  Conds = Map CondRef [Exp]
```


Appendix B

Agenda's GUISURFER Script Analysis

This appendix presents the script used to reverse engineer and reason the *Agenda* application.

B.1 Meta-model, Interactor and State Machine Extraction

```
ghc --make FileParser.hs -o FileParser -fglasgow-exts
ghc --make AstAnalyser.hs -o AstAnalyser -fglasgow-exts
ghc --make Graph.hs -o Graph -fglasgow-exts
```

```
FileParser Login.java
AstAnalyser "Login.java.ast" "main" "JButton, setEnabled, exit,
showMessageDialog, dispose, ContactEditor, Find, Login, MainForm"
Graph eventsFromInitState.gui initState.gui 0 "ContactEditor,
Find, Login, MainForm" windowName.gui "Login" "ClientDBjava" 1
dot -Tpng graph.dot -o graphClientDBjavaLogin.png
```

```
FileParser ContactEditor.java
AstAnalyser "ContactEditor.java.ast" "main" "JButton,
setEnabled, exit, showMessageDialog, dispose, ContactEditor,
Find, Login, MainForm"
Graph eventsFromInitState.gui initState.gui 0 "ContactEditor,
Find, Login, MainForm" windowName.gui "Login" "ClientDBjava" 1
dot -Tpng graph.dot -o graphClientDBjavaContactEditor.png
```

```
FileParser MainForm.java
AstAnalyser "MainForm.java.ast" "main" "JButton, setEnabled,
exit, showMessageDialog, dispose, ContactEditor, Find, Login,
MainForm"
Graph eventsFromInitState.gui initState.gui 0 "ContactEditor,
Find,Login,MainForm" windowName.gui "Login" "ClientDBjava" 1
dot -Tpng graph.dot -o graphClientDBjavaMainForm.png
```

```
FileParser Find.java
AstAnalyser "Find.java.ast" "main" "JButton, setEnabled,
exit, showMessageDialog, dispose, ContactEditor, Find,
Login, MainForm"
Graph eventsFromInitState.gui initState.gui 0 "ContactEditor,
Find,Login,MainForm" windowName.gui "Login" "ClientDBjava" 1
dot -Tpng graph.dot -o graphClientDBjavaFind.png
```

Appendix C

Agenda's Windows Behaviour Specification

The following specifications are generated automatically by GUISURFER from script in appendix B. These are written in *Haskell* programming language and define the behaviour of each *Agenda* application's window.

C.1 Login Window

```
-- Generated automatically by GuiSurfer
  GuiModel
  Data.Map
  EventRef = String
  CondRef = String
  WindowName = String
  ExpRef = Int
  GuiModel = Map (EventRef,CondRef) [ExpRef]
  Pres = Map ExpRef (EventRef,Bool)
  End = [ExpRef]
  Close = [ExpRef]
  NewWindow = Map ExpRef WindowName

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")]

close :: Close
close = [3]

```

C.2 MainForm Window

```

-- Generated automatically by GuiSurfer
  GuiModel
  Data.Map
  EventRef = String
  CondRef = String
  WindowName = String
  ExpRef = Int
  GuiModel = Map (EventRef,CondRef) [ExpRef]
  Pres = Map ExpRef (EventRef,Bool)
  End = [ExpRef]
  Close = [ExpRef]
  NewWindow = Map ExpRef WindowName

guimodel :: GuiModel
guimodel = fromList
          [ (("Exit","cond1"),[1]),
            ("Edit","cond2"),[2]),
            ("Edit","cond3"),[3]),
            ("Find","cond4"),[4]),
            ("Find","cond5"),[5]),
            ("init","condInit1"),[6,7,8,9,10,11,12,13,14,15]] ]

```

```

pres :: Pres
pres = fromList
      [ (10,("Exit",True)),
        (11,("Edit",True)),
        (12,("Find",True)) ]

end :: End
end = [1]

newWindow :: NewWindow
newWindow = fromList
           [ (2,"ContactEditor"),
             (4,"Find"),
             (6,"MainForm") ]

close :: Close
close = []

```

C.3 Find Window

```

-- Generated automatically by GuiSurfer
  GuiModel
  Data.Map
  EventRef = String
  CondRef = String
  WindowName = String
  ExpRef = Int
  GuiModel = Map (EventRef,CondRef) [ExpRef]
  Pres = Map ExpRef (EventRef,Bool)
  End = [ExpRef]
  Close = [ExpRef]
  NewWindow = Map ExpRef WindowName

guimodel :: GuiModel
guimodel = fromList
          [ (("Search","cond1"),[]),
            ("Cancel","cond2"),[1]),
            ("Show","cond3"),[]),
            ("init","condInit1"),[2,3,4,5,6,7,8]] ]

pres :: Pres
pres = fromList

```

```

        [ (6,("Search",True)),
          (7,("Cancel",True)),
          (8,("Show",True)) ]

end :: End
end = []

newWindow :: NewWindow
newWindow = fromList
            [ (2,"Find") ]

close :: Close
close = [1]

```

C.4 ContactEditor Window

```

-- Generated automatically by GuiSurfer
  GuiModel
  Data.Map
  EventRef = String
  CondRef = String
  WindowName = String
  ExpRef = Int
  GuiModel = Map (EventRef,CondRef) [ExpRef]
  Pres = Map ExpRef (EventRef,Bool)
  End = [ExpRef]
  Close = [ExpRef]
  NewWindow = Map ExpRef WindowName

guimodel :: GuiModel
guimodel = fromList
          [ (("Add","cond1"),[1,2]),
            ("Edit","cond2"),[],
            ("Remove","cond3"),[3,4]),
            ("Remove","cond4"),[5,6]),
            ("Cancel","cond5"),[7]),
            ("Ok","cond6"),[8]),
            ("init","condInit1"),[9,10,11,12,13,14,15,16,17]),
            ("init","condInit2"),[18,19,11,12,13,14,15,16,17]) ]

pres :: Pres
pres = fromList

```

```
    [ (1,("Edit",True)),
      (2,("Remove",True)),
      (5,("Edit",True)),
      (6,("Remove",True)),
      (9,("Edit",True)),
      (10,("Remove",True)),
      (17,("Add",True)),
      (3,("Edit",False)),
      (4,("Remove",False)),
      (18,("Edit",False)),
      (19,("Remove",False)) ]

end :: End
end = []

newWindow :: NewWindow
newWindow = fromList
    [ (11,"ContactEditor") ]

close :: Close
close = [7,8]
```


Appendix D

Agenda's Windows States Extraction

From script described in appendix B, the GUI SURFER tool generates automatically all windows states. This appendix presents for each *Agenda's* window the generated states.

D.1 *Login Window*

```
-- Generated automatically by GuiSurfer
   GuiModelStatesClientDBjavaLogin
   Data.Map
   GuiTypes

statesLogin :: (Map (StateRef,EventRef,CondRef,[ExpRef])
StateRef, Map StateRef State)
statesLogin = (fromList [
  (("state0","init","condInit1",[5,6,7,8,9]),"state1"),
  (("state1","Cancel","cond1",[1]),"state0"),
  (("state1","Ok","cond2",[2,3]),"state0"),
  (("state1","Ok","cond3",[4]),"state1")],
fromList [("state0",fromList []),
("state1",fromList [("Cancel",True),("Ok",True)])])
```

D.2 MainForm Window

```
-- Generated automatically by GuiSurfer
  GuiModelStatesClientDBjavaMainForm
  Data.Map
  GuiTypes

statesMainForm :: (Map (StateRef,EventRef,CondRef,[ExpRef])
  StateRef, Map StateRef State)
statesMainForm = (fromList [
  (("state0","init","condInit1",[6,7,8,9,10,11,12,13,14,15]),"state1"),
  (("state1","Edit","cond2",[2]),"state1"),
  (("state1","Edit","cond3",[3]),"state1"),
  (("state1","Exit","cond1",[1]),"state0"),
  (("state1","Find","cond4",[4]),"state1"),
  (("state1","Find","cond5",[5]),"state1")],
  fromList [("state0",fromList []),
  ("state1",fromList [("Edit",True),("Exit",True),("Find",True)])])
```

D.3 Find Window

```
-- Generated automatically by GuiSurfer
  GuiModelStatesFind
  Data.Map
  GuiTypes

statesFind :: (Map (StateRef,EventRef,CondRef,[ExpRef])
  StateRef, Map StateRef State)
statesFind = (fromList [
  (("state0","init","condInit1",[2,3,4,5,6,7,8]),"state1"),
  (("state1","Cancel","cond2",[1]),"state0"),
  (("state1","Search","cond1",[]),"state1"),
  (("state1","Show","cond3",[]),"state1")],
  fromList [
  ("state0",fromList []),
  ("state1",fromList [("Cancel",True),("Search",True),
  ("Show",True)])])
```

D.4 ContactEditor Window

```
-- Generated automatically by GuiSurfer
  GuiModelStatesClientDBjavaContactEditor
  Data.Map
```

GuiTypes

```

statesContactEditor ::
  (Map (StateRef,EventRef,CondRef,[ExpRef]) StateRef,
   Map StateRef State)
statesContactEditor = (fromList [
  (("state0","init","condInit1",
   [9,10,11,12,13,14,15,16,17]),"state2"),
  (("state0","init","condInit2",
   [18,19,11,12,13,14,15,16,17]),"state1"),
  (("state1","Add","cond1",[1,2]),"state2"),
  (("state1","Cancel","cond5",[7]),"state0"),
  (("state1","Ok","cond6",[8]),"state0"),
  (("state2","Add","cond1",[1,2]),"state2"),
  (("state2","Cancel","cond5",[7]),"state0"),
  (("state2","Edit","cond2",[]),"state2"),
  (("state2","Ok","cond6",[8]),"state0"),
  (("state2","Remove","cond3",[3,4]),"state1"),
  (("state2","Remove","cond4",[5,6]),"state2")],
  fromList [("state0",fromList []),
  ("state1",fromList [("Add",True),("Cancel",True),("Edit",False),
  ("Ok",True),("Remove",False)]),
  ("state2",fromList [("Add",True),("Cancel",True),("Edit",True),
  ("Ok",True),("Remove",True)])])

```


Appendix E

Agenda's Windows Events Sequences Extraction

From script described in appendix B, the GUI SURFER tool generates automatically test cases. This Section presents for each *Agenda's* window the obtained results (considering only sequences with one or two events).

E.1 Login Window

-- Generated automatically by GuiSurfer

GuiModelWaysLogin
GuiTypes

```
waysLogin :: [M]
waysLogin = [N ("condInit1",["init"]),
N ("condInit1","cond1",["init","Cancel"]),
N ("condInit1","cond2",["init","Ok"]),
N ("condInit1","cond3",["init","Ok"]),
N ("condInit1","cond3","cond1",["init","Ok","Cancel"]),
N ("condInit1","cond3","cond2",["init","Ok","Ok"]),
N ("condInit1","cond3","cond3",["init","Ok","Ok"])]
```

E.2 MainForm Window

-- Generated automatically by GuiSurfer

GuiModelWaysMainForm

GuiTypes

```

waysMainForm :: [M]
waysMainForm = [N(["condInit1"],["init"]),
N(["condInit1","cond2"],["init","Edit"]),
N(["condInit1","cond3"],["init","Edit"]),
N(["condInit1","cond2","cond2"],["init","Edit","Edit"]),
N(["condInit1","cond2","cond3"],["init","Edit","Edit"]),
N(["condInit1","cond2","cond1"],["init","Edit","Exit"]),
N(["condInit1","cond2","cond4"],["init","Edit","Find"]),
N(["condInit1","cond2","cond5"],["init","Edit","Find"]),
N(["condInit1","cond3","cond2"],["init","Edit","Edit"]),
N(["condInit1","cond3","cond3"],["init","Edit","Edit"]),
N(["condInit1","cond3","cond1"],["init","Edit","Exit"]),
N(["condInit1","cond3","cond4"],["init","Edit","Find"]),
N(["condInit1","cond3","cond5"],["init","Edit","Find"]),
N(["condInit1","cond1"],["init","Exit"]),
N(["condInit1","cond4"],["init","Find"]),
N(["condInit1","cond5"],["init","Find"]),
N(["condInit1","cond4","cond2"],["init","Find","Edit"]),
N(["condInit1","cond4","cond3"],["init","Find","Edit"]),
N(["condInit1","cond4","cond1"],["init","Find","Exit"]),
N(["condInit1","cond4","cond4"],["init","Find","Find"]),
N(["condInit1","cond4","cond5"],["init","Find","Find"]),
N(["condInit1","cond5","cond2"],["init","Find","Edit"]),
N(["condInit1","cond5","cond3"],["init","Find","Edit"]),
N(["condInit1","cond5","cond1"],["init","Find","Exit"]),
N(["condInit1","cond5","cond4"],["init","Find","Find"]),
N(["condInit1","cond5","cond5"],["init","Find","Find"])]

```

E.3 Find Window

-- Generated automatically by GuiSurfer

GuiModelWaysFind

GuiTypes

```

waysFind :: [M]
waysFind = [N(["condInit1"],["init"]),
N(["condInit1","cond2"],["init","Cancel"]),
N(["condInit1","cond1"],["init","Search"]),
N(["condInit1","cond1","cond2"],["init","Search","Cancel"]),
N(["condInit1","cond1","cond1"],["init","Search","Search"]),
N(["condInit1","cond1","cond3"],["init","Search","Show"]),
N(["condInit1","cond3"],["init","Show"])]

```

```

N(["condInit1","cond3","cond2"],["init","Show","Cancel"]),
N(["condInit1","cond3","cond1"],["init","Show","Search"]),
N(["condInit1","cond3","cond3"],["init","Show","Show"])

```

E.4 ContactEditor Window

-- Generated automatically by GuiSurfer

GuiModelWaysContactEditor

GuiTypes

```

waysContactEditor :: [M]
waysContactEditor = [N(["condInit1"],["init"]),
N(["condInit2"],["init"]),
N(["condInit1","cond1"],["init","Add"]),
N(["condInit1","cond1","cond1"],["init","Add","Add"]),
N(["condInit1","cond1","cond5"],["init","Add","Cancel"]),
N(["condInit1","cond1","cond2"],["init","Add","Edit"]),
N(["condInit1","cond1","cond6"],["init","Add","Ok"]),
N(["condInit1","cond1","cond3"],["init","Add","Remove"]),
N(["condInit1","cond1","cond4"],["init","Add","Remove"]),
N(["condInit1","cond5"],["init","Cancel"]),
N(["condInit1","cond2"],["init","Edit"]),
N(["condInit1","cond2","cond1"],["init","Edit","Add"]),
N(["condInit1","cond2","cond5"],["init","Edit","Cancel"]),
N(["condInit1","cond2","cond2"],["init","Edit","Edit"]),
N(["condInit1","cond2","cond6"],["init","Edit","Ok"]),
N(["condInit1","cond2","cond3"],["init","Edit","Remove"]),
N(["condInit1","cond2","cond4"],["init","Edit","Remove"]),
N(["condInit1","cond6"],["init","Ok"]),
N(["condInit1","cond3"],["init","Remove"]),
N(["condInit1","cond4"],["init","Remove"]),
N(["condInit1","cond3","cond1"],["init","Remove","Add"]),
N(["condInit1","cond3","cond5"],["init","Remove","Cancel"]),
N(["condInit1","cond3","cond6"],["init","Remove","Ok"]),
N(["condInit1","cond4","cond1"],["init","Remove","Add"]),
N(["condInit1","cond4","cond5"],["init","Remove","Cancel"]),
N(["condInit1","cond4","cond2"],["init","Remove","Edit"]),
N(["condInit1","cond4","cond6"],["init","Remove","Ok"]),
N(["condInit1","cond4","cond3"],["init","Remove","Remove"]),
N(["condInit1","cond4","cond4"],["init","Remove","Remove"]),
N(["condInit2","cond1"],["init","Add"]),
N(["condInit2","cond1","cond1"],["init","Add","Add"]),
N(["condInit2","cond1","cond5"],["init","Add","Cancel"]),
N(["condInit2","cond1","cond2"],["init","Add","Edit"]),

```

196 APPENDIX E. AGENDA'S WINDOWS EVENTS SEQUENCES EXTRACTION

```
N (["condInit2","cond1","cond6"],["init","Add","Ok"]),  
N (["condInit2","cond1","cond3"],["init","Add","Remove"]),  
N (["condInit2","cond1","cond4"],["init","Add","Remove"]),  
N (["condInit2","cond5"],["init","Cancel"]),  
N (["condInit2","cond6"],["init","Ok"])]
```


Appendix F

Agenda Script Reasoning through *Graph-Tool*

This appendix contains the complete *Python* script (*Graph-Tool*) which generates several metrics results of the *Agenda* application.

```
#!/usr/bin/env python
import sys, os
from pylab import *
from graph_tool.all import *
g = Graph()
v_age = g.new_vertex_property("string")
e_age = g.new_edge_property("string")
Findstate1 = g.add_vertex()
v_age[Findstate1] = "Findstate1"
Findinit = g.add_vertex()
v_age[Findinit] = "Findstate0"
Findclose = g.add_vertex()
v_age[Findclose] = "Findclose"
e = g.add_edge(Findinit,Findstate1)
e_age[e] = "init/condInit1/[2,3,4,5,6,7,8]"
e = g.add_edge(Findstate1,Findclose)
e_age[e] = "Cancel/cond2/[1]"
e = g.add_edge(Findstate1,Findstate1)
e_age[e] = "Search/cond1/[]"
e = g.add_edge(Findstate1,Findstate1)
e_age[e] = "Show/cond3/[]"
ContactEditorstate1 = g.add_vertex()
v_age[ContactEditorstate1] = "ContactEditorstate1"
```

198 APPENDIX F. AGENDA SCRIPT REASONING THROUGH GRAPH-TOOL

```

ContactEditorstate2 = g.add_vertex()
v_age[ContactEditorstate2] = "ContactEditorstate2"
ContactEditorinit = g.add_vertex()
v_age[ContactEditorinit] = "ContactEditorstate0"
ContactEditorclose = g.add_vertex()
v_age[ContactEditorclose] = "ContactEditorclose"
e = g.add_edge(ContactEditorinit,ContactEditorstate2)
e_age[e] = "init/condInit1/[9,10,11,12,13,14,15,16,17]"
e = g.add_edge(ContactEditorinit,ContactEditorstate1)
e_age[e] = "init/condInit2/[18,19,11,12,13,14,15,16,17]"
e = g.add_edge(ContactEditorstate1,ContactEditorstate2)
e_age[e] = "Add/cond1/[1,2]"
e = g.add_edge(ContactEditorstate1,ContactEditorclose)
e_age[e] = "Cancel/cond5/[7]"
e = g.add_edge(ContactEditorstate1,ContactEditorclose)
e_age[e] = "Ok/cond6/[8]"
e = g.add_edge(ContactEditorstate2,ContactEditorstate2)
e_age[e] = "Add/cond1/[1,2]"
e = g.add_edge(ContactEditorstate2,ContactEditorclose)
e_age[e] = "Cancel/cond5/[7]"
e = g.add_edge(ContactEditorstate2,ContactEditorstate2)
e_age[e] = "Edit/cond2/[]"
e = g.add_edge(ContactEditorstate2,ContactEditorclose)
e_age[e] = "Ok/cond6/[8]"
e = g.add_edge(ContactEditorstate2,ContactEditorstate1)
e_age[e] = "Remove/cond3/[3,4]"
e = g.add_edge(ContactEditorstate2,ContactEditorstate2)
e_age[e] = "Remove/cond4/[5,6]"
MainFormstate1 = g.add_vertex()
v_age[MainFormstate1] = "MainFormstate1"
MainForminit = g.add_vertex()
v_age[MainForminit] = "MainFormstate0"
MainFormend = g.add_vertex()
v_age[MainFormend] = "MainFormend"
e = g.add_edge(MainForminit,MainFormstate1)
e_age[e] = "init/condInit1/[6,7,8,9,10,11,12,13,14,15]"
e = g.add_edge(MainFormstate1,MainFormstate1)
e_age[e] = "Edit/cond2/[2]"
e = g.add_edge(MainFormstate1,ContactEditorinit)
e_age[e] = "Open ContactEditor window"
e = g.add_edge(MainFormstate1,MainFormstate1)
e_age[e] = "Edit/cond3/[3]"
e = g.add_edge(MainFormstate1,MainFormend)
e_age[e] = "Exit/cond1/[1]"
e = g.add_edge(MainFormstate1,MainFormstate1)

```

```

e_age[e] = "Find/cond4/[4]"
e = g.add_edge(MainFormstate1,Findinit)
e_age[e] = "Open Find window"
e = g.add_edge(MainFormstate1,MainFormstate1)
e_age[e] = "Find/cond5/[5]"
Loginstate1 = g.add_vertex()
v_age[Loginstate1] = "Loginstate1"
Logininit = g.add_vertex()
v_age[Logininit] = "Loginstate0"
Loginend = g.add_vertex()
v_age[Loginend] = "Loginend"
Loginclose = g.add_vertex()
v_age[Loginclose] = "Loginclose"
e = g.add_edge(Logininit,Loginstate1)
e_age[e] = "init/condInit1/[5,6,7,8,9]"
e = g.add_edge(Loginstate1,Loginend)
e_age[e] = "Cancel/cond1/[1]"
e = g.add_edge(Loginstate1,Loginclose)
e_age[e] = "Ok/cond2/[2,3]"
e = g.add_edge(Loginclose,MainForminit)
e_age[e] = "Open MainForm window"
e = g.add_edge(Loginstate1,Loginstate1)
e_age[e] = "Ok/cond3/[4]"
e = g.add_edge(Findclose,MainFormstate1)
e_age[e] = "Close Find window"
e = g.add_edge(ContactEditorclose,MainFormstate1)
e_age[e] = "Close ContactEditor window"

graph_draw(g, size=(30,30), layout="dot", vcolor="white",
  ecolor="black", output="graphTool-DBclientjava.pdf",
  vprops=dict([('label', v_age)]),
  eprops=dict([('label', e_age),
    ('arrowsize', 2.0), ('arrowhead', "empty")]))

graph_draw(g, size=(30,30), layout="dot",
  vcolor="white", ecolor="black",
  output="graphTool-DBclientjavaNumbered.pdf",
  vprops=dict([('label', g.vertex_index)]),
  eprops=dict([('label', g.edge_index),
    ('arrowsize', 2.0), ('arrowhead', "empty")]))

bv, be = betweenness(g)
bel = be
bel.get_array()[:] = bel.get_array()[:]*120+1
graph_draw(g, size=(70,70), layout="dot",

```

200APPENDIX F. AGENDA SCRIPT REASONING THROUGH GRAPH-TOOL

```
    vcolor="white", ecolor="gray",
    output="graphTool-Betweenness.pdf",
    vprops=dict([('label', bv)]),
    eprops=dict([('label', be), ('arrowsize', 1.2),
                 ('arrowhead', "normal"), ('penwidth', be1)]))

pr = pagerank(g)
graph_draw(g, size=(70,70), layout="dot",
           vsize = pr, vcolor="gray", ecolor="black",
           output="graphTool-Pagerank.pdf",
           vprops=dict([('label', "")]),
           eprops=dict([('label', ""), ('arrowsize', 2.0),
                        ('arrowhead', "empty")]))

g.set_edge_filter(None)
bv, be = betweenness(g)
be.a *= 10
graph_draw(g, pin=True, size=(8,8),
           vsize=0.07, vcolor=bv,
           eprops={"penwidth":be},
           output="GraphTool-nonfiltered-bt.pdf")

pr = pagerank(g)
print "pagerank", pr.a

vb, eb = betweenness(g)
print "betweenness", vb.a

vb, eb = betweenness(g)
print "central_point_dominance",
      central_point_dominance(g, vb)

print "isomorphism", isomorphism(g, g)

tc = transitive_closure(g)
graph_draw(tc, size=(15,15), layout="dot",
           output="graphTool-TransitiveClosure.pdf")

dist = shortest_distance(g, source=g.vertex(7))
print "shortest_distance from MainForm"
print dist.get_array()

dist = shortest_distance(g, source=g.vertex(11))
print "shortest_distance from Login"
print dist.get_array()
```

```
vlist, elist = shortest_path(g, g.vertex(11), g.vertex(6))
print "shortest path vertices", [str(v) for v in vlist]
print "shortest path edges", [str(e) for e in elist]

m = adjacency(g)
print m.todense()

for v in g.vertices():
    print v, v_age[v]

for e in g.edges():
    print e, e_age[e]
```