2

GUIsurfer: A Reverse Engineering Framework for User Interface Software

José Creissac Campos[1], João Saraiva[1], Carlos Silva[1] and João Carlos Silva[2]
[1]*Departamento de Informática, Universidade do Minho*
[2]*Escola Superior de Tecnologia, Instituto Politécnico do Cávado e do Ave*
Portugal

1. Introduction

In the context of developing tool support to the automated analysis of interactive systems implementations, this chapter proposal aims to investigate 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.

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. (Dix et al., 2003)), and can support acceptance testing processes. Moreover, the manipulation of the models supports the evolution, redesign and comparison of systems.

1.2 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 (Loer & Harrison, 2005).

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 (Myers, 1991).

At the same time it is important to ensure that GUI based applications behave as expected (Memon, 2001). The correctness of the GUI is essential to the correct execution of the software (Berard, 2001). 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 (Abowd et al., 1989; SC4, 1994).

The main objective of this Chapter consists in developing tools to automatically extract models containing the GUI behaviour. We call this reverse engineering the GUI source code. Models allow designers to analyse systems and could be used to validate system requirements at reasonable cost (Miller et al., 2004). 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 Chapter is to be able to reason about 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 (Melody, 1996). 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.

1.3 Structure of the chapter

This Chapter is structured into three main parts. Part 1 (Section 2) presents the reverse engineering area relating it to the GUI modelling area. Reverse engineering techniques' state of the art, related work and additional methodologies used within this research are firstly described. Then, the Section 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.

Part 2 (Sections 3, 4, 5 and 6) presents the approach proposed in this Chapter. Section 3 presents methodologies to retargetable GUI reverse engineering. Section 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. Then, Section 5, describe the research about GUI reasoning through behavioural models of interactive applications. Section 6 describes the application of GUISURFER to a realistic third-party application.

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

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 Section, reverse engineering and user interface modelling aspects are described (Campos, 2004; Duke & Harrison, 1993). Section 2.1 provides details about the reverse engineering area. Then, the type of GUIs models to be used is discussed in Section 2.2. Finally, the last Section summarizes the Section presenting some conclusions.

2.1 Reverse engineering

Reverse engineering is useful in several tasks like documentation, maintenance, and re-engineering (E. Stroulia & Sorenson, 2003).

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 (Chen & Subramaniam, 2001; Paiva et al., 2007; Systa, 2001). 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 (Moore, 1996). Within a dynamic approach the system is executed and its external behaviour is analysed.

Program analysis, plan recognition and redocumentation are applications of reverse engineering (Müller et al., 2000). 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.

2.2 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 rational, thus documenting the decision process undertook 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. 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.

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.
- *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. (Blandford & Young, 1993). 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 (Duke et al., 1998; Young et al., 1989);
- *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;
- *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;

- *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;
- And, finally, *Platform models* define the physical devices that are intended to host the application and how they interact with each other.

This Chapter 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.

2.3 Summarizing GUI reverse engineering

This Section 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*.

With the above in mind, this Chapter 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 Chapter will focus on event-based programming toolkits for graphical user interfaces development (*Java/Swing* being a typical example).

3. GUISURFER: A reverse engineering tool

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

This Section is organized as follows: Section 3.1 describes the architecture of the GUISURFER tool. A description about the retargetability of the tool is provided in Section 3.2. Finally, Section 3.3 presents some conclusions.

3.1 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 1. Hence, if there is the need of retargeting GUISURFER into another language, ideally only the language dependent phase should be affected.

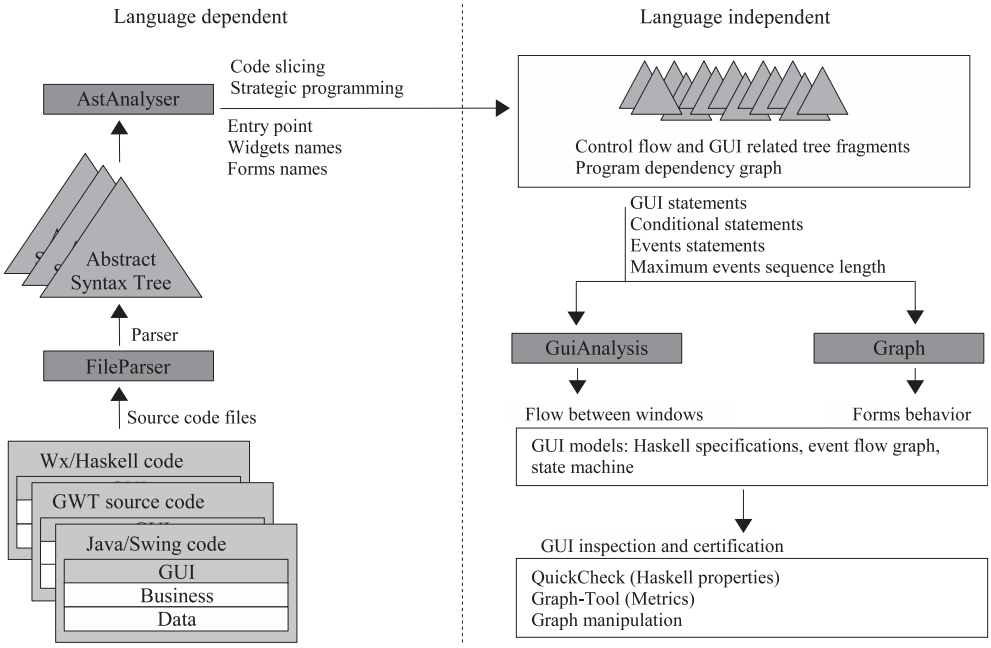


Fig. 1. GUISURFER architecture and retargetability

To support these two phases process, the GUISURFER 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.

3.1.1 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: 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.

3.1.2 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.

3.1.3 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 (Claessen & Hughes, 2000), 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 (Ellson et al., 2001). GUI reasoning is also performed through the use of *Graph-Tool*[1] for the manipulation and statistical analysis of graphs. In this particular case an analogy is considered between state machines and graphs.

3.2 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 (Hanson & Tacy, 2007). 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 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.

[1] see, http://projects.skewed.de/graph-tool/, last accessed 27 July, 2011

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

3.3 Summarizing GUISURFER approach

In this Section a reverse engineering tool was described. The GUISURFER 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 GUISURFER architecture was presented and important parameters for each GUISURFER's executable file were outlined. A particular emphasis was placed on developing a tool that is, as much as possible, language independent.

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 (Melody, 1996). 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.

4. GUI Reasoning from reverse engineering

The term GUI reasoning refers to the process of validating and verifying if interactive applications behave as expected (Berard, 2001; Campos, 1999; d'Ausbourg et al., 1998). 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 (Bumbulis & Alencar, 1995). 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) (Belli, 2001).

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 (Ivory & Hearst, 2001) 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.

The reverse engineering approach described in this Chapter allows for the extraction of GUI behavioural models from source code. This Section describes an approach to GUI reasoning from these models. To this end, the *QuickCheck Haskell* library (Claessen & Hughes, 2000), graph theory, and the *Graph-Tool*[2] are used.

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

[2] see, http://projects.skewed.de/graph-tool/, last accessed 27 July, 2011.

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 (ISO/IEC, 1999).

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 (Nielsen, 1993):

- 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. (Stamelos et al., 2002) used the Logiscope[3] 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 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 (Yoon & Yoon, 2007) 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 (Thimbleby & Gow, 2008) 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 GUISURFER. Indeed, by calculating metrics over the behavioural models produced by GUISURFER, relevant knowledge may be acquired about the dialogue induced by the interface, and, as a consequence, about how users might react to it.

[3] http://www-01.ibm.com/software/awdtools/logiscope/, last accessed May 22, 2011

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 *ContacEditor*, as shown in Figure 2.

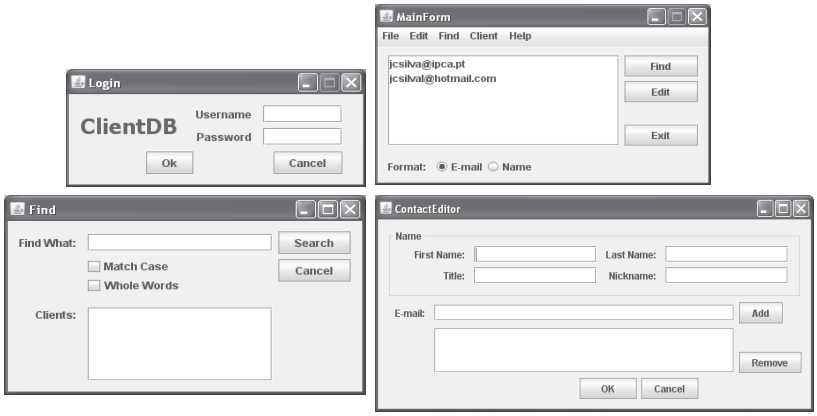


Fig. 2. A GUI application

We will use this example to present our approach to GUI reasoning. Let us discuss it in detail. The initial *Login* window (Figure 2, 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 2, 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 2, 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 2, 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.

4.1 *Graph-Tool*

Graph-Tool is an efficient python module for manipulation and statistical analysis of graphs[4]. 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.

[4] see, http://projects.skewed.de/graph-tool/, last accessed 27 July, 2011.

Now, for brevity, the graph described in Figure 3 will be considered. All vertices and edges

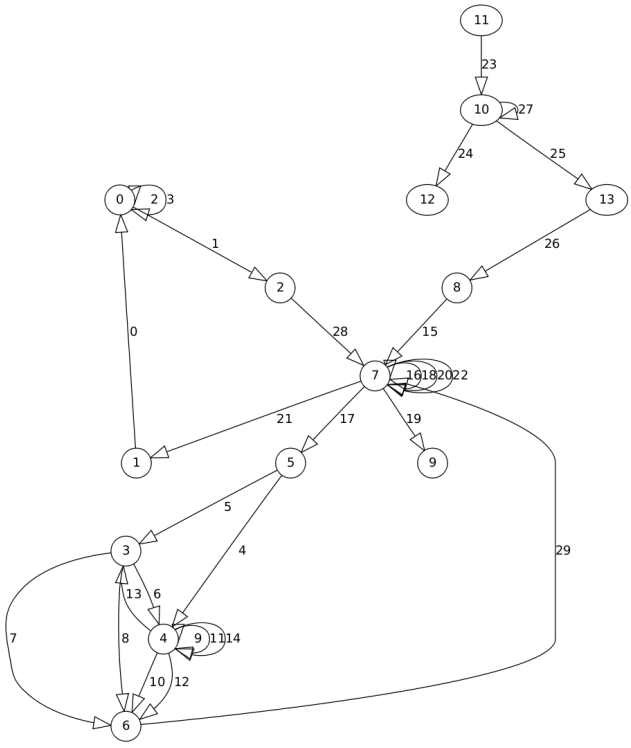


Fig. 3. *Agenda*'s behaviour graph (numbered)

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.

4.1.0.1 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 3.

```
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 indexs 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.

4.1.0.2 Pagerank

Pagerank is a distribution used to represent the probability that a person randomly clicking on links will arrive at any particular page (Berkhin, 2005). 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 4 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")]))
```

Fig. 4. *Python* command for Pagerank algorithm

Figure 5 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 to see vertices and edges description).

4.1.0.3 Betweenness

Betweenness is a centrality measure of a vertex or an edge within a graph (Shan & et al., 2009). 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 provides the *Python* command for applying this algorithm to the *Agenda*' graph model. Figure 7 displays the result. Betweenness values for vertices and edges are expressed visually. Highest betweenness edges values are represented with thicker edges. The Main window has the highest (vertices and edges values) betweenness, meaning it acts as a hub

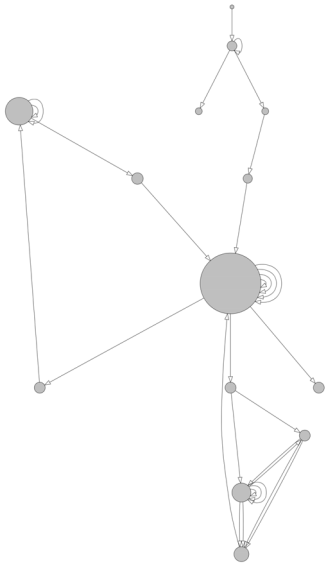


Fig. 5. *Agenda*'s pagerank results

```
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)]))
```

Fig. 6. *Python* command for Betweenness algorithm

from where different parts of the interface can be reached. Clearly it will be a central point in the interaction.

4.1.0.4 Cyclomatic complexity

Another important metric is cyclomatic complexity which aims to measures the total number of decision points in an application (Thomas, 1976). 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 $M = E\text{-}V + 2P$ (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.

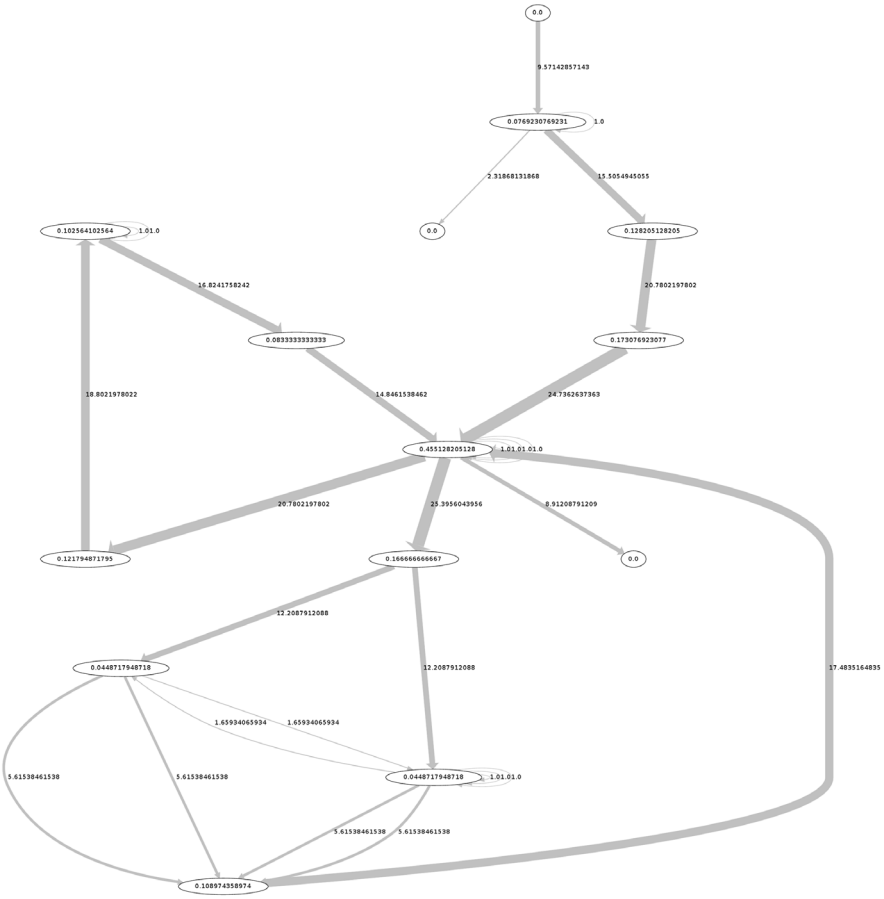


Fig. 7. *Agenda*'s betweenness values (highest betweenness values represented with thicker edges)

Considering Figure 5 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 (Thomas, 1976). 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 Chapter.

4.2 Summarizing GUI reasoning approach

In this Section 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.

5. HMS case study: A larger interactive system

In previous Sections, we have presented the GUISURFER tool and all the different techniques involved in the analysis an the reasoning of interactive applications. We have used a simple examples in order to motivate and explain our approach. In this Section, 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 Section 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 choosen interactive system is related to a Healthcare Management System (*HMS*), and can be downloaded from *Planet-source-code* website[5]. *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 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.

5.1 Bills management

This Section presents results obtained when working with the billing form provided in Figure 8. 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. Figure 9 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.

5.2 GUI Reasonning

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

Figure 10 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 Chapter. 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

Pagerank is a link analysis algorithm, that assigns a numerical weighting to 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.

[5] http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=6401&lngWId=2, last accessed May 22, 2011

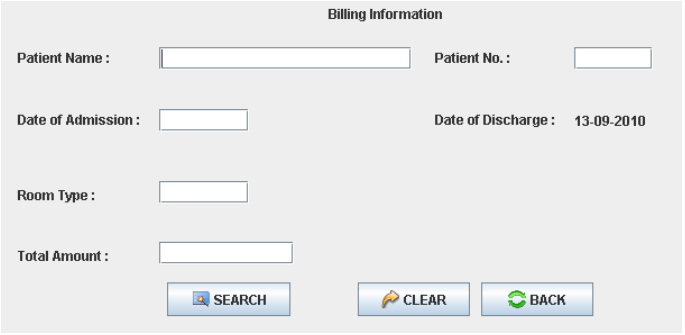


Fig. 8. *HSM*: Billing form

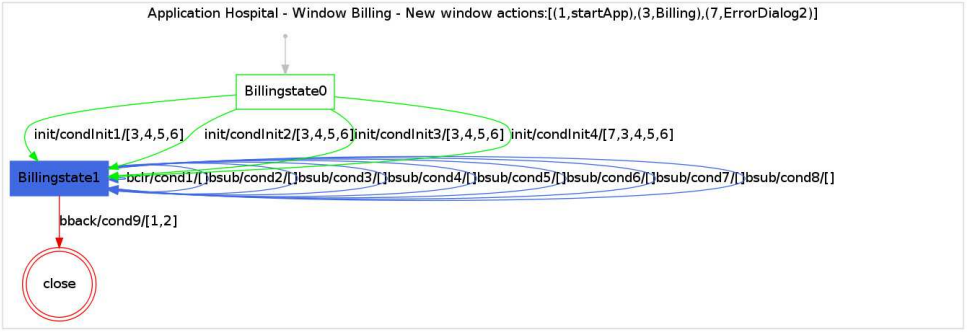


Fig. 9. *HSM*: Billing form behaviour state machine

Figure 11 provides the result obtained when applying the pagerank algorithm to graph of Figure 10. 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 12 provides the obtained result when applying the betweenness algorithm. 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. States *patStartstate0*, *patStartstate1* are the main states of the *patStart* window's state machine. Finally, *docStartstate0* and *docStartstate1* belong to *docStart* window's state machine (*docStart* is the main doctor window).

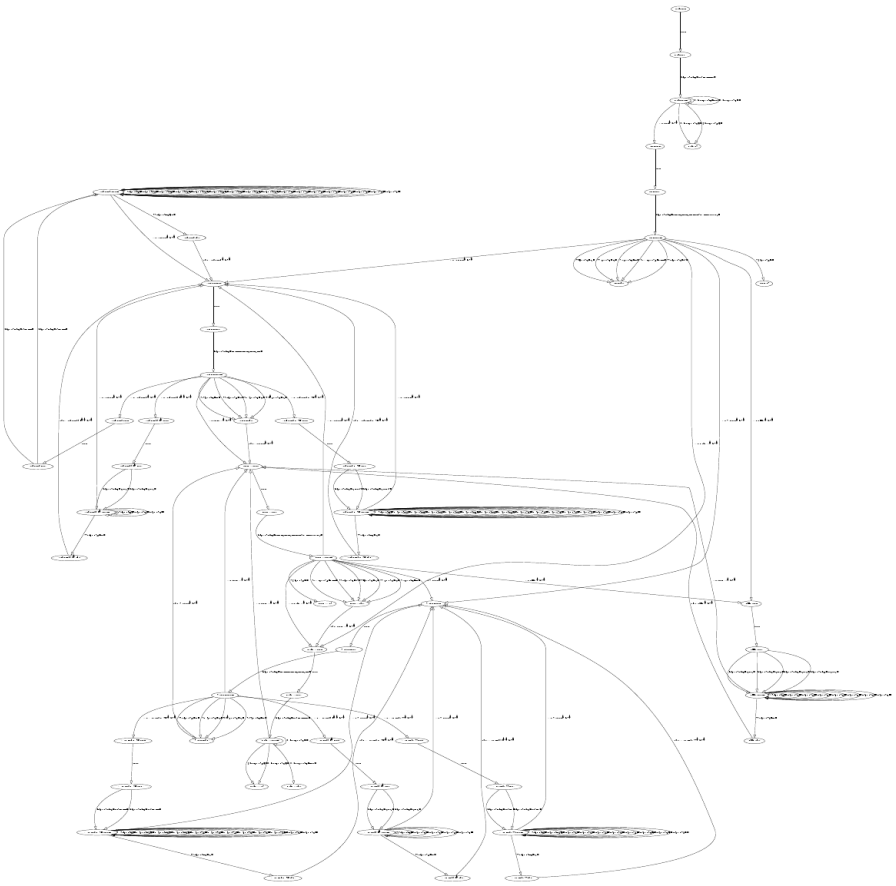


Fig. 10. *HSM*: The overall behaviour

5.3 Summarizing case study

This Section described the results obtained with GUISURFER when applying it to a larger interactive system. The choosen interactive system case study is related to a healthcare management system (*HMS*). The HMS system is implemented in *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 GUISURFER 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 GUISURFER enables the analysis of real interactive applications written by third parties.

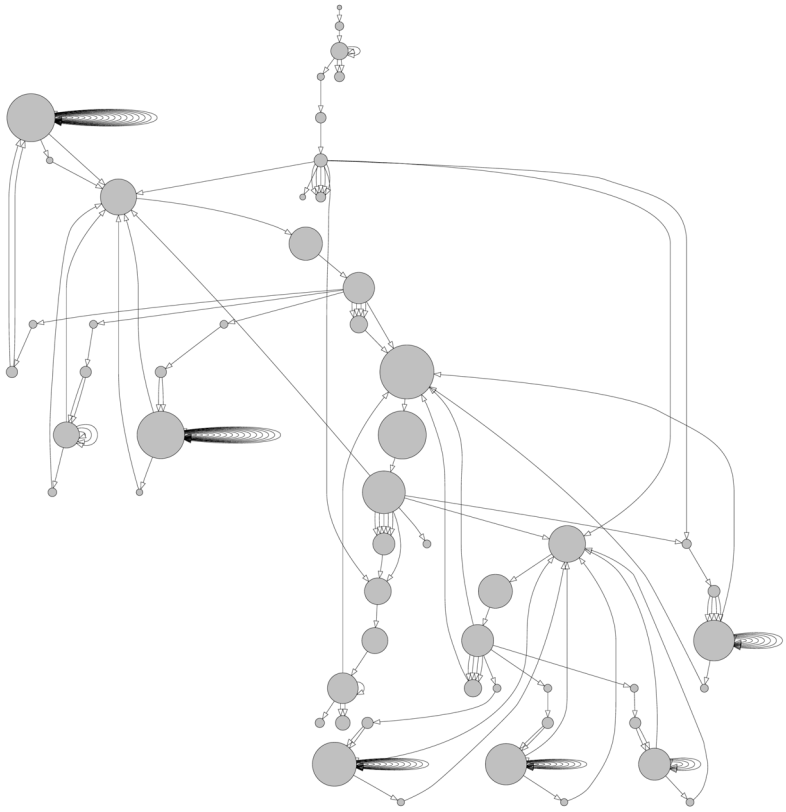


Fig. 11. *HSM*'s pagerank results

6. Conclusions and future work

This Chapter 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 describes the contributions of the Chapter. A discussion about GUISURFER limitations is provided in Section 2. Finally, the last Section presents some future work.

6.1 Summary of contributions

The major contribution of this work is the development of the GUISURFER 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 Chapter has presented a methodology to generate behavioural user interface models from the extracted information and to reason about it.

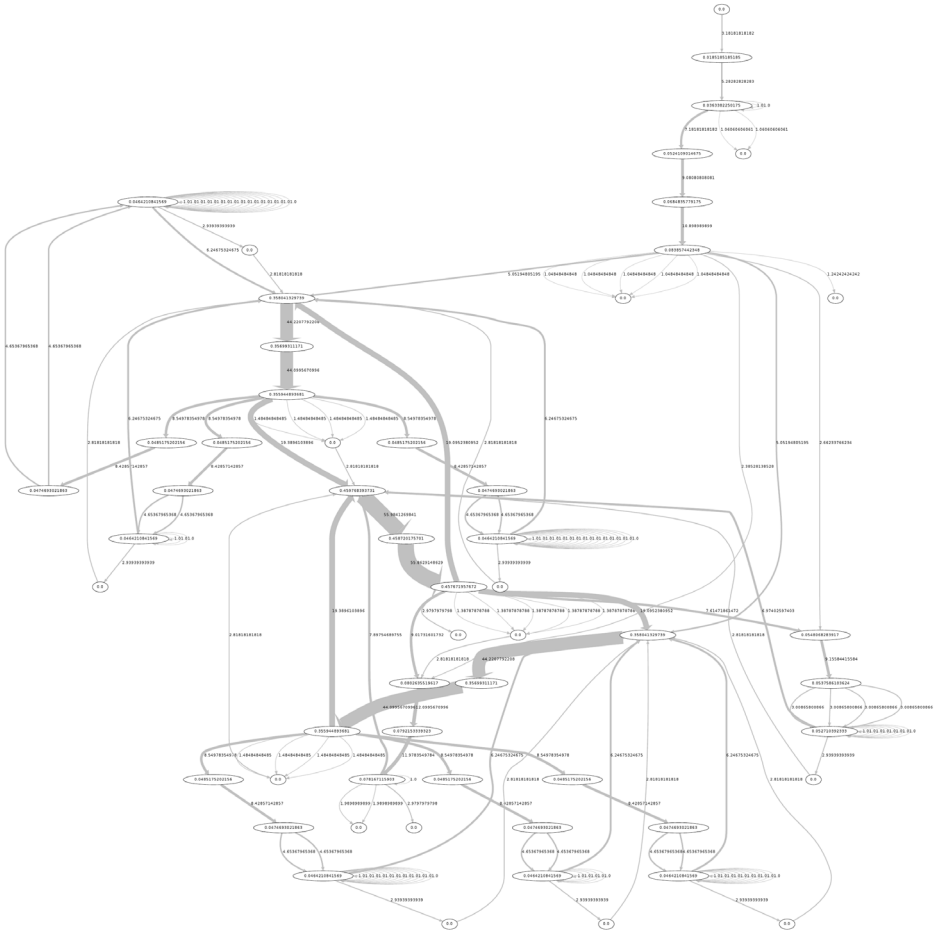


Fig. 12. *HSM*'s betweenness values

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

This work is an approach to bridging the gap between users and programmers by allowing the reasoning about GUI models from source code. This Chapter 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 forward. 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.

6.2 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. (Campos & Harrison, 2009)). 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 usability of a user interface.

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 due with layout management issues. GUISURFER cannot extract, for example, information about overlapping windows since this must be determined at run time. Thus, it can not be find out in a static way whether important information for the user might be obscured by other parts of the interface. A third issue relates to the fact that generated models reflect what was programmed as opposed to what was designed. Hence, if the source code does the wrong thing, static analysis alone is unlikely to help because it is unable to know what the intended outcome was. For example, if an action is intended to insert a result into a text box, but input is sent to another instead. However, if the design model is available, GUISURFER can be used to extract a model of the implemented system, and a comparison between the two can be carried out.

A number of others issues still needs addressing. In the examples used throughout the Chapter, only one windows 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).

6.3 Future work

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

6.3.1 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.

6.3.2 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. (Gamma et al., 1995). In software engineering, a design 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: (Stoll et al., 2008) 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 (Tidwell, 2005) 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 (Gilroy & Harrison, 2004). In any case these patterns have typically been used in a forward engineering context.

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 (Coutaz & Calvary, 2008) and generally help the maintenance and migration of GUI code.

7. References

Abowd, G., Bowen, J., Dix, A., Harrison, M. & Took, R. (1989). User interface languages: a survey of existing methods, *Technical report*, Oxford University.

Belli, F. (2001). Finite state testing and analysis of graphical user interfaces, *Proceedings.of the 12th International Symposium on Software Reliability Engineering, ISSRE 2001*, IEEE, pp. 34–42.
URL: *http://ieeexplore.ieee.org/iel5/7759/21326/00989456.pdf?tp=&arnumber=989456&isnumber=21326&arSt=34&ared=43&arAuthor=Belli%2C+F.%3B*

Berard, B. (2001). *Systems and Software Verification*, Springer.

Berkhin, P. (2005). A survey on pagerank computing, *Internet Mathematics* 2: 73–120.

Blandford, A. E. & Young, R. M. (1993). Developing runnable user models: Separating the problem solving techniques from the domain knowledge, *in* J. Alty, D. Diaper

& S. Guest (eds), *People and Computers VIII — Proceedings of HCI'93*, Cambridge University Press, Cambridge, pp. 111–122.

Bumbulis, P. & Alencar, P. C. (1995). A framework for prototyping and mechaniacally verifying a class of user interfaces, *IEEE* .

Campos, J. C. (1999). *Automated Deduction and Usability Reasoning*, PhD thesis, Department of Computer Science, University of York.

Campos, J. C. (2004). The modelling gap between software engineering and human-computer interaction, *in* R. Kazman, L. Bass & B. John (eds), *ICSE 2004 Workshop: Bridging the Gaps II*, The IEE, pp. 54–61.

Campos, J. C. & Harrison, M. D. (2009). Interaction engineering using the IVY tool, *ACM Symposium on Engineering Interactive Computing Systems (EICS 2009)*, ACM, New York, NY, USA, pp. 35–44.

Chen, J. & Subramaniam, S. (2001). A gui environment for testing gui-based applications in Java, *Proceedings of the 34th Hawaii International Conferences on System Sciences* .

Claessen, K. & Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs, *Proceedings of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000*.

Coutaz, J. & Calvary, G. (2008). HCI and software engineering: Designing for user interface plasticity, *The Human Computer Interaction Handbook*, user design science, chapter 56, pp. 1107–1125.

d'Ausbourg, B., Seguin, C. & Guy Durrieu, P. R. (1998). Helping the automated validation process of user interfaces systems, *IEEE* .

Dix, A., Finlay, J. E., Abowd, G. D. & Beale, R. (2003). *Human-Computer Interaction (3rd Edition)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Duke, D., Barnard, P., Duce, D. & May, J. (1998). Syndetic modelling, *Human-Computer Interaction* 13(4): 337–393.

Duke, D. J. & Harrison, M. D. (1993). Abstract interaction objects, *Computer Graphics Forum* 12(3): 25–36.

E. Stroulia, M. El-ramly, P. I. & Sorenson, P. (2003). User interface reverse engineering in support of interface migration to the web, *Automated Software Engineering* .

Ellson, J., Gansner, E., Koutsofios, L., North, S. & Woodhull, G. (2001). Graphviz - an open source graph drawing tools, *Lecture Notes in Computer Science*, Springer-Verlag, pp. 483–484.

Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Gilroy, S. W. & Harrison, M. D. (2004). Using interaction style to match the ubiquitous user interface to the device-to-hand, *EHCI/DS-VIS*, pp. 325–345.

Hanson, R. & Tacy, A. (2007). *GWT in Action: Easy Ajax with the Google Web Toolkit*, Manning Publications Co., Greenwich, CT, USA.

ISO/IEC (1999). Software products evaluation. DIS 14598-1.

Ivory, M. Y. & Hearst, M. A. (2001). The state of the art in automating usability evaluation of user interfaces, *ACM COMPUTING SURVEYS* 33: 470–516.

Loer, K. & Harrison, M. (2005). Analysing user confusion in context aware mobile applications, *in* M. Constabile & F. Paternò (eds), *PINTERACT 2005*, Vol. 3585 of *Lecture Notes in Computer Science*, Springer, New York, NY, USA, pp. 184–197.

Melody, M. (1996). A survey of representations for recovering user interface specifications for reengineering, *Technical report*, Institute of Technology, Atlanta GA 30332-0280.

Memon, A. M. (2001). *A Comprehensive Framework for Testing Graphical User Interfaces*, PhD thesis, Department of Computer Science, University of PittsBurgh.

Miller, S. P., Tribble, A. C., Whalen, M. W. & Heimdahl, M. P. (2004). Proving the shalls early validation of requirements through formal methods, *Department of Computer Science and Engineering* .

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

Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R. & Wong, K. (2000). Reverse engineering: a roadmap, *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, ACM, New York, NY, USA, pp. 47–60.

Myers, B. A. (1991). Separating application code from toolkits: Eliminating the spaghetti of call-backs, *School of Computer Science* .

Nielsen, J. (1993). *Usability Engineering*, Academic Press, San Diego, CA.

Paiva, A. C. R., Faria, J. C. P. & Mendes, P. M. C. (eds) (2007). *Reverse Engineered Formal Models for GUI Testing, 10th International Workshop on Formal Methods for Industrial Critical Systems*. Berlin, Germany.

SC4, I. S.-C. (1994). Draft International ISO DIS 9241-11 Standard, International Organization for Standardization.

Shan, S. Y. & et al. (2009). Fast centrality approximation in modular networks.

Stamelos, I., Angelis, L., Oikonomou, A. & Bleris, G. L. (2002). Code quality analysis in open source software development, *Information Systems Journal* 12: 43–60.

Stoll, P., John, B. E., Bass, L. & Golden, E. (2008). Preparing usability supporting architectural patterns for industrial use. Computer science, Datavetenskap, Malardalen University, School of Innovation, Design and Engineering.

Systa, T. (2001). Dynamic reverse engineering of Java software, *Technical report*, University of Tampere, Finland.

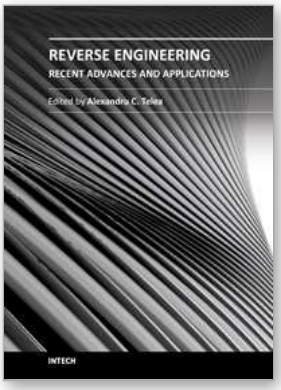
Thimbleby, H. & Gow, J. (2008). Applying graph theory to interaction design, *EIS 2007* pp. 501–519.

Thomas, J. M. (1976). A complexity measure, *Intern. J. Syst. Sci.* 2(4): 308.

Tidwell, J. (2005). *Designing Interfaces: Patterns for Effective Interaction Design*, O' Reilly Media, Inc.

Yoon, Y. S. & Yoon, W. C. (2007). Development of quantitative metrics to support UI designer decision-making in the design process, *Human-Computer Interaction. Interaction Design and Usability*, Springer Berlin / Heidelberg, pp. 316–324.

Young, R. M., Green, T. R. G. & Simon, T. (1989). Programmable user models for predictive evaluation of interface designs, *in* K. Bice & C. Lewis (eds), *CHI'89 Proceedings*, ACM Press, NY, pp. 15–19.



Reverse Engineering - Recent Advances and Applications
Edited by Dr. A.C. Telea

ISBN 978-953-51-0158-1
Hard cover, 276 pages
Publisher InTech
Published online 07, March, 2012
Published in print edition March, 2012

Reverse engineering encompasses a wide spectrum of activities aimed at extracting information on the function, structure, and behavior of man-made or natural artifacts. Increases in data sources, processing power, and improved data mining and processing algorithms have opened new fields of application for reverse engineering. In this book, we present twelve applications of reverse engineering in the software engineering, shape engineering, and medical and life sciences application domains. The book can serve as a guideline to practitioners in the above fields to the state-of-the-art in reverse engineering techniques, tools, and use-cases, as well as an overview of open challenges for reverse engineering researchers.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

José Creissac Campos, João Saraiva, Carlos Silva and João Carlos Silva (2012). GUIsurfer: A Reverse Engineering Framework for User Interface Software, Reverse Engineering - Recent Advances and Applications, Dr. A.C. Telea (Ed.), ISBN: 978-953-51-0158-1, InTech, Available from: http://www.intechopen.com/books/reverse-engineering-recent-advances-and-applications/guisurfer-a-generic-reverse-engineering-framework-for-interactive-system-analysis

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821