Universidade do Minho
Escola de Engenharia

Carlos Eduardo Bastos e Marques da Silva

Reverse Engineering of Rich Internet Applications

Setembro de 2010

Reverse Engineering of Rich Internet Applications
Carlos Eduardo Bastos e Marques da Silva
UMinho | 2010

Universidade do Minho
Escola de Engenharia

Carlos Eduardo Bastos e Marques da Silva

Reverse Engineering of Rich Internet Applications

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Orientador:
Professor Doutor José Creissac Campos
Co-Orientador:
João Carlos Silva

Setembro de 2010

Declaração

Nome: Carlos Eduardo Bastos e Marques da Silva

Endereço Electrónico: carlosebms@gmail.com

Telefone: 253418510

Cartão de Cidadão: 13290739

Título da Tese: Reverse Engineering of Rich Internet Applications

Orientador: Professor Doutor José Creissac Campos

Co-Orientador: João Carlos Silva

Ano de conclusão: 2010

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, Setembro de 2010

Assinatura: _____

And if we are to go forward,
if we are to make this a better world in which to live,
we've got to go back.
We've got to rediscover these precious values
that we've left behind.

Martin Luther King

Acknowledgments

I would like to express the deepest gratitude to my advisors, Professor José Creissac Campos and João Carlos Silva for the continuous support and encouragement throughout this work. Without their guidance and dedication this dissertation would not have been possible. Moreover, I would like to thank the other GuiSurfer team members: Professor João Alexandre Saraiva and Rui Oliveira, for the help, motivation and counseling provided.

Additionally, I am grateful to my family, for all the love and support given, and for always having confidence in me. I would like to thank all my friends in lab 3.20 for the inciting and pleasant environment created. Also, I am thankfull to Neil Mitchell (TagSoup creator), Arjun Guha (WebBits creator), Mark Wassell (HJS creator), and Andrey Chudnov for their support regarding the parsers' use.

Finally, a special thanks to the financial support from Fundação para a Ciência e a Tecnologia (FCT) under Research Grant (BI) number: BI1-2010_PTDC/EIA-CCO/108995/2008_UMINHO

Abstract

Web applications have gained significant popularity. Consequently, various new technologies for the development of these applications have arisen. However, as these are recent technologies, they are yet little mature and are in constant mutation and evolution. This makes the developed applications lack in structuring and planning.

The user interface of a Web application is an important part of the overall application. Most technologies have a clear division between the application layers, placing each of them either in the client or in the server. Normally, in the client there is only source code relevant to the user's interaction.

With the goal of contributing to the improvement of Web applications' development process, tools that enable the reverse engineering of their user interface layers have been developed. To that end, the GuiSurfer tool, a tool to reverse engineer Java/Swing and WxHaskell GUI code, generating models that capture the behaviour of user interfaces, was extended. In order to broaden the scope of this tool to Web applications development, two new modules were added to GuiSurfer, giving it the capability of analysing GWT (Google Web Toolkit) and Ajax applications.

Resumo

As aplicações Web têm vindo a despertar um crescente interesse. Por conseguinte, têm surgido inúmeras e variadas tecnologias para o desenvolvimento deste tipo de aplicações. No entanto, como estas tecnologias são recentes, são ainda pouco maturas e estão em constante mutação e evolução. Isto faz com que as aplicações desenvolvidas careçam de estruturação e planificação.

Neste tipo de aplicações, um segmento muito importante para análise são as interfaces com o utilizador. Visto que muitas destas tecnologias efectuam uma divisão entre a parte da aplicação que está no cliente e a que está no servidor, no cliente é normal estar presente apenas código relativo à interacção com o utilizador.

Com o intuito de contribuir para uma melhoria do processo de desenvolvimento de aplicações Web, foram desenvolvidas neste trabalho ferramentas de engenharia reversa da camada de interface com o utilizador a partir do código fonte deste tipo de aplocações. Neste sentido, foi extendida a ferramenta GuiSurfer, que fazia já engenharia reversa de interfaces em Java/Swing e WxHaskell, gerando modelos que capturam o comportamento dessas interfaces. Foram adicionados dois novos módulos a esta ferramenta, tendo em vista alargar o seu âmbito de utilização a duas tecnologias de desenvolvimento de aplicações de internet: GWT (*Google Web Toolkit*) e Ajax.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the computer primordial past, computers worked mostly standalone. Thus, each application had to be installed on each client computer. Sharing and exchanging information was difficult. With the advent of the Internet, computers began to be connected together, easing information sharing and exchange. The World Wide Web (or simply the "Web") represented a major milestone in the process of creating a global information medium, enabling seamless navigation between hypertext pages (Web pages) distributed across a multitude of servers (Web sites). At first, Web sites consisted mostly of collections of static pages with information users could browse. However, soon Web applications started to emerge. As with every new technology, they pose development and maintenance challenges that developers must face.

This thesis addresses some of these challenges. By investigating techniques and tools for the reverse engineering of Web applications, it aims at promoting a better understanding of existing systems and ease their maintenance. The focus is on the user interface layer of those systems. This is both a well defined layer of current software architectures, and of paramount relevance for a system's success.

1.1 Web applications

The main difference between a Web application and a Web site is mainly the fact that a Web application implements business logic, whereas a plain Web site does

not. The main distinction between a Web application and a traditional application is mainly that a Web application is remotely accessed over the internet, whereas a traditional application sits in the client computer.

Web applications have gained significant popularity over the last few years. Almost every company associated with business software and even some who are not, has converted their applications into Web applications, or plan to do so (Conallen, 1999). The limitations of early Web applications, when compared with desktop interfaces, were mainly their reduced usability and interactivity. Implementation technologies were more limited and application's responsiveness was lower, due to network latency. However, with ongoing technologies development, Web applications have evolved throughout the years into a new type of applications, providing a usage experience closer to that of desktop applications.

This evolution is often associated with the term Web 2.0, a marketing term that gained popularity when used by O'Reilly (2007). The term represented a shift in Web applications, making them more appealing and interactive towards the user. These applications were designated Rich Internet Applications (RIAs) and introduced new implementation technologies, enabling improved forms of user interaction and system responsiveness, bringing the user a richer experience. Unfortunately, this evolution of Web applications towards the desktop paradigm also meant that developing RIAs is based on a plethora of fast evolving technologies, making it hard to develop applications according to rigorous software engineering principles (Mikkonen and Taivalsaari, 2008). Hence, building and maintaining a RIA requires a significant amount of work, that can be reduced if aided by high-level model abstractions of the system, which enable a simpler system's specification and analysis.

There are several types of models that can be used during the implementation and maintenance of a software system. The type of model used depends on the users' purposes and objectives towards the system's abstraction.

1.2 User Interfaces

User interfaces are a vital component for the success of software applications. The same applies to Web applications. Indeed, user interfaces might be even more

important when it comes to Web applications, as it becomes easier to change from one application to another. User interface quality is typically measured by its usability, i.e., the effectiveness, efficiency, and satisfaction users experience when using the software ISO (1998). Usability is focused on the system?s design, not its implementation. However, from a software engineer perspective, the quality of the implementation is very relevant, since it is essential for the maintenance and evolution of the software. This is emphasized by the fact that the user interface is one of the components more prone to changes and updates.

As with software in general, models are useful in the development and maintenance of user interfaces. However, the constant changes and updates applications tend to be subject to, make it difficult to keep models and systems synchronized. A solution to this problem is having a reverse engineer tool to quickly abstract the system into a model.

Most of the existing tools and analysis approaches were developed targeting desktop applications. Nevertheless, Web applications development is an emerging area for these analysis. Indeed, these subjects have gained research popularity, and a number of works has appeared regarding this theme. Since tools targeting desktop applications are already developed and tested, the reutilization and expansion of these tools towards Web applications seems a suitable step. Moreover, an existing desktop-targeted tool, with the specific properties we wish to verify of Web applications (therefore appropriate to be reused and expanded), is already available: GuiSurfer.

1.3 Context

The work described in this thesis was carried out as a contribution to the CROSS[1] project. CROSS stands for "An infrastructure for Certification and Re-engineering of Open Source Software". The project's goal is to promote Open Source Software (OSS) quality by developing and combining new program understanding and analysis techniques. In particular, the project aims at the development of open source tools, enabling the analysis of open source software, for certifying

[1] twiki.di.uminho.pt/twiki/bin/view/Research/CROSS/

software quality. The project is being carried out by several researchers from *Universidade do Minho*, with funding from FCT (*Fundação para a Ciência e a Tecnologia*) under contract PTDC/EIA-CCO/108995/2008.

The project is divided into several specific tasks, each addressing different analysis aspects of the overall project. Of particular interest in this context is Task 3, named "Graphical User Interface analysis for OSS". This task aims to develop techniques and tools which permit the analysis of the user interface layer of software systems. GuiSurfer (Silva et al., 2009), a generic tool that reverse engineers user interfaces from source code, is being developed as part of this task. The challenge, then, is using the GuiSurfer tool to derive models from RIAs in order to reason over them.

1.4 Goals

The main purpose of this work is to continue the work previously done on reverse engineering user interfaces in the context of the CROSS project. More specifically, the development and enhancement of the GuiSurfer tool. This is to be achieved by extending its back-end. That is, generalising the approach to new languages and toolkits. The intention is to carry on previous work on generalizing GuiSurfer towards a greater number of languages, and develop two new back-ends.

Two different technologies will be considered: GWT (Google Web Toolkit) and AJAX. Hence the goals of the work are two generate two new modules for GuiSurfer. One enabling the reverse engineering of user interfaces programmed in Ajax, and another enabling GuiSurfer to reverse engineer user interfaces programmed in AJAX.

These candidates were selected because of the challenge to pursue this line of work towards a different paradigm, namely the Web applications paradigm. Therefore, it is also our objective to analyse and draw conclusions over the information generated by GuiSurfer in this new area of interest.

1.5 Structure of the document

This document presents the following structure:

- Chapter 2 makes an overview of the main concepts this document addresses. It starts by describing reverse engineering. Afterwards, several interface models are analysed and it concludes with a Rich Internet Applications synopsis.
- Chapter 3 describes the GuiSurfer tool in detail. It presents its architecture, an example of using GuiSurfer tool, and a description of GuiSurfer's implementation.
- Chapter 4 presents our research towards Google Web Toolkit. It begins with an overview of GWT, followed by an example application. Subsequently, is the exhibition of our approach and the results it produced.
- Chapter 5 introduces our approach regarding Ajax. An overview of Ajax technologies, our approach is described. Also GuiSurfer's new architecture for Ajax is presented.
- Chapter 6 is the conclusion of this dissertation and proposes some future work improvements.

All footnotes references to Web pages were validated during September 2010.

Chapter 2

Related Work

In this chapter, the main concepts and previous works in the area of interest for this dissertation are described. Specifically, this chapter starts by analysing the state of the art of Reverse Engineering (RE), with various applications and approaches in this area. Afterwards, as reverse engineering is a process of obtaining abstrations from a implementation, an analysis of the various interface models is presented. Concluding this chapter, a brief explanation of Rich Internet Applications is made.

2.1 Reverse Engineering

System's tendency for degradation, which can be provoked by software entropy (Jacobson, 1992), that is, the predilection for software to become onerous and expensive to manage over time, leads to a need to improve and maintain the system. This is one of the reasons reverse engineering has been an important subject to the software industry in the last few years. In order to maintain and improve a system, a process of reverse engineering is necessary. In other words, a system's model abstraction is required to obtain the system's relevant information. The abstraction can be performed on a mental level only, or through a reverse engineering tool which generates the system's abstraction. Obviously, as the system size increases, so it does the usefulness of using reverse engineering tools. Therefore, reverse engineering becomes an essential process to develop,

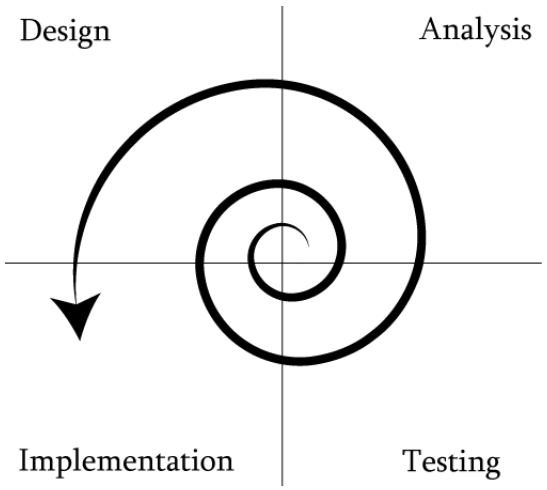

Figure 2.1: The spiral model

improve or maintain a computer system.

2.1.1 General Terms

In order to comprehend the reverse engineering concept it is important to understand the four main concepts related to this area that Chikofsky and Cross (1990) identified, namely: "Forward engineering", Reverse engineering, Restructuring and Reengineering.

Forward engineering is the traditional process of moving from high level abstractions and logical (implementation independent) designs to a system's physical implementation (Chikofsky and Cross, 1990). During forward engineering one moves through the requirements phase, to the design phase and from this to the implementation. This means that forward engineering implies moving downward through the abstraction levels (Rosenberg and Lawrence, 1996). In summary, in a software engineering context, forward engineering corresponds to one or more transitions, of a design or implementation artifact to a lower abstraction level.

Reverse engineering is, as the name implies, the inverse of forward engineering. It consists on a process of analysing a system in order to discover its components and their interrelationships, as well as to create a representation of the system. This representation may be another form of the system, or a representation at a higher abstraction level. The main purpose of using reverse

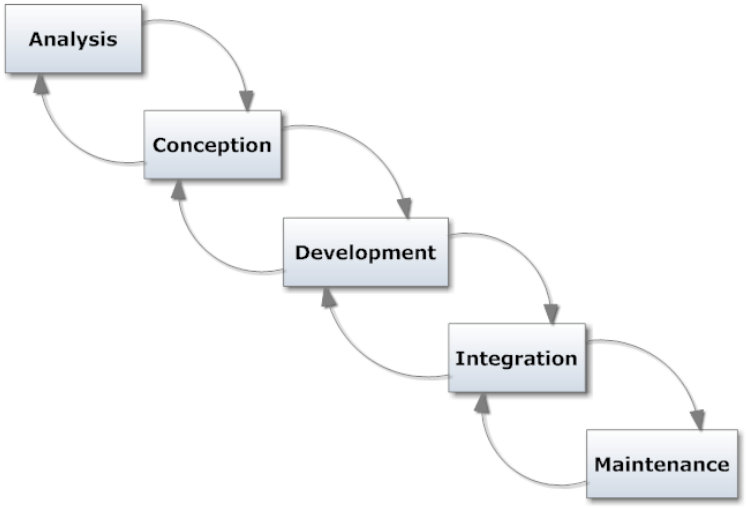

Figure 2.2: Waterfall model

engineering is to extract information from already existing application systems. This information is usually gathered by building or synthesizing abstractions that are less implementation-dependent.

Although reverse engineering usually starts with a fully functional system, this is not mandatory, it can also be used in a new system's construction. For instance, when developing software through the spiral model. The spiral model defined in (Boehm, 1986), and depicted in Figure 2.1, is a software development model that comprises the idea of evolutionary development. Thus, it is an iterative and incremental method. For example, in order to perform the analysis or the design in the second iteration, it is useful to have an idea of what has really been implemented in the first. Hence, the use of reverse engineering on a partial implementation.

Moreover, reverse engineering can be made from any abstraction level. As an example, consider the waterfall model, as depicted in Figure 2.2. It was the first model towards defining the procedure of developing software systems. The model is based on developing a system through a sequential process of several phases, starting with the system requirements analysis, and ending with the maintenance of the application. Thus, the visual aspect of a waterfall. Reverse engineering, however, can be seen as the inverse approach to the traditional development of software systems. It is the process of "going up" the waterfall. This process can

be used in any two sequential phases, or even more. In the extreme case, it can be the process of moving from the implementation phase into the requirements analysis phase, but starting from the implementation is not mandatory.

A common fallacy is that reverse engineering implies a transformation in the system or the creation of a new system. It should be seen as a process of examination, analysis, not a process of change, transformation or replication. Consequently, reverse engineering's general objective is to obtain missing knowledge when it is not available (Eilam, 2005).

Restructuring is the change from one representation to a new one at the equivalent abstraction level. The system should maintain the same level of functionality as well as semantics. Shortly, restructuring transforms the system but functionality remains the same. Moreover, system's transformations of this nature correspond generally to alterations in the code from an unstructured form to a structured one.

An example of a restructuring transformation is data normalization, which can be defined as a data to data transformation of the database in order to obtain a normalized logical data model. The restructuring process can be performed with the purpose of improving the software structure, which also implies an improvement of the software system maintainability (Eloff, 2002). Therefore, restructuring can be seen as a process of re-organization of the logical structure of a subject software system in order to develop characteristic attributes (Kang and Bieman, 1999), or build it less predisposed to errors in the future (Arnold, 1989).

Reengineering is described as the process of inspection and adaptation of a system in order to rebuild it in a new form and afterwards accomplish its implementation. Usually, reengineering is a process of reverse engineering, aimed at obtaining an abstract representation, followed by a forward engineering process in order to achieve the necessary alterations to the system (Chikofsky and Cross, 1990). 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, and afterwards the reformulation of the subject system. Reengineering may also modify the subject system's behaviour if there were modifications in the requisites. Reengineering's purpose is

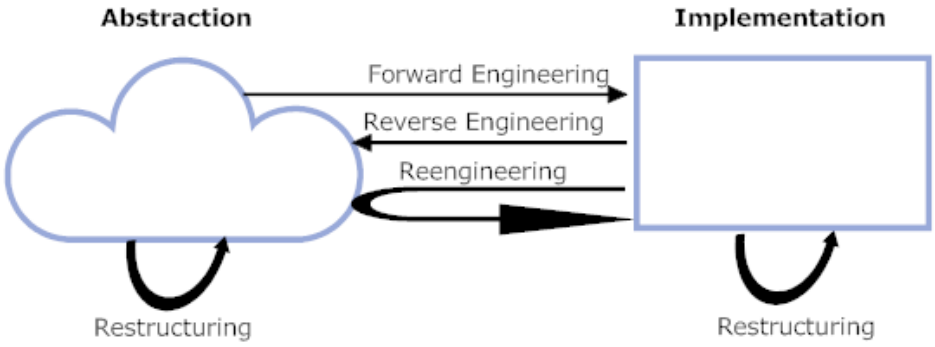

Figure 2.3: Reverse Engineering

to comprehend the software, and subsequently to apply modifications in order to improve system's functionality, performance or implementation (Rosenberg and Lawrence, 1996).

In Figure 2.3, the various concepts are presented together, in order to better illustrate their tasks during the software development cycle. Although only an abstraction and implementation are depicted, these concepts can also be applied between two abstractions.

2.1.2 Applications

Reverse engineering has various applications in software engineering, The most important, in increasing level of abstraction, are: program analysis, plan recognition, concept assignment, redocumentation and architecture recovery (Telea et al., 2002).

Program analysis is the most common application of reverse engineering. It consists of two processes: the creation of a program's model and the presentation of this model to the user. As the name implies, a program analysis tool requires source code examination. The analysis can be performed at several different levels of abstraction such as: implementation, functional, structural and domain levels (Ning, 1989). These levels of analysis are important as each will contain different system's informations.

Plan recognition has the goal of recognizing structural or behavioural patterns in the source code. The procedure is to search the source code in order to

find the most used algorithms or data structures. To compare the algorithms, pattern-matching heuristics are used to map patterns or models at higher levels of abstractions to lower level code (Baxter and Mehlich, 1997).

Since, usually, several pieces of the source code in a system are reused, simply by copy pasting and using them numerous times, plan recognition can be used to detect these situations. Therefore plan recognition helps reducing the problems associated with code reutilization, such as increased code size, and the need to replicate changes in all cloned copies (Tilley, 1998).

Concept assignment is a reverse engineering application that aims to search for concepts in a system, and assign them to their implementation counterparts (Biggerstaff et al., 1993). Concept assignment can be seen as conceptual pattern matching, a pattern matching performed at a semantic level. The process is achieved by identifying relevant concepts (e.g., a functional requirement) within the source code (Robillard et al., 2007), and afterwards building a representation of the code by associating the concepts to the code.

Redocumentation is based on the modification or creation of documentation for an existing software system. It is considered the simplest and oldest form of reverse engineering (Chikofsky and Cross, 1990). Since redocumentation is the transformation from source code to pseudo-code or prose, it can be considered as a transformation to a higher abstraction level, and therefore to be a reverse engineering task (Tilley, 1998).

Architecture recovery aims to recover the architectural aspects of the system. It may also be identified as structural redocumentation (Wong et al., 1995). Therefore, architecture recovery is the creation of documentation that defines the entire structure/architecture of large systems. In order to achieve that goal it generates high-level structural models that define the design architectural information of the system.

2.1.3 Approaches

There are two main approaches to the realization of a reverse engineering process. Namely, static analysis and dynamic analysis. As their names indicate they differ on how the analysis of the system is performed.

Static analysis implies the analysis of the software system without the actual execution of the software. Dynamic analysis implies analysing the system while running, that is, while the software system is being executed.

Static analysis involves analysing the system code, and therefore can produce some results that could not be found in a dynamic analysis since the code is where all the system's actions are specified. However, static analysis cannot discern what elements are really used and those who are not, it also cannot analyse the system's performance. Consequently, if our interest is in these system's aspects, a dynamic analysis must be used (Ritsch and Sneed, 1993).

Static Analysis

Static analysis performs a system analysis without executing it, and can be divided in two types: source code analysis and binaries analysis. Source code analysis is simpler to carry out, since it is easier to interpret source code than binary code. However, the problems are that source code is not always available, and that the final result is obviously dependent on the quality of the source code parser.

A number of static source code analysis reverse engineering tools, aimed at user interfaces, can be found in the literature. For instance, ReversiXML[1] a tool described in (Bouillon et al., 2005) applies derivation rules to reverse engineer an HTML web page into UsiXML[2], a modelling language for user interfaces (Limbourg et al., 2004).

Guha et al. (2009) describe a tool that performs a static control-flow analysis for JavaScript applications running in Web browsers. In the approach, a behaviour model is extracted, and, afterwards, an intrusion detection is performed from the server side. Their analysis, however, has a different focus from the one we intend to perform, as the model built is a flow graph of URLs whereas we will focus on the interface behaviour. Also worth mentioning, the same approach was successfully tested on the JavaScript code generated from a GWT application.

The GuiSurfer tool (Silva et al., 2009), performs static analysis to produce behaviour models of the GUI from a target source code, There are versions of

[1] ReversiXML - www.isys.ucl.ac.be/bchi/research/reversi/RevXMLUI.php
[2] UsiXML - www.usixml.org/

the tool aimed at Java/Swing and WxHaskell. This tool will be analysed deeper in Chapter 3.

Performing the analysis from the binaries has the advantage of easier access to the needed information. One recurring problem is that it can have legal issues, as tools that perform reverse engineering of binaries may be used on illegal acts such as discovering and recreating information about proprietary software. In order to prevent these situations, some programmers/compilers obfuscate their code, that is, write code which is difficult to understand on purpose. Therefore adding a greater adversity to the reverse engineering process.

Reverse engineering of binaries is accomplished with hex editors, decompilers or disassemblers. Hex editors, such as WinHex[3], read the programs from Random-access memory (RAM), and afterwards display the results in hexadecimal code. Decompilers, do the reverse of a compiler, they attempt to transform binary programs into readable source code. However, if there are parts they cannot decompile they transform them into assembly code. There are numerous decompilers available for several languages, for example the DJ Java decompiler[4], for Java and, the REC (Reverse Engineer Compiler) decompiler[5] that translates binaries into C pseudo-code. Disassemblers convert binary code into assembly code. Thus, in comparison with a decompiler, they differ as a decompiler translates binary to a high level language. An example of a decompiler is OllyDbg[6] a 32-bit assembler level analysing debugger.

Dynamic Analysis

Dynamic analysis aims to obtain a model of a system from its runtime behaviour. To this end, there are several types of tools. Namely, debuggers, profilers, or source code instrumentation (the insertion of source code fragments in order to enable the acquisition of system runtime information).

A dynamic analysis tool may have a technique to model the behaviour of the software systems, that technique is called dynamic visualization. Specifically,

[3] Winhex - X-Ways Software Technology - www.winhex.com/winhex/
[4] DJ Java Decompiler - members.fortunecity.com/neshkov/dj.html
[5] REC - www.backerstreet.com/rec/rec.htm
[6] Ollydbg - www.ollydbg.de

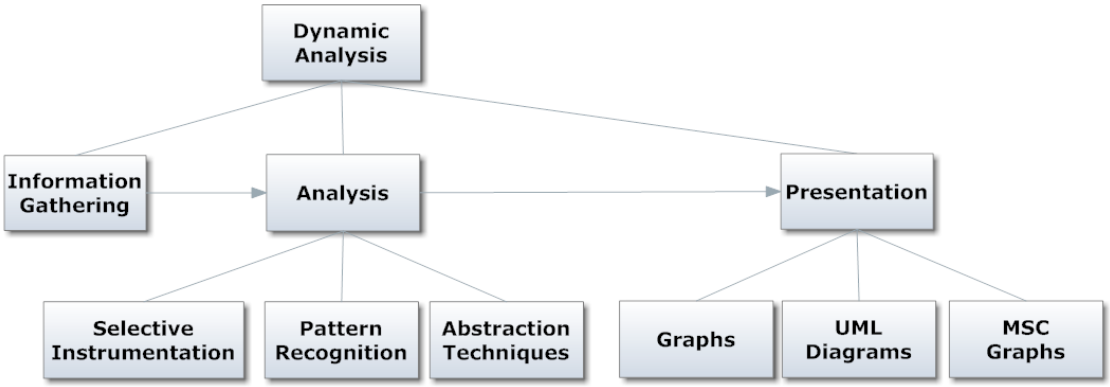

Figure 2.4: Dynamic Analysis

dynamic visualization is defined in three phases: gathering of information about the system's behaviour; analysis of the information collected; presentation of the outcome (Pacione et al., 2003).

The information gathering can be consummated by the execution of an event trace of the program's execution. Moreover, the event trace results may be acquired by instrumenting source code, object code, the environment, or executing the system under debugger or profiler monitoring.

To analyse the information there are three major techniques: selective instrumentation, that measures specific methods important to the analysis; pattern recognition, that aims to find behaviour patterns; and abstraction techniques, that try to aggregate the data gathered (Pacione et al., 2003).

In order to present the results there are three main diagramming techniques (Pacione et al., 2003): basic graphs representations which may be susceptible to scalability issues; Unified Modelling Language (UML) (Booch et al., 2005) diagrams such as class diagrams or sequence diagrams; message sequence charts (MSC) a popular visual formalism (Henriksen et al., 2000). The various concepts related to dynamic analysis are illustrated in the diagram of Figure 2.4.

Dynamic analysis has numerous implementations. For instance, Systä (1999) analyses the run-time behaviour of Java software by running the software in order to generate state diagrams. Chen and Subramaniam (2001) use reverse engineering to accomplish a specification-based testing of user interfaces. Users can graphically control test specifications that appear as Finite State Machines

(FSM) which abstract the run-time system. Memon et al. (2003) describe an application called Gui Ripping which consists in a dynamic process that transverses a GUI by opening all its windows and extracting all the widgets (GUI objects) and their information. Grilo et al. (2009) propose a tool that tries to automate GUI testing, the tool executes the application (uses the dynamic approach) and afterwards extracts structural and behaviour information from the user interface.

2.1.4 Hybrid approaches

There are approaches to reverse engineering that try to gather the positive aspects from both static and dynamic analysis, thus using both approaches simultaneously. For example, Li and Wohlstadter (2008) describe an hybrid approach that enables view-based maintenance of GUIs. This tool was tested on Java/Swing applications and its main concern is interface maintenance, while we focus on interface behaviour.

Furthermore, Gimblett and Thimbleby (2010) discover a model of an interactive system by simulating user actions. Models created are directed graphs where nodes represent system states and edges correspond to user actions. The approach is dynamic but it also considers access to a source code application is available. Similarly to GuiSurfer's, this tool was developed using the Haskell programming language.

2.2 Interface Models

As reverse engineering is a process of extracting models at higher levels of abstraction from more concrete representations, it is relevant to research what types of models exist in the current literature. As our approach is focused on the interface of a system, this section specifies the various interface models.

For a reverse engineering tool to be as comprehensive as possible, it needs to aspire deriving the different types of interface models since each model type serves unique purposes. Therefore, for a tool to be generic it cannot elide the different interface modelling types.

During these last years there has been a significant evolution in interface

models, mostly due to considerable interest in model-based user interface development. The importance of interface models has made this area proliferate and several types of interface models are currently accepted and implemented. For instance, (Puerta et al., 1994) developed a tool, named Mecano, that aims to automate interface design from data models. More recently, the UsiXML language (Limbourg et al., 2004) addresses the modelling of user interfaces for the requirements stage down to the implementation and is supported by a considerable numer of tools.

2.2.1 Data models

Data models are considered the pioneers of interface models. They perform an abstraction of the system based on the data structures it contains. However, abstracting an interface into a data model has little significance since data models do not contain any type of behaviour specification. Nevertheless, current model-based environments use data representations as a subcomponent of their interface model (Puerta, 1997).

2.2.2 Domain models

Domain models emerged naturally from the evolution of data models. Domain models are built from a conceptualization of the domain they symbolize. Fully declarative domain models are able to describe object relationships in a specific domain. Actually, domain models allowed the automatic creation of interface behaviour specifications therefore eradicating the main data models problem. However, domain models do not express the semantic functions associated with the domain's objects.

2.2.3 Application models

In order to solve the semantic functions lacuna Applications models arose. These models' primary objective is to ease interface behaviour specification. "The UIDE application model, for example, consists of application actions, interface actions, and interaction techniques" (Puerta, 1997). They permit the assignment of pa-

rameters, preconditions, and post conditions to each action allowing the control of the application behaviour.

2.2.4 Task models

Task models are models that express the actions a user performs in order to achieve his objectives. The goal of a task model is not to express how the user interface behaves, but rather how a user will use it to achieve specified goals. Task models are important in the application domain analysis and comprehension because they capture the main application activities and their respective relationships. Another important reason is the fact they permit the examination and evaluation of the system's usability. Besides, they are appropriate to improve conventional software engineering development since they add the user interaction aspects to development.

One of task models applications is measuring the easiness of how users will reach their goals. Accordingly, task models can also be used as a documentation method in order to support the users in using the system, and developers since they are an abstract model of the implementation.

Task models can be grouped into three different types: system task models that express the system implementation of the tasks; envisioned task model refers to the system designers ideas about the application and the interaction with users; user task models describe how users consider task should be implemented to accomplish their goals (Paternò and Alfieri, 2001).

There are distinctive task models variants. For instance, **Hierarchical Task Analysis (HTA)** was the pioneer method. It was proposed by (Annett and Duncan, 1967). The primary objective was to train users to execute particular tasks. Thus, the proposal was to express the activities logically structured in a number of levels. However, this method does not contain task ordering. Tasks should be implemented through a plan available for each hierarchic level. Tasks can be defined in terms of the goals attained when the task is completed. Moreover, a goal has a status and conditions to be satisfied associated (Limbourg and Vanderdonckt, 2003). The main disadvantage of this approach is the archaic relationship between activities, as the information they express is quite scarse.

With the arrival of graphical interfaces, this subject gained significant relevance. This motivated the genesis of the first method aiming at a systematic approach to the design of user interfaces: **Goals, Operators, Methods and Selection rules (GOMS)** (Card et al., 1983). GOMS models are composed by the four concepts that constitute the name GOMS. Goals express the tasks that users might perform. Operators describe the physical and cognitives actions a user fulfills in order to achieve a goal. Methods, a concept introduced by this method are a description of how a task is performed, therefore, they are a sequence of goals and operators. Selection rules are the factor that provides the method selection to achieve a purpose. It is also important to reference that GOMS introduced hierarchical logical structures to task models (Paternò et al., 1997). Therefore, this approach is useful for the prediction of user tasks performance. Nevertheless, GOMS has limitations, since it considers behaviours have no errors, and sequential tasks only.

ConcurTaskTrees(CTT) is a graphical notation developed by Paternò et al. (1997) to describe tasks at different abstractions levels. It is composed by five main concepts: tasks, objects, actions, operators and roles. This notation divides tasks in four groups:

- *User tasks*, where the user is the unique intervener, therefore they require cognitive or physical activities without system interaction;
- *Application tasks* refer to tasks performed exclusively by the system;
- *Interaction tasks*, as the name suggests, are tasks accomplished by the user interaction with the system;
- *Abstract tasks* are used to structure models, thus they aggregate different subtasks groups.

Operators are based on the LOTOS (Language of Temporal Ordering Specifications) specification language (Bolognesi and Brinksma, 1987) and express temporal relationships between tasks at the same abstraction level (Paternò et al., 1997).

Figure 2.5 depicts a simple example of a CTT model for a Login Web page authentication window. As illustrated, a CTT model is a tree structure, composed

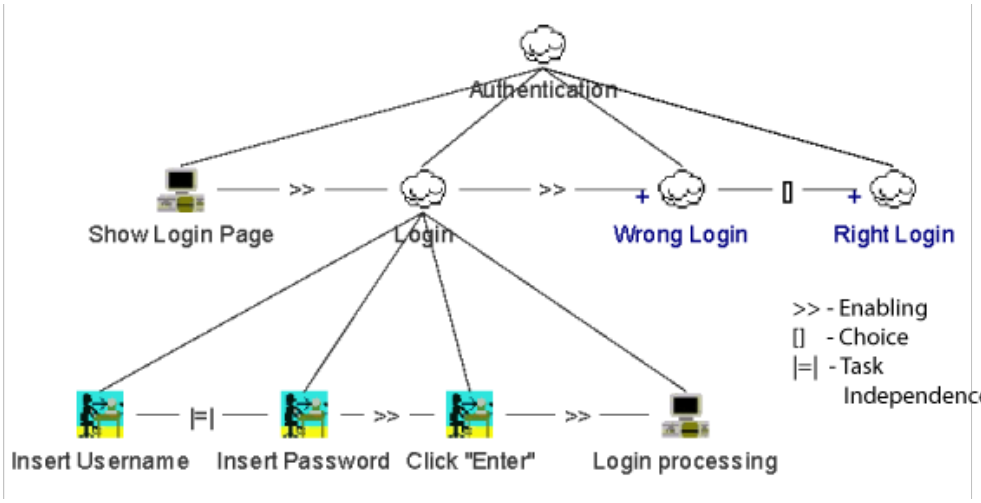

Figure 2.5: An authentication CTT model

by a root, that designates the goal of the task, and contains the various subtasks. This particular task starts with the system presenting the Web login page to the user. Afterwards, the user inserts his username and password and clicks the "Enter" button. The information is then processed by the system. Consequently, the system reacts with either of two abstractions, correct login and wrong login, each with specific subtasks models (not represented in the figure).

2.2.5 Dialog models

Dialog models express the syntactic structure of the interaction between human and computer (Schlungbaum, 1996). To this end, they stipulate what the application presents to users. The buttons, commands, all the controls the user can interact with, and the results of that interaction on the system. This modelling type has several different implementations, for example: dialog nets (Janssen et al., 1993), attributed grammars (Paakki, 1995), state transition diagrams (Grosu et al., 1996), and dialog templates (Szekely et al., 1993). This diversity of modelling languages can also be seen as a downside, since all these different types of dialog models have not merged into a single abstraction technique.

2.2.6 Presentation models

Another type of interface models are Presentation models that represent the materialization of widgets in the various dialog states. Moreover, a presentation model is composed by the different windows layouts of an application, the widgets each window contains', and their visual dependencies. Ideally, presentation models express the state and behaviour of the interface without any implementation specific controls or widgets making the implementation of the model possible in any GUI framework. This type of models is closely linked to dialog models, thus being often merged together.

2.3 Rich Internet Applications

Rich Internet Applications (RIAs) are an emergent set of technologies whose primary goal is to develop web applications with the strengths of desktop applications. The RIA term was first introduced in a Macromedia paper by Allaire (Allaire, 2002) and it meant the unification between traditional desktop and web applications. This unification aspect made this set of technologies gain an enormous acceptance, and consequently a great development during these last few years.

The principal advantages of desktop applications in comparison to traditional web applications are (Noda and Helwig, 2005):

- the absence of page reloading;
- no need for an online connection;
- support for interaction with other desktop applications;
- superior interaction and usability.

However, traditional web applications, applications accessed through the Web browser, also have specific advantages such as:

- no deployment/installation or updates in every desktop;

- easier access, since only an internet connection is required;
- availability in more platforms;
- concentrated information eases security and backup processes.

RIAs merge both types of applications in order to acquire the advantages of both. Bozzon et al. (2006) groups RIAs in four categories:

- Scripting-based approaches, where the client-side logic is performed by scripting languages like JavaScript, and interfaces are composed by HTML and CSS. There are several frameworks that use different programming languages for development but the final result is a JavaScript application. For example, GWT, Vaadin[7] and ZK[8].
- Plugin-based approaches, are built with a programming language and afterwards the interface is created by browser plug-ins. Flash, Flex[9], Laszlo[10] (uses Flash player) , Xamlon (lets developers use XAML to build applications) and Silverlight[11] all fall into this category;
- Browser-based approaches, the browser is capable of interpreting a XML language and afterwards creating the interface, an example is Mozilla's XUL[12] (XML User Interface Language);
- Web-based desktop technologies, define applications that require Web download but are executed outside the browser, Java Web Start (JavaWS) and Windows Smart Client are examples of this category.

From all the different RIAs frameworks, this research will focus on two technologies, namely: Google Web Toolkit (GWT) and Asynchronous JavaScript And XML (Ajax). This occurs as we decided to focus on a specific category of RIAs.

[7] Vaadin - http://vaadin.com/home
[8] ZK - http://www.zkoss.org/
[9] Flex - http://flex.org/
[10] Laszlo - http://www.openlaszlo.org/
[11] Silverlight - www.silverlight.net/
[12] XUL - https://developer.mozilla.org/en/xul

Scripting-based approaches were selected because they are widely accepted and do not need additional plugins.

Moreover, from the various Scripting-based approaches these particular frameworks were chosen for different reasons. GWT was analysed due to the fact that applications' are developed in Java code. Since GuiSurfer is a tool that works with Java Swing, an extension of the tool to another Java source code language seemed to be a logical step. Ajax was analysed because is one of the most popular RIA technologies. A considerable percentage of Internet Web sites is implemented in Ajax. Both GWT and Ajax will be discussed in detail in Chapter 4 and Chapter 5 respectively. Before that, however, the GuiSurfer tool will be introduced (Chapter 3).

Chapter 3

The GuiSurfer Tool

GuiSurfer is a generic tool to reverse engineer GUI code (Silva et al., 2009).

GuiSurfer was developed in order to reach the following goals:

- Automatically extract models expressing GUI behaviours from source code. The behaviours of interest are: the occurrences of GUI events, the interactive actions performed, the GUI states created.
- Enable reasoning about GUI behaviours to study the usability, quality and interactivity of the application.
- Be a generic tool, to allow posterior adaptations to other programming languages and different programming paradigms.

In summary, GuiSurfer aims to recognize components in the user interface through functional strategies and formal methods. "These components include user interface objects and actions" (Silva et al., 2006).

GuiSurfer's approach is focused on the applications' behaviour, that is, it performs a system analysis based on the events which take place, after a starting point in the application. Furthermore, for each event discovered it analyses the associated conditions, the actions that are executed, and which are the future application states. This approach enables the acquisition of information regarding the application's usability and also the quality of the implementation. A simple example of GuiSurfer's execution is presented on Figure 3.2 (see Section 3.2).

The tool is capable of analysing the source code of applications programmed with Java/Swing (Loy et al., 2002), and afterwards generate behavioural models of their user interfaces. Thus, this reverse engineering work enables us to analyze, e.g., via model checking, models of the user interface generated from Java code. Naturally, and as it is a generic tool, it evolved and is nowadays also capable of analysing WxHaskell (Leijen, 2004), a portable and native GUI library for Haskell.

3.1 Architecture

Since one of GuiSurfer's development objectives was making it generic, the tool is composed of two phases: a language dependent phase and a language independent phase, as shown on Figure 3.1. Hence, if there is the need of retargeting GuiSurfer into another language, only the language dependent phase should ideally be transformed.

The GuiSurfer architecture is composed by three modules (Silva et al., 2009): GUI layer extraction; GUI behavioural abstraction; and GUI analysis.

3.1.1 Gui layer extraction

The first step GuiSurfer performs is the creation of an Abstract Syntax Tree (AST). This is achieved by using a parser on the source code. An AST is a formal representation of the abstract syntactical structure of a source code. Moreover, the AST represents the entire code of the application. However, GuiSurfer focus is the GUI layer of applications, not the entire source code. Therefore, techniques to retrieve only the GUI relevant code fragments are needed. To this end, GuiSurfer was built using two generic techniques: strategic programming and code slicing.

Strategic programming is a form of generic programming based on the concept of functional strategy, a generic action of data processing that is capable of transversing heterogeneous data structures while aggregating uniform and type-specific behaviours (Lämmel et al., 2002). This type of programming greatly improves transversal approaches, thus permitting programmers to define tree transversal functions with a high level of conciseness, composability, structure-

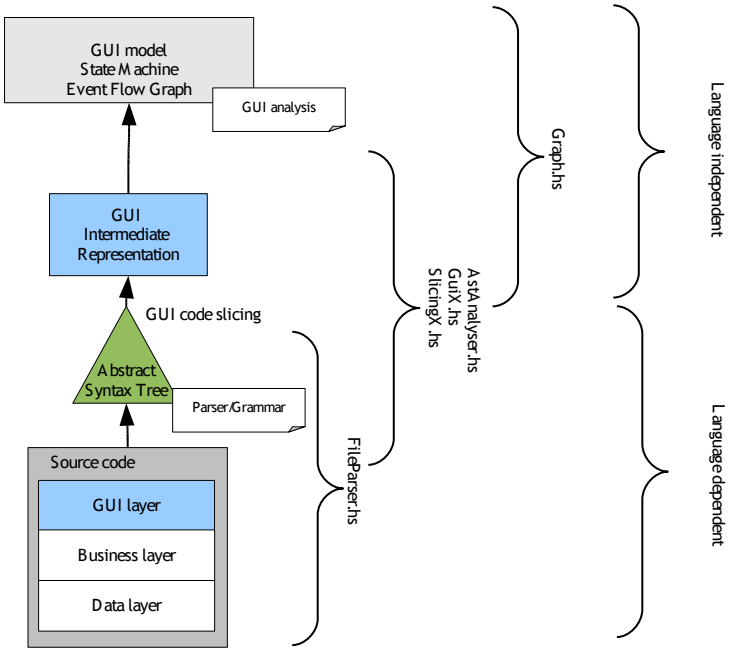

Figure 3.1: GuiSurfer's tool architecture. Adapted from Silva et al. (2010a)

shyness, and traversal control (Visser and Saraiva, 2004).

Code slicing is a programming language independent technique that allows extracting relevant information from a program source code, based on the program dependency graph and a slicing criteria (Tip, 1995). A slice is a subset of the main program. It can be obtained, for example, by traversing the entire AST of the application, and returning a sub-tree of that AST.

In order to implement these techniques an Haskell GUI code slicing library was developed that includes a generic set of transversal functions with the objective of transversing any AST.

3.1.2 GUI behavioural abstraction

Once the AST has been created, GUI behavioural abstraction is the following step. It consists in abstracting the user interface behaviour and structure. The relevant abstractions are (Silva et al., 2007):

- User inputs - widgets that enables users' data input;

- User selections - widgets that allow users to choose between different options;
- User actions - the actions executed as a consequence of user input or user selection;
- Output to user - the interactive responses the system gives to the user.

In this phase, the textual representations of the behavioural GUI models is created. Therefore, a GUI intermediate representation is created in this phase. I.e., the files GuiModel.hs and GuiModelFull.hs are produced in the conclusion of this phase. GuiModel.hs and GuiModelFull.hs are meta-models created by GuiSurfer, they specify the behaviour of the application interface.

A GuiModel is defined in GuiSurfer as the following Haskell type:

```
type GuiModel = Map (EventRef,CondRef) [ExpRef]
```

Hence, it can be described as a mapping between events and conditions, and their associated actions references. This meta-model is where the information of the interface behaviour is defined. It also contains other relevant information, such as the window name and initial state. Moreover, it has the window close and end actions references, as well as new windows action references. Section 3.3 presents an example of this type of model.

3.1.3 GUI analysis

After producing the interface behaviour models, it is imporant to perform reasoning over the generated models. As an example, GuiSurfer models can be tested by using the Haskell QuickCheck tool (Claessen and Hughes, 2000), a tool that tests Haskell programs automatically. Thereby, the programmer defines certain properties functions should satisfy, and afterwards tests those properties through the generation of random values.

For a better visual experience and easier reasoning of the results Guisurfer is 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 (FSM) models of the interface. The

FSM models are illustrated through state diagrams in order to make them visual appealing. These different diagrams GuiSurfer produces are a form of representation of dialog models. GuiSurfer's image models are created by the usage of Graphviz[1] (Ellson et al., 2002), an open source set of tools that allows the visualization and manipulation of abstract graphs.

3.2 GuiSurfer example

Figure 3.2 depicts the result of executing GuiSurfer over a simple Java/Swing application. The application is an example of a login window. It is composed of two input boxes, enabling the user to input his name and password, and by two buttons. One *OK* button to confirm the operation and proceed with the validation, and a *Cancel* button that closes the application.

Guisurfer's execution over the login application produces a state machine, as represented on the right side of the figure. Each state from the state machine represents a GUI window in a particular state. Arrows denote event triggered transitions between states. Each event has an associated condition and a sequence of actions. Therefore, a transition is only performed when the related condition is verified and the associated actions are then executed. The actions are represented by the list of numbers associated with each event.

The type of diagram in Figure 3.2 enables analysis of the dialogue supported by each application window. Although this is a very simple example, there are several conclusions made after analysing this particular source code abstraction. For example, the login window is composed by two states, the initial state, *state 0*, and *state1*, the state the application has after being initialized. As illustrated on Figure 3.2 the final states can be "end", which means the end of the application, or "close" which implies the window closing, but not the application end.

Moreover, by analysing the transitions between states, it can be concluded that the event Ok can trigger two different transitions, depending on conditions *cond2* and *cond3*. After pressing Ok, the application may be in the same state or it can close the login window. By analysing the conditions we can determine

[1] http://www.graphviz.org/

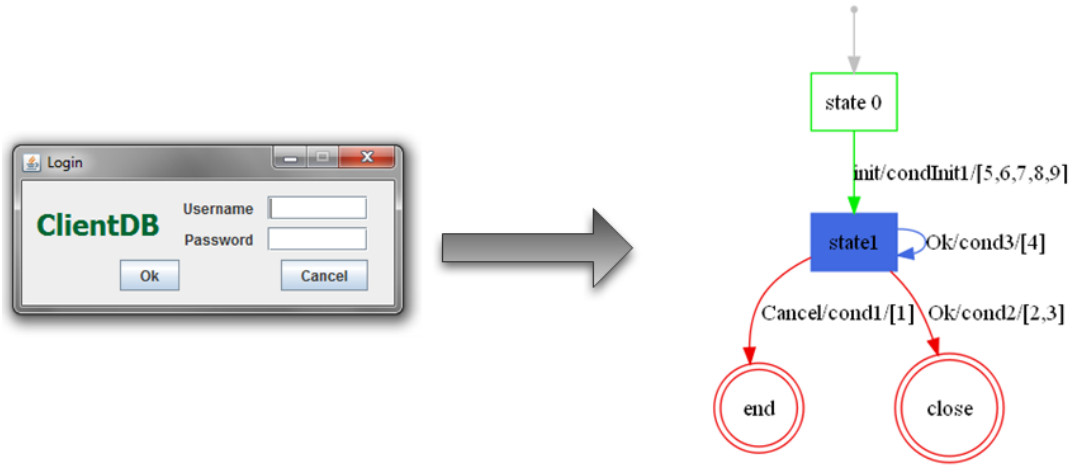

Figure 3.2: GuiSurfer's execution over a Java/Swing application

wheter the interface is predictable or not, and under which conditions. For example, if *cond2* and *cond3* are not mutually exclusive, then pressing Ok will have an unpredictable effect on the interface.

The following section explains how the model depicted in Figure 3.2 was generated.

3.3 GuiSurfer Implementation

The GuiSurfer tool encompasses three executable tools. Namely: FileParser, AstAnalyser and Graph. These tools were all implemented using the Haskell programming language. The executables receive a few obligatory arguments. These arguments, described in detail below, are important for defining the focus of the analysis.

As an example, to apply GuiSurfer over the login window of Figure 3.2, the following code is necessary:

```
>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"
```

The first tool, **FileParser** is responsible for parsing the source code object of our analysis. Since this is the only tool that only requires one argument, the file we want to parse, merely the command "FileParser Login.java" is needed for its execution. Afterwards, a file named "Login.java.ast" is created with the AST obtained from the Login class.

The next tool used is named **AstAnalyser** and slices the AST, therefore retaining only the interface layer related branches of the tree. It is composed by a slicing library, including a group of transversal functions, thus enabling the traversal of any AST. This library is composed by the files SlicingX.hs and GuiX.hs. SlicingX is composed by the generic slicing functions. For example, a function to slice an AST to find all elements that match a specific constructor given as a parameter. GuiX uses the general slicing functions from SlicingX.hs and contains more specific slicing functions. For instance, to slice all the existing expressions in an AST, in Java is done by using the "*Exps*" constructor.

AstAnalyser is executed with three parameters, namely: the AST file; the entry point in the source code, in other words, the method where the tool starts its analysis; a list with all widgets the slicing process will focus upon, and the relevant windows the target window will interact with, if this is not known all the application windows may be inserted.

The simplest command needed to execute this tool with the Login class would be: "AstAnalyser 'Login.java.ast' 'main' 'JButton,Login,MainForm'". Meaning that we want the process to begin in the main method, and only information related to "Jbutton" should be considered. Moreover, only the transitions with the MainForm window were identified. After the execution of this command, the GUI layer is extracted from the Login AST. Consequently, the AstAnalyser tool produces three files, namely: "initState.gui", "eventsFromInitState.gui" and "windowName.gui", containing the initial state information, the events that occur

from the initial state, and the target window name respectively.

The last executable, the **Graph** tool, receives the files generated from AstAnalyser, i.e., "initState.gui","eventsFromInitState.gui" and "windowName.gui", as parameters. Moreover, it further receives, as parameters, the names of windows composing the application and the application name, which in this example was "ClientDBjava". It therefore creates a few meta-data files, such as: "actions.txt", "events.txt", "conds.txt", where the action, states, events, conditions retrieved from the source code are stored. This tool also produces the files "GuiModel.hs" and "GuiModelFull.hs" with GUI specifications in the Haskell programming language.

The GuiModel produced after the execution of GuiSurfer over the Login window is the following (compare with the state machine in Figure 3.2:

```
guimodel :: GuiModel
guimodel = fromList
[
(("Cancel","cond1"),[1]),
(("Ok","cond2"),[2,3]),
(("Ok","cond3"),[4]),
(("init","condInit1"),[5,6,7,8,9])
]
```

This window has three events, Cancel, Ok and init. The init event is a specific event that represents the window initialization process. Each event has the related conditions, and a reference, defined as numbers to the connected actions. For instance, `(("Cancel","cond1"),[1])` means that the event "*Cancel*" has a condition, which GuiSurfer automatically named as "*cond1*", and executes the action referenced by the number 1. The AST slices that corresponds to condition *cond*, and action *action1* is held in the "actions.txt" and "conds.txt", respectively. Therefore, all the visual information available in Figure 3.2 originates from the textual representation meta-model defined in this GuiModel.hs file.

When performing an analysis of a system, all three tools are executed through a certain order, that is, FileParser followed by AstAnalyser and afterwards Graph. Finally, there is the possibility to generate the visual models, such as the model

on the right side. These models are created from the meta-models using the GraphViz tool.

As stated at the beginning of this chapter, GuiSurfer is capable of analysing both Java Swing code (as ilustrated with the example), and WxHaskell code. In the remainder of this thesis, its extension to deal with GWT and JavaScript will be studied.

Chapter 4

Google Web Toolkit

Google Web Toolkit (GWT) (Dewsbury, 2008) is a technology that provides a Java-based environment for the development of Web applications. As it can be deduced by its name this technology was developed by Google. The first version of GWT was released in 2006. This chapter refers to version 2.0, the latest version at the time of writing.

GWT is a set of development tools, programming utilities, and widgets that enables a programmer to create Ajax-based rich internet applications. Furthermore, GWT aims to make the coding of RIAs as simple as possible while allowing interaction with existing JavaScript code. The goal is to make it easy to develop complex cross-browser applications. To this end, GWT provides a set of ready-to-use user interface widgets that can immediately be used to create new applications. Moreover, it also provides a simple way to create original widgets by combining the existing ones. Developing the application in the Java language, allows GWT to bring all of Java's benefits to RIAs. Since GWT produces a JavaScript application, it does not require browser plug-ins additions, and there is also no need of possessing an application server if the applications comprehend just the client-side.

GWT was chosen as the first RIA technology to be analysed because it comprises Web applications and the Java language as the programming environment. This is important because the GuiSurfer tool has a module that works with Java Swing, therefore easing the development of the new module focusing on

GWT. Amongst the various frameworks to create Web applications that use Java language we chose to focus on GWT due to its popularity. Other open-source frameworks like ZK[1] could have been the target of this research. In ZK the source code is written Java and the final result is an Ajax Web application (Staeuble and Schumacher, 2008). Additionaly, there is a visual designer to aid in the application development called ZeroCode[2]. Furthermore, ZK has it's own markup languages designated ZUML (ZK User Interface Markup Language). This framework is more server based, as most of the information will originate from the server, and as such was considered less interesting.

4.1 Features

This section describes the main advantages and features of developing a Web application using GWT, such as: GWT development using the Java language; GWT's integration with JavaScript; GWT's widgets; JUnit integration; client-server communications and history management and internationalization.

4.1.1 Java Language

By making the development be coded in the Java programming language, GWT inherits many of Java benefits. One of these benefits is that it enables a better application management than most RIA technologies thus making GWT a proper solution for the development of Web applications with significant size. This occurs as Java is an object oriented language, therefore allowing Java projects to generally be easy to communicate and comprehend.

Another advantage of using the Java language arises as it enables using any Java Integrated Development Environment (IDE) in the application development. Java IDEs improve development as they provide several tools to help developers, for instance, code completion or error checking, and even tools to help debugging the application. Despite GWT being often associated with the Eclipse IDE[3]

[1] ZK - http://www.zkoss.org/
[2] ZeroCode - http://sourceforge.net/projects/zerokode/
[3] Eclipse - www.eclipse.org

(probably because the GWT web page explains how to develop with Eclipse), other Java IDEs, such as Netbeans IDE[4] or IntelijIDEA[5] can also be used. In the context of this work they were all tested and work without problems with GWT, each with its unique advantages just like in traditional Java applications development.

JavaScript is a loosely typed language, meaning variables can be declared without a type or not declared at all. Java, however, is a strongly typed language, as all variables have a defined type. Therefore, by using the Java language, one benefits also from Java type checking, decreasing the number of application errors. There is also an improvement on JavaScript debugging, as errors are noticed in compilation time instead of execution time. This happens as JavaScript is an interpreted language whereas Java is a compiled language.

4.1.2 JavaScript Integration

Though the codification in Java is useful, there is sometimes the need to write direct JavaScript calls. GWT addresses this need with the JavaScript Native Interface (JSNI) that permits the mix of JavaScript code into the Java source code and vice-versa. It also allows throwing exceptions across Java/JavaScript boundaries, and reading and writing Java fields from JavaScript. Notice that exceptions should be solved ("catched") in the JavaScript code portions.

This integration is achievable because the GWT compiler can combine native JavaScript code with the JavaScript code that is generated from Java. However, there is the risk that writing JSNI code could introduce memory leaks, or problems in the cross-browser application domain, as the JavaScript portions of code do not have the same protection as the Java code segments.

4.1.3 Widgets

GWT comes with a set of commonly used widgets. The widgets are very similar to the AWT/Swing widgets, consequently making it easy for GWT interfaces to look like a Java Swing desktop application. In Appendix A a table is presented

[4]Netbeans - http://netbeans.org/
[5]IntelijIDEA - www.jetbrains.com/idea/

that depicts the different widgets available for both Swing and GWT. As it's perceptible, almost all Swing widgets have a GWT correspondent, except widgets that are not relevant to the Web paradigm.

Some widgets with a greater complexity degree are available on the incubator or on different libraries on the internet. For instance, GWT-EXT, SmartGWT, or gwt-dnd (a library whose purpose is to add drag and drop features to GWT).

4.1.4 JUnit integration

Unit testing is a method in which the several smallest pieces of testable software are isolated and tested in order to determine whether they operate as supposed. There are a number of unit testing frameworks for JavaScript, for example, JsUnit[6] (Hieatt and Mee, 2002). However, GWT applications are developed in Java, so ideally a unit testing framework for that language should be used.

The most popular unit testing framework for Java is JUnit[7]. JUnit is a java tool that helps building and organizing automated tests for the application. To help testing the applications GWT provides JUnit integration.

To this end, GWT utilizes and extends the JUnit framework in order to supply a method to test AJAX code as simply as any other Java code. The GWT framework provides a *GWTTestCase* base class that extends from *TestCase* class in the JUnit testing library, and that can be used to define test cases. Therefore using GWT leads to the creation of a final JavaScript product but simultaneously it is also a testing solution over the built application.

4.1.5 Client-Server Communications

In order to communicate with the server, GWT provides several different options to send and retrieve data. A Remote Procedure Call (RPC) implementation enables connection to a Java servlet. Therefore, invoking methods on the server is as simple as making a local method call. However, if there is no possibility of running Java on the backend, communications can be made via Hypertext Transfer Protocol (HTTP), by providing generic HTTP classes to handle requests, and

[6] JsUnit - www.jsunit.net
[7] JUnit - www.junit.org

JSON and XML client classes to process responses.

The correct use of the client-server communications, by using asynchronous calls, makes it possible to separate the UI logic into de client and the Business logic into the server. This is one of the factors that can be used to improve the performance of the application, reduce the web server load, and even present a better user experience.

4.1.6 History Management and Bookmarking

One of the main Ajax problems is the browser's history problems it usually creates. An Ajax application tends to have issues when the user presses the "Back" button, because a dynamic page does not register the assynchronous calls actions in the browser's history. Therefore, pressing the "Back" button may cause users to navigate into places they were not expecting.

There are a number of workaround solutions to this problem that usually require the creation of a hidden frame, and some scripting. However, GWT already addresses this problem by implementing GWT's *HistoryListener* interface and its *onHistoryChanged* method.

Moreover, GWT's history support also solves bookmarking issues that sometimes arise. Ajax applications bookmarking issues occur when for example a web page content is dynamically changed, however, the URL remains the same. Hence, not allowing the bookmarking of a particular state of the application.

4.1.7 Internationalization

Another GWT feature is that it supports a set of techniques whose goal is to assist with internationalization. In other words, application adaptation to different languages or cultures. Those techniques can be divided in two groups:

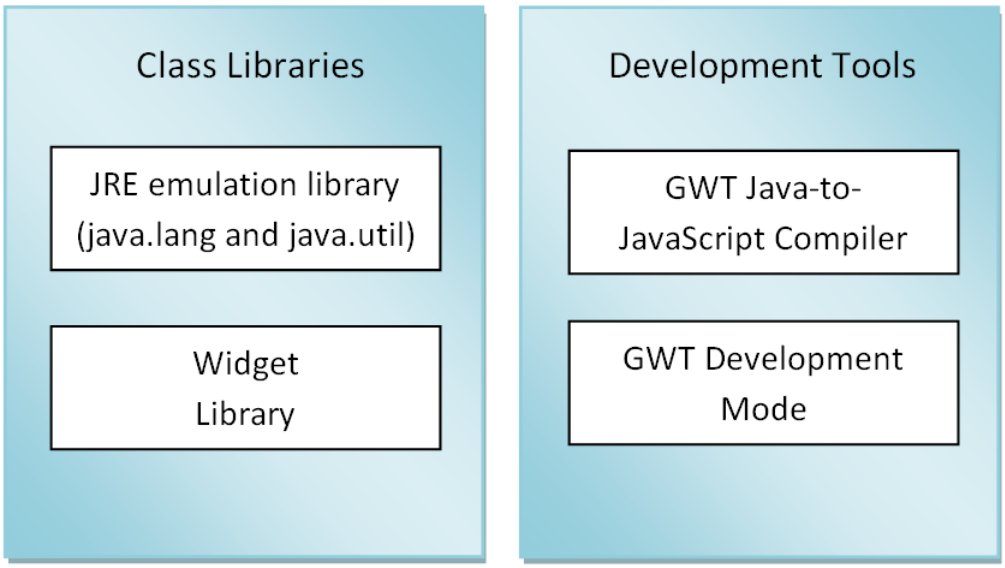

Figure 4.1: GWT Architecture

- Static string internationalization works at compile time and is achieved by implementing the Constants or Messages Interface.
- Dynamic string internationalization is supported by the class Dictionary and does not need the applications recompilation, since it works at runtime.

4.2 Architecture

GWT has an architecture composed of four elements that can be grouped into two major groups as seen on Figure 4.1: the development tools, which include the compiler and the development mode; and the class libraries, which contains the JRE emulation library and the widget library.

4.2.1 Java to JavaScript Compiler

The Java to JavaScript compiler is the module responsible for the compilation from the Java programming language to the JavaScript language. However, the GWT compiler does not perform a simple compilation only; it also performs several code optimization tasks throughout the compilation process. The most

significant optimization tasks are:

- Dead code elimination: only classes and methods that are actually used in the application are translated into JavaScript.
- Method calls in-lining: a method call is replaced by the real code of the method being called, thereby eliminating the overload on the linking phase.
- String interning: an optimization method that stores only one copy for each unique string object, with this copy being shared by all the equivalent strings.

4.2.2 Development mode

In order to ease GWT applications' development, as of version 2.0, GWT introduced "development mode", which enables the utilization of any (supported) browser to view the page being debugged. However, to actually be able to see the application, it is necessary the use of a browser plugin, called Google Web Toolkit Developer Plugin. Therefore, this GWT mode only works on browsers which allow the addition of plugins. At the time of this publication, GWT development mode supports popular browsers' recent versions such as Internet Explorer, Firefox, Google Chrome and Safari. This mode allows for a single debugging session to test all the different browsers at the same time.

GWT versions before 2.0 had a different process to ease the debugging. It was called hosted mode. Hosted mode used to embed a modified browser into the Java Virtual Machine(JVM) to allow running the Java version of the application during development. In this mode, the application runs as Java in the JVM without compiling to JavaScript, thus improving the time it takes to execute the application. Consequently, GWT presents the final application result to the user, in an emulation environment.

4.2.3 JRE Emulation library

Since GWT does a Java to JavaScript conversion, there is also the need of supporting the Java libraries. In order to solve this need, GWT has a library whose

purpose is to emulate the Java libraries. However, GWT does not support all the Java libraries, only the most important ones, namely:

- java.lang
- java.io
- java.util

Nevertheless, it is important to emphasize that some classes have features that may subtly differ from the features of the original Java classes.

4.2.4 Widget Library

The widget library is a GWT library that comprises a set of widgets whose goal is to ease client-side UI development. GWT provides the most popular widgets used on Web applications. For example, the *Button* widget to create button elements, or the *Table* widget to produce table elements.

Widgets are the elements responsible for the user interaction with the application, and are incorporated in panels. Panels determine the widgets arrangement on the page. As an example of a panel, the *HorizontalPanel*, is used to position widgets horizontally.

4.3 Frameworks and libraries

There are several GWT frameworks and libraries available whose goal is to ease the building of GWT applications. Libraries provide additional functionality or classes to developers. For example, a few widgets are available only on the GWT incubator[8]. The incubator is a Web site where widgets are managed by a Google team, and available to everyone, before they are added to the core toolkit in normal releases. Moreover, there are also other libraries with additional widgets, for instance: gwt-ext[9], Smart GWT[10] or MyGWT[11]. As shown on Appendix A

[8] GWT incubator - http://code.google.com/p/google-web-toolkit-incubator/
[9] gwt-ext - http://code.google.com/p/gwt-ext/
[10] Smart GWT - http://code.google.com/p/smartgwt/
[11] MyGWT - http://www.gwtsite.com/mygwt-widget-library/

(page 95), there are some widgets available on Java Swing that are only available in GWT through the use of the libraries.

Furthermore, there are libraries which have other purposes besides adding new widgets to GWT. Some examples include: gwt-dnd[12] a library which adds drag and drop support to GWT; Gilead[13] a library that eases the use of persistent entities with GWT; GIN (GWT Injection)[14] a library that enables automatic dependency injection to GWT client-side code. The term dependency injection was first applied by Fowler (2004) and it is a design pattern in which objects are arranged by external entities, thus reducing the number of factory classes in the Java code.

Besides libraries, a number of frameworks is also available to ease the development of GWT applicaations. IDEs like Netbeans provide WYSIWYG (What-You-See-Is-What-You-Get) editors to build Java Swing applications, where the application is build by manipulating the widgets layout without the need of coding the process. The developer simply drags and drops the widgets into a drawing area, and the framework automatically produces the source codes. The Netbeans editor was used to create the Java Swing applications tested with GuiSurfer. In order to have a similar process to produce GWT applications, there is a framework called GWT Designer[15]. The only downside of this framework is that it is paid. One can however freely evaluate it. The company that made this product also has a Java Swing and Java SWT designer. All three tools are plug-ins used within the Eclipse IDE.

Also worth mentioning here is a framework named Vaadin[16] (Grönroos et al., 2010) which enables the development of applications at an even higher abstraction level. Vaadin is built on top of GWT and adds server side validation to it's actions. Therefore, it is considered to be a server-driven framework, while GWT is client-driven as it works only on the client side. It also has a WYSIWYG editor, however at the moment it is marked as experimental. Any GWT component can be used in Vaadin, however the available Vaadin components are already

[12] gwt-dnd - http://code.google.com/p/gwt-dnd/
[13] Gilead - http://noon.gilead.free.fr/gilead/
[14] GIN - http://code.google.com/p/google-gin/
[15] GWT Designer - www.instantiations.com/gwtdesigner/
[16] Vaadin - http://vaadin.com/home

numerous and appropriate for most developers' requirements.

4.4 The Agenda example implemented in GWT

One of the applications thoroughly tested and used by the GuiSurfer team as an example is a simple interactive agenda of contacts. As depicted on Figure 4.2 the application is composed by four panels: *Login*, *MainForm*, *Find* and *ContactEditor*. Since this application was developed in Java Swing and WxHaskell, we build it in GWT in order to allow an easy comparison of the generated models.

A panel is the GWT equivalent name to a Desktop application window, as the GWT application is executed in a browser web page. Obviously the application begins with the Login panel (Figure 4.2, top-left panel), used to authenticate the users of the application. In order to authenticate himself, a user must fill his username and password and press the *Ok* button. Afterwards, if the information is correct, the Login panel is replaced with the Mainform panel. However, if the information is wrong, a warning pop-up appears, and the input fields are cleared. Likewise, the *Cancel* button, when pressedm also clears the input fields.

The MainForm panel (Figure 4.2, top-right panel) is the main application panel, it has the list of the various contacts, and buttons that enable users to find or edit them. The *Exit* button allows a user to logout, therefore returning the application to the Login panel.

When the *Find* button is pressed, the Find panel (Figure 4.2, bottom-left panel) appears. This panel allows a user to search for a specific contact by name. The search results appear afterwards in the lower panel section.

By clicking the *Edit* button, the user navigates to the ContactEditor panel (Figure 4.2, bottom-right panel). As the name implies, this panel enables changing a contact's information. Namely: first name, last name, title, nickname and the various e-mails. Since each contact can have several e-mails, they are presented in a list. The Add, Remove and Edit buttons are used to modify it. If the e-mails list is empty the remove button is automatically disabled.

The GWT version of the application was built aiming to be as similar as possible to the interactive agenda of contacts in Java Swing presented in Silva et al. (2009) (both in terms of presentation, and in terms of coding structure),

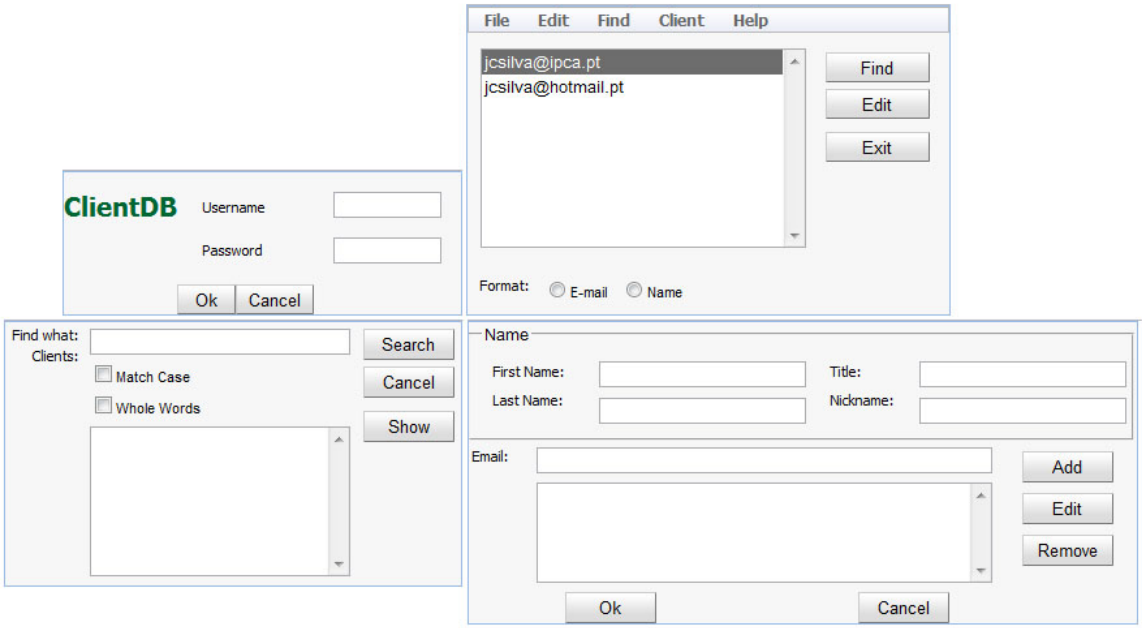

Figure 4.2: GWT Contacts Agenda Application

thus showing GWT's applications similarity with Java Swing applications, despite being Web applications. For example, the GWT application could have been written all in the same Java class. To maintain GuiSurfer's approach of analysing one Java class at a time, we divided the code into classes with the same name as the Java Swing application. In order to achieve this, the GWT application contains a class, named *MainEntryPoint*, which implements the *EntryPoint* interface. That is the class where the application begins, similar to the main classes on Java applications. In this class we added the four classes that represent each window, and set their beginning visibility. Thus, the login class stayed visible, and all the others were set invisible.

This *MainEntryPoint* class is the application manager, thus it has a method to change each window visibility. As an example, to change from the login window to the mainform window only the following code is necessary in the login class.

```
this.setVisible(false);
view.setMainForm();
```

The class visibility is set to false, and then the method setMainForm on the

MainEntryPoint class is called, that method just sets the mainform panel to visible. In order to call the *MainEntryPoint* class, each panel defines a private variable of type *MainEntryPoint*, named *view*, which is set to point towards the main class. However, notice that it is not possible to set the Mainform panel to visible from within the Login class, since the panell are unaware of each other Thus, the need of having a *MainEntryPoint* class to act as a manager.

An important note is that the possibility of leaving more than one window opened simultaneously was not implemented. Despite being possible to open several frames to edit multiple contacts at the same time in a RIA, that approach is not usual, and it would require a completely different Contacts Agenda GWT application, with a complexity far greater than what we intended to examine. Our GWT application only turns panels visible/invisible to simulate the Java Swing or WxHaskell change between windows. As there is only a panel per window, there can only be one window opened at a time. This means that models that depict simultaneous windows in the application will not be created. For instance, state machines whose purpose it also to discover the number of windows opened at the same time in a state are not analysed.

4.5 Reverse Engineering GWT

In this section the applicability of GuiSurfer to GWT code is discussed. Adapting GuiSurfer to reverse engineer GWT code was a two step process. In the first step, an assumption was made that the GWT code would be structured as similarly as possible to the Java Swing code. This was helpful to assess the viability of using GuiSurfer on GWT, and to identify a number of small improvements to the tool that were needed. Using these assumptions a new GWT module for GuiSurfer was developed.

On a second step, the GWT module was extended in order to loosen the above restrictions as much as possible, and generally improve support for panel handling. For instance, GuiSurfer requires a method name as a parameter, that defines the beginning state of the application. Usually on Java Swing that is the main method. However, in GWT some applications are defined completely within the class constructor, and can, in the worst case scenario, have no methods at all.

Therefore, in order for those applications to be correctly interpreted by GuiSurfer, a little code rearrangement is needed. That is, the creation of a method and the reallocation of the constructor code into that method.

GuiSurfer's architecture remained exactly the same, as shown on Figure 3.1. Changes performed to extend GuiSurfer to a new programming language, specifically GWT, didn't reflect on architectural alterations. This is a good indication that GuiSurfer can be easily adapted to new languages, as the core structure remained identical.

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

A second issue arose related to differences in the functionality of both toolkits (Swing and GWT). Since a GWT application is a web application, the closing window (panel, in GWT) actions available in Java Swing are not present. Closing a web application is an unusual action, and thus there is no direct support in GWT for doing it. Nevertheless, it can be achieved by invoking native JavaScript. A third issue occurred in detecting a change from a window/panel to another. In Swing this is achieved by invoking the "dispose" method on a class. In GWT this is accomplished by manipulating the visibility attribute of the panels. Again, changes were introduced to address this situation by changing the method GuiSurfer analyses to the "setVisible" method.

As GWT is written in the Java programming language, the first GuiSurfer's phase, FileParser, remained exactly the same. GuiSurfer's second executable, AstAnalyser, had a few alterations. Those changes were mainly performed on the AstAnalyser import file named "GuiX". GuiX contains the major GuiSurfer analysis tools. Thus, most of the issues explained above were solved here.

Table 4.1 depicts the total number of language dependent lines of code for

Table 4.1: Total language dependent lines of code

Module/Language	Java Swing	WxHaskell	GWT
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)

the five GuiSurfer modules. The column Java Swing contains the number of lines each Java Swing module has. As Java Swing was the first approach made by GuiSurfer, the other languages were made by re-targeting this initial approach. Hence, the other columns, specifically, WxHaskell and GWT, contain the number of lines each GuiSurfer module has. Additionaly, between parentheses, there is also information on how many lines were changed. Notice that, lines changed include code lines erased and source code lines altered.

As an example, the FileParser module developed for Java Swing has 54 source code lines. FileParser for WxHaskell has 26 lines, and 41 lines from the Java Swing source code were altered (mostly deleted). GWT's FileParser has 54 lines and no line was changed, thus it is identical to the Java Swing file.

Files were compared using a visual file comparison tool named ExamDiff[17]. Furthermore, to enable a better comparison, and since the tool does not recognize Haskell comments, these were all removed, or copied into both languages files, in order to obtain good results on the source code changed.

From this table, we can conclude that GuiSurfer is indeed a retargetable tool, as new programming languages analysed incur in few source code alterations. Moreover, and focusing on our approach to GWT, just a few line codes were changed. Also this table depicts that GuiSurfer still can have improvements in it's language dependent and language independent, because the module Graph.hs is part of the language independent phase, thus should not have been necessary to alter it to work with other languages. In GWT this happened because of the button event method which was different from the Java Swing method. To solve

[17] ExamDiff - http://www.prestosoft.com/edp_examdiff.asp

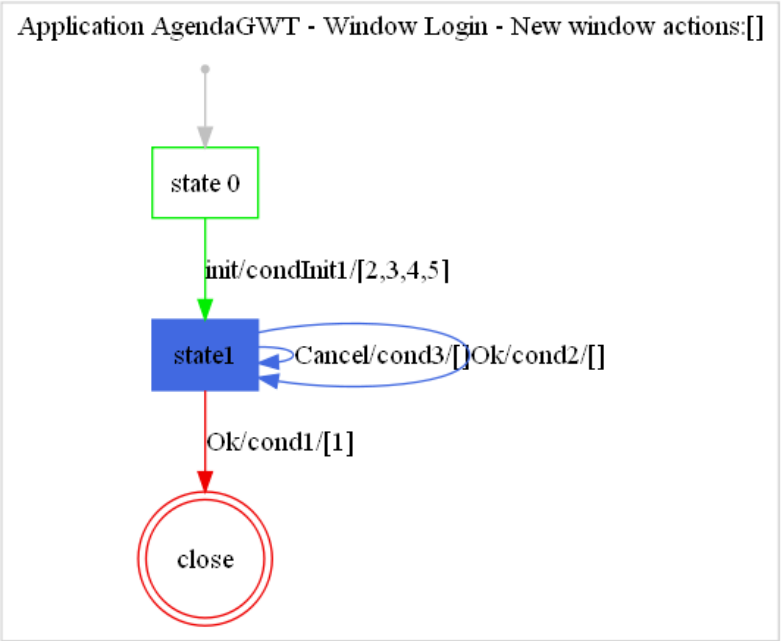

Figure 4.3: GWT Login window FSM

this, the method name to be analysed as the event trigger, should be passed as a parameter to the Graph application.

4.6 Results

The execution of GuiSurfer over the Contacts Agenda application produced various GUI behaviour meta-models, such as GuiModel and GuiModelFull. In order to perform reasoning over these models, as well as to compare these models with the models generated with the other programming languages, the models are going to be presented as finite state machines generated with the GraphViz tool.

Figure 4.3 depicts the FSM generated after executing the GuiSurfer tool over the GWT login class. The GWT login application has an inicial state, *state0*. *State0* progresses to *state1* when the *init* event is triggered, which represents the application start. When the user presses the Cancel button, the state remains the same. However, when the user presses the Ok button and the credentials are correct, the panel closes, thus changing into the mainform panel.

In comparison with the FSM produced when executing GuiSurfer over the Java Swing login class shown on Figure 3.2, some differences are immediately noticeable. For instance, the GWT login application does not have an “end“ state. This occurs because it does not make sense for a web application to simply close and leave the browser with a clear page. Our approach was to design the GWT login application with a different functionality when the Cancel button was pressed, namely to clear the login window textboxes. Hence, the Cancel button activation maintains the application in the same state. This difference in the model is actually highlighting differences in the user interaction supported by two version of the application.

When the Ok button is pressed two different situations may occur. If the login is correctly validated (*cond1*), the application moves into the panel close state. If the login is not validated (*cond2*), the application sends a JavaScript alert stating "Username/Password not valid". This alert is not perceived by GuiSurfer as a new state, thus the panel state is maintained.

Some of the actions described above, for example a JavaScript alert, may be thought of as another state. GuiSurfer, however, as it is still a work in progress, has a particular way of defining states, namely, it makes the assumption that new states occur only when some interface widgets are enabled or disabled. There are many other interface alterations that could be recognized as new interface states. However, this is still being developed at the moment. Hence, for the time being, JavaScript alerts are not perceived as new application states.

GuiSurfer is also capable of producing models that reflect the various windows of an application. By giving each window’s name as a parameter to the Graph application, GuiSurfer merges the models from each window and assembles them into a few different models.

For example, the model in Figure 4.4 depicts the several windows states, and the number of events that might trigger transitions between them. All information present on this model is also present on the models generated for each window, as in Figure 4.3 for the login window. However, this model presents the same information in a different way, that is, instead of showing the various events, it depicts the number of events between each state. An example of this model for the Java Swing application was presented in Figure 4 in (Silva et al.,

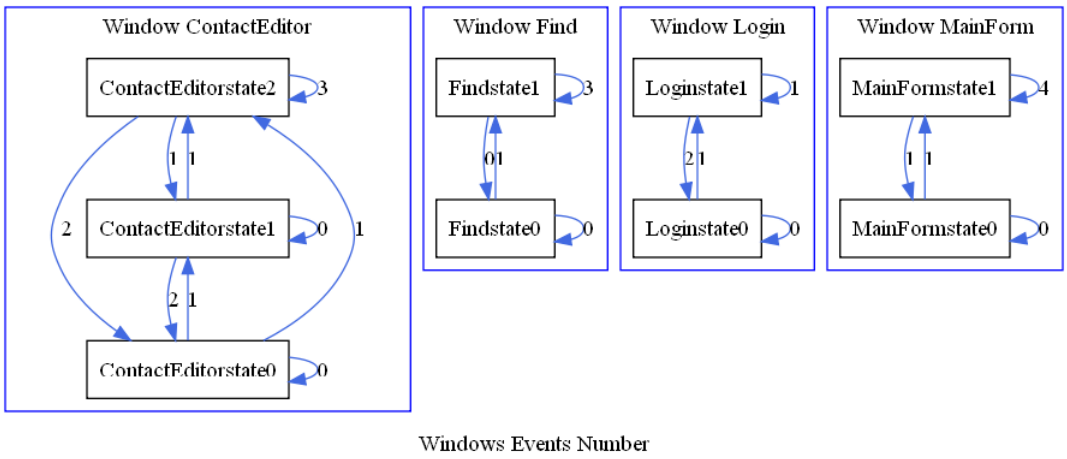

Figure 4.4: GWT Agenda number of events

2010b).

A model like this one enables us to analyse several aspects of the subject application related to the complexity of the interaction with each of its windows. For instance, one can immediately notice that ContactEditor is the only of the four windows that has three states. All the other windows are composed by two states. However, notice that cancel and exit states are not depicted in Figure 4.4 as they were not considered important to this model type specification.

This model is important as it shows one of the extensions currently being developed in GuiSurfer. Since GuiSurfer is applied over a single window, it is also important to encompass the entire application, that is to give an abstraction of all the windows behaviour and how they relate to each other. Thus, work is currently being made to generate models that depict all windows states and the transitions between different windows.

In order to generate the models from several windows, a new graphical tool is being used, namely, Graph-tool[18], a python module for manipulation and statistical analysis of graphs. Moreover, this tool allows the calculation of several metrics over the graphs produced. For instance, the shortest distance between vertices, this graph metric is very useful in user interfaces, as it expresses the minimal way for a user to perform a task. Furthermore, it enables the analyse

[18]Graph-tool - http://projects.forked.de/graph-tool/

of user tasks complexity, as longest paths represent complex tasks while shortest paths represent simpler tasks.

4.7 Using GuiSurfer in real-life applications

Through the various articles published about GuiSurfer, and through discussion about the tool with other researchers from this area, a recurrent question was identified: how well does GuiSurfer perform when applied on an application other than the ones we developed and tested ourselves. To that end, a website is currently being developed to enable users to upload their applications source code, and see the various models produced by GuiSurfer.

Other issue usually asked is if the target system needs adjustments before being able to produce satisfactory results. In other words, if GuiSurfer works with source code rearranged to a specific format only.

As far as GWT is concerned, we researched and applied GuiSurfer to various examples of GWT applications available on the internet. Through this research, some important GuiSurfer's vulnerabilities were discovered. Most of the examples which GuiSurfer had problems with were due to parser problems. For instance, the current parser being used for Java is for Java version 1.1. Therefore, it does not recognize Java annotations, and even stops the parsing whenever one is found. In several examples on the internet the annotation *@Override* is used whenever the element is intended to override a method declaration in the superclass. Hence, examples that contained this and others annotations were unsuccessfully tested with GuiSurfer. To attain good results, all annotations should be removed.

Another example of a problem that arose through the Java parser use was related to the variables. Variables that start with a '_' (underscore) character also give parse error, thus they have to be altered prior to GuiSurfer's execution over the application.

The next section describes an example of an open source GWT application and the results GuiSurfer produced by analysing it.

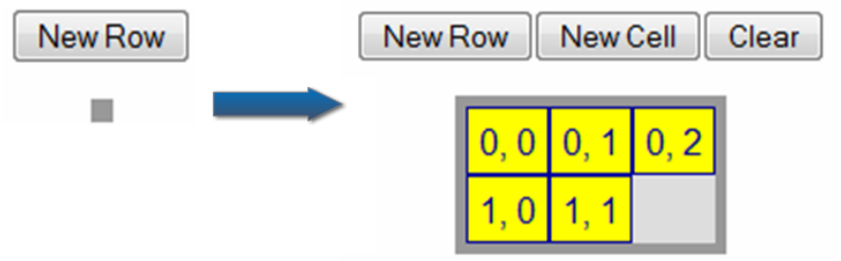

Figure 4.5: FlexTable application

4.7.1 Example

FlexTable [19] is an example of an open source GWT application available on the internet. It is a widget, based on an HTML table and has 62 lines of code.

The application is depicted on Figure 4.5. 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 clicked, the table has a new row, and the application set two more buttons visible, namely: *New Cell* and *Clear*. The button *New Cell* adds a new cell to the table, in the last created row. Each of the table cells has the value of its respective coordinate on the table. Thus, to create the table shown on the right side of the image, the following sequence of buttons was pressed:

```
New Row -> New Cell -> New Cell -> New Row -> New Cell
```

In summary, the *New Row* button increases the table downwards, adding new rows while *New Cell* increases the current row by adding new cells. Moreover, the button *Clear*, clears the table, thus returning to the initial, empty table state, as depicted on the left side of Figure 4.5.

We applied GuiSurfer to the source code of the application. However, on a first attempt it did not return a successfull result. The final model image just had the initial state, nothing more. After a close look at the code, the reason was spotted: in the source code, every button action was made using *ClickListener*. ClickListener is a method that was used in older versions of GWT. Newer versions use ClickHandler instead, hence, ClickListener is now deprecated. In the

[19] FlexTable - http://examples.roughian.com/index.htm#WidgetsFlexTable

application author's homepage, he refers that precise issue, that his applications were made to GWT 1.5, thus the use of the deprecated method.

Therefore we performed just that source code alteration, to change the buttons ClickListeners into ClickHandler. For example, button *downButton* was rearranged as shown below on the right side, the original code is on the left.

```
Button downButton = new
    Button("New Row", addACellDownListener);

ClickListener addACellDownListener =
    new ClickListener(){
        public void onClick(Widget sender){
            //button code here
        }
    };
```

```
Button downButton = new Button("New Row");

downButton.addClickHandler(new ClickHandler(){
    public void onClick(ClickEvent event) {
        addACellDownListener();
    }
});

public void addACellDownListener(){
    //button code here
}
```

After updating the code of all the application buttons, to be according to version 2.0. GuiSurfer was again executed on the source code. Results were better. All actions, events and conditions were discovered by the tool.

However, as discussed before, GuiSurfer was using the enabling and disabling of widgets to indentify new window states. Thus, the application buttons appearance and disappearance were not found as new states. To solve this issue, some alterations were made to the GuiSurfer tool, allowing it to trigger new states when alterations to the widgets visibility happened. This was done by changing the source code of the file *Graph.hs*.

After performing the alterations and executing GuiSurfer a third time, it produced the model as depicted on Figure 4.6. Notice that the method used as the starting point for the analysis was the method where the application starts, that is, *FlexTableDemo()*.

An analysis of Figure 4.6 enables several conclusions to be made about the Flex Table application. For instance, the interface has 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*).

It is important to emphasize the accurate definition of the two states, that is, *state1* and *state2*. As there are two buttons being set to visible or invisible,

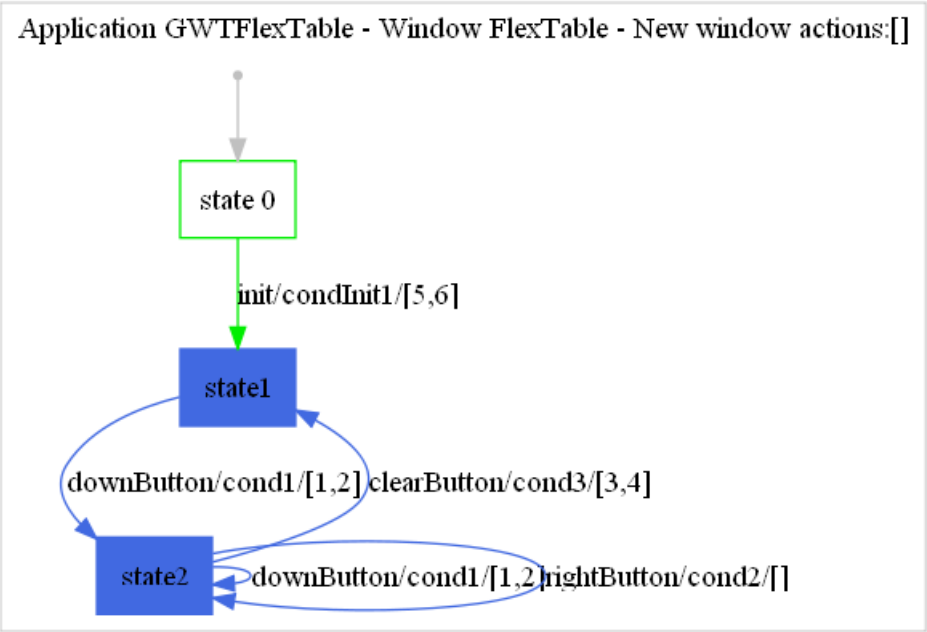

Figure 4.6: FlexTable FSM model

GuiSurfer could have created a state for each. However, GuiSurfer correctly analysed the application, and since the two buttons are always set visible or invisible simultaneously, only one state was considered. Because the application always has either one or three buttons, GuiSurfer accurately defined the two states.

Moreover, GuiSurfer properly identified the two different events after pressing the button *New Row*. If the table is empty, it will perform a transition to the following state, *state2*, otherwise it will remain in the same state. This aspect is also relevant because despite the source code of button *New Row* having the two buttons, *Clear* and *New Cell* set to visible, GuiSurfer analysed that the changing to visibility would just occur under certain conditions. Normally, a button would produce two events only if there is a conditional statement defined in it. This example enable us to conclude that GuiSurfer precisely defines two events even when there is no explicit conditional statement.

Furthermore, Figure 4.6 also depicts that there is no final states, that is, there are no ”close“ or ”cancel“ states. This would enable us to conclude that the Flex

Table application does not end and that there is just a single window in that application. Obviously, the application not ending relates to the fact that it is always enabled on that specific Web page, because Web applications can always end by changing the Web page currently being browsed.

In retrospect, the adaptation of GuiSurfer to a new programming language, specifically, GWT, was achieved sucessfully. Consequently, the necessary testings to achieve solid results were performed over two GWT applications examples. The main limitations of the new module are related to the parser limitations. These limitations should be overcome as a future work.

Chapter 5

Ajax

Asynchronous JavaScript And XML (Ajax) is a set of technologies combined for the purpose of creating highly interactive web sites and web applications. The term was first applied by Garrett (2005), in a paper where he grouped all the already existent technologies with the goal of achieving a higher level of interactivity in HyperText Markup Language (HTML), and named that collection of technologies Ajax. The technologies themselves were already available for many years, but their aggregation was only considered by few people previously (Hadlock, 2007).

The technologies in question are:

- XHTML and Cascading Style Sheets (CSS) to define the presentation;
- The Document Object Model (DOM) for dynamic display manipulation;
- XML and XSLT for data interchange and manipulation;
- The XMLHttpRequest object to handle asynchronous data calls;
- JavaScript as the language that combines all the technologies;

The idea is to make what is on the Web appear to be local by providing a rich user experience, offering features that usually only appear in desktop applications. By working as an extra layer between the user's browser and the web server, Ajax handles asynchronous server communications, submitting server requests

and processing the returned data. The results may then be integrated seamlessly into the page being viewed, without that page needing to be refreshed or a new one loaded. The end user does not notice these processes and therefore only observes a smooth and uninterrupted application.

One of the main advantages of Ajax over other RIAs is that there is no need to install tools or plug-ins, neither to run nor to develop an Ajax application. Another aspect of Ajax is that it has been widely accepted by the main industry companies, such as Google, Yahoo, Amazon, and Microsoft amongst many others.

In what concerns this work, JavaScript it the more relevant of all the techonologies listed above.

5.1 JavaScript

JavaScript appeared in the Netscape Navigator Web browser around 1995 as a scripting language that would enable basic validation features. It was first named as LiveScript. With Netscape Navigator 2, a browser that supported the inclusion of Java applets, Netscape altered the name LiveScript to JavaScript, as a Netscape publicity stunt. Thus, JavaScript is not related to Java, as the name seems to imply. The language gained significant popularity amongst Web developers and was therefore included in other Web browsers, such as Internet Explorer.

However, at the time, different implementations arose. For example, Microsoft developed JScript for Internet Explorer. In order to aggregate the various implementations, there was a need for a standard, cross-browser, scripting language. The major companies involved gathered, and defined a new scripting language named ECMAScript (International, 2009). Nowadays, all browsers scripting languages come from their implementations of ECMAScript.

Despite JavaScript and ECMAScript often being used as the same concept, a JavaScript application is composed of three parts (Zakas, 2009), namely:

- ECMAScript
- The Document Object Model (DOM)

- The Browser Object Model (BOM)

A thorough description of JavaScript is outside the scope of this thesis. Instead, the following subsections briefly describe each of these JavaScript components.

5.1.1 ECMAScript

JavaScript is an ECMAScript dialect. ECMAScript (International, 2009) was a compromise primarily between Netscape and Microsoft, to standardize their languages, JavaScript and JScript respectively. ECMAScript is object based, and its syntax resembles the Java language.

ECMAScript defines several aspects of the language, in order for its implementations to be standard, such as: types, values, objects, properties, functions, and program syntax and semantics. Moreover, an implementation must be able to interpret the Unicode Standard. All current browsers have ECMAScript implementations that follows ECMAScript guidelines.

ECMAScript is updated through the releases of new editions, which browsers implement as soon as possible. As of the time this thesis is written, the latest approved edition of ECMAScript is the fifth (International, 2009).

For an example of ECMAScript compliant code, consider the following JavaScript source code which assigns the text "alert" to variable a, and then sends an alert with that variable:

```
var a = "alert";
alert(a);
```

5.1.2 Document Object Model (DOM)

The numerous objects defined on a Web page, i.e., document, are arranged in a tree structure. As an example, consider a simple Web page:

```
<html>
  <head>
    <title> Example Web Page </title>
  </head>
  <body>
    <p> Testing </p>
  </body>
</html>
```

As the example shows, the content of a HTML page is usually started by an <htlm> tag, followed by the <head> and <body> tags. The tags are paired (cf. <html> and </html>), and each tag pair defines an HTML element. Inside these elements, other elements can be placed, therefore enabling the construction of more complex Web pages. Thus, the HTML language makes it possible to easily transform its source code into a hierarchy of nodes.

JavaScript considers each of the document's tree items to be an object. These objects are also referred to as tree nodes. Using JavaScript each node can be accessed, modified, added, removed, replaced. If a node is composed by an HTML element it is named an element node. If it is not, it is called a text node. Obviously, an element node can contain another element node or a text node.

For instance, the following JavaScript code, uses the DOM to create a button in an HTML page.

```
var button = document.createElement('button');
```

5.1.3 Browser Object Model (BOM)

The Browser Object Model (BOM) allows access and manipulation of a Web Browser. The BOM can be defined as the set of objects composing a browser window, therefore DOM is a subset of BOM. Using BOM, a Web page developer can interact with the browser.

However, as of this moment, there is no standard implementations for BOM, making it the only JavaScript part which differs when different browsers are used. The only aspect the different browsers converge on is having defined a window

and a navigator object. The other objects, methods and properties are specific to the browser used.

As BOM contains DOM, it is also composed by a hierarchy of objects. The higher level contains the window object, the object that represents instances of the browser window. Consequently, the following level contains the navigator, screen, history, location and document objects. The navigator object has information on the browser, like the browser name and version. A screen object provides knowledge about the user's screen, such as its dimensions or color depth. The history object contains the various Uniform Resource Locator (URL) the client browser visited. The location object enclose data on the current URL. Finally, the document object is the DOM.

As an example, the following JavaScript code uses BOM to send an alert containing the name of the current URL:

```
alert(location.href);
```

5.2 Reverse Engineering Ajax

A JavaScript application is very different from applications developed in previous GuiSurfer target programming languages. For example, a JavaScript Web application contains three main languages. JavaScript as the programming language, where the logic is present; HTML the mark-up language, where the visual elements of the application are defined; and CSS which contains the Web page layout. This huge difference implies a total restructuration of GuiSurfer's architecture, as instead of parsing one language, it needs to parse two languages, JavaScript and HTML. CSS is not considered because our focus is on the interface behaviour, not its layout.

Another important difference is that, although a JavaScript Web application may be completely contained in a single HTML file, it is usually divided into an HTML file, and one or more JavaScript files. Thus, GuiSurfer must be capable of extracting information from the two languages merged in one single file, or detached into several files.

The first step into extending GuiSurfer to handle JavaScript applications

was the research of existing HTML and JavaScript parsers. As the GuiSurfer application is developed in Haskell, the research was mainly focused on parsers developed in this programming language.

5.2.1 HTML parser

Choosing the HTML parser was a challenging task, as a number of different implementations were available. A number of parsers were considered, namely:

- HaXml, a set of utilities for XML in Haskell, which included an HTML parser and pretty printers for HTML.
- HXML, a parser designed for efficiency, consuming minimal memory.
- TagSoup, a library for parsing HTML; it parses well-formed, unstructured or malformed HTML.
- HXT, the Haskell XML Toolbox, introduces a different approach to parsing XML, based on arrows (Hughes, 2000) instead of filters. It includes TagSoup to parse HTML.

From all the above libraries the only one exclusively made for HTML was TagSoup. A first approach was tried with this library, thus trying to reduce the program total size by choosing the smaller library. TagSoup was simple to manipulate, just a couple of source code lines, and an HTML file was parsed. After parsing an HTML file with TagSoup a list of Tags is produced. However, a tree data structure would be more suitable.

A Tag is defined in Haskell as follows:

```
data Tag  str
 = TagOpen str [Attribute str]
 | TagClose str
 | TagText str
 | TagComment str
 | TagWarning str
 | TagPosition !Row !Column
```

A tag is composed by a marker such as TagOpen, TagClose, TagText, and by a String. If the tag is an open tag there is also a list of the possible attributes an opening tag can have. The final structure produced by TagSoup was not the kind of structure we expected. HTML is a declarative programming language, i.e., it specifies what needs to be done, but not how to do it. Specifically, HTML is a mark-up language, therefore it is composed by a set of tags. However, a list of tags, just like TagSoup produced, was not a structure we were looking forward to work with, because it would be hard to slice the HTML. Notice that, representing a Web page as a list of tags means that information regarding the hierachical structure of the page, induced by the nesting of HTML elements, is not explicitly represented.

For example, with the other programming languages GuiSurfer is applied over one window, or screen, at a time. Therefore, there was a need to slice only the equivalent of a window from an HTML file, in order to maintain the same philosophy, which was important in order to effectively reuse GuiSurfer. This requirement to slice the HTML prevented the use of the list of tags from TagSoup. For instance, in order to slice a *TagOpen "div"* we would have to search for the respective closing tag to extract that list portion. However, since a div tag can contain div tags, it would take a lot of computing resources to calculate the respective closing tag, for each slice we would perform.

Moreover, HTML is an easy language to convert into a tree, if each element is seen as a tree branch and the text inside tags is transformed into the tree leafs, a tree is easily generated. Consequently, we searched for a parser that produced a tree, therefore having a structure similar to the AST GuiSurfer creates with the other programming languages.

TagSoup did have a solution to our needs, there is a module contained in TagSoup called Tree, namely: *Text.HTML.TagSoup.Tree*. This module makes it possible to convert a list of Tags into a Tree structure, by creating a TagTree. A TagTree is defined by the following data type:

```
data TagTree  str
  = TagBranch str [Attribute str] [TagTree str]
  | TagLeaf (Tag str)
```

A TagTree is a simple HTML tree structure, it is composed by two elements: TagBranch and TagLeaf. A TagBranch has its description, the list of attributes and the TagTree it contains. This type of structure was more like what we were looking for. However, when executing this module, a warning was produced stating this Tree module was deprecated, and not quite ready to use. Nevertheless, using it over the simple HTML example presented in Section 5.1.2 produced the following tree:

```
[TagBranch "html" []
    [TagLeaf (TagText "\n  "),
     TagBranch "head" []
          [TagLeaf (TagText "\n    "),
           TagBranch "title" []
                 [TagLeaf (TagText " Example Web Page ")],
           TagLeaf (TagText "\n  ")],
     TagLeaf (TagText "\n  "),
     TagBranch "body" []
          [TagLeaf (TagText "\n    "),
           TagBranch "p" []
                 [TagLeaf (TagText " Testing ")],
           TagLeaf (TagText "\n  ")],
     TagLeaf (TagText "\n")],
TagLeaf (TagText "    ")]
```

The tree generated was what we were looking for, so before pursuing this approach, the deprecated warning was researched. After discussing this matter with the developers of the tool, it was concluded that the module was deprecated because it had not had too many users, and, as such, was not thoroughly tested. Rather than deprecated it would have been preferred to mark it as experimental.

This module was just composed by the tree data type, the function to transform between the list of tags into a tree called *tagTree*, and three functions to transform the tree, namely: *flattenTree* that transforms a tree back into a list of tags; *universeTree* a function that given a list of trees, returns those trees and all

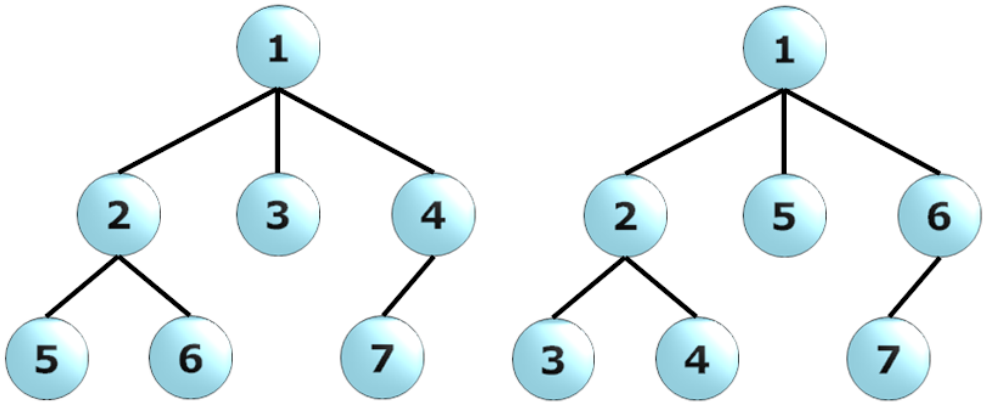

Figure 5.1: BFS and DFS traversals

the children trees at any level; *transformTree* that given a list of trees and a function, applies the function to every tree in a bottom-up manner. However, some important functions were missing, for example, functions to efficiently traverse the tree were not available.

Since there was a lack of functions in the TagTree module, we decided to create a library with functions that would prove useful to our future requirements. Moreover, an AST structure as produced by GuiSurfer to the other programming languages is significantly different from our tree of tags. For instance, a straightforward difference is the data types, a TagTree has the data type shown before while an AST data type in GuiSurfer is defined as a list of Strings.

An important function needed was a tree traversal function. That is, a function that enables visiting each node in a tree data structure, thus allowing to sequentially perform operations on an a tree, one node at a time. A tree traversal may be performed in several manners. In this library we built both a Breadth First traversal and a Depth First traversal.

A Breadth First Search (BFS) is an algorithm to traverse a tree or a graph that begins at the tree root, and then visits every tree nodes on each level, thus also being called level-order. A little example is presented in the left side of Figure 5.1, the numbers in each tree node depict the sequence of the tree traversal. As seen on the figure, the search progresses in a top-down manner, and it can be seen as searching sequentially each tree level. This type of traversal was implemented

as in some cases there is the need to find HTML tags that exist only in the tree top levels, thus there is no need to assess the low levels of the tree.

The Depth First Search (DFS) is an algorithm to perform a tree traversal that begins at the tree root and afterwards search each branch till a condition is satisfied, or a leaf is found. Subsequently, the algorithm does backtracking in order to search the rest of the tree. An example of how this algorithm works is depicted in the right side of Figure 5.1. This traversal is advantageous as it enables te search for a node in a tree and then when the node is found to search the rest of the tree, without visiting the rest of the branch.

For instance, we use DFS traversal to find all the *button* tags in an HTML file. As a button tag cannot contain another *button* tag, whenever we find a *button* node, the search in that branch stops and it continues in the unvisited branches, if they exist. This aspect of stopping the search in that branch is important as the rest of the branch does not need to be visited. This same example performed with a BFS would require the entire tree to be processed, in order to achieve the same results.

5.2.2 JavaScript Parser

According to our research, there were mainly two JavaScript parsers written in Haskell, namely HJS and WebBits. *HJS*[1], is a parser and interpreter that works with JavaScript 3rd edition, and has some additions from JavaScript 5th edition. *WebBits*[2] is based on JavaScript 5th edition, and unlike HJS, enables working with JavaScript embedded in HTML.

Despite WebBits being a parser with more recent updates, thus more current development, it also had less documentation. For example, the HJS package comes with a folder called *testsuite* with several examples of JavaScript files. Furthermore, the main file from the HJS package served to compile the examples on the testsuite folder. In order for the parser to produce the desired Abstract Syntax Tree (AST), a flag ShowAST is placed every time the parser is used.

Some experiences with HJS were performed, and even some functionality was

[1] http://www.haskell.org/haskellwiki/Libraries_and_tools/HJS
[2] http://hackage.haskell.org/packages/archive/WebBits/0.9.1/WebBits.cabal

Table 5.1: Number of characters generated by both parsers

Parser/AST	AST	AST without code positions
HJS	448	419
WebBits	560	163

developed to work with the resulting AST. For instance, to slice the AST and find a particular function, or to get a list with all the function names present in the JavaScript file. However, HJS has a problem: it does not work with embedded HTML. That is, Web applications that have their JavaScript directly in the HTML file. Although two examples of embedded functions in an HTML file are presented in the testsuite, both did not return any results after being parsed. As the goal of this work is to work with generic internet applications that use JavaScript, this parser was found unsuitable. Thus the approach leaned towards WebBits.

Moreover, WebBits produces a less complex data structure than HJS. For example, the following source code depicts a JavaScript file with one of the simplest functions possible. The function *print* receives a variable, named *txt*, as a parameter and it adds that variable to the document, that is, the web page.

```
function print(txt) {
return document.write(txt);
}
```

After executing both parsers over the function, HJS generated an AST with 448 characters, while WebBits generated an AST with 560 characters, as illustrated on Table 5.1. However, despite having more characters, the WebBits structure associates, to each tree node, the source code position of the source element, namely it's line and column. Whereas HJS just associates the source code position of each beginning line. After removing all the source code positions from the two ASTs, HJS' tree contained 419 characters while WebBits' tree contained 163 characters. Therefore, there is a considerable difference in the data structure of both parsers. In this work context the smaller data structure was suitable, since all the added verbosity HJS has was not necessary towards our needs.

Another reason to choose WebBits is that every element has associated the source code position, while in HJS only sentences have source positions. This is important because GuiSurfer also stores every element source position in GuiModelFull. Although generated models are not being directly influenced by source positions, GuiModelFull contains these references to be prepared for possible future requirements. For example, if the source code was reconstructed from our models, the source positions would be required.

5.2.3 Architecture of the Ajax module

Extending GuiSurfer to handle Ajax applications required a significant change in GuiSurfer's original architecture (described in Section 3.1). The main difference relates to the first GuiSurfer step, the parsing of the target application. While on other approaches, GuiSurfer's first step was using a parser towards the particular language source code, in an Ajax application GuiSurfer must use two parsers. Specifically, an HTML and a JavaScript parser.

An Ajax application may be built in several different ways. That is, the JavaScript source code may be embedded in the HTML, or it might be in one or more external files. JavaScript in an HTML file is identified by the use of the <script> element. The JavaScript source code is then placed inside the element. The following source code represents a simple javascript function, that sends and alert, using the <script> tag. Using this approach one can produce a JavaScript application entirely in a single HTML file.

```
<script type=''text/javascript''>
    function showAlert(){
        alert(``ALERT'');
    }
</script>
```

Another method is to store the JavaScript source code in a separate file or files. To this end, the <script> element has an attribute, *src*, that may be used to specify the name of the file where the JavaScript source code is present. As an example consider:

```
<script type=''text/javascript'' src=''example.js''>  </script>
```

The file in the example has the "*.js*" extension but this is not mandatory, the browser will not check for the extension. Thus, any extension can be used. Additionally, notice that when the *src* attribute is used, there should not be any JavaScript code between the <script> and </script> tags. Moreover, an HTML page can have multiple of these instances, thus the JavaScript source code can be dispersed in several files. Furthermore, the *src* attribute also enables the inclusion of JavaScript files from outside domains. In other words, the JavaScript files may be stored on an URL, thus *src = "http://www.example.com/example.js"* may also occur in an Ajax application.

Consequently, GuiSurfer has to address all these different methods of including JavaScript source code in an HTML web page. Thus, a new architecture was designed specifically for Ajax. GuiSurfer's Ajax module architecture is depicted on Figure 5.2 (compare it with Figure 3.1).

In this approach to Ajax, GuiSurfer starts by parsing the HTML file using the TagSoup parser, and an HTML tree is produced. Afterwards, the HTML file is also inspected to discover if there is embedded JavaScript source code. The WebBits parser is used to parse the HTML file and the external files and produce the JavaScript ASTs.

In this first approach the external files are inserted, in the command line, as parameters. In the future, the external files will be parsed as they are found in the HTML source code. After parsing the JavaScript if there is more than one AST, they are merged into a single JavaScript AST.

Afterwards, only the GUI behaviour parts are extracted from the application. GuiSurfer approaches to other programming languages is performed only one window at a time. In GWT this matter was also discussed, and the application was applied to a panel at a time.

5.2.4 Implementation

The implementation of the Ajax module represented three distinct challenges:

- identifying Ajax windows;

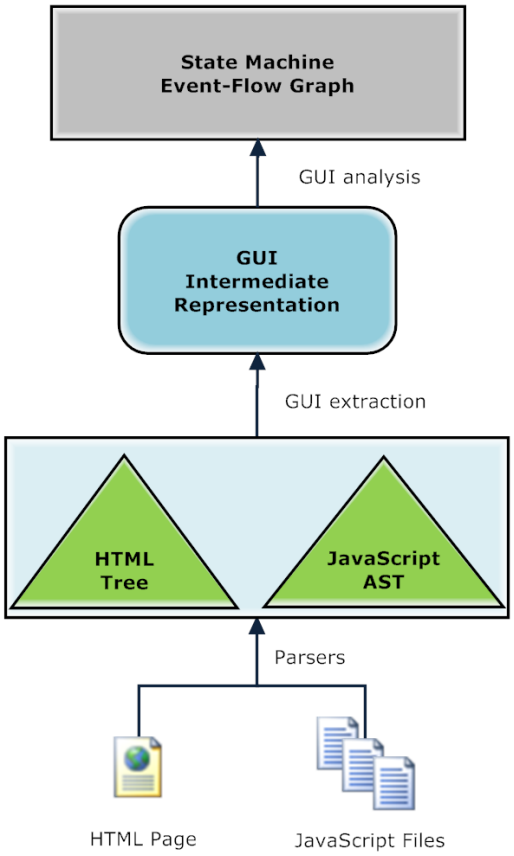

Figure 5.2: GuiSurfer's Ajax module architecture

- recognizing Ajax events;
- and identifying end states.

Each of them was addressed as described next.

Identifying Ajax windows

HTML is by nature page-oriented. The concept of a window or panel in Ajax applications is an artificial construct which can be implemented in several ways. For instance, a window can be an element with a tag which is afterwards set visible or invisible according to whether the window is active or inactive. Furthermore, a window can also be an element with a <table> tag. This is what

happens to the HTML souce code when it is generated with GWT, the window panels are mapped into tables in HTML.

As a window may be different elements, and those same elements may have the same tags inside (for instance, a window table may contain another table inside). The approach was to identify the windows through an identifier. The identifier was placed by using the *id* attribute, which assigns an unique identifier to an HTML element. Therefore, when building Ajax pages, whenever a window was defined, we used an identifier on the window defining element. As an example, the login window in HTML started with the following tag:

```
<table cellspacing="0" cellpadding="0" id="login"
 class="container" style= "width: 300px; height: 100px;">
```

Consequently, to retrieve the information of a single window, the window identifier is given as a parameter. Afterwards, the HTML tree is examined to discover the given identifier. When the identifier is found, only the HTML subtree of that identifier is analysed by GuiSurfer. By using this approach, the GuiSurfer module for Ajax has the same targets as the other GuiSurfer modules.

Recognizing Ajax events

In order to fetch the events of the application only, the module currently analyses the button elements of HTML. For example, the *Ok* button from the login window is defined as follows:

```
<button type="button" tabindex="0" class="gwt-Button"
 style="width: 45px; height: 23px;"
 onclick="loginOK()">Ok</button>
```

The button is defined by using the button tag, the text of the button indicates it's name. However, notice that several other ways of identifying a particular button are possible. For instance, using the *id* attribute, or using the *name* attribute. All these options should be evaluated to determine the correct button identifier.

Moreover, button actions are defined by the use of the *onclick* event attribute. It is important to refer that a *onclick* attribute receives a script as a value. In this case, the script points to function *loginOK()*, defined in the JavaScript source code. However, it is also usual to sometimes write the function between a *try/catch* statement. To also cover these situations, the cancel button was coded this way:

```
<button type="button" tabindex="0" class="gwt-Button"
 style="width: 60px; height: 23px;"
 onclick="try{loginCancel();}catch(ex){}">Cancel</button>
```

In order to obtain only the function name from the button, regular expressions were used. Regular expressions enable powerful, flexible and efficient text processsing (Friedl, 2006). To use regular expressions within Haskell, the module *Text.Regex.Posix* was imported. As an example, the Haskell source code to retrieve functions with no arguments is the following:

```
reg = "[a-zA-Z0-9]*[(][)]"
fnames :: String -> String
fnames x = let y = x =~ reg :: String
           in [c|c<-y,c/='(',c/=')']
```

This code defines a variable *reg* which contains the regular expression as a String, and a function named *fnames* which receives a String and returns the String after being evaluated through the regular expression. Notice the special operator $=\tilde{}$ which is the function defined in *Text.Regex.Posix* that enables the interpretation of the *reg String* as a regular expression. The above regular expression, defined as *reg*, matches all alphanumeric characters with an opening parenthesis followed by a closing parenthesis.

There are other attributes that can be used in a button tag to refer to possible actions triggered by the button. The event attributes supported by the <button> tag and the actions that trigger each attribute are presented on Table 5.2. All these different event attributes have to be considered to obtain the events and the behaviour from the target application. After the HTML elements are discovered,

Table 5.2: Event attributes for a <button> tag

Attribute	Attribute trigger
onblur	an element loses focus
onclick	a mouse click
ondblclick	a mouse double-click
onfocus	an element gets focus
onmousedown	mouse button is pressed
onmousemove	mouse pointer moves
onmouseout	mouse pointer moves out of an element
onmouseover	mouse pointer moves over an element
onmouseup	mouse button is released
onkeydown	a key is pressed
onkeypress	a key is pressed and released
onkeyup	a key is released

they are mapped into the JavaScript functions they are associated to in the JavaScript ASTs.

Afterwards, the JavaScript AST is analysed to retrieve relevant information. For each function, the various conditions are searched for, and each is associated with a unique identifier. The identifier is related to their position on the source code, as each has a number which designates it's discovery order, for instance, "*cond1*", "*cond2*", etc.

Furthermore, all the actions present in the JavaScript source code functions are associated with a number, also meaning the order they appear in the source code. These numbers are employed to reduce the size of the generated GUI model, as well as to make it more compressed, as instead of AST pieces there is a number which represent those pieces. Moreover, they are also important to produce the graphical models depicting the several actions connected to the GUI events.

Identifying end states

To discover the end state of a window, it's name, which is given by a parameter at the beginning of GuiSurfer execution, is used. The JavaScript AST is traversed,

searching for that window name element being set to invisible. An element can be set to invisible in JavaScript by using the display attribute from the element's style and setting it to '*none*'. The element can be assessed by using the DOM, namely through the element id. For example, to set the login window invisible, the following JavaScript code is used:

```
var login = document.getElementById("login");
login.style.display = 'none';
```

To set the window back to visible, instead of '*none*', an empty single quotation mark is used: ''. Therefore, to find the close state of a target window, we search for pieces of code similar to the one above. When the state is found, the corresponding action number is returned.

These informations are then gathered to produce the GUI intermediate representation, that is, the GuiModel and GuiModelFull files. Afterwards, the process unfolds as normal. These files are analysed by a module called GuiAnalysis, which is the same for all the different GuiSurfer approaches. GuiAnalysis is responsible for the creation of the GraphViz files, which are then used to produce the state machine and behaviour models.

5.3 Case Study

In order to test our prototype Ajax GuiSurfer module, the login window from the contacts agenda application was coded in Ajax. As GWT produces an Ajax application, the HTML and CSS source code produced by the GWT application was reused. However, some modifications were made in order for the code to follow the assumed coding conventions mentioned in the previous section.

The JavaScript code was rewritten, since GWT dynamically binds event handlers when pages are loaded. This approach is outside of the scope of the conventions mentioned above. The HTML tag that began the login window, was given the identifier attribute of "*login*" (as referred above, the tag was a <table> tag). Moreover, the button tags were changed to point event handlers to the correct JavaScript functions. A function "loginOk()" was written, with the actions that

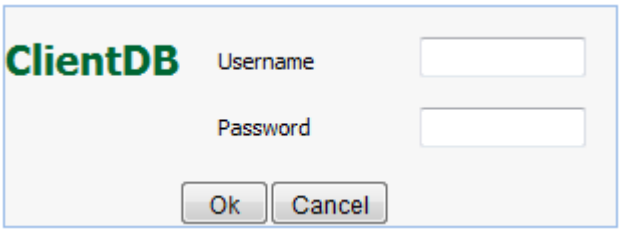

Figure 5.3: Ajax Login window

were applied after pressing the "Ok" button, and a function "loginCancel()" was made for the "Cancel" button. Both these functions are presented in Appendix B.

As the HTML had just some minor changes, and the CSS file was the same as the one generated by the GWT application, the interface remained exacly the same, as depicted in Figure 5.3.

In the *loginOk* function, the only validation made to accept an user is if the name equals "abc". This was done in all other three contacts agenda applications, as our interest is not in the validation, but in the behaviour of the user interfaces only. Furthermore, just as in GWT, if the authentication is wrong an alert is sent to the inteface, and if the "Cancel" button is pressed the text fields are cleared.

To execute GuiSurfer over the login application the following script is used:

```
>ghc --make Graph5.hs -o Graph5 -fglasgow-exts
>Graph5 login.html example.js login
>ghc --make GuiModelAnalysis.hs -o GuiModelAnalysis -fglasgow-exts
>GuiModelAnalysis 3 "Login" "AgendaAjax"
>dot -Tpng graph.dot -o graphAgendaAjaxLogin.png
```

A difference is immediately noticed when compared to the script in Section 3.3. Instead of having three executables, namely FileParser, AstAnalyser and Graph, there is only one named Graph5. All three steps were merged into a single executable. This was done mainly because since there was no need to use the slice libraries of AstAnalyser and FileParser, just parse the file, both executables were merged into the Graph module. The Graph5 module receives three parameters, the HTML file, the JavaScript file, and the window name, that is, the identifier of the element to be analysed.

This Graph5 module is just a prototype, future versions will not receive the JavaScript source file, as it's name will be read from the HTML. Moreover, this will allow Graph5 to work with HTML files that are associated with multiple JavaScript external files. After executing Graph5, GuiModelAnalysis is compiled and executed, this will generate files that can be imported and used with GraphViz. Thus, the final script sentence is using GraphViz to generate the state machine image file.

5.3.1 Results

The entire GuiModel.hs file produced by this script is present on Appendix C. A piece of the file is the following:

```
guimodel :: GuiModel
guimodel = fromList
[
(("Ok","cond1"),[1,2,3,4]),
(("Ok","cond2"),[5]),
(("Cancel","cond3"),[6,7,8,9]),
(("init","condInit1"),[])
]
```

By analysing this piece of source code, several conclusions can be made about the Ajax GuiSurfer prototype. For instance, the initial actions, that is, the actions that appear associated with the *init* event are not gathered. This situation happens because all the current actions are discovered by analysing the JavaScript source. However, in these examples, before a button is pressed, there is no JavaScript source code that is executed. Therefore, this particular situation of the behaviour actions executed when starting the application not being found must be analysed in the future.

Consider, for instance, an action that sets a button to invisible, and happens when the application starts. That is, before any user interaction occurs, the button is invisible. This action would be a part of the *init* event in GuiModel. In an Ajax application there are three main ways of achieving this effect: by

setting the HTML attribute *style* to "display:none"; by setting the display status on the CSS file; or by using JavaScript code that does the same in an *onload* event handler. An *onload* event handler enables a function to be executed when the element is loaded.

Using the JavaScript procedure would be easier to implement in our prototype. The HTML and CSS technique would pose difficulties, because the HTML or the CSS (which currently is not even parsed) should be analysed to retrieve the actions. Therefore, display and layout technologies (HTML and CSS) would have to be analysed to identify behaviour actions.

By examining the GuiModel code presented above, the various events, conditions, and respective actions can be compared to the original source code. For instance, the four actions in *cond1* (see the *Ok* event) correspond to the four sentences in the first branch of the If clause, present in Appendix B. Furthermore, the model distinguishes the closing action as the action number four, which by analysing the JavaScript source code, can be ascertained as correct. Thus, we can consider that the prototype is correctly interpreting the source code.

Another issue with this prototype version is that it does not detect the pre-conditions associated to each event. In the GuiModel file presented in Appendix C, the pre-conditions results have an empty list of *Pres*.

```
type Pres = Map ExpRef (EventRef,Bool)
pres :: Pres
pres = fromList
[]
```

Pre-conditions in GuiModel consist in an action (*ExpRef*), the associated event reference (*EventRef*) and the state of a widget after executing that action. Since initial versions of GuiSurfer only address the enabling or disabling of widgets, the state of the widget is merely a boolean (*Bool*) which defines if the widget is enabled or not. The fact that pre-conditions is not defined is related to the initial actions not being retrieved in this version of the prototype. This occurs because the actions from the pre-conditions are a reference to the actions defined in the *init* event.

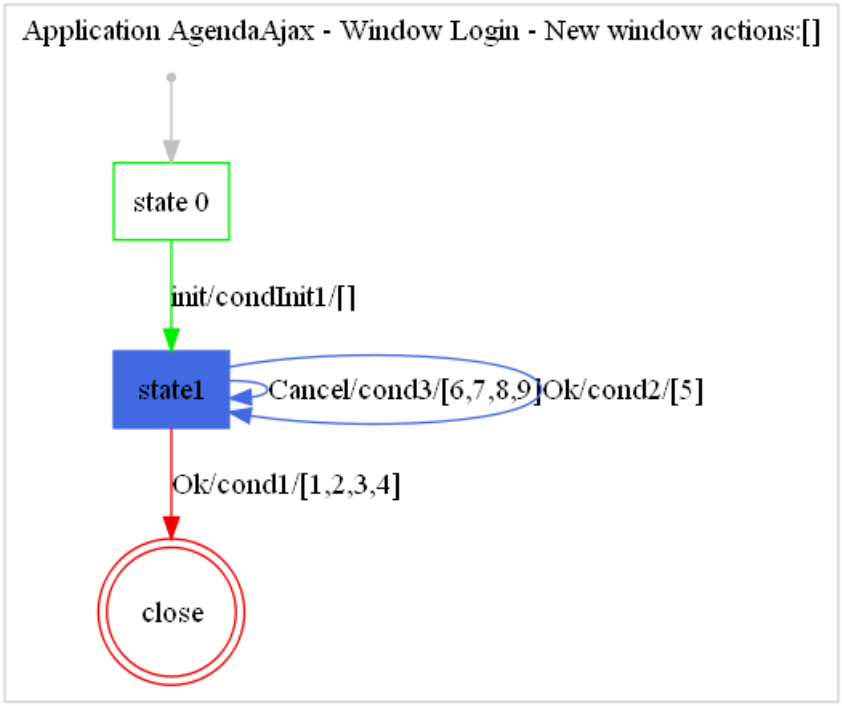

Figure 5.4: Ajax Login window FSM

Furthermore, the pre-conditions from the GuiModel meta model play an important role on how GuiSurfer defines the several states a window may have. GuiSurfer's analysis phase is responsible for extracting from the GuiModel, and specifically from the pre-conditions, the various states GuiSurfer recognizes from a target window. Therefore, this limitation is important, and will be addressed in the future.

By using GuiModelAnalysis command over the generated GuiModel and GuiModelFull, and afterwards using the GraphViz tool over the results, Figure 5.4 was created. Figure 5.4 is very similar to Figure 4.3, the state machine generated by GuiSurfer after analysing the GWT login window. The main differences relate to the actions associated with each events, while GWT module has issues identifying actions in some events, the Ajax module cannot discern the initial actions.

Moreover, neither prototype can distinguish the new windows names, that is, which windows are opened after the target window is closed. In GWT this issue

was already discussed, in Ajax, however, this situation occurs as it is difficult to distinguish when a new window is opened. For instance, to open the mainform window, that element is called in JavaScript by its identifier and is therefore set to visible. However, a completely different element could be set to visible, for instance a button, and that is not a new window action. Consequently, the difficulty of discerning the Ajax windows.

To recapitulate, in this chapter the Ajax technologies were described, the new module architecture and implementation was expound and a case study was analysed. The various tests perfomed with the Ajax module produced satisfactory results. However, this module has some limitations that need to be solved in future work. The main restraint is not recognizing the pre-conditions, because windows with more than one state will not generate accurate models. Moreover, the module can not identify new windows names and initial actions.

Chapter 6

Conclusions and Future Work

The user interface layer of Web applications is based on a combinations of technologies, brought together to increase their interactivity and responsiveness towards the end user. Given the fast pace of technological development, and the plethora of technologies available, it can become quite difficult to keep a full understanding of a developed systems, and to ensure application behaviour is the one expected.

Reverse engineering is a technique to generate models at a higher level of abstraction from an application or another model thereof. These models can then be used to gain a better understanding of a developed system. The ability to reverse engineer the GUI layer of a Web application would be an invaluable help in promoting Web applications quality.

With the above in mind, this work aimed to extend GuiSurfer, a static analysis reverse engineer tool, by providing support for the analysis of Web applications. From all the different Web applications frameworks available, the object of our analysis were GWT and Ajax applications.

6.1 Results

Following on from the goals set forth for the thesis, the two main results of this work are two new GuiSurfer modules. One module supporting the reverse engineering of GWT applications, another supporting the reverse engineering of

Ajax applications.

The GWT module was based on GuiSurfer's support for Swing based user interfaces. Chapter 4 describes GWT, its architecture and major features, and the challenges that were faced when adapting GuiSurfer to this technology. Furthermore, the GWT module implementation, and its execution over some examples were analysed. As a result of this work contributions were made to two publications: one published in the proceedings of EICS 2010, the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (Silva et al., 2010b); another at Interacção 2010, the 4th Human-Computer Interaction conference of the Eurographics Portuguese chapter (Silva et al., 2010c).

The adaptation of GuiSurfer to Ajax applications was studied in Chapter 5. The Ajax language was explored, and a brief introduction to JavaScript provided. The nature of Ajax applications implied a deeper restructuring of the language dependent part of GuiSurfer. The module developed to add Ajax support to GuiSurfer, and an example of reverse engineering an authentication window in Ajax, are also described in this chapter.

The examples provided in both chapters illustrate that appropriate models and state machines were successfully created, reflecting the corresponding applications behaviour. Moreover, the models created produced results as expected. With great similarities with the models already created from Java/Swing and WxHaskell, when similar applications were being analysed. Highlighting the differences between the different user interfaces, when user interface behaviour differed from application to application. Furthermore, the example also illustrated some of the analysis that could be carried out from the generated models.

Overall, the examples indicate that it was possible to produced new useful GuiSurfer modules, enabling the quick creation of easily understandable, simple, manageable models, that still capture relevant behavioural information. Hence, a third result of this work is that it proves the retargetability of the GuiSurfer tool, as GuiSurfer now works with a new paradigm. Proving that this tool can easily be reused and expanded. Since GuiSurfer possesses an architecture with a well defined distinction between the language dependent modules and language independent modules, the goal of generalization to RIAs was made easier, because there is only the need of restructuring the language dependent part.

Other contributions of this work include an introduction to reverse engineering, and its two main approaches: static analysis and dynamic analysis. A number of different reverse engineering implementations have also been referred. Moreover, GuiSurfer has been thoroughly analysed, as it is the tool extended in this work. Rich Internet Applications (RIAs) were also introduced and various different RIA technologies referred to. It was concluded that RIAs exhibit a far greater division between the interface code and the rest of the code than traditional desktop applications. Such division happens because a RIA application's code is divided into client-side and server-side. This division is important because it eases GuiSurfer GUI code slicing needs, as the code is more partitioned. Finally, different interface modelling approaches were also discussed and analysed, with a special emphasis on CTT models. From this analysis, it can be inferred that incrementing the GuiSurfer's model generation capabilities is essential to allowing new analysis types. Since it already produces state models, a type of dialog models, it is considered important to also abstract user task models because they permit a user-centered evaluation of the system's usability.

6.2 Future work

Both modules are currently usable, providing basic support for the reverse engineering of Web applications. A number of limitations has nevertheless been identified, and should be addressed in the future.

The GWT module requires just a few adjustments. These are related with relaxing the assumptions made on the structure of the code. This will enable it to reverse engineer a broader range of applications, without the need of source code readjustments to make them in accordance with current assumptions.

The Ajax module is more experimental. The focus of the work was on assessing the viability of extending GuiSurfer to a very different technological solution to the programming of user interfaces, setting the ground for the approach. The current version of the module requires further work in order to support the acquisition of initial actions, and the pre-conditions associated to the various events of the target window. Alternatives for this were discussed in Chapter 5, and a possible solution put forward. These enhancements will also enable better support

for windows which behaviour is composed by more than one state.

Another important update to GuiSurfer would be to further work with even more RIA technologies. For instance, a possible improvement would be to allow GuiSurfer to work with HTML5 applications, since it is a technology that gained some popularity recently.

Other line of work would be to improve the GuiSurfer's front-end, that is, to allow GuiSurfer to generate other type of models. For instance, from the analysis done in Section 2.2 it would be interesting to add the capability of generating CTT models, as they would enable different analysis to be made from target applications.

Furthermore, a different aspect could also be pursued: to transform GuiSurfer into a re-engineering tool. This could be achieved by allowing GuiSurfer models to me altered by the user, and afterwards to recreate the applications to reflect these changes in the models.

Bibliography

Jeremy Allaire. Macromedia Flash MX-A next-generation rich client. Technical report, Macromedia, March 2002.

J. Annett and K. D. Duncan. Task analysis and training design. *Occupational Psychology*, 41:211–221, 1967.

Robert S. Arnold. Software restructuring. In *Proceeding of the IEEE*, volume 77, pages 607–617. IEEE Press, April 1989.

Ira D. Baxter and Michael Mehlich. Reverse engineering is reverse forward engineering. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 104–113, Washington, DC, USA, 1997. IEEE Computer Society.

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

Barry W. Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2nd edition, 2005.

Laurent Bouillon, Quentin Limbourg, Jean Vanderdonckt, and Benjamin Michotte. Reverse engineering of web pages based on derivations and transformations. In *Proceedings of third Latin American Web Congress LA-Web'05*. IEEE Computer Society Press, 2005.

Alessandro Bozzon, Sara Comai, Piero Fraternali, and Giovanni Carughi. Conceptual modeling and code generation for rich internet applications. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 353–360, New York, NY, USA, 2006. ACM.

Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1983.

J. Chen and S. Subramaniam. A GUI environment to manipulate FSMs for testing GUI-based applications in Java. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, volume 9, page 9061, Washington, DC, USA, 2001. IEEE Computer Society.

Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

Jim Conallen. Modeling Web application architectures with UML. *Communications of the ACM*, 42(10):63–70, 1999.

Ryan Dewsbury. *Google Web Toolkit Applications*. Prentice Hall, 2008.

Eldad Eilam. *Reversing - Secrets Of Reverse Engineering*. Wiley, 2005.

John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz — open source graph drawing tools. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 594–597. Springer Berlin / Heidelberg, 2002.

Jacques Eloff. Software restructuring: implementing a code abstraction transformation. In *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 83–92, Republic of South Africa, 2002. South African Institute for Computer Scientists and Information Technologists.

Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, January 2004.

Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

Jesse James Garrett. Ajax: A new approach to web applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005.

Andy Gimblett and Harold Thimbleby. User interface model discovery: towards a generic approach. In *EICS '10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 145–154, New York, NY, USA, 2010. ACM.

André M. P. Grilo, Ana C. R. Paiva, and João Pascoal Faria. Reverse engineering of GUI models. In Luís Rodrigues e Rui Lopes, editor, *Actas do INForum – Simpósio de Informática 2009*. Faculdade de Ciências da Universidade de Lisboa, 2009.

Marko Grönroos et al. *Book of Vaadin*. Vaadin Ltd, 2010. URL http://vaadin.com/book.

Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State transition diagrams. Technical Report TUM-I9630, Technische Universität München, 1996.

Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *WWW '09: Proceedings of the 18th International Conference on World Wide Web*, pages 561–570, New York, NY, USA, 2009. ACM.

Kris Hadlock. *Ajax for Web Application Developers*. Sams Publishing, 2007.

Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar, and P. S. Thiagarajan. On message sequence graphs and finitely generated regular MSC languages. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 675–686, London, UK, 2000. Springer-Verlag.

Edward Hieatt and Robert Mee. Going faster: Testing the web application. *IEEE Software*, 19(2):60–65, 2002. ISSN 0740-7459.

John Hughes. Generalising monads to arrows. *Science of Computer Programing*, 37(1-3):67–111, 2000.

E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fifth edition, December 2009.

ISO. ISO 9241-11:1998 Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability. Technical report, International Organization for Standardization, 1998.

I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, June 1992.

Christian Janssen, Anette Weisbecker, and Jürgen Ziegler. Generating user interfaces from data models and dialogue net specifications. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 418–423, New York, NY, USA, 1993. ACM.

Byung-Kyoo Kang and James M. Bieman. A quantitative framework for software restructuring. *Journal of Software Maintenance*, 11(4):245–284, 1999.

Ralf Lämmel, Eelco Visser, and Joost Visser. The essence of strategic programming, 2002.

Daan Leijen. wxHaskell: a portable and concise GUI library for Haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, New York, NY, USA, 2004. ACM.

Peng Li and Eric Wohlstadter. View-based maintenance of graphical user interfaces. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 156–167, New York, NY, USA, 2008. ACM.

Quentin Limbourg and Jean Vanderdonckt. Comparing task models for user interface design. In D. Diaper and N. Stanton, editors, *The Handbook of Task Analysis for Human-Computer Interaction*, chapter 6, pages 135–154. Lawrence Erlbaum Associates, 2003.

Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. UsiXML: a language supporting multi-path development of user interfaces. In *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, pages 11–13. Springer-Verlag, 2004.

Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java Swing*. O'Reilly, 2nd edition, 2002.

Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 260, Washington, DC, USA, 2003. IEEE Computer Society.

Tommi Mikkonen and Antero Taivalsaari. Web Applications – Spaghetti Code for the 21st Century. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 319–328, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

Jim Qun Ning. *A knowledge-based approach to automatic program analysis*. PhD thesis, Yale University, Champaign, IL, USA, 1989.

Tom Noda and Shawn Helwig. Rich Internet Applications – Technical Comparison and Case Studies of AJAX, Flash, and Java based RIA. Technical report, UW E-Business Consortium, University of Wisconsin-Madison, November 2005.

Tim O'Reilly. What is Web 2.0: Design patterns and business models for the next generation of software. *International Journal of Digital Economics*, 65: 17–37, 2007.

Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

Michael J. Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualisation tools. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 80, Washington, DC, USA, 2003. IEEE Computer Society.

Fabio Paternò and Via V. Alfieri. Task models in interactive software systems. In *In Handbook of Software Engineering and Knowledge Engineering*, pages 817–836. World Scientific Publishing Co, 2001.

Fabio Paternò, Cristian Mancini, and Silvia Meniconi. ConcurTaskTrees: A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction Pages*, pages 362–369. Chapman & Hal, July 1997.

Angel R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4):40–47, 1997.

Angel R. Puerta, Henrik Eriksson, John H. Gennari, and Mark A. Musen. Beyond data models for automated user interface generation. In *HCI '94: Proceedings of the conference on People and computers IX*, pages 353–366. Cambridge University Press, 1994.

H. Ritsch and H. Sneed. Reverse engineering programs via dynamic analysis. In *Proceedings of the 1st Working Conference on Reverse Engineering*, pages 192–201, 1993.

Martin P. Robillard, David Shepherd, Emily Hill, K. Vijay-Shanker, and Lori Pollock. An empirical study of the concept assignment problem. Technical Report SOCS-TR-2007.3, School of Computer Science, McGill University, 2007.

Linda H. Rosenberg and Hyatt E. Lawrence. *Software Reengineering*. NASA, 1996.

Egbert Schlungbaum. Model-based user interface software tools current state of declarative models. Technical Report GIT-GVU-96-30, Graphics, Visualization and Usability Centre, Georgia Institute of Technology, 1996.

J. C. Silva, J. C. Campos, and J. Saraiva. Models for the reverse engineering of java/swing applications. In J. M. Favre, D. Gasevic, R. Lämmel, and A. Winter, editors, *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies (ateM 2006) for Reverse Engineering*, number 1/2006 in Informatik-Bericht series. Johannes Gutenberg-Universität Mainz, Institut füur Informatik - FB 8, October 2006.

João C. Silva, José C. Campos, and João Saraiva. Combining formal methods and functional strategies regarding the reverse engineering of interactive applications. In *Interactive Systems. Design, Specification, and Verification*, volume 4323 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, May 2007.

João C. Silva, João Saraiva, and José C. Campos. A generic library for GUI reasoning and testing. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 121–128, New York, NY, USA, 2009. ACM.

João C. Silva, João Saraiva, and José C. Campos. GUI inspection from source code analysis. In *Proceedings of the Fourth International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)*, 2010a.

João C. Silva, Carlos Silva, Rui Gonçalo, João Saraiva, and José C. Campos. The GUISurfer tool: towards a language independent approach to reverse engineering gui code. In *EICS 10: Proceedings of the 2nd ACM SIGCHI symposium*

on Engineering interactive computing systems, pages 181–186, New York, NY, USA, 2010b. ACM Press.

João C. Silva, Carlos Silva, João Saraiva, and José C. Campos. GUI behavior from source code analysis. In *Interacção 2010*. Grupo Português de Computação Gráfica, 2010c.

Markus Staeuble and Jurgen Schumacher. *ZK Developer's Guide: Developing responsive user interfaces for web applications using Ajax, XUL, and the open source ZK rich web client development framework*. Packt Publishing, 2008.

Tarja Systä. Dynamic reverse engineering of java software. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 174–175, London, UK, 1999. Springer-Verlag.

Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: model-based interface tools. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 383–390, New York, NY, USA, 1993. ACM.

Alexandru Telea, Alessandro Maccari, and Claudia Riva. An open toolkit for reverse engineering data visualisation and exploration. In *Proceedings of the symposium on Data Visualisation 2002*, volume 22 of *ACM International Conference Proceeding Series*, pages 241–249, 2002.

Scott Tilley. A reverse-engineering environment framework. Technical Report CMU/SEI-98-TR-005/ESC-TR-98-005, Carnegie Mellon University, 1998.

Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

Joost Visser and Joao Saraiva. Tutorial on strategic programming across paradigms. In *Proc. 8th Brazilian Symposium on Programming Languages*, Niteroi, Brasil, 2004.

Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.

Nicholas C. Zakas. *Professional JavaScript for Web Developers*. Wiley Publishing, Inc., 2nd edition, 2009.

Appendix A

Java Swing vs. GWT

In this appendix, the main differences between the Java Swing and GWT language specifications widgets are presented.

	Java Swing	GWT
Containers	JPanel	Panel
	JTabbedPane	TabPanel
	JSplitPane	HorizontalSplitPanel/VerticalSplitPanel
	JScrollPane	ScrollPanel
	JToolBar	ToolBar (gwt-ext/SmartGWT)
	JDesktopPane	-
	JInternalFrame	(gwt-dnd)
	JLayeredPane	GlassPanel (incubator)
Controls	JLabel	Label
	JButton	Button
	JToggleButton	ToggleButton
	JCheckBox	CheckBox
	JRadioButton	RadioButton
	JButtonGroup	-
	JComboBox	ComboBox
	JList	ListBox
	JTextField	TextBox
	JTextArea	TextArea
	JScrollBar	ScrollBar(SmartGWT)
	JSlider	SliderBar(incubator)/Slider(SmartGWT)
	JProgressBar	ProgressBar (incubator/ SmartGWT)
	JFormattedTextField	-
	JPasswordField	PasswordTextBox
	JSpinner	Spinner(incubator)/ SpinnerItem(SmartGWT)
	JSeparator	Separator
	JTextPane	-
	JEditorPane	-
	JTree	Tree
	JTable	Table

	Java Swing	GWT
Menus	JMenuBar	MenuBar
	JMenu	-
	JMenuItem	MenuItem
	JCheckBoxMenuItem	CheckMenuItem (gwt-ext)
	JRadioButtonMenuItem	-
	JPopUpMenu	PopUpMenu (gwtcomp)
Windows	JDialog	DialogBox
	JFrame	Frame
	JColorChooser	ColorPallete(gwt-ext)/ ColorPicker(SmartGWT)
	JFileChooser	-
	JOptionPane	-

Appendix B

JavaScript functions

JavaScript functions from the login window Ajax application.

```
function loginOK(){
    var textBox = document.getElementById("textBox");
    var textBox2 = document.getElementById("textBox2");

    if (textBox.value=="abc") {
        var mainform = document.getElementById("mainform");
        mainform.style.display = '';
        var login = document.getElementById("login");
        login.style.display = 'none';
    }
    else
        alert("Username/Password not valid");
}

function loginCancel(){
    var textBox = document.getElementById("textBox");
    var textBox2 = document.getElementById("textBox2");
    textBox.value = '';
    textBox2.value = '';
}
```

Appendix C

Example Guimodel

Guimodel.hs file generated after executing GuiSurfer over the login Ajax window.

```
-Generated automatically by GuiSurfer

module GuiModel where

import Data.Map

type EventRef = String
type CondRef = String
type WindowName = String
type ExpRef = Int

type GuiModel = Map (EventRef,CondRef) [ExpRef]

type Pres = Map ExpRef (EventRef,Bool)
type End = [ExpRef]
type Close = [ExpRef]
type Window = WindowName
type NewWindow = Map ExpRef WindowName

guimodel :: GuiModel
```

```
guimodel = fromList
[
(("Ok","cond1"),[1,2,3,4]),
(("Ok","cond2"),[5]),
(("Cancel","cond3"),[6,7,8,9]),
(("init","condInit1"),[])
]

pres :: Pres
pres = fromList
 []

end :: End
end = []

window :: Window
window = "login"

newWindow :: NewWindow
newWindow = fromList
 []

close :: Close
close =
[4]
```