UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Dipartimento di Informatica e Sistemistica

Dottorato di Ricerca in Ingegneria Informatica ed Automatica, XXIV ciclo

Coordinatore: *prof. Francesco Garofalo*

Ph.D. Thesis:

Reverse Engineering and Testing of Rich Internet Applications

Tutor: *prof. Anna Rita Fasolino*

Ph. D. Student: *Domenico Amalfitano*

November 2011

ad Antonia

Abstract

The World Wide Web experiences a continuous and constant evolution, where new initiatives, standards, approaches and technologies are continuously proposed for developing more effective and higher quality Web applications.

To satisfy the growing request of the market for Web applications, new technologies, frameworks, tools and environments that allow to develop Web and mobile applications with the least effort and in very short time have been introduced in the last years.

These new technologies have made possible the dawn of a new generation of Web applications, named ***Rich Internet Applications*** (RIAs), that offer greater usability and interactivity than traditional ones. This evolution has been accompanied by some drawbacks that are mostly due to the lack of applying well-known ***software engineering*** practices and approaches. As a consequence, new research questions and challenges have emerged in the field of web and mobile applications ***maintenance*** and ***testing***.

The research activity described in this thesis has addressed some of these topics with the specific aim of proposing new and effective solutions to the problems of ***modelling***, ***reverse engineering***, ***comprehending***, ***re-documenting*** and ***testing*** existing ***RIAs***.

Due to the growing relevance of mobile applications in the renewed Web scenarios, the problem of testing mobile applications developed for the ***Android*** operating system has been addressed too, in an attempt of exploring and proposing new techniques of ***testing automation*** for these type of applications.

Index

Introduction

The original World Wide Web was a platform for accessing static or dynamic content encoded in hypertext markup language. User interaction was limited to navigating links and entering data in forms. This thin-client architecture was simple and universal (no client installation required) but severely limited the quality of the applications that could be delivered over the Internet. Early attempts at extending interface functionality (such as Java applets and client-side scripting) enriched HTML-based navigation with interactive objects, animated presentation effects, and input validation.

Modern Web solutions resemble desktop applications, enabling sophisticated user interactions, client-side processing, asynchronous communications, and multimedia.

In such scenario nowadays Rich Internet Applications (RIAs) play a prominent role.

The term RIA refers to a heterogeneous family of solutions, characterized by a common goal of adding new capabilities to the conventional hypertext-based Web. RIAs combine the Web's lightweight distribution architecture with desktop applications' interface interactivity and computation power, and the resulting combination improves all the elements of a Web application (data, business logic, communication, and presentation) [11].

On the other hand while the use of RIAs' technologies positively affects user-friendliness and interactiveness of web applications it comes at a price. Indeed, RIAs' advent creates

an articulated research landscape with issues that include the language and architectural standards used to develop RIAs, the software frameworks built on top of these standards that enhance development productivity and solution quality, and the development tools and methodologies backing the RIA life cycle's development activities.

A specific crucial issue for RIAs is that of finding suitable approaches and technologies for supporting all the software lifecycle activities effectively, and the maintenance and testing activities above all. The relevance, complexity, expensiveness and criticalities of software maintenance processes are well known for any type of software application. However, these problems are even more relevant in the context of RIAs, since these applications are usually developed in short times by programmers that don't use well known practices of Software Engineering, and often use frameworks and tools that, on the one hand simplify RIAs' development, on the other hand produce complex code that is difficult to understand, at the expense of the quality of the final product.

Moreover, both the asynchronous and the heterogeneous nature of the RIAs, which are developed by means of several technologies and are based on a client-server architectural model in which the communication between the client and server may be asynchronous, make the RIAs harder to comprehend and consequently difficult to maintain and test.

To solve these problems with success, specific research questions must be addressed. First of all, suitable models for representing the dynamic behaviour and the heterogeneous nature of RIAs are needed. Reverse Engineering approaches that allow reconstructing these models by exploiting dynamic analysis techniques must be introduced. Moreover, comprehension processes that are based on these models and techniques will have to be proposed, in order to allow the code of RIAs to be analysed and understood efficiently.

Finally, new testing methods and techniques for verifying the quality of such applications are needed, as well as novel maintenance processes must be introduced to support the complete life cycle's development of RIAs.

On the other hand, we're now entering a new era of the Web in which Smartphones, gadgets and consumer electronics are more and more Internet-enabled. In the coming

years, billions of devices will be connected to the Internet, and they'll access and share information through the Web. New kinds of mobile and Web apps are on the horizon that will be more ubiquitous and smarter than current apps and will be accessible anytime, anywhere, and from any kind of device [143].

Nowadays most mobile applications are usually small-sized and developed by a small team (one or two people) that has the responsibilities for conceiving, designing and developing them [144]. The team usually works in strict times, under the pressure of short time-to-market, using powerful development tools and frameworks, but rarely adopting any formal development process. This approach may be suitable for small or medium size applications. However, as mobile applications become more complex and business-critical, it becomes essential to use well-defined Software Engineering techniques. In particular, to satisfy the need for quality of these applications, greater efforts and attention have to be devoted to the testing activity.

On the other side, due to the huge growth of mobile applications developed for the Android platform recorded in the last months, finding effective testing techniques, strategies and tools for Android applications is a relevant research topic too

In the last years, the scientific community has dealt with great interest the topics outlined above, trying to propose suitable approaches for developing and maintaining RIAs. At the same time, a lot of research and industrial initiatives aiming at defining effective testing principles, techniques and tools for mobile applications have been carried out too [145, 146, 147, 148, 149, 150, 151].

The research activity described in this thesis addressed some of these topics too, with the specific aim of proposing new and effective solutions to the problems of analysing, reverse engineering, comprehending and testing existing Rich Internet Applications. Moreover, due to their growing relevance in the renewed Web scenarios, the problem of testing mobile applications developed for the Android operating system has been addressed too, in an attempt of exploring and proposing new techniques of testing automation for these kind of applications. The results of this research activity will be

presented in this thesis that is organized as it follows. In the first chapter, after a brief historical excursus that shows the evolution of the Web from its beginnings until today, the main characteristics of RIAs are analysed and Ajax is introduced. As we'll read in this thesis RIAs are a novel kind of Web applications and Ajax is a set of Web technologies that allow to develop them.

Afterwards in the same chapter we show the new issues introduced by the advent of RIAs and how the scientific community of the Software Engineers are facing them.

In chapter 2 we address the problem of modeling the RIAs, proposing both a suitable model able to describe the characteristics of this new kind of Web application and a Reverse Engineering process that allows to obtain this model exploiting techniques of dynamic analysis. In chapter 3 we present the results of an experimentation performed in order to assess the effectiveness of the Reverse Engineering process proposed in chapter 2.

In chapter 4 we introduce an original process of Reverse Engineering for Rich Internet Application that may be defined "Agile". This technique is more effective than the one proposed in chapter 2.

In the chapters 5 and 6 of this thesis we address the problem of RIAs testing. In chapter 5 we propose a testing technique that allows to detect crashes on the client side of a Rich Internet Application. In chapter 6 we propose a classification framework that characterizes existing RIA testing techniques from different perspectives.

In chapter 7 we present DynaRIA, a tool developed to analyse Web applications from different perspectives. The tool offers a user-friendly environment and can be used to execute activities of program comprehension, testing, debugging and quality assessment. Moreover, the chapter shows the effectiveness of the tool in performing several program comprehension activities involving different real RIAs.

In chapter 8 we address the problem of software re-documentation and propose a novel, tool-supported process for re-documenting Web applications. The process is based on the RIA model and the related Reverse Engineering technique for obtaining it that have been introduced in previous chapters. The process is semi-automatic and has been employed for

re-documenting a real Rich Internet Application.

Finally, the 9th chapter of the thesis will be dedicated to the problem of testing Android mobile applications and will present a new testing technique based on a GUI crawler for crash testing of Android applications.

Chapter 1

RICH INTERNET APPLICATION FUNDAMENTALS

As Leon Shklar and Rich Rosen wrote in the preface of their book *Web application architecture: Principles, Protocols and Practices* [12], "The expression 'web time' connotes a world in which rapid change is the norm, where time is exponentially condensed. Technological advances that once upon a time might have taken years to transpire now occur in a matter of months or even days. What's more these advances often result in radical paradigm shifts that change the way we interact with our technology and with the world at large."

This phrase synthesizes the technological revolution that we are living every day and in particular the way the web has changed so rapidly in the last two decades.

1.1 The evolution of the World Wide Web. From the Hypertext to RIAs

Tim Berners-Lee at the CERN of Geneva in Switzerland, in 1989, presented a proposal for an information management system to share knowledge and resources over a computer network. Nowadays this system is known as World Wide Web (WWW) or more simply "The Web". The Web poses its basis on existing Internet protocols and services and actually represents an ubiquitous network that is able to provide information and communication services to hundreds of millions of people around the world.

From its humble beginnings the web has expanded exponentially to serve a wide variety of purposes for a wide variety of people.

The Web is nowadays so common in ordinary life that concepts, technologies and

definitions related to it have become in ordinary usage, such as Client, Server, hypertext, HTML, HTTP, URL, Web Browser, Web 2.0, Web application, etc.

In particular the term "hypertext" represent a set of documents related each other through "key-words", it can be considered like a network whose nodes are the documents. The hypertext can be read in non sequential manner, unlike the static text of print media, each document of the network can be the next one on the basis of the choice of the reader selecting a key-word as link. The choice of a key-word actually opens a new document. Hypertext was intended for use with an interactive computer screen and could be connected to other pieces of hypertext by "links". In practice a hypertext was text containing links to other text and is one of the major features of the World Wide Web. Berners-Lee married together the notion of hypertext with the power of the Internet, promoting the web as a virtual library useful to share information resources among researchers via on-line documents that could be accessed via a unique document address, a universal resource locator (URL). An URL is a sequence of characters used to identify or name a resource, such as a Web page, uniquely on Internet. The evolution of the hypertext was the Web page that is a document or information resource that is suitable for the World Wide Web and can be accessed through a Web browser.

Web pages was richer than simple hypertext including, usually, information as to the colors of text and backgrounds and very often also contain links to images and sometimes other types of media to be included in the final view. Web pages are written in HyperText Markup Language (HTML) that is a markup language rather that a programming language, that allows to define, by the means "tags", paging, formatting and graphical layouts both of the textual and not textual contents of a Web page, providing, moreover, navigation to other web pages via hypertext links. .

A set of web pages interrelated each other is defined as a Web Site. A web site is a structure of hypertext, hosted in a Web server, accessible by a user through a Web Browser. Web pages are requested and served from web servers using Hypertext Transfer Protocol (HTTP).

The served content by web sites in response to a user request usually consists of a HTML document: a web browser downloads from one o more web server the HTML content, and any related documents and processes them, interprets the code, in order to generate the display of the page on the computer screen.

Web site was static so that the user could navigate only through the Web pages. The advent of the dynamic web which resulted from the birth of scripting languages server side, such as CGI, a new generation of application was born, the so called Web applications. By means of the client side scripting languages was possible to. By the means of these languages was possible to access to various resources, creating dynamically and returning web pages depending on the client requests.

Web applications are client-server application that uses a web browser as their client program and deliver interactive services through web servers distributed over the Internet (or an intranet). Unlike a web site that simply delivers content from static files a web application can present dynamically tailored content based on request parameters, tracked user behaviours and security considerations. Web applications not only provide information and interact with site visitors, but also collect and update information, maintain access controls and support on-line transactions. As the web matured, more server-side scripting languages appeared, examples of which include PHP, Python, Ruby, Java Server Pages (JSP), and Active Server Pages (ASP).

Since 2000, another major trend has arisen in the Web, incorporating applications that support user-generated content, on-line communities. and collaborative mechanisms for updating on-line content. This trend is often referred to as Web 2.0, because it is closely tied to advances in web technology that herald a new generation of web applications.

Another kind of Web application is emerged in the last years and is actually a new way to think to the Web as a platform for accessing and develop applications. This is possible due both to the evolution of scripting languages client-side and to networks that present more and more larger bandwidth. This new kind of Web application is called Rich Internet Applications (RIAs) and will be discussed in the next chapters.

1.2 Rich Internet Applications: a new generation of Web Applications

The term "Rich Internet Application" (RIA) was introduced for the first time in a White Paper in 2002 by Jeremy Allaire of Macromedia [13] to denote a unification of traditional desktop applications and Web applications with the aim, on the one hand, to exploit the advantages of both and the other of trying to overcome the disadvantages of the two architectures.

In the past the original World Wide Web was a platform for accessing static or dynamic content encoded in hypertext markup language. User interaction was limited to navigating links and entering data in forms. This thin client architecture was simple and universal (no client installation required) but severely limited the quality of the applications that could be delivered over the Internet [14].

During the last years, processes, technologies and tools for developing Web applications have evolved considerably, so that a new generation of Web applications characterized by an enhanced usability of their interfaces and providing a more interactive, dynamic, and satisfactory user experience has come. The advent of RIAs has evolved into an authentic technological revolution, providing Web information systems with many of the features and functionality of traditional desktop applications [15]. RIAs are client/server applications that exist at the intersection of two competing development cultures: desktop and Web applications providing most of the deployment and maintainability benefits of Web applications while supporting a much richer and responsive client user interface (UI) [15]. These capabilities represent a way to make programs easier to use and more functional, thus both enhancing the user experience and overcoming problems with traditional Web applications such as slow performance and limited interactivity.

The concept of "Rich" has two aspects: the richness in the data model and rich user interface. Rich in the data model means that the user interface can represent and manipulate more complex data structure within the client, reducing the server-side computational load, and allows the transmission and reception of data asynchronously.

The advantage is that the application resides on the client that requests more specific and essential data to the server, reducing at the same time the interactions with it. The server continues to have the task of generating and transforming the responses to the client's requests, these responses not are whole HTML pages but individual portions of them.

The richness of the user interface means an approach both aesthetic and functional to the models of the user interface of the desktop applications. Indeed the most evident difference between a Rich Internet Application and a traditional web application regards their presentation levels. More precisely, at the presentation level a traditional web application can be considered as a form-based software system [16] that uses a multi-page interface model where the user submits some input on the current web page and requesting the elaboration to the server, the server executes some data processing and responds by presenting a new page.

Vice-versa, the interface model of a RIA can be considered as a single-page model where changes can be made to each page components without the need of refreshing the page entirely.

With the new rich user interfaces is possible to move from a model in which the server's response affects the entire interface to another where changes are specific only of an area of the application's interface in which the request of change was originated or necessary.

This means that the interfaces are divided into different areas of components that can be added, deleted, modified independently. The result is the ability to manage the interface of the client as a set of components that reflect much better the richness and complexity of data and business logic offered by the web application.

The common element of RIAs technologies is to create Web applications featuring sophisticated interfaces moving part or all of the layers involved in the presentation, interaction and application logic from server to the client using the so called client engine that is loaded automatically at the beginning of the session, as well as to exploit the same engine to require the needed data to the server.

The RIAs provide the load, at the beginning of the session, of a client engine both to

manage the communication between the client and the server and to manage the showed user interface. Although it may seem that adding a new layer to the application may slow down its execution, in fact the opposite is true.

Usually the client engine is loaded as part of the application instantiation, and if needed itself could download supplementary portions of code from the server, actually the client engine acts as if it were an extension of the browser, standing between the client and the server avoiding that all the user interactions generate requests to the server, which in the classical model of interaction would lead to make HTTP requests. In fact in the case of RIAs, the same user interactions generate only requests to the client engine which tries to manage them on its own without any interaction with the server, unless it is necessary. Moreover if the client engine needs some information or some processing, it makes requests, usually asynchronous, to which the server sends only the data that the client engine needs to update the page or to perform other kinds of elaborations, in contrast, rather, to the traditional Web applications in which the server responds by sending to the client and building the entire HTML page.

The presence of the engine allows that the RIAs has fast response time to the user requests and reduce the amount of network traffic and so will meet clients with limited bandwidth connections and giving them the opportunity to fully enjoy the richness of the application.

RIAs are developed using Web 2.0 techniques and technologies, such as Ajax [17] and Ajax based framework [108] such as GWT [120], or Ajax platforms such as ASP.NET Ajax [121], or non Ajax platforms such as Microsoft Silverlight [122], Adobe AiR [123], and Adobe Flex [124], or Sun Microsystems' Javafx [125].

Nowadays Web developers use mainly Ajax technologies to build Web applications with improved performance and interactivity, as well as responsive user interfaces. Ajax will be discussed in the next section.

1.3 AJAX

Jesse James Garrett, president of the Adaptive Path product-design consultancy, coined the

acronym Ajax for Asynchronous JavaScript (JS) and XML in 2005 [17] to indicate a set of Web 2.0 technologies now supported by all major browsers.

As Garret said Ajax isn't a technology. It's really several technologies, each flourishing in its own right, coming together in powerful new ways. Ajax incorporates the following technologies [18]:

- **Dynamic HTML.** Ajax applications take advantage of dynamic HTML, which consists of HTML, Cascading Style Sheets, and JavaScript glued together with the document object model. The technology describes HTML extensions that designers can use to develop dynamic Web pages that are more animated than those using previous HTML versions. For example, when a cursor passes over a DHTML page, a color might change or text might get bigger. Also, a user could drag and drop images to different places.
- **XML.** Ajax uses XML to encode data for transfer between a server and a browser or client application.
- **Cascading Style Sheets.** CSS gives Web site developers and users more control over how browsers display pages. Developers use CSS to create style sheets that define how different page elements, such as headers and links, appear. Multiple style sheets can be applied to the same Web page.
- **Document Object Model.** The DOM is a programming interface that lets developers create and modify HTML and XML documents as sets of program objects, which makes it easier to design Web pages that users can manipulate. The DOM defines the attributes associated with each object, as well as the ways in which users can interact with objects. DHTML works with the DOM to dynamically change the appearance of Web pages. Working with the DOM makes Ajax applications particularly responsive for users.
- **JavaScript.** JavaScript (JS) interacts with HTML code and makes Web pages and Ajax applications more active. For example, the technology can cause a linked page to appear automatically in a popup window or let a mouse rollover change

text or images. Developers can embed JavaScript, which is openly and freely available, in HTML pages. Ajax uses asynchronous JavaScript, which an HTML page can use to make calls asynchronously to the server from which it was loaded to fetch XML documents. This capability lets an application make a server call, retrieve new data, and simultaneously update the Web page without having to reload all the contents, all while the user continues interacting with the program.

- **XMLHttpRequest.** Ajax can use JavaScript-based XMLHttpRequest (XHR) objects to make HTTP requests and receive responses quickly and in the background, without the user experiencing any visual interruptions. Thus, Web pages can get new information from servers instantly without having to completely reload. For example, users of an application with XHR objects could type in a centigrade amount in one box of a temperature conversion application and have the Fahrenheit amount appear instantly in another box.

Figure 1.1 and Figure 1.2, in the next page, show in details the differences between the classic web application model and the Ajax web application model. In the classic web application model most user actions in the interface trigger an HTTP request back to a web server. The server does some processing (retrieving data, crunching numbers, talking to various legacy systems) and then returns a formatted HTML page to the client. As showed in Figure 1.2 the user activity is blocked waiting for the responses of the server, the user have to wait at every step usually staring at a blank browser window and an hourglass icon, waiting around for the server to do something.

Ajax applications eliminate the start-stop-start-stop nature of interaction on the Web by introducing an intermediary, the so called Ajax engine, between the user and the server.

Instead of loading a webpage, at the start of the session, the browser loads an Ajax engine written in JavaScript that runs within the JavaScript browser engine. This engine intercepts user inputs, displays requested material, and handles interactions on the client side and is responsible for both rendering the interface the user sees and communicating with the server on the user's behalf. The Ajax engine allows the user's interaction with the

application to happen asynchronously independently of communication with the server. The user activity is continuous, and is never blocked as is showed in figure, in fact every user action that normally would generate an HTTP request takes the form of a JavaScript call to the Ajax engine instead. Any response to a user action that doesn't require a trip back to the server, such as simple data validation, editing data in memory, and even some navigation, the engine handles on its own. If the engine needs some information from the server in order to respond, if it's submitting data for processing, loading additional interface code, or retrieving new data, the engine makes those requests asynchronously, usually using XML, without stalling a user's interaction with the application.

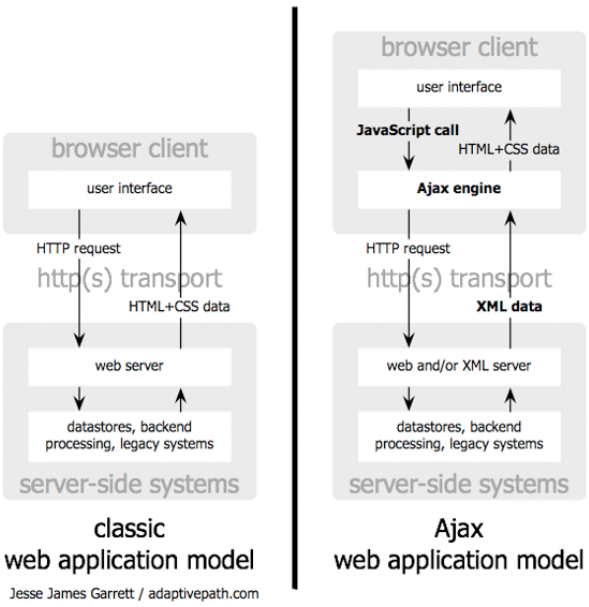

Figure 1.1: The traditional model for web applications (left) compared to the Ajax model (right)

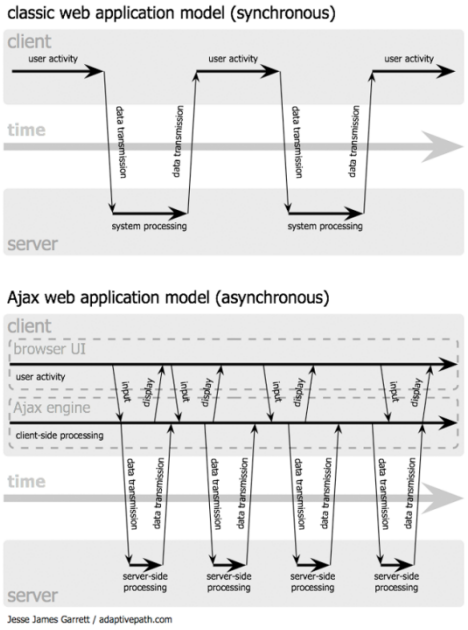

Figure 1.2: The synchronous interaction pattern of a traditional web application (top) compared with the asynchronous pattern of an Ajax application (bottom).

Ajax is not only technically sound, but also practical for real-world applications. Google is making a huge investment in developing the Ajax approach. All of the major products Google has introduced over the last year, Orkut, Gmail, Google Groups, Google Suggest, and Google Maps, are Ajax applications. For instance, when users hold down the left mouse button and slide the cursor over an image on the Ajax-based Google Maps to retrieve a part of the map not shown on the screen, the updates occur smoothly and the image appears to move and change immediately. With typical Web applications, users

must spend time waiting for entire pages to reload, even for small changes.

Flickr uses Ajax in some parts of its Web site, on which users post and share photographs. For example, Ajax enables the site to let users add and view photo annotations. Expedia has produced features such as pop-up calendars on its travel site via Ajax. Amazon's A9.com search engine applies Ajax.

All these projects demonstrate that Ajax isn't another technology that only works in a laboratory. Ajax applications can be any size, from the very simple, single-function Google Suggest to the very complex and sophisticated Google Maps.

1.4 RIA Modelling

In the last years much of the efforts of the scientific community were focused on proposing suitable models of RIAs to be used in several fields such as architectural design, maintenance, testing, comprehension, model-driven approaches, accessibility, migration from legacy systems, etc. In this section we present some of the most well-known models introduced in the last years in the field of the research on the Rich Internet Applications.

With respect to traditional web applications RIAs technologies introduce several differences in all the web application lifecycle too. According to [19, 20] the development process of a RIA needs to be changed in order to take into account the new RIAs' capabilities, and traditional methods, models and techniques used for designing a web application need to be adapted or extended in order to take into account the new aspects.

In their work Preciado et al. [19] in 2005 presented the main characteristics of RIAs and reviewed several methodologies for the Web, Multimedia, and Hypermedia application development, in order to asses if they was suitable for modelling RIAs. In their work the authors demonstrated that none of the methodologies in the selected fields were suited to model applications that respond to the main requirements of the RIA technology. However, they proved that their combination presents most of the required characteristics, emphasizing strongly the necessity to provide a modelling approach for the RIA technologies, since at the moment of the paper methodologies in this area were missing

and the current methodologies couldn't be directly applied to model and generate RIAs. Moreover the same development process of a Web Application has to be changed in order to take into account the new RIAs' capabilities, and traditional methods, models and techniques used for designing a web application need to be adapted or extended in order to cope with the new aspects of the RIAs. As an example, the approaches proposed in the literature to model a traditional web application are clearly unsuitable to specify the behaviour of an application whose processing is no more exclusively performed by the server, but can be performed on either the client, or the server, or both sides of the application.

At the same time, approaches, techniques and models traditionally used for testing [21], maintaining [22] and evolving a web application will need to undergo substantial modifications too. In particular, new solutions will have to be investigated in order to reverse engineering software representation models that can support the expensive tasks of comprehending, evolving and validating existing RIAs' implementations. In this context, open issues consist of defining both effective representation models, and reverse engineering processes that are able to reconstruct them using both traditional and innovative analysis techniques.

The Web engineering community is well-aware that the RIA development is a new and difficult challenge requiring that the traditional methodologies are modified.

Before all, new models capable to represent the interactive nature of the RIAs must be introduced. Recently, the Web engineering community has advocated adopting the model-driven development (MDD) paradigm for RIAs [14]. MDD refers to a family of development approaches based on using models as a primary artifact in the development life cycle. Researchers have extended numerous existing methodologies, originally conceived for traditional Web applications, to cope with the new modelling issues appearing in RIAs [14]. However, this extension is far from trivial because of RIAs incorporate many novel features, such as presentation behaviours, data and processing distribution, flexible event handling and communication. Consequently, RIA models can

quickly grow too complex for developers to be understood or managed.

In the last years, several methodologies and models for developing Web applications have been proposed in the literature. The first methodologies, such as RMM [23, 24], HDM [25] and OOHDM [26], derived from the area of hypermedia applications, while other ones were developed specifically for web applications [27, 28, 29]. A user-centered design process for web applications was described by Cloyd [30]. WebML (Web Modelling Language) [31, 32] is another language for the high-level description of a Web system built on several previous proposals for hypermedia and Web design languages, including HDM, RMM, and OOHDM. Jim Conallen [33, 34] proposed some extensions to the UML notation to make the UML suitable to model web applications. UWE (UML-Based Web Engineering) [35] is a modern proposal for the development of Web applications based on UML and the Object Oriented model. Further approaches for developing web applications are reported in [19]. Unfortunately, due to substantial differences between a RIA and a traditional web application, these methodologies and the corresponding models for representing a web application are not suitable for RIAs.

Moreover RIA methodologies are relatively new and don't yet cover all design concerns usually encountered in state-of-the-art software engineering.

Recently many efforts have been made in order to find models capable to represent the heterogeneous nature of the RIAs, and how these models could be applied in effective processes of RIAs development. All these approaches have proven successful for functional concerns such as domain, navigation, and presentation. Broadly, they propose a set of RIA-specific abstractions. Unfortunately, RIA researchers tend to overlook architectural and technological aspects. Consequently, these RIA proposals have a gap between the problem concepts that capture their models and how these concepts are ultimately implemented through components or RIA frameworks. For this reason, these methods must realize a set of assumptions and select predefined architectures and technologies that often aren't the most appropriate with regard to the solution the customer seeks.

Moreover, the need to have the most appropriate architecture is especially important in methodologies that provide a code-generation environment (such as RUX, WebML, OOWS and OOH4RIA). Including solution-space abstractions would thus decrease the set of predefined architectural decisions that these methodologies usually make when generating code in such environments.

Melia et al. [36] in this context have proposed a new approach called OOH4RIA which proposes a model driven development process that extends OOH methodology. It introduces new structural and behavioural models in order to represent a complete RIA and to apply transformations that reduce the effort and accelerate its development. The developed RIA is implemented on the Google Web Toolkit (GWT) framework.

Preciado et al. [20] have proposed an integrated Web Engineering approach based on the WebML and the RUX-Model conceptual models for supporting a high-level design of Rich Internet Applications and their automatic code generation.

Valverde et al. in [37] tried to establish the foundations for supporting the UI technological perspective of the Web 2.0 in a MDE scenario. The technological complexity of RIA development is abstracted to the analysts, applying the MDE principles,. To achieve this goal, first of all, the authors have defined a RIA meta-model to support the new expressivity required in this new kind of Web applications.

The meta-model that has been defined is generic enough to be extended and related to different MDE methods. Furthermore, it can be used as a basis to define concrete UI meta-models to address the UI modelling for different RIA technologies.

The proposed RIA meta-model has been defined without taking into account a specific method. The authors explained how is possible the integration of their RIA meta-model with the OOWS Web Engineering method [38].

OO-Method [39] is an automatic code generation method that produces an equivalent software product from a system conceptual specification. OOWS was defined to extend OO-Method with the principles proposed by the Web Engineering community. To achieve this goal, OOWS introduces a new set of models for

supporting the interaction concern between the user and a Web application.

An interesting proposal of a new architectural style for Ajax applications, named SPIAR, has been presented by Mesbah and van Deursen [40]. The authors state that the SPIAR style can be used when high user interaction and responsiveness is desired in web applications. The SPIAR style considers three categories of architectural elements, namely *processing*, *data*, and *connecting elements*. An overview of these elements is showed in Figure 1.3.

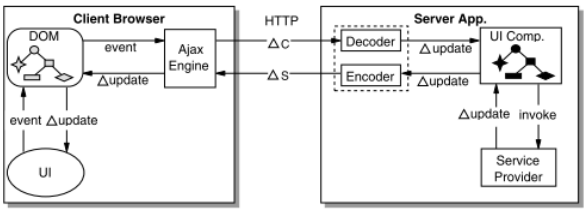

Figure 1.3: Processing View of a SPIAR-based architecture

Processing Elements are defined as those components that supply the transformation on the data elements, including the Client Browser, the Ajax Engine, the Server Application, the Service Provider, the Delta Encoder/Decoder and the UI Components. Data Elements contain the information that is used and transformed by the processing elements, such as the Representational Model, a run-time abstraction of how a UI is represented on the client browser through the DOM, and the Delta communicating messages, that form the means of the delta communication protocol between client and server. SPIAR makes a distinction between the client delta data (DELTA-CLIENT, ΔC) and the server delta data (DELTA-SERVER, ΔS). Finally Connecting Elements serve as the glue that holds the components together by enabling them to communicate. The connecting elements are the Events, the Delta connectors and the Delta Updates. Figure 1.3 shows a processing view to describe how the elements work together to form the architecture. The view concentrates on the data flow and some aspects of the connections among the processing elements with respect to the data, showing the interactions of the different components at some time after the initial page request (the engine is running on the client). User activity on the user interface fires off an event to indicate some kind of component defined action which is delegated to the AJAX engine. If a listener on a server-side component has registered itself with the event, the engine will make a DELTA-CLIENT message of the current state

changes with the corresponding events and send it to the server. On the server, the decoder will convert the message, and identify and notify the relevant components in the component tree. The changed components will ultimately invoke the event listeners of the service provider. The service provider, after handling the actions, will update the corresponding components with the new state which will be rendered by the encoder. The rendered DELTA-SERVER message is then sent back to the engine which will be used to update the representational model and eventually the interface. The engine has also the ability to update the representational model directly after an event, if no round-trip to the server is required. The same authors in another work [41] faced the challenge of migrating web applications to single page Ajax applications, introducing a Single-page Meta-model. The meta-model is depicted in Figure 1.4 showing that a Web Rich Internet Application is composed by an Ajax single-page that in turn is composed of widgets. Each widget, in turn, consists of a set of user interface (UI) components. The UI components are components of Input, Output, Navigation or Layout. As an example Button, Text, Anchor and File are Input UI Components, and Image, Label and Data are Output UI components.

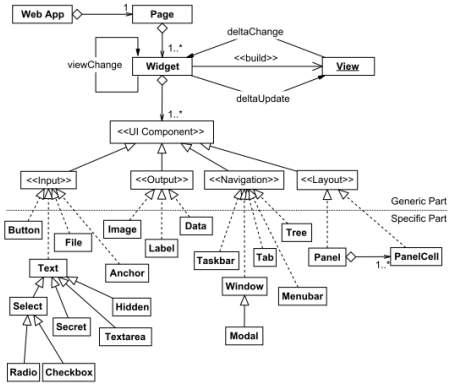

Figure 1.4: The meta-model of a single-page AJAX application composed of UI components

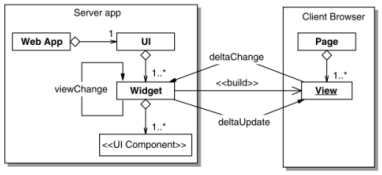

Figure 1.5: A single page web application composed of UI components

The client side page is composed of client-side views, which are generated by the server-side widgets/components as showed in Figure 1.5

Navigation is through view changes. For each view change, merely the state changes are interchanged between the client and the server, updating or changing dynamically the singles widgets that compose the page as opposed to the full-page retrieval approach in multi-page web applications.

The SPIAR and the single page proposal are relevant contributions in the direction of defining suitable representation models for specifying RIAs' characteristics. However, further models besides the architectural one are necessary for representing a RIA from other relevant points of view, such as the behavioural one.

From this point of view, being a RIA an hybrid between a web application and a desktop application, suitable models for representing its behaviour may be considered and selected from the ones usually adopted for modelling GUIs in event-driven software. In RIAs, indeed, like in Graphical User Interfaces the interface is sensible to a set of user-generated and system-generated events that act on the interface widgets. As a consequence suitable models for representing the RIAs' behaviour include the ones usually adopted for modelling GUIs evolution in event-driven software, such as the Event Flow Graphs or Finite State Machines (FSM) [50].

In the field of GUI modelling and reverse engineering, Memon et al. [44] presented a GUI model and a technique (called GUI Ripping) for reverse engineering it from the executing GUI. This model includes both a representation of the hierarchical nature of a GUI (called GUI Forest and made up of GUI's windows, each one containing a set of widgets, a set of properties of these widgets, and a set of values associated with the properties, and flow between windows) and a representation of its execution behaviour (made up of an Event-flow Graph). The model has been used with success in several GUI testing activities [42].

EFG models are successfully used by Memon in the context of test automation of EDS Systems in particular for the GUI of desktop applications [43]. In particular to automate the process of GUI testing, a graph-traversal model, event flow graph (EFG), and its later version, event interaction graph (EIG)[44-47] and event sequence graph (ESG) [48-49], have been proposed in recent years to generate sequences of events for creating test cases. In this field of research, the event flow graph (EFG) was proposed as the core-enabling model. In EFG, each vertex represents an event. All events which can be executed immediately after this event are connected with directed edges from it. A path in EFG is a legal executable sequence which can be seen as a test case. EFGs can be generated

automatically using a tool called GUI Ripping ([44]) Traversing the EFG with certain strategy can generate test cases. EFG was first proposed in [45], its definition being as follows.

- *Definition:* An event-flow graph for a component **C** is a quadruple **<V, E, B, I>** where:
  - **V** is a set of vertices representing all the events in the component. Each v ∈ **V** represents an event in **C**;
  - **E** ⊆ **V** × **V** is a set of directed edges between vertices. Event e_i follows e_j iff e_j may be performed immediately after e_i. An edge (v_x, v_y) ∈ **E** iff the event represented by v_y follows the event represented by v_x;
  - **B** ⊆ **V** is a set of vertices representing those events of **C** that are available to the user when the component is firstly invoked; and
  - **I** ⊆ **V** is the set of restricted-focus events of the component.

In the definition, a **GUI component C** is an **ordered pair <RF, UF>**, where **RF** represents a model window in terms of its events and **UF** is a set whose elements represent modeless windows also in terms of their events. Each element of **UF** is invoked either by an event in **UF** or **RF**.

Another example of a model representing the flow of events in a GUI has been proposed by Belli et al. [51]. This model is also proposed for testing a GUI and consists of an event sequence graph representing the system behaviour and the facilities from the user's point of view while interacting with the system. However, this model is a more abstract representation compared with State transition diagrams or Finite State Machines, since it disregards the detailed internal behaviour of the system.

FSMs provide another convenient way to model software behaviour from a black-box perspective, and several techniques have been proposed in the literature to reverse engineering them from existing software applications [52-54] and from traditional web applications [55].

Finite State Machine (FSM) is one of the most widely used models in software design and

software testing, especially for GUI modelling [50]. To represent the GUI behaviour by means of a FSM is mandatory define the concepts of GUI State and State Transition. There are many ways to define the states of a GUI application, usually the graphical user interface of a given application is treated as a series of interfaces. Each interface can be regarded as a state. This state can be used to construct a finite state machine for GUI test automation. A GUI's state is modeled as a set of opened windows and the set of objects (label, button, text, etc.) contained in each window.

As definition,

- at a particular time t, the GUI can be represented by its constituent windows:
  - **W** ={w1, w2, …, w_n} and their objects **O** ={ O1, O2, …, O_n},

    where Oi={o(i,1), o(i,2),…o(i, m_i)}, i=1, 2, …, n;

    each object contains properties
    - **P** = { P(1,1), P (1,2), …, P (1, m_l),

      P (2,1), P (2,2), …, P (2, m_2),

      ….,P (n,1), P(n,2), …., P(n, m_n)},

      where

P(i,j) = { p(i, j, 1), p(i, j, 2), …., p(i, j, k_{ij}) }; i=1, 2, …, n; j= 1, 2, …., m_i ;

    - and their corresponding values

      V(i,j) = { V(i,j,1), V(i,j,2), …., V(i,j, k_{ij}) };

      where

V(i,j, k) = { v(i,j, k, 1) , v(i,j, k, 2) , …, v(i,j, k, L_{ijk}) ,},

i=1, 2, …, n; j= 1, 2, …., m_i; k= 1, 2, …., k_{ij};

At a certain time t, the set of windows and their objects constitutes the state of the GUI. All the objects are organized as a forest. A GUIs state is then modeled as a quadruple (**W**, **O**, **P**, **V**). Events *{e_1, e_2... e_q}* performed on the GUI may lead to state transitions. The function notation $S_j = e_i(S_i)$ is used to denote that S_j is the state resulting from the execution of event e_i at state S_i. Such a state and transition can be considered as a finite state machine.

Recently FSMs have been successfully used by Marchetto et al. in [56] and Mesbah et al. in [57, 83] as reference model to describe the behaviour of the RIAs GUI in their-model based testing processes.

1.5 Open Issues

As we discussed in the previous sub-sections, Rich Internet Applications present a richer functionality and enhanced usability than the traditional Web applications. On the other hand, there are some aspects of RIAs that introduce new challenges in several fields of research.

In particular RIAs, as well as Web sites and Web applications, usually are deployed at a fast pace not only by experts but also by individuals programmers without the required training and knowledge to implement well-structured systems. For this reason the field of the Web applications, and of the RIAs too, has been characterized by a lack of well defined design methodologies and development processes. Often RIAs are developed without adequate phases of design and testing so that the quality of resulting software is drastically decreased.

Moreover RIAs are obtained by means of a successful combination of heterogeneous technologies, multiple programming languages, frameworks and communication models that has contributed to the complexity's growth of these web systems.

All these problems are true for RIAs developed with several technologies, however they have been concretely faced in the literature just for RIAs developed in Ajax.

In the field of testing, Marchetto and al. [56] stated that the advent of Ajax adds novel problems to those already known in the Web testing area. Since Ajax Web applications are heavily based on asynchronous messages and DOM manipulation, the authors expect that the faults associated with these two features are relatively more common and widespread than in other kinds of applications. Hence, Ajax testing should be directed toward revealing faults related to incorrect manipulation of the DOM. For example the DOM structure can become invalid during the execution due to page manipulations by JavaScript code. Another example is an inconsistency between code and DOM, which

makes the code reference an incorrect or nonexistent part of the DOM.

Often Ajax programmers make the assumption that each server response comes immediately after the request, with nothing occurring in-between. While this is a reasonable assumption under good network performance, when the network performance degrades, we may occasionally observe unintended interleaving of server messages, swapped callbacks, and executions occurring under incorrect DOM state. All such faults are hard to reveal and require dedicated techniques.

Mesbah et al. in [58] stated that with the new change in developing web applications comes a whole set of new challenges, mainly due to the fact that AJAX shatters the metaphor of a web 'page' upon which many web technologies are based. Among these challenges are the following:

- **Searchability** ensuring that AJAX sites are indexed by the general search engines, instead of (as is currently often the case) being ignored by them because of the use of client-side scripting and dynamic state changes in the DOM;
- **Testability** systematically exercising dynamic user interface (UI) elements and states of AJAX to find abnormalities and errors;
- **Accessibility** examining whether all states of an AJAX site meet certain accessibility requirements.

The same authors in [57] asserted that while the use of AJAX technology positively affects user-friendliness and interactiveness of web applications, it comes at a price: AJAX applications are notoriously error-prone due to, e.g., their stateful, asynchronous and event based nature, the use of (loosely typed) JavaScript, the client-side manipulation of the browser's Document-Object Model (DOM), and the use of delta-communication between client and web server.

Matthijssen et al. in [59] stated that although Ajax allows developers to create rich web applications, Ajax applications can be difficult to comprehend and thus to maintain. Before the dawn of Ajax, Hassan and Holt already noted that "Web applications are the legacy software of the future" and "Maintaining such systems is problematic" [60].

Moreover, software processes often devote little effort to the production of end user documentation due to budget and time constraints, or leave it not up-to-date as new versions of the application are produced. In particular, in the field of Web applications, due to their quick release time and the rapid evolution, end user documentation is often lacking, or it is incomplete and of poor quality.

In this thesis we examine the new Software Engineering challenges in the field of Ajax-based RIAs, in particular in the context of maintenance, comprehension, testing and re-documentation processes. First of all, we address the problem of finding the most suitable models for representing the peculiar features of the RIAs. As a consequent step we propose Reverse Engineering processes that allow to obtain these models automatically, by exploiting techniques of dynamic analysis. Then we show how the obtained models can be used to carry out cost-effective processes of comprehension, testing and re-documentation of Rich Internet Applications. These arguments will be discussed in details in the next chapters of the thesis.

Chapter 2 [1]

REVERSE ENGINEERING THE BEHAVIOUR OF RICH INTERNET APPLICATIONS

Software technologies, processes and development paradigms for producing Internet applications are evolving in constant and rapid way, offering always new methods and solutions for producing more effective Web applications. A recent output of this trend is represented by Rich Internet Applications (RIA), a new generation of Web applications which exploit specific web technologies for overcoming usability limitations of traditional web applications and offering greater usability and interactivity to their users.

The term AJAX, originally proposed as an acronym for Asynchronous JavaScript and XML [17], is just an approach for developing RIAs using a combination of Web technologies such as XML, XMLHttpRequest, JavaScript, CSS, and DOM: these technologies give RIAs new client-side elaboration capacity, new presentation features, and different communication mechanisms between client and server side. The main characteristic of an Ajax application is that it introduces an intermediary — an Ajax engine — between the user and the server. Instead of loading a webpage, at the start of the session the browser loads an Ajax engine, written in JavaScript. This engine is responsible for both manipulating the interface the user sees, and communicating with the server on the user's behalf. The interface evolves dynamically on the basis of the user's interaction with its single DOM components, while the engine communicates with the server by the XMLHttpRequest (XHR) object. This object allows asynchronous retrieval of arbitrary

[1] Part of this chapter was published in the Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008).

data from the server without the need of refreshing the current page and leaving at the same time the user able to perform other tasks independently. This aspect has produced an important shift in the Internet's default request/response paradigm, and now that major browsers have added support for it, web applications have gained the ability to provide richer user experiences, becoming more and more similar to desktop applications.

In this chapter, the problem of modelling the behaviour of a RIA using Finite State Machines will be addressed, and a reverse engineering approach for obtaining this model from Ajax-based RIAs will be presented. A key challenge of the proposed approach consists of obtaining the model on the basis of an analysis of just the client side of the application.

The reverse engineering approach will reconstruct and model the behaviour of an existing RIA by analysing both the evolution of its user interface, and other relevant information about the processing performed by it at run-time. The approach is based on a two-step process where the former step is devoted to tracing the RIA executions, and the latter one exploits abstraction techniques based on client interface clustering rules for generating the FSM from the information collected during the RIA execution.

The process execution is supported by a tool that provides an integrated environment for performing dynamic analysis and collecting information about the RIA execution using non-invasive techniques. Moreover, the tool implements automatically clustering criteria that are used for abstracting the FSM-based model of the RIA behaviour.

2.1 Modelling the Behaviour of a Rich Internet Application

Finite State Machines (FSM) provide a convenient way to model software behaviour from a black-box perspective, and several techniques have been proposed in the literature to reverse engineering them from existing software applications. Many black-box reverse engineering techniques generate FSMs by analysing the software user interface run-time evolution [52-54] and making hypotheses for generating states and transitions of the corresponding FSM. Some of these techniques have also been applied with success to reverse engineer FSMs from traditional web applications [55].

These approaches are applicable to RIAs, provided that key differences between a RIA and a traditional web application presentation levels are taken into consideration.

In particular, at the presentation level, a traditional web application can be considered as a form-based software system [16, 61] where human-computer interaction is session-based and composed of an alternating exchange of messages between user and computer. The web application, indeed, is based on a multi-page interface model, where the user submits some input on the current web page, the server executes some data processing and responds by presenting a new page.

Vice-versa, the interface model of a RIA can be considered as a single-page model where changes are made to the single page components without the need of refreshing the page entirely. Page changes are produced by elaborations triggered on the page components by several types of events (such as user events, time events, or other asynchronous events) and performed by event handlers.

As an example, in RIAs implemented with Ajax-based technologies, the evolution of the client interface corresponds to the run-time evolution of the DOM associated with the RIA web page. In particular, the DOM (e.g., Document Object Model) is the document model proposed by the W3C providing a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them [62]. The DOM of a web page defines a tree data structure, made up of components (e.g., element nodes), that is processed by the browser for rendering it on the client interface.

During the execution of an Ajax-based RIA, external events trigger elaborations (e.g., event handlers) which may both involve the DOM tree element nodes[2], and other related data structures instantiated at run-time (such as XHR requests). Each triggered elaboration will certainly produce a change in the client-side accessible data structures of the application, and potentially will result in a new configuration of the client interface.

[2] In Ajax, several types of DOM events allow various *event handlers* (or listeners) to be registered on the DOM element nodes, where each event handler is a piece of code that is executed when particular events occur.

This client-side run-time behaviour of a RIA can be specified by the UML class diagram reported in Figure 2.1.

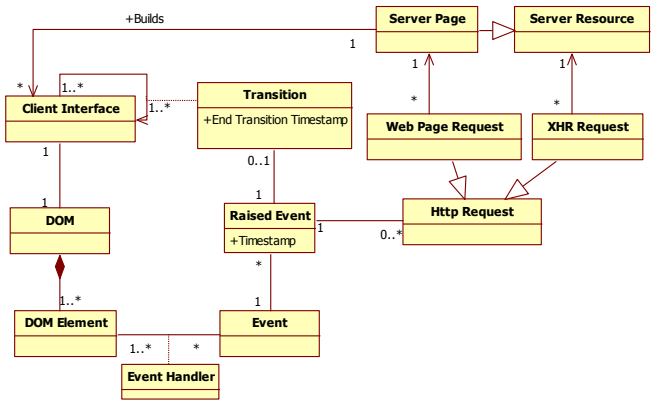

Figure 2.1: The conceptual model of a RIA client-side behaviour

The model describes the behaviour in terms of the Client Interfaces shown by the RIA as far as several types of events are triggered, and explicitly represents registered events, event handlers, raised events and some information about the processing triggered by raised events. In particular, the model shows the various RIA's Client Interfaces where each Client Interface is associated with the corresponding DOM configuration. A DOM is composed of DOM elements, a DOM element can be associated with 0 or more Events registered with the element, and this pair is associated with the corresponding Event Handler. An Event Handler may be either declared in the script code explicitly, or may be implicitly pre-defined (such as for the default event handlers of click events on hyperlink objects, or on form submit buttons).

During the RIA execution, the occurrence of each Event at a given Timestamp will be registered by a Raised Event, and the processing triggered by the Raised Event may result in a Transition that links the starting Client Interface to an ending one reached at the End Transition Timestamp. Moreover, the handling of a raised event may instantiate a set of HTTP Requests, which may either be Web Page Requests to Server Pages, or XHR Requests directed to any Server Side Resource.

As well as the behaviour of a traditional web application can be described by a Finite State Machine whose states and transitions can be deduced by analysing the sequence of different web pages rendered by the browser during the web application execution [63, 55], a similar approach can be used for a RIA too.

In particular, in the case of a RIA, we propose to specify the client-side behaviour of the application by a FSM=(S, T) made of a set S of states, and a set T of transitions, where each state from S is associated with a distinct Client Interface generated by the RIA at run-time, and each transition from T corresponds to a raised event that produced a new Client Interface. Of course, such a definition of the FSM exposes to a well-known problem of state explosion. Indeed, each new DOM configuration that is reached after an event-driven processing is a distinct state of the FSM. To solve this problem, equivalence criteria can be exploited for determining both equivalent states and equivalent transitions inside the State Machine, and simplifying it accordingly. Such criteria can be defined by considering both the structure of each client interface, and the processing that can be triggered by interacting with its elements. Several proposals of these criteria have been defined in the context of a Reverse Engineering process that was designed for generating the FSM of an existing RIA. This process and proposed criteria will be presented in the next section.

2.2 Reverse Engineering Process

The Reverse Engineering process proposed in this section aims at reconstructing a Finite State Machine modelling the behaviour of an existing Rich Internet Application using a combination of dynamic analysis and clustering techniques involving the RIA user interfaces. The process is based on two sequential steps of Extraction and Abstraction, respectively. The Extraction step implements the RIA dynamic analysis for tracing the sequence of event-driven Client Interfaces produced during the RIA execution, as well as some event handling related processing information. The information collected during this step can be used to build a direct graph, the RIA Interface Transitions Graph (TG), whose nodes represent the RIA client interfaces, and edges represent events that cause the

generation of new interfaces. Of course, using dynamic analysis techniques for collecting information about the RIA execution behaviour requires that two main problems are addressed. The former is a problem of adequacy of the set of traced executions for representing all the relevant behaviours of the RIA. This problem can be addressed by adopting suitable strategies for assuring that traced executions capture all possible behaviours of a RIA during the execution of its use cases. The latter problem is a problem of explosion of the number of different client interfaces produced during the RIA executions. To solve this problem, the proposed reverse engineering process introduces the Abstraction step, where equivalence criteria are exploited for clustering together equivalent client interfaces and equivalent transitions between interfaces of the TG, and generating a FSM correspondently. Additional details about both Extraction and Abstraction steps will be presented in the next sub-sections.

2.2.1 Extraction step

The Extraction step is actually a tracing activity of user sessions where the RIA is executed in a controlled environment in order to trigger sequences of events (making up the execution of specific use cases of the application) and to register corresponding results available on the client side of the application.

This tracing activity can be modelled by the statechart diagram shown in Figure 2.2 that includes two main iterative states, the Event Waiting and the Event Handling Completion Waiting one.

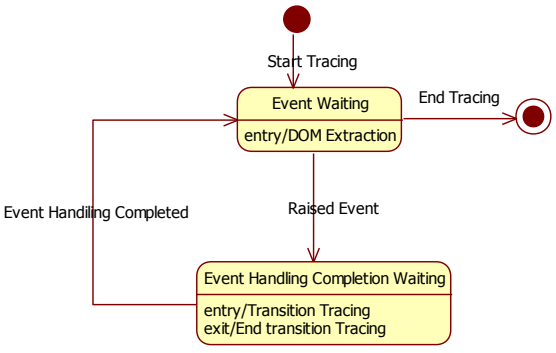

Figure 2.2: The Tracing Activity of the Extraction Step

When the Start Tracing initial state is entered, a new Web page is loaded and rendered by the browser, and the Event Waiting state is reached where the page remains frozen until any event rises.

Entering in the Event Waiting state, the DOM Extraction activity, consisting of extracting and storing information about the structure of the currently rendered DOM, is carried out.

The occurrence of a raised event causes a transition from the Event Waiting state towards the Event Handling Completion Waiting state.

Three main types of events are able to trigger this transition:

- user events, corresponding to user actions made on any input device, such as mouse or keyboard;
- time events, due to the occurrence of a timed condition;
- http response event, due to receptions of responses to some http request, such as a request for a web page, or an asynchronous Ajax (XHR) request.

During the reverse engineering process, in order to avoid loss of data or inconsistent data, the execution of the RIA should be controlled by delaying a new event occurrence until the DOM Extraction activity in each Event Waiting state has been completed. While entering in the Event Handling Completion Waiting state, a Transition Tracing activity is carried out consisting of the extraction and storage of information related to the raised event, such as its type, its timestamp of raising, and the DOM element node which it has been raised on. When the event handling is completed, the time of the event handling termination is stored (End Transition Tracing activity) and the reverse engineering process returns in the Event Waiting state, for continuing the Tracing activity. While in the Event Waiting state, the Tracing activity can be stopped by the reverse engineer in order to exit from the Extraction step of the process.

2.2.2 Abstraction step

At the end of the Extraction step, the reverse engineering process enters the Abstraction one, where data extracted during the tracing activity is analysed in order to obtain the FSM modelling the RIA behaviour.

The abstraction of this model is accomplished in two steps: in the first step, a Transition Graph whose nodes represent the generated RIA client interfaces, and edges represent events that caused the generation of new interfaces, is built.

In the second step, this graph is analysed and Clustering techniques are used for grouping together its equivalent nodes and transitions. The resulting graph is submitted to a Concept assignment process, which will finally generate the FSM.

The clustering techniques exploit equivalence criteria of the client interfaces based on the analysis of the corresponding DOM configurations. In particular, we have considered (and experimented with) several heuristic criteria such as the one that considers two client interfaces to be equivalent if their DOMs include the same set (or sub-sets) of 'active element nodes', that is elements with registered events of selected types (such as user event, time event, asynchronous events, etc.) and having the same event handlers.

As a consequence, transitions of the TG between equivalent client interfaces and associated with the same type of event will be considered equivalent and clustered together too.

After the completion of the clustering task, a simplified TG will be obtained. This graph will be submitted to a Concept Assignment task where each node of the TG will be initially assumed as a distinct state of the State Machine, and the software engineer knowledge and experience will be needed for validating or discarding this hypothesis. The transitions between states will be deduced accordingly.

At the end of this task, the FSM modelling the behaviour of the RIA will be finally obtained.

2.3 The Reverse Engineering Tool

The proposed Reverse Engineering process can be executed with the support of the RE-RIA (Reverse Engineering RIA) tool that provides an integrated environment where the different activities of the process can be performed, and their intermediate results are persistently managed.

The tool architecture is shown in Figure 2.3.

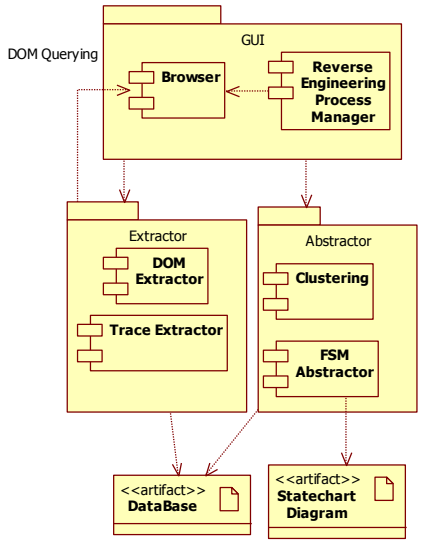

Figure 2.3: The RE-RIA tool architecture

The RE-RIA tool comprehends three packages, named GUI, Extractor and Abstractor, and a relational database that stores the extracted information and produced abstractions. A description of the package components, their functionality and of technological solutions used for implementing them follows in the next sub-chapters.

2.3.1 GUI Package

The GUI package comprehends two components, namely Browser and Reverse Engineering Process Manager. The Browser is actually the instantiation of a Mozilla Firefox Browser inside a Java GUI, allowing RIAs to be navigated trough the GUI, while their structure and behaviour can be at the same time accessed by other components of the tool. In particular, we used the Standard Widget Toolkit (SWT) [64], that is an open source widget toolkit for Java providing abstract classes and packages for instantiating HTML Browsers. Moreover, the browser uses the same rendering engine Gecko [65] used by the Mozilla Firefox browser for rendering the DOM. The Reverse Engineering Process Manager component is another Java GUI providing the user with several functionalities for the reverse engineering process management. The component allows starting and

stopping a user session tracing, starting a clustering session, and setting some process parameters, such as types of data that will be captured during the RIA execution, as well as the Equivalence Conditions that will be applied by the clustering sessions. A screenshot depicting the Reverse Engineering Process Manager GUI of the RE-RIA tool is shown in Figure 2.4.

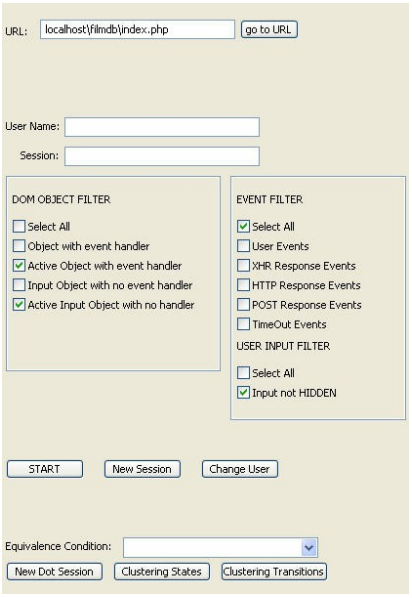

Figure 2.4: The Reverse Engineering Process Manager GUI

2.3.2 Extractor Package

The Extractor package comprehends two components, the DOM Extractor and the Trace Extractor. The DOM Extractor is a Java component interacting with the Browser in order to extract information about currently instantiated DOM element nodes that are rendered by the browser. The access to the DOM elements is made possible by the JavaXpCom library [66], that allows the interaction with the Mozilla browser embedded in SWT by using the full range of public Mozilla Interfaces. The DOM Extractor stores the structural information about the visited RIA interfaces into the database.

The Trace Extractor is a Java component interacting with the Browser Emulator in order to trace and collect information about both event raising and termination of the execution of their event handlers. The information collection has been realised by inserting non-

invasive probes that exploit the bubbling and capturing standard mechanisms defined by the W3C for DOM event dispatching (cfr. [67]). Event dispatching is the technique used by the script engine of Web browsers for propagating a raised event to the event listeners registered to the same event. In the W3C Event model, event handlers can be registered with an event associated with a DOM object (hereafter, the Target object) and characterized by a flag that assumes two values, namely 'capture' and 'bubble', defining the propagation order of events to the DOM element nodes. The Event Dispatch mechanism operates according to a procedure comprehending three sequential phases, called Capture phase, Target phase and Bubble phase, respectively. In the capture phase, the set of the ancestors of the Target object is analysed in descending order, from the DOM root (i.e. the window DOM object) to the Target object. Event handlers flagged as 'capture' are launched in this phase. In the target phase, the event handler associated to the Target object is executed. In the bubble phase, the ancestor set is visited in reverse order, from the Target object to the root, and the event handlers flagged as 'bubble' are launched too. The Trace Extractor is able to capture and trace the events of given types occurred in a tracing session thanks to two types of event handlers, named H_{raise} and $H_{termination}$, that are added to the window element (that is the DOM root) at the start of the session for each type of event that needs to be captured. The H_{raise} handlers are flagged as 'capture', so that each of them will be the first handler executed after the raising of any event of the specified type, and will be responsible for storing information about the raised event into the database. The $H_{termination}$ handlers are flagged as 'bubble', so that each of them will be executed after the termination of all the event handlers associated to the same type of raised event, and will be responsible for storing information about the termination of the event handling.

2.3.3 Abstractor Package

The Abstractor package comprehends two components, the Clustering component and the FSM Abstractor, respectively. The Clustering component is a Java component implementing the clustering techniques described in section 3.3 for simplifying the Trace

Graph obtained after the event tracing activity. The resulting Trace Graph is stored in the database. The FSM abstractor is another Java component designed to support the Concept Assignment task performed by the software engineer for abstracting the FSM from the Trace Graph stored in the database. In particular, the component offers a GUI where choices made during the abstraction task can be inputted and stored into the database.

2.4 Case Study

This section presents a case study that was carried out for exploring the feasibility and effectiveness of the proposed reverse engineering approach. In particular, the case study involved a medium-size open source Rich Internet Application which was submitted to the reverse engineering process. Using the support provided by the RE-RIA tool, some FSMs describing how the application behaves were deduced. The reverse engineering process we performed and its results are described in the following. The subject of the experiment was an Ajax-based RIA named FilmDB [68] that provides registered users with several functionalities for the management of a personal movie archive, such as visualisation of a movie description, insertion, modification, deletion of a movie description and search for movies in the archive, management of movie loans and so on. The server side of this application is implemented by 99 PHP server pages (624 kBytes) that generate client pages containing several scripts (implemented in JavaScript) able to realise a complex user interface. Moreover, FilmDB interacts with server side resources (in particular with the imdb.com web site to obtain movie data) by exploiting Ajax requests. A user documentation of FilmDB is available on line; it provides some indications about its main user functions, but it does not describe detailed information about the behaviour of its interface. Several use cases offered by the application were submitted to the reverse engineering process. In the Extraction step each use case was executed several times with different input data and actions in order to exercise various use case scenarios. These user sessions were traced automatically by the tool and registered by a number of execution traces. In the Abstraction step, for each use case, information collected by the tool about the related execution traces was used for deducing an FSM associated with the RIA

corresponding behaviour. This step required two tasks: the first task involved the production of a Transition Graph (TG) associated with collected traces. The second task was devoted to the TG analysis, and Clustering techniques were used for grouping together its equivalent nodes and transitions. The resulting graph was submitted to a Concept assignment process, which finally generated the FSM. In the following, we report data about the process that was performed for two specific use cases, the former one allowing a user to enter his/her personal movie area, and the latter for exiting this area. The first use case was characterised by three alternative scenarios (unsuccessful login of unregistered user, successful login of the administrator user, and successful login of a generic user), so that three executions were necessary for exercising all them. The exit use case presented just a single scenario that was exercised twice. The process was performed using the RE-RIA tool, whose Trace Extractor component captured and stored various data about monitored session. Table 2.1 reports synthetic data about these sessions captured by the tool, such as #Client interfaces that were generated, #DOM element nodes of these interfaces, #traced events (of various types).

Table 2.1: Synthetic data about traced sessions

#Client interfaces	60
# Extracted DOM elements	6015
# Traced transitions	59
# Total traced User events (of which)	42
# Click on a DOM element	11
# Mouseover on a DOM element	9
# Mouseout on a DOM element	8
# Keydown	14
# XHR response reception events	8
#Client interface reception events	4
# Timeout events	5

The Transition Graph associated with these execution traces was complex enough, including 60 nodes and 59 edges. In the Abstraction step, various clustering heuristic techniques were used for simplifying this Graph, grouping together its equivalent nodes and transitions. As to the client interfaces grouping, two different clustering criteria were experimented with: the former criterion (C1) considered two client interfaces to be equivalent if they included the same set of ‘active’ DOM elements (e.g., having the same

registered events and event handlers), while (C2) criterion considered two client interfaces to be equivalent if they included the same set of 'active' DOM elements and have the same set of instantiated XHR, Http Requests, or timeout listeners. As to transition grouping, the criterion that considers being equivalent two transitions of the TG if they link equivalent client interfaces and are produced by the same type of event was used. As to the simplified TG produced by criterion (C1), it comprised 8 nodes and 22 transitions (with respect to the 60 nodes and 59 edges of the initial graph).

This TG was submitted to a Concept Assignment task, where each node of the TG was initially assumed as a distinct state of the Finite State Machine, and the software engineer knowledge and experience were needed for validating or discarding this hypothesis.

This concept assignment step revealed that most TG nodes could be associated with meaningful states of the FSA, while some nodes could not be associated with meaningful states but had to be further split.

As an example, this problem was encountered with respect to a node of the TG that clustered together several RIA's client interfaces having the same DOM element nodes, but differing just for the set of instantiated XHR objects, Http Requests, and timeout listeners.

As to the simplified TG produced by criterion (C2), it comprised 12 nodes and 23 transitions (with respect to the 60 nodes and 59 edges of the initial graph).

The concept assignment revealed that all nodes of this TG could be associated with meaningful states of the RIA behaviour. Indeed, graph nodes associated with RIA's client interfaces having the same DOM element nodes, but differing just for the set of instantiated XHR objects, Http Requests, and timeout listeners were correctly considered non equivalent.

Therefore, this graph reconstructed correctly the behaviour of the RIA, thanks to the correct consideration of Ajax synchronization communication mechanisms.

This result confirmed that clustering criteria are able to influence the effectiveness of the FSM reverse engineering process.

The resulting FSM is shown in Figure 2.5, while Table 2.2 describes the meaning assigned with each state of the FSM.

Table 2.2: FSM states – criterion (C2)

1	Home Page, no logged users
2	Wait for login form
3	Login form
4	Wait for server authentication
5	Obtained authentication, wait for synchronization
6	Login failed
7	Obtained authentication, wait for page reload
8	Home Page, user logged
9	Wait for logout
10	Logged out, wait for synchronization
11	Logout notification, wait for home page reload
12	Logout notification

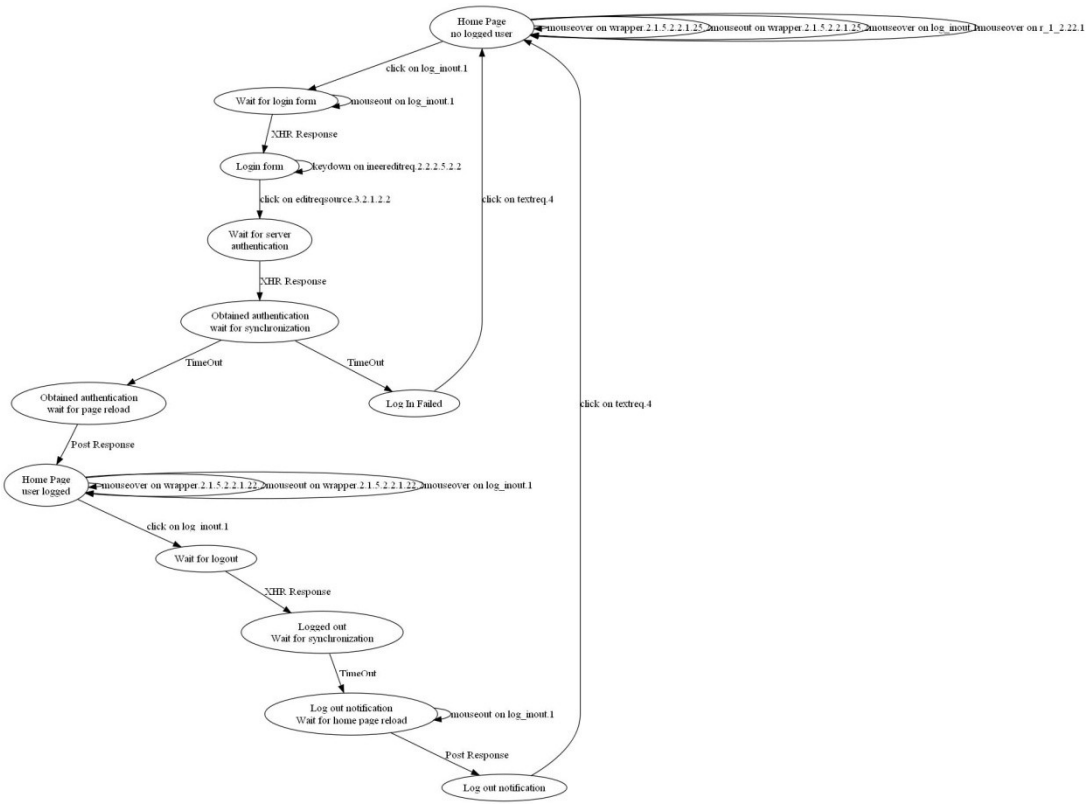

Figure 2.5: The Resulting FSM

2.5 Conclusions

In this chapter we have presented the initial results of a reverse engineering research project that aims at defining and validating effective reverse engineering processes and techniques for reconstructing suitable representation models of Rich Internet Applications.

The research preliminarily addressed the problem of modelling the client-side behaviour of a RIA by reverse engineering techniques based on the dynamic analysis of the application. Dynamic analysis is a necessary technique for reconstructing all possible behaviours exhibited by event-driven applications, but it exposes to several problems, such as the problem of assuring a full coverage of all possible RIA behaviours, as well as a problem of potential state explosion.

To solve the first problem, well known input selection strategies assuring the needed coverage (and already used for software testing aims) can be adopted with success. Vice-versa, a possible approach for managing the state explosion problem is offered by clustering techniques that exploit equivalence criteria for recognizing equivalent behaviours and classifying them correctly.

We have proposed to cope with the second problem using heuristic clustering criteria, whose effectiveness has been assessed in a preliminary experiment. The case study we performed showed the feasibility of the proposed reverse engineering approach, highlighting future works to be addressed. In particular, we plan to extend the experimentation with further case studies in order to assess the scalability of the approach. Moreover, the adequacy of the proposed model for supporting maintenance and testing activities involving the RIA, will be addressed, too.

In the next chapter we present an experimentation of the presented technique that we performed in order to assess its effectiveness.

Chapter 3 [3]

EXPERIMENTING THE REVERSE ENGINEERING TECHNIQUE FOR MODELLING THE BEHAVIOUR OF RICH INTERNET APPLICATIONS

In the previous chapter we have proposed the use of Finite State Machines (FSMs) to represent the behaviour of AJAX [17] applications, and presented a reverse engineering technique and a tool for obtaining them from existing applications using dynamic analysis [1]. In this chapter, we present the results of an experiment that aimed at assessing the effectiveness of this technique in reconstructing a FSM model of the RIA behaviour that can be used for maintenance, evolution, or re-documentation purposes

3.1 The proposed FSM model and the reverse engineering technique

Finite State Machines, which have also been used with success for modelling traditional Web applications [63], provide an abstract view of a system in terms of states and transitions among them. More precisely, a FSM representing an RIA behaviour will be a couple (S, T) where S is a set of states reached by the RIA during its processing, T is the set of transitions between states.

We propose of representing in the FSM all the elaboration states where the RIA receives any input solicitation by its user (state abstraction criterion), and of describing each state of the RIA by the User Interface shown to the user at that interaction time (representation criterion). In this model, transitions between states will be associated with user

[3] This chapter was published in the Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009).

interactions (e.g. user events) that triggered the RIA migration towards the new state. This FSM-based model of the RIA behaviour can be obtained by a four-step dynamic analysis based technique that, in a first step, records a set of execution traces of the RIA from user sessions. An execution trace will be modeled as a sequence of couples (I_i, $event_i$), where each I_i represents a user interface state and each $event_i$ is the user event occurred on that interface during the execution. For obtaining these execution traces, we use a non-invasive technique that does not instrument the code of the application directly, but rather the browser that renders it [1]. Once a set of execution traces (representative of all possible RIA behaviours) has been collected, the second step of the technique addresses the problem of detecting and filtering out redundant information contained in this set. In particular, user interfaces with the same set of 'active widgets' (i.e. elements with registered event listeners) and offering the same interaction behaviour to their users (by means of the same set of event handlers) will be considered as equivalent, and will be substituted by the corresponding equivalence class. Table 3.1 reports the definitions of three interface structural equivalence criteria *C1*, *C2*, and *C3* that have been proposed for finding equivalent user interfaces.

Table 3.1: Interface equivalence criteria

C1: two client interfaces I1 and I2 are equivalent if the same active widgets of I1 are also included in I2 and vice versa, and they have the same indexed path, the same type of corresponding listeners, and corresponding event handlers with the same name.
C2: two client interfaces I1 and I2 are equivalent if the same active widgets of I1 that are visible and enabled are also included in I2 and vice versa, and they have the same indexed path, the same type of corresponding listeners, and corresponding event handlers with the same name.
C3: two client interfaces I1 and I2 are equivalent if the same active widgets of I1 that are visible and enabled are also included in I2 and vice versa, and they have the same un-indexed path.

When the trace collection activity ends and its included equivalent user interfaces have been found, the FSM abstraction step of the technique (third step) can be entered in order to obtain a machine modelling the behaviour of the analysed application. The resulting FSM=(S, T) will include a set S of states corresponding to all interface equivalence classes discovered by a considered equivalence criterion, while the set T of transitions will be defined on the basis of recorded transitions between consecutively visited client interfaces.

Finally, a model validation step (fourth step) is required for assessing the correctness/adequacy of the reconstructed FSM, and for assigning each validated state with a meaningful description. Generally speaking, the correctness of such a model depends on the objectives of the task the model was produced for (such as comprehension, testing, maintenance, etc.), and its evaluation will be based on the judgment of an expert of the task. The proposed Reverse Engineering technique can be executed with the support of the RE-RIA (Reverse Engineering RIA) tool that provides an integrated user-friendly environment where execution traces collection, traces analysis and classification, and FSM abstraction and validation activities can be performed.

3.2 The experiment

This section illustrates an experiment that was carried out using a set of real RIAs. The experiment was designed *(1) for assessing the effectiveness of the reverse engineering technique*, and *(2) for analysing its cost-effectiveness ratio*.

For evaluating the effectiveness of the technique in reconstructing a behavioural model of an RIA we analysed the correctness of the FSM model produced by it. The FSM correctness can be evaluated by comparing two FSM models, e.g. the model *M* produced by the technique from a given set of execution traces *T* of a RIA, and a reference model *O* (the so-called Gold Standard) which would have been produced by an expert from the same set of execution traces *T*. Since both the expert and the technique actually distribute the set of visited interfaces *I* into a set of partitions (i.e. the states of the FSM models), we decided of comparing the models *M* and *O* by evaluating the edit distance *d(M, O)* proposed in [69] between these partitions. Using such an approach, we measured the reverse engineering technique effectiveness by the following *Correct Interface Ratio (CIR)* metric:

$$CIR\ (M) = 1 - d(M, O)/\ |I|$$

where *CIR* = 100% indicates that *M* and *O* partitions are exactly the same. As to the evaluation of the cost *C* of the proposed technique, we took into account the costs of its

single steps. These costs include: C_{coll} that is the cost of collecting user session traces (semi-automatic task), $C_{analysis}$ that is the cost needed for classifying analysed interfaces into a set of equivalence classes on the basis of the selected equivalence criterion (automatic task), C_{abstr} that is the cost needed for defining the FSM on the basis of the recovered interface equivalence classes (automatic task) and C_{val} that is the cost needed to validate the obtained FSM (manual task). C_{val} includes the cost C_v of validating the proposed state for each interface and the cost C_{mov} of moving incorrectly classified interfaces towards the expected classes of the Gold Standard model. Intuitively, C_{coll}, C_{abstr} and C_v depend on the number of analysed trace interfaces, and grow with it. $C_{analysis}$ depends both on the number of analysed trace interfaces, and on the number of active widgets included in analysed interfaces, while C_{mov} grows with the number of interface move operations needed for correcting the reconstructed model, that is with the partition edit distance $d(M, O)$.

If we consider as negligible all automatic activity costs, the most relevant cost factors include C_{coll} and C_{mov}.

In the experiment we used the following materials and procedures.

Subject applications included the following four distinct available online RIAs:

- W1: http://app.ess.ch/tudu/welcome.action
- W2: http://www.pikipimp.com
- W3:http://www.agavegroup.com/agWork/theList/theListWrapper.php
- W4: http://www.buttonator.com

We involved in the experiments two software engineers and five under-graduate students from the Software Engineering courses held at the University of Naples, in Italy.

A set of two/three students per application were trained about the application use cases (and their normal and alternative scenarios), and were asked for collecting a set of user session traces.

We asked each student for covering each use case of the application at least two times with their user sessions. This task was accomplished with the support of the RE-RIA tool

and returned a set of execution traces ET per application.

The experts produced a FSM reference behaviour model for each Web application, the so called *'Gold Standard' (GS)* model, to be used for comparative analysis. Each *GS* model was obtained by analysing the set of all collected execution traces *ET* (with the support of RE-RIA tool) and provided a specific partitioning of execution trace interfaces.

Table 3.2 reports, for each application, the number of considered use cases *(UC)* and alternative scenarios *(SC)*, the number of collected User Session Traces *(UST)* and interfaces (*I*), and the number of the corresponding *GS* States (*GS-s*) and transitions (*GS-t*).

For each application and for each analysed execution trace, three FSM models *M1*, *M2*, and *M3* were finally obtained automatically on the basis of a different interface equivalence criterion (*C1*, *C2*, and *C3*, respectively). Each model provided a different partitioning of execution trace interfaces.

For each model *M*, the partition distance *d (M, GS)*, and the *CIR* metric values were finally computed.

Table 3.2: Data about subject applications

RIA	UC	SC	UST	I	GS-s	GS-t
W1	8	17	30	1885	15	52
W2	1	2	8	533	4	16
W3	3	10	11	731	4	9
W4	1	8	11	829	19	54

3.2.1 Experimental results

For discovering the factors affecting the technique effectiveness, we analysed the correctness of the reconstructed FSM models as the subject RIA, the Interface equivalence criterion and the considered execution trace changed.

For brevity, the following Table 3.3 reports summary data about the experiments involving just the W1 application.

In particular, the table shows the number of states (#S) of the FSM and *GS* models, and the corresponding values of *CIR* which we obtained as the length of the trace (*TL* in the table)

and the equivalence criterion varied.

Table 3.3: Experimental data about W1

	C1		C2		C3		GS
TL	CIR	#S	CIR	#S	CIR	#S	#S
35	68%	9	37%	15	100%	8	8
51	49%	17	63%	23	94%	10	10
93	48%	24	61%	34	85%	10	15
141	27%	28	38%	42	90%	10	15
251	23%	67	25%	95	91%	10	15
604	13%	142	16%	204	88%	10	15

Table 3.3 shows that the FSM models produced by criterion C3 well approximated the Gold Standard model, whatever the length (*TL*) of considered trace (indeed, the *CIR* values were always not less than 85%). This trend was not so good for those models produced by criteria *C1* and *C2*. In order to explain the effectiveness difference among the different criteria, we analysed the characteristics of RIA interfaces included in the set of analysed traces. We deduced that the *C3* criterion worked well (that is, it classified equivalent interfaces effectively) if the RIA interfaces mostly presented collections (such as tables or lists) of active widgets with the same tag, but with different and dynamically defined sizes of the collections. Vice-versa, *C2* worked well in case of interfaces without this type of collections. Finally, *C1* was the less effective criterion in both types of interfaces, since it did not consider the visibility and enabling properties of active widgets. Hence, we concluded that the interface equivalence criterion actually influenced the effectiveness of the technique.

As to the cost-effectiveness, we also studied the trends of the main cost factors C_{coll}, and C_{val} as the subject RIA, the Interface equivalence criterion and the considered execution trace changed. Since experimental data showed that the *CIR* values did not significantly improve with the size of the trace, for reducing the cost of the technique without affecting its effectiveness we deduced that it would be necessary to find the shorter execution trace that allows the abstraction of the FSM model having the best *CIR* value.

As the data in Table 3.3 show, the number of states of FSM models produced by *C3* definitely tended to a stable value likewise the *GS* number of states, while it did not happen for models produced by *C1* and *C2*. Hence, we could hypothesize a possible

criterion for selecting the execution trace and the equivalence criterion for producing a suitable FSM model with the minimum cost. This cost-effective selection criterion indicates (1) *of choosing the criterion where the number of states of the reconstructed FSM assumes a stable value*, and (2) *of choosing the model produced by this criterion from the smaller trace in correspondence of which the number of FSM states assumes the stable value: the related FSM model will be the most cost-effective one.* Using such a criterion, we were actually able to select the most cost-effective model for each RIA, hence we validated the proposed criterion. Table 3.4 reports summary data about the selected models. In particular, for W1, W2 and W3, this model was produced by the C3 criterion, while for W4 two acceptable FSM models were reconstructed both by C2 and C3 criteria.

Table 3.4: Data about FSM models with the best cost-effectiveness ratio

RIA	Trace length	Best Criterion	FSM states	CIR	Edit distance
W1	93	C3	10	85%	14
W2	23	C3	2	65%	8
W3	40	C3	4	100%	0
W4	60	C2	19	100%	0
W4	60	C3	19	62%	23

3.3 Conclusions

In this chapter we presented the results of a validation experiment involving four real Web applications that showed the cost-effectiveness of the proposed reverse engineering technique for obtaining a model of the behaviour of a Rich Internet Application by dynamic analysis. The experiment showed that a key point of the proposed reverse engineering technique is the interface equivalence criterion that allows dynamically produced execution traces of the application to be analysed and simplified, in order to abstract a representative model of the RIA behaviour. Experimental data showed that these criteria are able to influence the effectiveness of the technique, as well as its cost-effectiveness. However, these criteria are general and reusable for any type of client interfaces of RIAs, differently from the technique [56] that requires that specific features allowing the correct classification of equivalent states be tailored manually with

application-specific mechanisms. Moreover, their effectiveness on discriminating different states is not dependent on the choice of any similarity threshold, differently from the Levenshtein distance-based technique proposed by [58]. The proposed reverse engineering approach is actually a waterfall process made by three steps, Extraction, Abstraction and FSM model validation, that have to be sequentially executed. In the next chapter we'll present an iterative agile reverse engineering process more effective than the one proposed in the previous chapter.

Chapter 4 [4]

AN ITERATIVE APPROACH FOR THE REVERSE ENGINEERING OF RICH INTERNET APPLICATION USER INTERFACES

Comprehending and modelling the behaviour of user interfaces exposed by Rich Internet Applications (RIAs) are important activities in software maintenance, testing, and evolution. This chapter presents an 'agile' process for the reverse engineering of Rich Internet Application User Interfaces: the process is based on dynamic analysis of the application, is iterative and exploits heuristic clustering criteria for reducing the data gathered by dynamic analysis. Moreover, it is based on the continuous validation feedback of the process executor, and allows the incremental reconstruction of a Finite State Machine for modelling the behaviour of RIA GUIs.

4.1 Introduction

As Rich Internet Applications (RIAs) are the new generation of Web applications that, besides the traditional server-side elaborations, provide client-side processing and asynchronous communication with the server, which make them more dynamic, interactive, responsive, and usable than traditional Web applications. In particular a RIA offers a rich user interface, which is programmatically built at run-time on the basis of user interactions with the application. For comprehending the characteristics and the behaviour of this type of interface, static analysis of the application code does not suffice, while dynamic analysis techniques can be used.

[4] This chapter was published in the Proceedings of the 5th International Conference on Web Applications and Services (ICIW 2010).

According to Cornelissen et al. [70], using dynamic analysis in program comprehension contexts has the benefits of the precision with regard to the actual behaviour of the software system, and of enabling goal-oriented analysis strategies where only the parts of interest of a software system can be exercised. On the other side, dynamic analysis limitations include the incompleteness, which depends on its inability in covering all possible program executions, and the scalability that is associated with large amounts of data collected at run-time.

For solving the scalability issue, over the last years several reduction techniques based on abstractions or heuristic criteria have been proposed in the literature to group parts of the program executions having similar properties. Reduction techniques can be either applied a-posteriori, once all the execution data have been collected, or step-by-step during the data collection activity. The former strategy has the advantage that the reduction techniques do not impact the process of execution trace collection at all, but their feedback usually comes too late, while the latter ones are able to provide an early feedback on the collection process. A possible classification of trace reduction techniques and a methodology for assessing them have been recently presented in [10].

Specific reduction techniques have also been proposed in the context of dynamic analysis of RIAs performed with the aims of reverse engineering [1, 2], crawling [58] or testing the RIA [56, 72, 57, 83], respectively. We, in particular, have proposed [1] and experimented with [2] some heuristic criteria for clustering similar user interfaces of a RIA that are built at run-time. The criteria have been defined in the context of a reverse engineering process that recovers from execution traces of the RIA a Finite State Machine (FSM) modelling the GUI behaviour. This process requires that the heuristic criteria be applied after the trace collection activity, and involve the complete set of collected interfaces. Moreover, the abstractions proposed by the criteria must be validated by a human expert on the basis of his knowledge about the application.

To improve the overall effectiveness of that approach, in this chapter we propose an improved, more ‘agile’ version of the reverse engineering process. This process version is

iterative, the feedback about the reduction techniques is provided at each process iteration, and it supports the incremental reconstruction of the FSM of the RIA.

This approach is supported by CReRIA, an integrated reverse engineering environment that provides automatic facilities for executing the process and incrementally recovering and validating the FSM. In this chapter both the approach and the environment will be presented, as well as the results of an experiment that was carried out for preliminarily assessing the feasibility of this technique.

4.2 The Iterative reverse engineering Process

In previous chapters we addressed the problem of reconstructing a FSM for modelling the behaviour of the client side of a RIA, and proposed a dynamic analysis technique, supported by the ReRIA tool, that exploits data collected from user sessions for abstracting this model. Since we aimed at obtaining a model of the interaction of a user with the RIA user interface, we decided to represent in the FSM all the interface states where the RIA receives an input solicitation by its user, and for describing each state of the RIA by the user interface shown to the user at the interaction time. As to the transitions between states, we associated them with the user interactions (e.g., user events) that moved the RIA towards the new state.

For obtaining this model we proposed the three-step process illustrated in Figure 4.1, including the sequential activities of Execution Trace Collection (performed by an instrumented Web browser), Trace Analysis (using a set of heuristic clustering criteria), and FSM Model Abstraction where an expert of the application analyses the abstractions proposed by the criteria, validate or refuse them on the basis of his personal knowledge about the application, and finally obtains a FSM of the application.

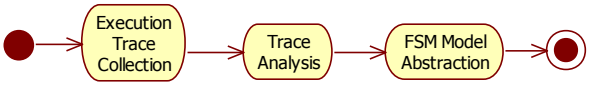

Figure 4.1: RIA reverse engineering process proposed

Like any waterfall-like process, this process has some limitations, such as: (1) the FSM is obtained only at the end of all the three steps, (2) the process does not provide any feedback from the late steps to the early ones, and (3) the FSM abstraction step relies on a costly and human intensive validation activity. On the contrary, the comprehension approach proposed in this chapter and illustrated in Figure 4.2 is not affected by these limitations since it assumes that the FSM model can be obtained incrementally by an iterative process including, at each iteration, the steps of User Interaction, Extraction, Abstraction and Concept Assignment.

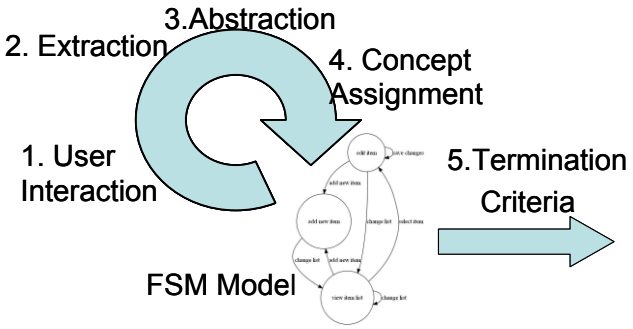

Figure 4.2: The iterative comprehension process

The process can be performed by any software engineer who aims at gaining an understanding of the RIA user interface behaviour just acting as a user of the application. The process starts with the User Interaction step where the user interacts with the RIA and fires an event on its current user interface: this interaction must be, of course, performed in a controlled navigation environment (such as the one offered by the CReRIA tool) that registers all the interactions and the needed information about them.

In the successive Extraction step, information about current interface, fired user event and the user interface that is obtained after the event processing, must be extracted and persistently stored. Using a controlled navigation environment, this step can be performed in a straightforward manner too.

The Abstraction step is performed using some heuristic criteria (such as the ones discussed in section 4.3) that evaluate the degree of similarity of the current user interface with the previously produced ones, as well as the similarity among occurred events. Each distinct

heuristic criterion creates a different clustering of interfaces (and events) into equivalence classes.

The Concept Assignment is actually a comprehension [73] and validation step where the software engineer has to validate the clustering proposed by the heuristic criteria and accepts or refuses them. If the clustering is refused, he has to propose the correct concept to be assigned. In this way, the expert incrementally reconstructs a FSM modelling the behaviour of the RIA GUI, since he either associates the current interface with a new class of interfaces (and a new FSM state), or with an already existing interface class (and FSM state). Analogously, he associates the current event either with a new or an already existing transition between states of the FSM. The proposed iterative process ends on the basis of a termination criterion, such as the one which considers the event coverage reached by the process, or the coverage of known scenarios of the application, or the effort (e.g, the time spent) devoted to the whole process.

The main difference between the processes reported in Figure 4.1 and Figure 4.2, respectively, is that the former process first collects execution traces, hence clusters their content by heuristic criteria, and then requires an expert to validate all proposed interface clustering and event clustering for abstracting the FSM. This validation task requires a huge (and too expensive) effort that makes it almost impracticable. In contrast, in the latter process the interface clustering and event clustering are produced iteratively, so that the validation effort is smaller and manageable (at each iteration). Moreover, it is possible to use the continuous feedback provided by process iteration data for executing the successive iterations of the process in a more effective manner and to determine when the FSM model has been completely reconstructed.

4.3 The Reverse Engineering Environment

CReRIA is the integrated environment for dynamic analysis designed for supporting comprehension processes of RIAs. CReRIA is the subsequent version of the tool RERIA presented in chapter 2. CReRIA both improves the functionalities offered by its ancestor

and offers new ones. The functionalities implemented by CReRIA include:

- offering a Web browser for navigating the rich internet application and performing user sessions tracing;
- extracting and recording relevant information about traced user sessions, such as user interfaces and events that occurred during the navigation;
- clustering of interfaces and events according to different abstraction criteria;
- supporting the Concept Assignment task on the basis of information collected or abstracted in the previous steps of the process.

The software architecture of CreRIA is illustrated by the package diagram of Figure 4.3.

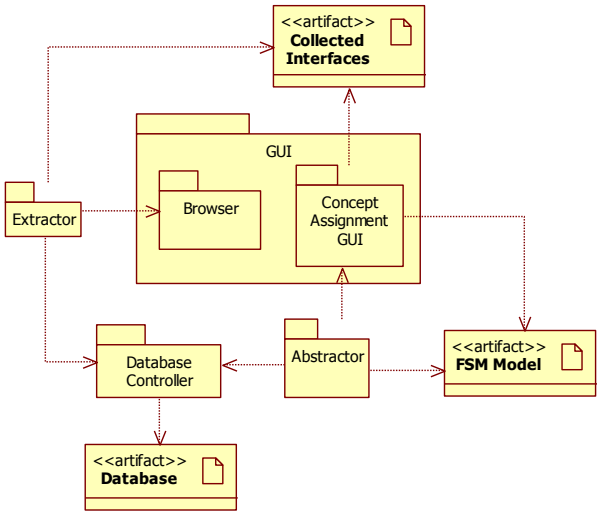

Figure 4.3: The CreRIA tool architecture

The architecture includes five packages named Browser, Concept Assignment GUI, Extractor, Abstractor, and DataBaseController, respectively, and a relational database that stores the extracted information and the produced abstractions. The architecture has been developed with Java technologies and MySQL DBMS. The Browser package supports the user navigation of the RIA. The package actually includes the instantiation of a Mozilla Firefox Browser inside a Java GUI implemented by the SWT Standard Widget Toolkit [64], which allows RIAs to be explored and navigated in a controlled environment where the runtime behaviour can be easily traced.

The Extractor package includes the software components that are responsible for extracting and collecting data about the RIA run-time.

In particular, these components capture both data about the structure and content of user interfaces, and data about user events that were fired on user interfaces during the navigation.

The output of this extraction activity is persistently stored in the Database component of the tool architecture, which is accessed through the DataBase Controller package. Moreover, the Extractor package produces another type of artifact, e.g., a set of Collected Interfaces that are the HTML version of the user interfaces captured during the RIA navigation.

The Abstractor package is responsible for clustering interfaces and events into equivalence classes. Several heuristic clustering criteria have been implemented in CReRIA, which produce different partitioning of both interfaces and events. The Concept Assignment GUI package implements a GUI that offers several aids to the user who performs the Concept Assignment phase of the comprehension process.

As an example, after the generation of a new user interface, this GUI displays the possible clusterings produced by each heuristic criterion, and for each class shows examples of interfaces belonging to the same class.

This GUI also provides the facility for assigning a concept to a new class of interfaces or events, and up-to-dating the FSM accordingly. The resulting FSM model is encoded by the tool in order to be automatically rendered by the graph viewer Dotty [74]. Another functionality offered by this GUI consists of providing some statistics about the FSM such as the number of states and transitions currently making up the model and the number of new states and transitions discovered in the last process iterations: this last datum can be used as an indicator of the stability of the reconstructed FSM model, and can be used as a termination criterion of the comprehension process.

Further details about the Extractor and Abstractor packages are provided in the following sub sections.

4.3.1 The Extractor Package

As to the user interface data extraction, the Extractor captures by the DOM API [62] all the data necessary for describing each user interface by a sub-set of its widgets (such as buttons, text fields, forms, …) having specific properties and values of these properties [44]. As an example, the Extractor characterizes the interface widgets that are potentially able to trigger an event-driven client-side processing of the RIA by the properties listed in Table 4.1. As to the user events, which were fired during the navigation, the Extractor captures the user event type (i.e. click, mouseover, mouseout, etc…) and the id of the widget on which the user event was fired, and records the ids of the current interface and of the one reached after the event handling completion (next interface).

Table 4.1: Widget Property Description

Widget Property	**Description**
Id, Type	Identifier and Type of the widget
InputType	Type attribute for widgets defined by an *INPUT* Html tag
Listeners	Set of Listeners of user event attached to the widget
Event handlers	Set of scripts responsible of event handling.
Active property	a Boolean attribute indicating whether the widget is actually accessible (that is, *clickable*) to end-users, or not.
Absolute Indexed Path (AIP)	DOM path that links the DOM root element with the considered widget.
Un-indexed Path (UP)	UP is similar to AIP, but it does not include the information about the indexed position of the DOM path component elements.
Un-indexed path with id (UPid)	UPid is similar to UP, but it also includes the id attribute of the DOM path component elements.

Figures 4.4-a and 4.4-b show an example interface and the corresponding HTML code of the page body.

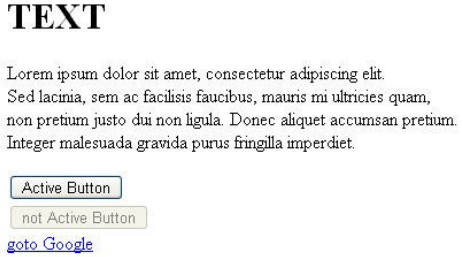

Figure 4.4-a: Example interface

```
 7. <body>
 8. <h1>TEXT</h1>
 9. <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.<br />
10. Sed lacinia, sem ac facilisis faucibus, mauris mi ultricies quam,<br />
11. non pretium justo dui non ligula. Donec aliquet accumsan pretium.<br />
12. Integer malesuada gravida purus fringilla imperdiet.</p>
13. <div id="" container1="">
14. <input onclick="buttonFunction();" value="Active Button"
15.  type="button" /></div>
16. <div id="container2">
17. <input onclick="otherButtonFunction();" disabled="disabled"
18.  value="not Active Button" type="button" /></div>
19. <a href="www.google.it">goto Google</a>
20. </body>
```

Figure 4.4-b: Html code of the interface in Figure 4.4-a

The Extractor gets from this page just three elements, namely the two buttons and the hyperlink, since they are able to trigger an event-driven client-side processing, and stores the attributes reported in Table 4.2. If the user had pressed the 'Active Button', CReRIA would extract about this event the information reported in Table 4.3.

Table 4.2: An Example of Interface Widgets Data Extracted by CReRIA

ID	1	2	3
Listener	Click	Click	Click
Handler	"buttonFunction();"	"otherButtonFunction();"	"www.google.it"
Type	Button	Button	Null
Active	True	False	true
AIP	html(1)/body(1)/div(1)/input(1)	html(1)/body(1)/div(2)/input(1)	html(1)/body(1)/a(1)
UP	html/body/div/input	html/body/div/input	html/body/a
UPid	html/body/div[@id='container1']/input	html/body/div[@id='container2']/input	html/body/a

Table 4.3: An Example of User-event Data

Id Event	User event	Id Widget	Id current interface	Id next interface
30	"click"	2	24	25

As to the event management, the Extractor package is not only responsible for capturing the event rising, but also for detecting the termination of the execution of the event handlers. To obtain this information, a technique based on two types of event handlers, named H_{raise} and $H_{termination}$ respectively, that act as probes and exploit the bubbling and capturing standard mechanisms defined by the W3C for DOM event dispatching [67] has been implemented. Moreover, a monitor component has been developed and embedded in CReRIA for tracing the data traffic between client and server side of the RIA and detecting the completion of synchronous and asynchronous server requests triggered by user events. This component and the $H_{termination}$ handler can be used to trace at run-time the termination of user event management.

4.3.2 The Abstractor Package

For the aim of reducing the information collected by dynamic analysis, the Abstractor Package implements some heuristic clustering criteria. At the moment, for the interfaces

four heuristic criteria named C1, C2, C3 and C4, respectively, have been implemented in CReRIA.

The former two criteria consider two interfaces as equivalent if they include the same set of clickable widgets, and some specific widget properties have the same values. In particular, according to C1, two interfaces are equivalent if they have the same set of active widgets, and the values of their Handler, Listener, Type, and AIP widget properties are exactly the same. C2 criterion considers two interfaces to be equivalent if they satisfy the same C1 criterion, and in addition the values of the widget Active attributes are exactly the same. In other words, C2 assumes as equivalent all the interfaces offering exactly the same elaboration to their users.

As an example, the interfaces I1 and I2 shown in Figure 4.5 will be considered equivalent by C1, but not equivalent by C2 since they have the same set of widgets, but one of them (cfr. 'Admins' button) is not clickable in the first interface, while it is clickable in the second one. It can be deduced that the event-driven elaboration provided by the interfaces is not exactly the same.

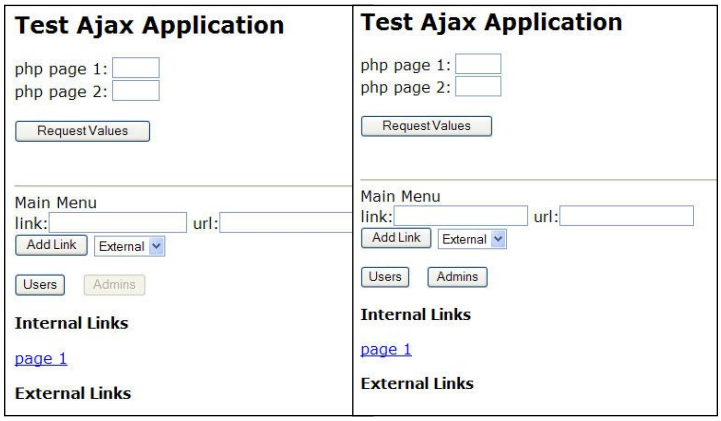

Figure 4.5: Two example interfaces (I1 and I2)

As to C3 and C4 equivalence criteria, they assume two interfaces to be equivalent if each of them includes a set (of any size) of clickable widgets having the same type of nesting position in the DOM tree (where the nesting position is defined either by the UP property or the UPid one).

As an example, Figure 4.6 reports two interfaces I3 and I4, which are considered as equivalent by C3 criterion (since I3 include a set of two hypertextual links, and I4 a set of three links with the same UP property value), but not equivalent by C4 criterion (since the links in I4 belong to different page divisions and have different UPid property values). As a further example, interfaces I2 (from Figure 4.5) and I3 (from Figure 4.6) aren't considered to be equivalent by C2 criterion (since I3 has an additional Hyperlink), while they are equivalent according to criteria C3 and C4.

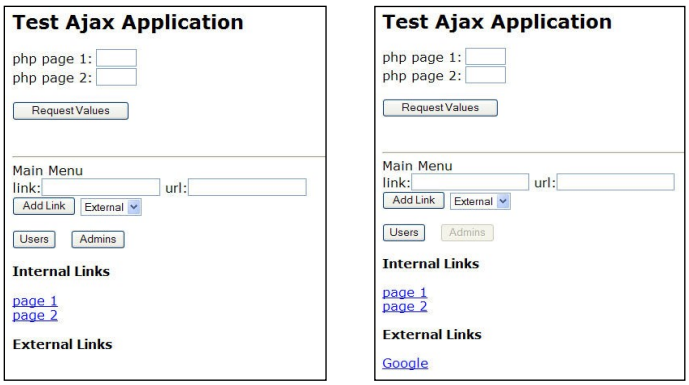

Figure 4.6 Two example interfaces (I3 and I4)

As to the reduction of data about user-events fired during user session registration, some heuristic equivalence criteria of events have been defined too. In CReRIA a traced session is defined by the sequence:

…<I_i, $event_i$>, <I_{i+1}, $event_{i+1}$>,..

where each user event is fired on a user interface widget, and I_{i+1} is the interface reached after the event management.

Hence, according to the first equivalence criterion T1, two events of the same type (such as click, mouseover, mouseout, etc.) and fired on two widgets with the same UPid will be considered as equivalent.

Moreover, according to the second criterion T2, two events of the same type and fired on two widgets with the same UPid will be considered as equivalent only if they are fired on two equivalent interfaces, and the interfaces reached after the event managements are equivalent too.

The third criterion T3 is similar to T2, but doesn't require the equivalence of the interfaces on which the events are fired.

4.4 Examples

The following two examples show the support to comprehension processes actually provided by CReRIA environment.

The RIA application involved in these examples is named Tudu, and is an open source application available at http://app.ess.ch/tudu, offering functionalities for the management of lists of tasks (the so-called 'todos') such as adding, deleting, searching for todos, organizing lists of todos, and so on.

Tudu is a meaningful example of a simple (but not trivial) RIA whose server side is implemented by php files, while its client side includes typical 'rich' pages that modify themselves at run-time on the basis of the user interaction with the pages.

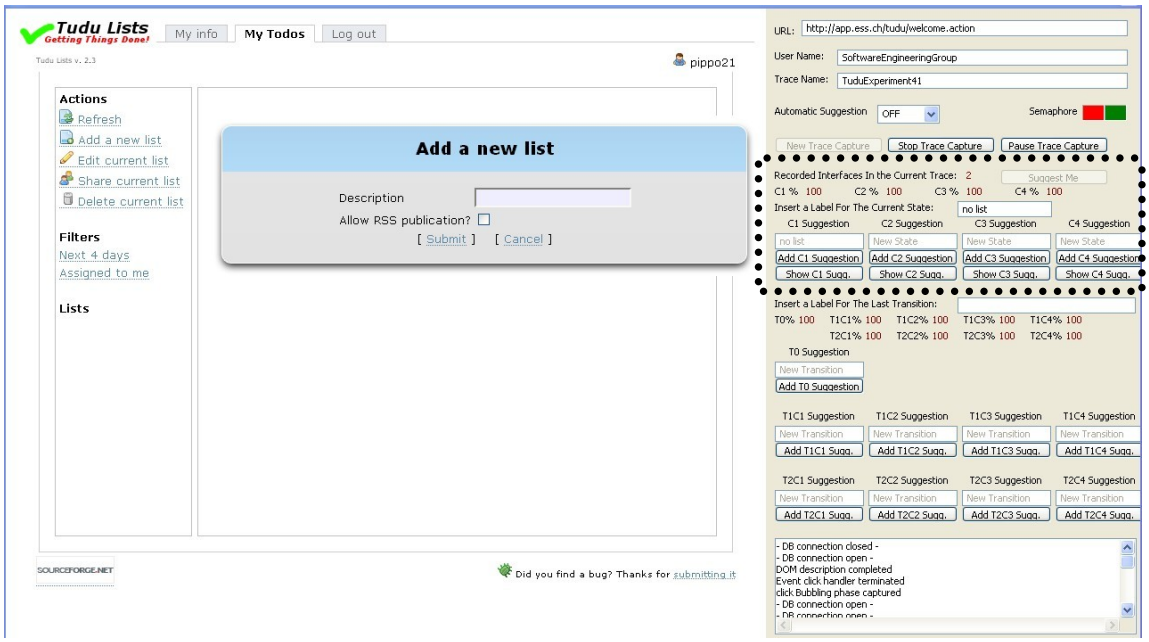

Figure 4.7-a: An Example of CReRIA GUI during the navigation of Tudu

Figure 4.7-a reports a CReRIA screenshot captured during a Tudu navigation session, the left side of the screenshot contains the CReRIA Browser window that renders the current

interface of Tudu, while the right side contains the Concept Assignment GUI.

A zoom on a part of the Concept Assignment GUI is also reported in Figure 4.7-b: this GUI contains the following items (from up-side to down-side): an output field reporting the number of recorded interfaces of the current trace (e.g., #2), the Accepted Suggestion Ratio (e.g., C1%, C2%, C3%, C4% output fields) of each interface clustering criterion[5], an input field "Insert a label for the Current State", the four suggested clusterings, and two lists of buttons for accepting a suggestion ('Add C Suggestion' buttons) or for showing a suggestion ('Show C Suggestion' buttons).

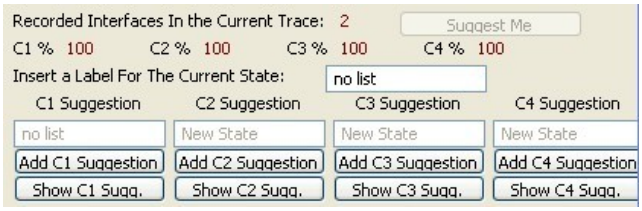

Figure 4.7-b: Zoom on the Concept Assignment GUI of CReRIA

For the current Tudu interface, the C1 criterion suggests that the interface belongs to the interface class already named 'no list', while C2, C3 and C4 suggest that it is a new type of interface, providing a new FSM state. If the user accepts one of the latter three suggestions, he will have to label the current new state using the input field "Insert a label for the Current State". Figure 4.8-a illustrates another CReRIA screenshot showing a new Tudu interface.

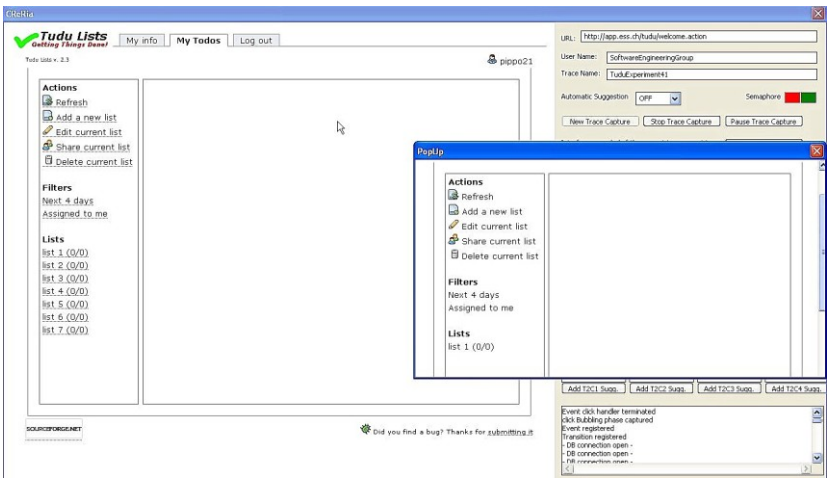

Figure 4.8-a: A second Example of CReRIA GUI during the navigation of Tudu

[5] The *Accepted Suggestion Ratio* metric is defined in Section 5.

Figure 4.8-b provides a zoom on the clustering suggestions for this interface (in this case two criteria, C3 and C4, have the 100% Accepted Suggestion Ratio). In order to aid the decision of the user about the current interface class, CReRIA shows a pop-up (see the right side of Figure 4.8-a) displaying another Tudu interface belonging to the same interface class suggested by the C4 criterion.

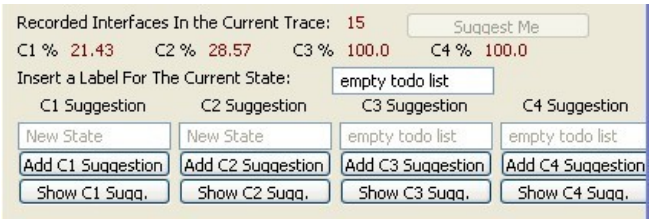

Figure 4.8-b: Zoom on the Concept Assignment GUI of CReRIA

The FSM resulting from the complete execution of the proposed reverse engineering process is shown in Figure 4.9.

This FSM has been generated by CReRIA as an input file for the graph viewer Dotty.

In that figure, nodes represent states of the FSM, while arcs represent transitions between states. Labels correspond to the ones proposed by means of the Concept Assignment GUI.

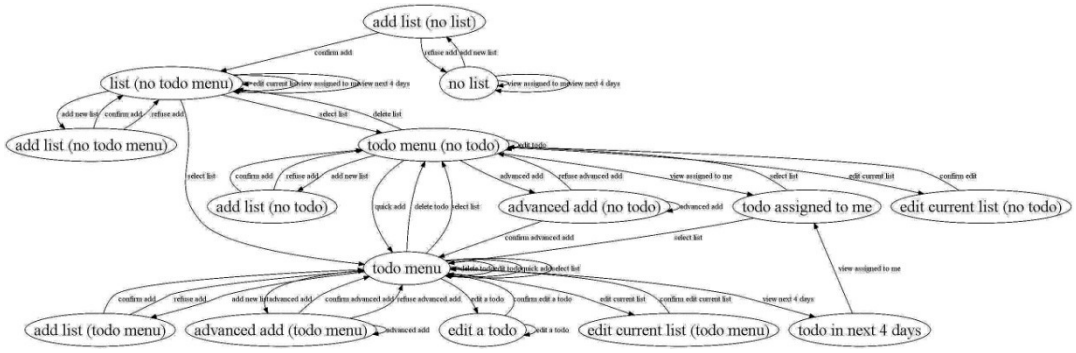

Figure 4.9: The reconstructed FSM for the RIA application Tudu

4.5 Case Studies

4.5.1 First Case Study

The iterative comprehension process of RIAs proposed in this chapter is based on the

assumption that it allows a more efficient process than traditional ones based on the simple navigation of the application.

The improved efficiency should depend on the clustering suggestions proposed iteratively by the CReRIA tool.

To show the validity of this assumption, we carried out a case study where the dynamic analysis of three distinct RIAs was carried out using the CReRIA environment, and the actual utility of the clustering suggestions provided by the tool heuristic criteria was assessed.

The following two metrics were used for this evaluation:

- Accepted State Suggestion Ratio (ASR-S) of criterion C, that is the ratio between the number of suggested state clusterings made by C and accepted by the user, and the number of suggested state clusterings made by C;
- Accepted Transition Suggestion Ratio (ASR-T) of criterion C that is the ratio between the number of suggested event clusterings made by C and accepted by the user, and the number of suggested event clusterings made by C.

The subject applications were three real AJAX applications with a rich user interface and that were available online.

Besides Tudu (W1 in the following), the second one W2 (TheList) is a demo application providing functionalities to manage a list of task descriptions.

The third one W3 (Buttonator) is a simple utility for web developers that offers functionalities for generating buttons with different shapes, size, and colors. The applications were analysed by three software engineers who faced the task of understanding the RIA client side behaviour and modelling it by a FSM.

Their previous knowledge about the applications was that of generic users of the applications, since no specific knowledge about RIA technologies is needed for using CReRIA.

During the experiment, CReRIA was configured to propose one more interface clustering suggestion, the so called CBest one, which represents the suggestion proposed by that

criterion (from the set of criteria from C1 to C4) whose suggestions were accepted by the engineer in most cases until the current iteration.

As an example, at a given process iteration, if the ASR-S of criteria C1, C2, C3, and C4 had the percentage values of 80%, 30%, 25%, and 90%, the CBest suggestion would coincide with the C4 one.

Analogously, a similar TBest event clustering suggestion was added too.

Table 4.4 reports a summary of the characteristics of these applications: the number of Use cases (UC) and Scenarios (SC) that were exercised during the navigation sessions, the number of different states and transitions composing the final FSM models that were reconstructed by the engineers, and the number of interfaces (NI) and fired events (NE) encountered during the reverse engineering process.

Table 4.4: Data about Subject Applications

	RIA	UC	SC	States	Transitions	NI	NE
W1	Tudu	8	17	15	52	610	609
W2	TheList	3	10	3	11	557	556
W3	Buttonator	1	8	19	72	825	824

Table 4.5 shows, for each application, the number of different states that were proposed by the clustering criteria C1, C2, C3, C4 and CBest respectively, the number of proposed state suggestions (Accepted Suggestion #) of each clustering criterion that were accepted by the engineer, and the consequent ASR-S values.

Table 4.5: Performance of State Clustering Criteria

		C1	C2	C3	C4	CBest
W1	Proposed FSM state #	142	204	10	14	14
	Accepted State Identification Suggestion #	78	97	519	605	605
	ASR-S	13%	16%	85%	99%	99%
W2	Proposed FSM state #	143	143	3	3	3
	Accepted State Identification Suggestion #	195	195	557	557	557
	ASR-S	35%	35%	100%	100%	100%
W3	Proposed FSM states #	3	19	7	19	19
	Accepted State Identification Suggestion #	91	825	825	371	825
	ASR-S	11%	100%	100%	45%	100%

Analogously, Table 4.6 shows, for each application, the number of different transitions that were proposed by the clustering criteria T1, T2, T3 and TBest, the number of proposed transition suggestions (Accepted Suggestion #) of each clustering criterion that were accepted by the engineer, and the consequent ASR-T values.

Table 4.6: Performance of State Clustering Criteria

		T1	T2	T3	TBest
W1	Proposed FSM transition #	28	34	47	47
	Accepted Transition Identification Suggestion #	304	426	518	518
	ASR-T	50%	70%	85%	85%
W2	Proposed FSM transition #	9	8	10	10
	Accepted Transition Identification Suggestion #	483	434	495	495
	ASR-T	87%	78%	89%	89%
W3	Proposed FSM transition #	12	45	68	68
	Accepted Transition Identification Suggestion #	107	461	700	700
	ASR-T	13%	56%	85%	85%

Data reported in the tables show that ASR-S and ASR-T values vary with the clustering criterion and with the application. However, in this experiment, the ASR-S and ASR-T values of CBest and TBest criteria were never lower than 85%, meaning that at least 85% of their suggestions were accepted by the engineer. This datum indicates the validity of suggestions proposed by the heuristic criteria, and the actual improved efficiency of the CReRIA supported navigation process with respect to a suggestion-less one.

These experimental data produced a further consideration: in the considered comprehension processes, there is not a single clustering criterion that is able to provide the most reliable suggestions for a given application. As a consequence, any dynamic analysis, which uses only a single clustering criterion, will be less effective than an *'adaptive'* analysis where several criteria are used at the same time, and the CBest and TBest criteria offer a prevision about the most reliable suggestion. This aspect shows the relevance of having a feedback from past iterations to current ones, which can be obtained only by adopting an incremental and iterative reverse engineering approach.

4.5.2 Second Case Study

The comprehension process supported by CReRIA is an iterative approach that at each iteration builds an intermediate version of the FSM model of the RIA GUI behaviour. In this type of approach, a relevant role is played by termination criteria that establish when the process iterations should stop. A first category of termination criteria is based on the evaluation of the coverage of some known RIA characteristics (such as its functionality, use cases, execution scenarios, event-driven processing, etc..) achieved by the considered navigation session. These criteria are, of course, not applicable in explorative navigation processes, when these RIA characteristics are not known before executing the process. A second category, vice-versa, is applicable in explorative navigation processes, since they consider some properties of the performed process iterations (such as number of performed iterations, time spent, size of the reverse engineered model, etc.) as a feedback for process termination. In the second case study, we decided to investigate the effectiveness of two types of process properties as possible indicators of explorative process termination: (1) the number of states VS(i) of the FSM model produced at the end of the i-th process iteration, and (2) the number of transitions VT(i) of the FSM model produced at the end of the i-th process iteration. To this aim, we considered the same comprehension processes carried out in the first case study, and we analysed the values of VS(i) and VT(i) as the iterations proceeded. The plots of these values are reported in Figure 4.10 and Figure 4.11 respectively, for the first analysed application Tudu. A similar trend was observed for the other applications.

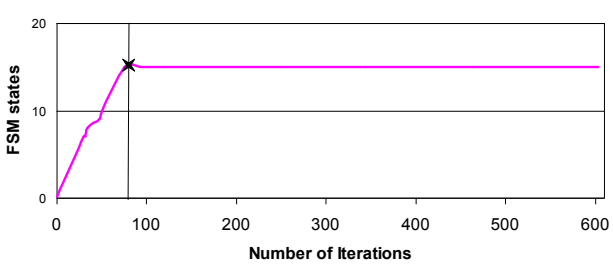

Figure 4.10: Plot of the VS number of FSM states discovered during the execution of the process on W1 (Tudu)

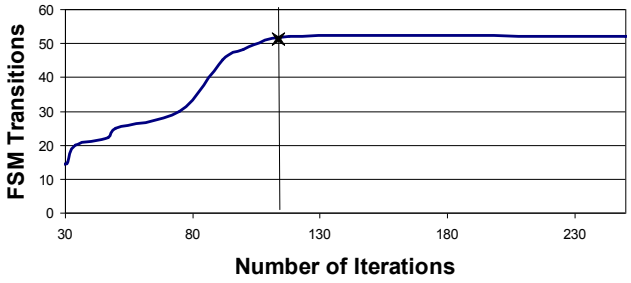

Figure 4.11: Plot of the VT number of FSM transitions discovered during the execution of the process on W1 (Tudu)

As these figures show, the FSM model version obtained at the 75th iteration (and further ones) has the same number of states of the model obtained at the last process iteration (that is the 610th). Analogously, the FSM model version obtained at the 120th iteration has the same number of transitions of the model obtained at the last process iteration (that is the 610th). These data indicate that, in this case, the process could be terminated after only 120 iterations. Hence, a possible termination criterion is the one that stops the process when a 'sufficient' number of iteration did not reveal any new state or new transition. As an example, a possible termination criterion could be the one that stops the process when the percentage of iterations that did not reveal any new state or transition (with respect to the number of performed iterations) is greater than a given threshold (such as 30%).

Of course, this technique for stopping the navigation of the application is not applicable to traditional analysis approaches that are not iterative, and do not produce incrementally any intermediate version of the FSM at each process iteration.

In conclusion, these case studies showed that the adoption of an iterative and incremental comprehension approach is beneficial for the overall reverse engineering process because:

The feedback provided by CBest and TBest criteria during the iterative process is able to improve the efficiency of the concept assignment and FSM reconstruction tasks;

The availability of a partial version of the FSM model of the analysed application at each process iteration supports the evaluation of effective termination criteria for stopping the RIA navigation during explorative comprehension processes.

4.6 Conclusions

In the last years, the principles and practices of the agile development [75, 76] are becoming more and more diffused, and their adoption is being experimented in different types of software contexts [77, 78].

Proposing and validating agile processes and methods also for software maintenance, evolution, and comprehension aims is an interesting new research area, and the first proposals of using agile approaches in the fields of Reverse Engineering and Reengineering are emerging in the literature [79, 80].

The work presented in this chapter provides an attempt to define an 'agile' process for the reverse engineering of Rich Internet Applications that is iterative, based on the continuous feedback of the process executor, and allowing the incremental reconstruction of a Finite State Machine modelling the behaviour of RIA GUIs. This approach is supported by CReRIA, an integrated reverse engineering environment that provides automatic facilities for executing the process and incrementally recovering and validating a FSM model.

The approach has been experimented with some case studies, which involved the dynamic analysis of existing RIAs. Results of the experiment showed the approach feasibility and how it simplifies the concept assignment tasks needed for software comprehension. The experiment highlighted further aspects to be investigated in future work, such as the definition and validation of additional iteration termination criteria. Moreover, future work will address the definition of further clustering criteria for reducing the data collected at run-time, and will investigate to which extent distinct clustering criteria are effective in supporting different reverse engineering and comprehension tasks.

In next chapters we propose techniques of testing automation, comprehension and re-documentation of RIA developed in Ajax that exploit the proposed models and the dynamic analysis techniques.

Chapter 5 [6]

RICH INTERNET APPLICATION TESTING USING EXECUTION TRACE DATA

In this chapter we present a technique for testing RIAs that generates test cases from application execution traces, and obtains more scalable test suites thanks to testing reduction techniques. Execution traces provide a fast and cheap way for generating test cases and can be obtained either from user sessions, or by crawling the application or by combining both approaches. The proposed technique has been evaluated by a preliminary experiment that investigated the effectiveness of different approaches for execution trace collection and of several criteria for reducing the test suites. The experimental results showed the feasibility of the technique and that its effectiveness can be improved by hybrid approaches that combine both manually and automatically obtained execution traces of the application.

5.1 Introduction and Related Works

As we introduced in section 1.2 Rich Internet Applications and are usually developed using Web 2.0 technologies, such as Ajax [17], Silverlight [122], or Flex [124].

Ajax [17], in particular, is a set of technologies (JavaScript, XML, XHR objects) that can be used for implementing RIAs. The user interface of an Ajax-based RIA is made up of Web pages (or, at least, of a single page), whose status changes thanks to run-time client-side elaborations performed on the page by JavaScript event handlers triggered by user

[6] This chapter was published in the Proceedings of the 3th International Conference Software Testing, Verification, and Validation Workshops (ICSTW 2010).

events or other external events (such as timeout events or asynchronous responses by the server). Event handlers access and manipulate the Web page using the Document Object Model (DOM) [62] interface. These handlers, besides synchronous requests, can also send asynchronous requests (for data or elaborations) to the server which introduce parallelism between the client and the server, leaving the user interface active. Due to the event-driven elaborations, the interface of Ajax-based RIAs may be considered like an event-driven software system (EDS) or similar to the GUI of a desktop application. Moreover in RIAs the dynamic elaboration of the user interface may be not deterministic but usually depends on the current state of the application. With these new characteristics, a set of new challenges have come such as the one of finding effective techniques, models and approaches for testing RIAs. Indeed a RIA may potentially exhibit specific types of failures due to incorrect manipulations of the DOM, unintended interleaving of server messages, swapped callbacks, etc. [72]. As a consequence, models, techniques, and strategies already proposed for traditional Web application testing [21, 63, 81, 82], may not be suitable for them. Indeed, the traditional web testing approaches are based on the assumption that the interaction between the user and the Web pages is limited to clicks on navigational links and to the insertion of data in forms, and that the business logic of the application is entirely implemented on the server side of the application.

Some research contributions to the RIA testing topic have recently been proposed in the literature [56, 57, 72, 83]. These papers essentially present different model-based testing techniques applied to Ajax applications, which require that a model of the application is preliminary obtained (either by semi-automatic approaches, or automatically by crawling techniques) and different approaches are used for generating test cases that cover these models. Two crucial points of these techniques are the expensiveness of the processes needed for reconstructing the Ajax application's model and the often unmanageable size of generated test suites. Another promising approach that should be investigated for RIA testing is the user-session based one that has been already applied with success both for traditional Web application testing [81, 84], and for GUI automated testing [85]. This

approach aims at automatically generating test cases composed of event sequences which are deduced by analysing user interactions with a version of the application. The obtained test cases can be either replayed for testing the same application, or for regression testing (provided that the user interfaces of modified versions are still the same of the original one), or for generating a model of the application which can be used for deriving test cases automatically. RIA user session data have also been exploited by the reverse engineering technique, presented in chapter 2 and validated in chapter 3 [1, 2], that aims at obtaining a state-based model of the user interface of RIA applications.

Examples of user-session based techniques are the ones proposed by Elbaum et al. [81, 86], who investigated the fault-detection capability and the cost-effectiveness of user-session based techniques, and by Sampath et al. [84] who explored the possibility of using concept analysis for achieving scalability in user-session based testing of Web applications.

As to the automated testing of Web applications, most testing proposals are based on the usage of capture and replay tools which can be used to record user interactions with a version of the Web application and to replay them. However, since the interactions must be recorded manually and obtained test cases are mostly usable for exercising just the considered version, the applicability of these techniques is actually limited.

Several automated techniques have been proposed for GUI test case generation. In particular, [44] remarked the necessity of automated GUI rippers to obtain automatically a model of the GUI behaviour that can be used to design test cases. In [87] a GUI smoke regression testing process called DART that automates GUI smoke testing is presented: the process is based on a GUI ripper that automatically reconstructs a model of the GUI (the so called Event-Flow Graph) and on a test case generator component that generates smoke test cases made up of sequences of interacting events that may be executed on the GUI. Yuan et al. [88] present a new automated model-based testing technique that uses the feedback from the execution of an initial and automatically obtained test suite to obtain new and improved test cases for a GUI, while [89] presents two studies showing the

improved effectiveness of the feedback-based technique with respect to other techniques.

Some research contributions to the RIA testing topic have recently been proposed in the literature: Marchetto et al. [56] investigated the feasibility of a state-based testing technique based on semantically interacting events, and in [72] the same authors propose an improvement of this technique that exploits a search-based approach (based on the hill-climbing algorithm) for obtaining longer interaction sequences having higher fault exposing capability and keeping the test suite size reasonable. Mesbah et al. [57, 83] have explored the Ajax automatic testing field, and have proposed an approach that uses an automatic crawler to infer a flow-graph of client-side interface states, and generates a test suite covering the paths obtained during crawling for identifying specific types of fault that can occur in those states. Finally, the Selenium testing framework [90] has added some specific constructs (such as waitForResponse) which can be used for replaying correctly the interactions with Ajax applications that exploit asynchronous messaging between client and server.

At the moment, no user-session based technique has been yet investigated in the literature for Rich Internet Applications testing.

In this chapter we present a preliminary investigations about using execution traces of an RIA for the aims of testing. In this context, we have defined a testing technique that exploits concrete execution traces of an application, either produced by real users (or tester users) or automatically by a Web crawler, to transform traces into executable test cases. For achieving the technique scalability, a test suite selection technique is employed for reducing the size of obtained test suites.

For exploring the feasibility and effectiveness of this technique, we developed an integrated set of tools for implementing it and carried out an experiment. In the experiment, different approaches for execution trace collection and several criteria for reducing the test suites were analysed and the characteristics of resulting test suites were evaluated and compared. The preliminary results showed the fault detection capability of obtained test suites and that the scalability of the technique can be improved by means of

suitable reduction techniques, that do not impact its effectiveness. Moreover the experiment revealed that the effectiveness of the technique can be improved by hybrid approaches that combine both manually and automatically obtained execution traces of the application.

5.2 Generating Execution Trace Based Test Cases for RIAs

Finite State Machines (FSM) are one of the most popular models used for representing the behaviour of a software system and testing it. They provide an abstract view of a system in terms of states and transitions among them and have also been used with success for modelling and testing object oriented systems [101], traditional Web applications [63] and GUIs [51]. These models are usually obtained manually or by semi-automatic techniques, and need state abstraction functions for managing the state-explosion problem.

Moreover, in state-based testing processes, the FSM model can be used for generating test cases given by paths (sequences of events) on the FSM that assure a requested coverage of the FSM model. A limitation of such an approach is that not all the possible FSM paths can be translated into executable test cases, and specific analysis techniques need to be used to select only paths that are actually executable on the application.

In this chapter we propose a testing technique where real execution traces of an RIA (both manually, and artificially obtained) are transformed into executable test cases, and a FSM of the application is not used for deriving test cases, rather it provides one possible reference model for reducing the test suite into a smaller one. More precisely, the proposed technique is implemented by the following steps:

- Collection of a set of execution traces of the application;
- Test suite generation;
- Test suite reduction.

These steps are described in the following sub sections.

5.2.1 Execution Traces Collection

The goal of this activity consists of obtaining a set of execution traces of the application which are representative of the behaviours to be tested. These execution traces may be

either recorded from user sessions of real users/testers of the application like it was suggested in chapters 2 and 3, or they can be obtained from an automatic exploration of the RIA user interfaces, such as the one produced by a crawler of the application.

A crawler is a tool that can be used for reconstructing a model of a Web application by exercising its client-side code: as an example, an Ajax crawler triggers all events that are accessible through the Web page widgets (buttons, forms, anchors, …) either by breadth-first or depth-first visiting strategy, so that the corresponding JavaScript functions are invoked. Finally it registers the new user interface states (DOM states) reached after the JavaScript execution. In this way the crawler can reconstruct a state flow-graph whose nodes capture the states of the user interface and edges represent possible transitions between them. However, since the same states can be regenerated during the crawling process, some techniques for recognising similar states must be used. As an example, in [91] a technique using hash values of the state content is used for discovering similar states, while in [58] the same problem is solved by using the edit distance between DOM trees. We have implemented an Ajax crawling technique in a tool called CrawlRIA (further details about this tool are provided in section 5.3). The tool can be also used for generating a set of execution traces of the application. The crawler starts from the initial page of the application and triggers its events in a depth-first manner; each time a new DOM state is reached, its similarity with already visited states is evaluated using one of the interface equivalence criteria proposed in chapter 3. If the interface state has not been already visited, the crawler continues its navigation process otherwise it stops the exploration, saves the sequence of DOM states and events as an execution trace, and restarts the crawling from the initial page of the application. Each resulting execution trace is defined as a sequence of couples:

…, <Interf. $State_i$, $event_i$>, <Interf. $State_{i+1}$, $event_{i+1}$>, … (1)

An open issue with RIA crawling techniques is the management of interfaces including forms where the user has to insert specific input values: an automatic crawler can solve this problem either using a pre-defined set of input values to populate the forms, or

excluding the forms from its analysis. Both solutions will yield to resulting execution traces that may not be representative of real interactions of human users with the application. Moreover, the termination criteria used by the crawler to stop the exploration of DOM states are likely to produce execution traces associated with short/ simplified paths of interactions with the application, far from the semantically-rich interactions of a real user.

As possible solutions to these problems and to obtain more representative and meaningful execution traces, we suggest integrating the crawled execution traces with traces from user sessions. To obtain such traces, a tool like CReRia, presented in chapter 4, can be used. The tool besides collecting traces is able to classify the generated DOM states using the already cited interface equivalence criteria, and to generate a trace with the same format defined in (1).

5.2.2 Test Suite Generation

Our testing approach generates test suites by transforming each available execution trace into a test case. This transformation is not straightforward but requires that some questions are solved, such as the definition of the pre-conditions of each test case and the definition of the expected output of a test case.

To solve the former problem, since in general the behaviour of an RIA will depend on the current state of the application data as well as by its environment and session data, it is necessary to get the RIA state before recording each execution trace, or to set it to a known reference value. This state will provide the preconditions of each test case and during the test case execution specific set-up and tear-down methods will have to be executed to manage it.

The second problem requires a testing oracle to define the PASS/FAIL result of a test case execution. Some authors solved this problem by checking specific types of failures of an RIA, such as state-invariant violations [83], or asynchronous message passing anomalies (such as unintended interleaving of server messages, swapped callbacks, …) [56]. Another proposed solution is that of checking the consistency of the concrete state sequence with

respect to the expected state sequence on an FSM model of the application [56]. In this chapter, we evaluate test case results by checking the occurrence of JavaScript crashes, and use a dynamic analyser of Ajax application executions (DynaRIA) to detect their occurrence automatically.

5.2.3 Test Suite Reduction

For obtaining a test suite with a manageable size, the typical testing problem of test suite reduction has to be addressed. Given an initial test suite, this problem can be solved by test case selection techniques that produce a test suite smaller than the original one yet still satisfying the original suite's requirements [92, 93, 94].

Several properties and models of the analysed software can be considered for test suite reduction. As an example, if a FSM model of the application is available, a starting test suite TS can be reduced into a smaller one including either its (1) test cases that cover the same set of FSM states covered by the original suite, or (2) the ones covering the same set of FSM transitions (or events) covered by TS. Analogously, if just the source code of the application is available, reduced test suites can be obtained by selecting the test cases covering the same set of code components (such as functions, modules, LOC, etc…) as the original test suite.

The selection techniques can be implemented using the generic reduction algorithm proposed in [95] that applies to any binary coverage matrix M where each row corresponds to a test case of TS and each column corresponds to a generic item $x \in X$. The elements of the matrix M are defined as follows:

m(i, j) =1 iff the test case i covers the item j

m(i, j) =0 iff the test case i does not cover the item j

The reduction algorithm uses the following essentiality and dominance criteria for generating the reduced set of test cases.

- *Essentiality criterion*: a test case tc is essential if it is the only test case from TS that covers a given item x;
- *Row dominance criterion*: a test case tc_i is dominated by a test case tc_j if all the items

covered by tc_i are covered by tc_j too;

- *Column dominance criterion*: an item x_i dominates another item x_j if x_i is covered by all test cases that cover x_j.

The first criterion identifies test cases associated with essential rows and they will have to be included in the final reduced test suite. The row dominance criterion identifies test cases associated with dominated rows and these test cases will be excluded from the final test suite.

The algorithm iteratively analyses the matrix M and progressively reduces it by discarding its rows or columns using the following rules: ***1) the essential rows are discarded from the matrix, the corresponding test cases will be added to the reduced test suite, and the set of columns associated with the objects covered by those essential rows will be discarded too; 2) dominated rows and dominant columns are discarded from M. The algorithm ends when the matrix becomes empty, providing the reduced subset of test cases.***

5.3 Experiment

In this section describes an experiment that was performed for evaluating the proposed testing approach and the supporting tools we developed.

5.3.1 Research questions

The proposed testing approach allows different types of execution traces to be used for test suite generation and different characteristics of an RIA to be considered for test suite reduction.

We investigated nine testing techniques that were obtained by combining three execution trace collection approaches (e.g., by user-sessions, by crawling, and both by user-sessions and crawling) and three reduction techniques that considered different types of RIA characteristics, namely **M1** (considering FSM states), **M2** (FSM transitions), and **M3** (JavaScript functions). The reduction techniques M1 and M2 assume that each subject application has been associated with a FSM model that was produced automatically from

the available execution traces using the abstraction technique presented in chapter 3. For the reduction aim of M3, the considered JavaScript functions include just the distinct static script functions called during the trace execution (also including library functions). We also considered three additional techniques (**B1**, **B2**, and **B3**) where no test suite reduction was performed. Table 5.1 reports the 12 techniques involved in the experiment.

Table 5.1: The testing techniques considered in the experiment

Technique	Execution trace collection	Reduction Technique
B1	By user sessions	-
B2	By crawling	-
B3	By user sessions and crawling	-
T1	By user sessions	M1
T2	By user sessions	M2
T3	By user sessions	M3
T4	By crawling	M1
T5	By crawling	M2
T6	By crawling	M3
T7	By user sessions and crawling	M1
T8	By user sessions and crawling	M2
T9	By user sessions and crawling	M3

The experiment was designed to address the following research questions:

RQ1. *How effective are the testing techniques B1, B2, and B3?* This question concerns the performance of the B1, B2, and B3 techniques in terms of the coverage and fault-detection they provide.

RQ2. *How effective are the reduction-based T1… T9 techniques with respect to the B1, B2, and B3 techniques?* This question concerns the relationship about the performance of the B1, B2, and B3 techniques with reference to the T1… T9 techniques, in terms of the coverage and fault-detection they provide.

5.3.2 Measured Variables

In the experiment, we measured the following variables:

- *FSM State Coverage (ts):* percentage of FSM states covered by at least one test case of the test suite ts.

- *FSM Transition Coverage (ts):* percentage of FSM transitions covered by at least one test case of the test suite ts.
- *JavaScript function Coverage (ts):* percentage of JavaScript functions executed during the ts execution w.r.t. the number of script functions contained by the JavaScript modules of the application.
- *JavaScript LOC Coverage (ts):* percentage of JavaScript function LOC executed during the ts execution w.r.t. the LOC of JavaScript functions of the application.
- *Fault detection effectiveness (ts):* percentage of faults detected by ts (section 5.4.1 provides further details on the faults used in the experiment).
- *Test Suite Size:* number of test suite test cases.
- *Test Suite Event Size:* number of events exercised by the test suite test cases.

5.3.3 Experimental process and supporting tools

The experimental process has been carried out with the support of a set of tools developed by the authors. The set of tools comprehends CreRIA, CrawlRIA, Test Case Generator, Test Case Reducer and DynaRIA which are briefly described in the following.

CReRIA is the tool, presented in chapter 4, for dynamic analysis of RIAs designed for supporting comprehension processes. The functionalities implemented by CReRIA include:

- offering a Web browser for navigating the rich internet application and performing user sessions tracing;
- extracting and recording relevant information about traced user sessions, such as user interfaces and events that occurred during the navigation;
- clustering of interfaces and events according to different abstraction criteria;
- abstraction of the FSM.

The user session traces (sequences of interfaces and events) and the corresponding paths on the abstracted FSM (sequences of states and transitions) are stored in the FSM & Trace Repository implemented by a MySQL database. Further details about CreRIA are described in chapter 4.

CrawlRIA is a prototype crawler that automatically generates execution traces of an RIA by triggering the events found in RIA interfaces. CrawlRIA is also able to cluster interfaces and events with the same technique implemented by the CReRIA tool, to generate an FSM model, and to store them in the FSM&Trace Repository.

Test Case Generator is a tool able to transform the execution traces stored in the FSM & Trace Repository in a test suite composed of executable test cases. The current Test Case Generator prototype generates test cases both in a format executable by the Selenium suite [90] and by the DynaRIA tool.

Test Case Reducer is a tool able to reduce a test suite ts into a smaller one that satisfies the same ts coverage requirements. The output of the tool is a reduced test suite ts' in the same format of the input test suite ts. The Test Case Generator tool and the Test Case Reducer are integrated in a general tool called TestRIA.

DynaRIA is a tool for dynamic analysis and testing of RIAs. It is able to execute the test cases produced by the Test Case Generator or the Test Case Reducer tool, to monitor their execution in a browser environment, and to produce a report of detected crashes. It also evaluates and reports the coverage measures described in sub-section 5.3.2.

The experimental process we carried out is shown in Figure 5.1. Chosen the subject RIA, execution traces were both manually collected (using the CreRIA tool) and automatically by the CrawlRIA tool. Produced traces were stored in FSM&Trace Repository realised as a MySQL database. The Test Case generator tool produced test cases from the collected execution traces. The Test Case Reducer applied the minimization techniques and produced reduced test suites. The produced test suites were submitted to the DynaRIA tool for the execution. The DynaRIA tool

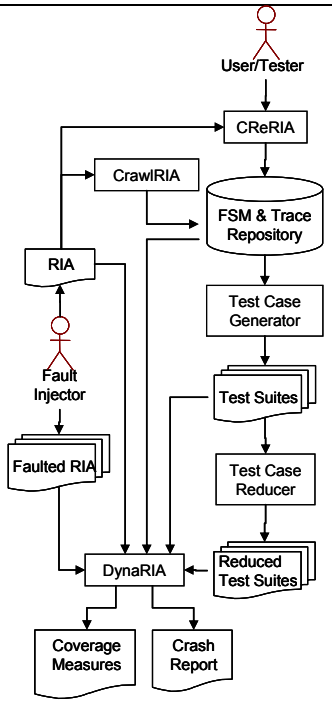

Figure 5.1: The experimental testing process

evaluated the results of all test case executions both on the original version of the RIA, and on a set of RIA versions in which an expert injected some faults. A description of the injected faults is reported in the sub section 5.4.1.

5.4 Subject Application

Our experiment involved Tudu, an open source application available from http://tudu.sourceforge.net offering 'todo' list management facilities (such as adding, deleting, searching for todos, organizing lists of todos, and so on). This application is a well-known example of open source RIA [56, 72] consisting of about 10,000 LOCs of Java, JSP and JavaScript, and it uses JavaScript frameworks. The persistent data related to the users and their todos are stored in a MySQL database.

5.4.1 Fault Seeding

As we wanted to evaluate the fault detection capability of proposed testing techniques, we injected faults of different types in the JavaScript (JS) code of the subject application. We focused on faults that are able to produce JS crashes. Since the JS code of an Ajax application is interpreted at run-time by a browser component, a JS crash is managed by interrupting the current execution (event handling). The notification of the crash may be shown or not on the user interface. For this reason, depending on the fault's position in the JS code, JS crashes may emerge or not on the GUI. Hence, we had to analyse the flow of the program in order to inject both faults whose effects are visible on the client interface DOM, and faults that do not produce visible effects on the DOM. Both types of fault are automatically detectable by DynaRIA thanks to the functionality of JavaScript execution tracing.

The faults were representative of typical JS programming errors, such as: JS function call instructions with undefined, incorrect, or missing parameters, JS syntax errors, array out of bound errors, server requests of missing resources or JS files, etc. 19 faults were injected and 19 versions of Tudu were produced, each one containing just one fault.

5.4.2 Data Collection

In the experiment, the Tudu application was crawled by the CrawlRIA tool that collected 1,684 interfaces belonging to 203 execution traces by triggering 1,481 events on the interfaces. Moreover, one of the authors who knew the main functionalities offered by the application recorded 21 user sessions by the CReRIA tool, triggering 518 events and navigating 539 interfaces. These user sessions were able to exercise all the application's known use cases and their scenarios. All the traces were collected by starting from the same initial conditions of the application state. In particular, the session data were always reset to zero and the database was reset to a reference dump.

All the collected traces were used for automatically abstracting a reference FSM model of the application. Table 5.2 reports the overall number of collected interfaces and triggered events, the number of states and transitions of the resulting FSM, the total number of distinct JavaScript function definitions found in the code of collected interfaces and their size in LOC.

Table 5.2: Overview information about collected execution traces

Collected Interfaces	2223
Triggered Events	1999
FSM States	19
FSM Transitions	61
Defined JS Function #	1018
Defined JS Function LOC	6150

The execution traces were used to obtain three initial test suites: **US** (generated from user session traces), **CR** (generated from CrawlRIA traces), and **HY** (obtained by merging the test suites US and CR).

Each test suite was then reduced using the three minimization techniques M1, M2 and M3, and 9 reduced test suites were obtained called US-M1, US-M2, US-M3, CR-M1, CR-M2, CR-M3, HY-M1, HY-M2, and HY-M3, respectively.

The test suites were automatically executed on all the faulty versions of Tudu by the DynaRIA tool that also evaluated the coverage measures and the number of detected faults of each test suite. The following tables report collected data for the test suites obtained

from user sessions (Table 5.3), from crawler traces (Table 5.4) and both from user sessions and crawled traces (Table 5.5), respectively.

Table 5.3: Data about user session test suites

	US	US-M1	US-M2	US-M3
Test Case #	21	3	9	10
Event #	518	81	232	235
Covered States	19	19	19	19
Covered States %	100%	100%	100%	100%
Covered Transitions	56	40	56	56
Covered Trans. %	91,8%	65,6%	91,8%	91,8%
Covered Functions	172	160	163	172
Covered Funct. %	16,9%	15,7%	16,0%	16,9%
Covered LOC	1016	967	980	992
Covered LOC %	16,5%	15,7%	15,9%	16,1%
Revealed faults #	19/19	16/19	19/19	19/19

Table 5.4: Data about test suites from crawled traces

	CR	CR-M1	CR-M2	CR-M3
Test Case #	203	5	20	23
Event #	1481	42	134	273
Covered States	14	14	14	14
Covered States %	73,7%	73,7%	73,7%	73,7%
Covered Transitions	35	16	35	35
Covered Trans. %	57,4%	26,2%	57,4%	57,4%
Covered Functions	160	141	153	160
Covered Funct. %	15,7%	13,9%	15,0%	15,7%
Covered LOC	949	875	929	937
Covered LOC %	15,4%	14,2%	15,1%	15,2%
Revealed faults #	17/19	9/19	17/19	17/19

Table 5.5: Data about test suites obtained from user sessions and crawled traces

	HY	HY-M1	HY-M2	HY-M3
Test Case #	224	3	21	24
Event #	1999	81	261	283
Covered States	19	19	19	19
Covered States%	100%	100%	100%	100%
Covered Trans.	61	40	61	61
Covered Trans.%	100%	65,6%	100%	100%
Covered Funct.	192	160	164	192
Covered Funct.%	18,9%	15,7%	16,1%	18,9%
Covered LOC	1042	967	987	1022
Covered LOC%	16,9%	15,7%	16,0%	16,6%
Revealed faults #	19/19	16/19	19/19	19/19

5.5 Discussion

To answer the research question RQ1 on coverage and fault detection effectiveness of the considered B1, B2 and B3 techniques, we analysed the results achieved by the initial test suites US, CR, and HY, which are reported in the second columns of Tables 5.3, 5.4 and 5.5, respectively.

When we analyse the coverage of JavaScript functions and of FSM states and transitions,

we can observe that US covers 172/1018 (16.9 %) functions, CR covers 160/1018 (15.7%) functions, HY covers 192/1018 functions (18.9%). However, US exclusively exercises 32 functions of the overall 192, while CR exclusively exercises only 20 functions of 192.

This small coverage of JS functions of both US and CR can be explained because the Tudu application largely includes library functions from frameworks, but just a little part of these library functions are actually used by it. As to the LOC coverage of functions, analogous coverage data were obtained.

As to the FSM coverage, US and HY cover all FSM states (19), while CR only covers 14/19 (73.7%) states. Moreover, US covers 56/61 transitions (91.8%), CR covers 35/61 (57.4%) and HY covers all transitions.

As to the fault detection effectiveness, US and HY discover all known faults (19/19), while CR only 17/19 faults. For understanding why CR did not reveal these faults, we analysed them, and discovered that they were not exercised by the crawler execution traces because the sequence of events that caused the crash was not triggered by the crawler. Vice-versa, this sequence of events was executed by the user sessions, since it belonged to a well-known functionality of Tudu.

As a result, the test suite US (obtained from user session traces) reached a wider coverage of the FSM model, JS code, and faults than CR (that is automatically obtained by the crawler), and its size in number of test cases (21) and covered events (518) was smaller than the size of CR (203 test cases and 1481 events).

On the other hand, even if CR included more test cases than US, their average length (in number of executed events) was smaller (about 7 events against 25) and they discovered about 90% faults. Moreover, deriving CR is less expensive than generating US since CR can be derived automatically, while US always requires human intervention for trace collection As to the test suite HY, it was larger than US and CR, but it covered quite the same JS functions of US and of CR, and had the same fault detection capability of US. Moreover, the costs of generating HY is approximately the same of generating US, being CR automatically obtained.

In conclusion, in this experiment the B3 testing technique proved to be more effective than the B1 and B2 ones, but B1 and B3 effectiveness are quite similar. However, we can observe that the technique B2 always provides an automatic and fast solution to the problem of generating test suites that detect about 90% of known defects.

To answer the research question RQ2 about the relationship between techniques without test suite reduction (B1, B2, and B3) and the techniques with reduction (T1, …, T9), we first compared the size of the US test suite against the size of its reduced test suites. Similar comparisons were made for CR and HY and their respective reduced test suites.

As to the US test suite, the M1 technique significantly reduced the US size (3 test cases against 21, covering 81 events against 518), but US-M1 lost the coverage of 16 transitions of the FSM, as well as of 12 JS functions, and revealed only 16 faults (rather than 19). Vice-versa, the M2 and M3 reduction techniques allowed a smaller reduction of the test suite size (232/518 and 235/518 covered events, respectively), but preserved the FSM transition coverage and fault detection capability.

As to the automatically obtained test suite CR, we recognized that the size reductions provided by M1, M2 and M3 techniques were actually relevant (CR-M1, CR-M2, and CR-M3 covered 42, 134, and 273 events with reference to the 1481 events covered by CR). On the other hand, while the fault detection capability of reduced test suites CR-M2 and CR-M3 remained the same (17/19 faults) of CR, the capability of CR-M1 of revealing faults decreased (only 9 faults/ 17 detected by CR).

As to the HY test suite, the size reductions of the M1, M2, M3 techniques were relevant and similar to the ones observed for the US test suite. While the fault detection capability of the reduced suite HY-M1 get worse (e.g., 16/19 with respect to 19/19 of the HY), for the HY-M2 and HY-M3 it remained the same of HY.

In conclusion, these results showed that all analysed test suite reduction techniques significantly reduced the size of test suites automatically obtained by crawling, and the fault detection capability of test suites reduced by M2 and M3 did not get worse. Hence, the testing techniques with reduction (T1, …, T9) were comparable to the techniques

without reduction (B1, B2 and B3) as to the coverage and the fault detection effectiveness.

5.6 Conclusions

In this chapter we have proposed a testing technique for RIAs that transforms execution traces of an existing application into executable test cases. For achieving the technique scalability, a test suite selection technique is employed that reduces the size of obtained test suites. For exploring the feasibility and effectiveness of this technique, we carried out an experiment involving an open-source RIA application, where different approaches (both human-based, and automatic) for execution trace collection and several criteria for reducing the test suites were analysed.

The experimental results showed that test suites produced automatically by means of a crawler of the RIA user interface are not more effective than suites derived from execution traces, but the former ones have the advantage of being automatically obtained and of revealing a good percentage of RIA faults. As a consequence, we believe that a more effective testing strategy should combine test cases obtained by both approaches: first, test cases automatically obtained by an RIA crawler and by reduction techniques should be used for discovering the most of application defects. Since these test cases are usually made up by shorter sequences of events than the ones generated by user session traces, they will also have the advantage of being executed and debugged faster. Then, if user session data will be available, test cases based on these data could be employed to obtain a wider coverage of defects. Of course the validity of obtained experimental results is reduced, due to several limitations of the experiment we performed, such as the single RIA application involved, the small number of collected user sessions, the single user involved in the collection, and the single initial state of the application that was considered during trace collection. Moreover, the faults that were injected in the application were just of a particular type (i.e. faults causing JS crashes), while faults affecting the RIA behaviour without causing crashes were not considered. Finally, the technique adopted for abstracting the FSM model of the RIA may provide just an approximate model of the RIA behaviour. To overcome these limitations, further investigations and a wider

experimentation will be carried out in future work. In this chapter we've proposed a testing technique in order to detect JavaScript crashes of Rich Internet Applications. In next chapter we propose a classification framework that characterizes RIA testing techniques from different perspectives.

Chapter 6 [7]

TECHNIQUES AND TOOLS FOR RICH INTERNET APPLICATIONS TESTING

The User Interfaces of Rich Internet Applications (RIAs) present a richer functionality and enhanced usability than the ones of traditional Web applications which are obtained by means of a successful combination of heterogeneous technologies, frameworks, and communication models. Due to its increased complexity, dynamicity, and responsiveness, testing the user interfaces of an RIA is more complex than testing the user interfaces of a traditional Web application and requires that effective and efficient testing techniques are proposed and validated. In this chapter we analyse the most critical open issues in RIA testing automation and propose a classification framework that characterizes existing RIA testing techniques from four different perspectives. Driven by this classification, we present a set of testing techniques that can be used for automatically and semi-automatically generating test cases, for executing them and evaluating their results. Some examples of applying the proposed techniques for testing real Ajax applications will also be shown.

6.1 Introduction

Rich Internet applications provides a more satisfactory user experience than the one offered by traditional Web applications. This improvement is obtained thanks to a combination of Web techniques and technologies that allow several advantages, such as

[7] This chapter was published in the Proceedings of the 12th International Symposium on Web Systems Evolution (WSE 2010).

the possibility of implementing most of the business logic of the application on the client-side rather than exclusively on the server-side, of communicating with the server in both synchronous and asynchronous ways, of exchanging with the server just small amounts of data and, finally, of manipulating the inner Web page components of the user interface independently at run-time.

Unfortunately, while these applications are actually more usable, interactive and responsive that traditional Web applications, testing them may be a more complex and challenging task.

Indeed, the traditional Web testing approaches such as the ones proposed in [21, 63, 81, 82] are all based on the assumption that the business logic of the application is entirely implemented on the server side of the application and that the interaction between the user and the Web pages is limited to clicks on navigational links and to the insertion of data in forms. On the contrary, the user interface of an RIA may be considered like an event-driven software system whose behaviour is a not-deterministic one, since it usually depends on the current state of the application. As an example, the user interface of an Ajax-based RIA is made up of Web pages (or, at least, of a single page), whose status changes thanks to run-time client-side elaborations performed on the page by JavaScript event handlers triggered by user events or other external events (such as timeout events or asynchronous responses by the server). Event handlers access and manipulate the Web page using the Document Object Model (DOM) interface [62]. These handlers, besides synchronous requests, can also send asynchronous requests (for data or elaborations) to the server that introduce parallelism between the client and the server, leaving the user interface active. With these new characteristics, an Ajax-based RIA may potentially exhibit specific types of failures due to incorrect manipulations of the DOM, unintended interleaving of server messages, swapped callbacks, etc. [72].

RIA testing automation is a relevant research topic. Some research contributions to this topic have recently been proposed in the literature [5, 56, 72, 83]. These papers present different testing techniques which require that a model of the run-time behaviour of the

application user interface is preliminary obtained (either by semi-automatic techniques or automatically by crawling techniques) and different approaches are used for generating test cases that cover these models. Some of these techniques are applicable for regression testing of RIAs; other ones are also usable in other testing contexts.

However, automated testing of RIAs requires that specific problems are solved with systematic and effective solutions. As an example, suitable techniques and tools are needed for the automatic definition of test cases, for the generation of a testing oracle and for evaluating the results of a test execution.

In this chapter we analyse the critical open issues in RIA testing automation and discuss possible techniques and tools that can be used to solve them.

6.2 A Framework for RIA testing technique Classification

RIA testing techniques can be characterized from different perspectives. Here we propose to classify them on the basis of the following categories: 1) testing goal, 2) technique used for generating test cases, 3) testing oracle and 4) types of tool supporting the testing process. In this section we present these categories and their definitions.

6.2.1 Testing goal

Finding defects is the classic objective of testing: a test is run in order to trigger failures that expose defects. However, other types of testing can be executed, such as acceptance, regression, stress, load testing, etc. [96]. Hereafter we focus on RIA testing whose goal is to find application defects.

Of course, there may be several types of defect in a Web application. Guo and Sampath [97] proposed a classification of Web application faults that distinguishes them on the basis of two main dimensions, the physical location of a fault and the effect of the fault. The considered fault categories hence include: Data store faults, Logic faults, Form faults, Appearance faults, Link faults and Compatibility faults. Marchetto et al. [98, 99] have proposed a Web application fault taxonomy that explicitly takes into account some specific characteristics and sub-characteristics of a Web application in order to find possible classes of faults affecting each sub-characteristic. However, both the former and

the latter classification proposals have been defined for traditional Web applications, but they have not been thought for Rich Internet applications.

Generally, we look for defects in all interesting parts of the software application. However, whatever their position, RIA application faults either produce effects that are directly visible on the User Interface of the application, or not-visible ones. Faults producing user-visible effects can be in turn divided into generic-faults (that cause the violation of generic and implicit requirements of the UI, such as the HTML syntax validity, the accessibility requirements, the absence of broken links or server error messages, etc.) and application-specific ones. Application-specific faults produce violations of specific functional requirements of the application (as an example, an application specific fault is the one that produces an incorrect, incomplete, or inappropriate DOM configuration at a given point of an application execution).

As to the faults that produce non user-visible effects, we intend faults that do not emerge with effects on the UI but whose effects can be detected either by monitoring the application execution [100] for checking the occurrence of abnormal events, such as JS crashes, or by testing any post-condition of the application.

In conclusion, we distinguish the following testing goals:

- To detect generic/application-specific faults with no user-visible effects;
- To detect generic/application-specific faults with user-visible effects.

6.2.2 Test Case generation technique

A test case for an interactive application with an event-based User Interface (UI) can be described as a sequence of events and input values to be submitted to the application, plus a set of pre-conditions that must be verified before the test case execution. Moreover, the test case definition requires a testing oracle that provides the expected output and a set of post-conditions that has to be verified after the test execution. Techniques for generating inputs will be discussed in this subsection while oracles and post-conditions will be discussed in the next one.

Besides the traditional code-based and requirement-based testing approaches, two further

techniques are suitable for RIA test case generation: the model-based approach and the one based on real execution traces of the application. The model-based one requires that a model of application is available and test cases are selected so as to assure an expected coverage of the model components. The main limitation of this approach consists of the difficulty of obtaining a trusted model of the RIA, since models produced by the development process do not usually match the actual implementation of the application, while reverse engineered models require costly semi-automated and human-intensive processes [56, 101, 102]. Test cases can be also generated by real execution traces of the application which can be obtained in three different ways as stated in chapter 5 [5]:

- From user sessions;
- By crawling techniques;
- By hybrid approaches (e.g., mix of user session and crawler executions).

Using execution traces produced by real users (or by testers) of the application is a cheap and effective technique for obtaining test cases, already used for testing both traditional and rich Web applications [5, 81, 84, 86, 94]. Usually, these test cases allow the coverage of the most common scenarios of the application execution. This technique needs non-invasive Capture tools that can be used by more users contemporarily, for obtaining a wider set of traces.

A point of weakness of this technique is that it may produce very huge test suites that cover only the most commonly used scenarios of the application, while the rarest scenarios may not be navigated by any user. In order to improve the scenario coverage, traces produced by testers with the aim of exercising the rarest scenarios should be considered too. On the other hand, in order to reduce the size of test suites, reduction techniques that are able to select a minimal sub-set of test cases on the basis of some equivalence criterion can be used too, such as the ones introduced in chapter 5 [5].

The use of a Web crawler allows a completely automatic generation of execution traces. The realization of a crawler for the automatic navigation of dynamic Web applications is a well known challenge for the builders of search engines. The wide diffusion of Ajax

applications, which exploit techniques of dynamic generation of JavaScript code, leads to new difficulties that make this problem very challenging.

A fundamental aspect with Ajax crawling techniques is the termination criterion adopted to stop the user interface exploration when an already known interface is encountered. This problem can be solved by heuristic approaches based on user interface similarity or equivalence criteria. Two feasible solutions to this problem have been recently proposed by the authors in [1, 2, 4] and Mesbah et al. [58].

Another feasible approach for generating execution traces is offered by hybrid approaches that mix test cases obtained from execution traces produced either by real users or by testers, with test cases generated automatically by a crawler. Hybrid approaches are able to improve the effectiveness of the technique in terms of its fault detection capability and code coverage [5].

As to the pre-conditions of a test case, normally they are defined both by the internal state of the application (made up by the state of the resources, global and session variables of the application) and by the state of the execution environment of the application (such as the state and type of the browser, the state of the Web server, the communication infrastructure and protocols, the system clock, the concurrency of other executions, data sources and so on). Of course, while the internal state of the application may be usually accessed and set before a test case execution, it is difficult and, at least impossible, to set the state of the execution environment to a given known state. As a consequence, test case pre-conditions usually will include only the settable conditions and will not consider the uncontrollable one. While pre-conditions must be settable, analogous considerations can be done for test case post-conditions that vice-versa must be observable.

6.2.3 Testing Oracle

In software testing the role of the oracle is that of defining the expected output of a given execution.

When an application execution does not produce a user visible output, its effects may be alternatively deduced by testing any post-condition of the application (such as the state of

persistent data managed by the application, or the state of environmental variables). Of course, the techniques usable for accessing the post-conditions of an execution will depend on the techniques and technologies used for implementing the application itself, and sometimes these post-conditions cannot be assessed at all by the tester.

Vice-versa, when the effects of an application execution are visible on its UI, the tester just needs an oracle to define which will be the post-execution state of the RIA UI. In this case, the oracle can be either automatically provided by a previous version of the same application (but this approach is feasible just in regression testing processes) or it must be manually defined, on the basis of a knowledge of the application specifications. The manual oracle that punctually defines the Web interface obtained by an output execution is very expensive to obtain, so that it is used very seldom in the practice of Web applications.

A more efficient approach for defining the expected output of an execution is the one used in invariant-based testing approaches [83]. An invariant is defined as a property of the application that must be true and it can be checked by means of an assertion. An assertion is a boolean expression that defines necessary conditions for correct execution [101]. There are both implementation-specific assertions and implementation-independent ones. Van Deursen et al. [83] distinguish between invariants on the DOM-tree, between DOM-tree states, and application-specific invariants that are based on a fault model of Ajax applications.

More in general, we propose to classify an invariant for RIA testing according to the applicability scope of the invariant (that is, the scope of the components of the Web application which the invariant refers to) and the invariant generation technique.

As to the applicability scope, the invariant can refer either to properties that must be true for any Web application (we call it a universal invariant), or for any UI state of a given application (e.g., application level invariant), or for a specific sub-set of UI states (e.g., application group of states level invariant), or only for specific states of the UI (e.g., application single state level invariant).

The generation of invariants is a challenging task for testing and in particular for testing

automation. An invariant can be defined manually by a programmer or by an expert of the application implementation who defines the assertions that must be checked. In some particular cases (when a previous version of the application is available) the invariant can be obtained 'automatically' by analysing the output of the execution of the previous version. In other cases, a hybrid approach can be used that exploits some technique for deducing possible invariants automatically and requires human intervention for validating them. An example of this approach is presented in [3, 103] where a technique for detecting specific features of Web pages that can be used for defining invariant properties is proposed. Other semi-automatic approaches may deduce invariants by analysing the similarity of Web pages according to specific UI equivalence criteria. As an example, a simple (but often ineffective) criterion is the one that considers equivalent two interfaces if and only if they have exactly the same HTML and JavaScript code. More effective interface clustering criteria have been presented in the literature, such as the structural equivalence criteria proposed in [1, 2, 4] and the Levenshtein distance based criterion proposed in [58] that provide suitable approaches for finding equivalent interfaces of Ajax-based applications.

6.2.4 Testing automation tools

There are several categories of tool that executes basic tasks needed for RIA testing automation. They include:

- *Crawler*, that is able to interact automatically with the RIA under test and to generate execution traces by firing events on the RIA user interface. An execution trace is defined as a sequence of pairs: (widget, fired event).
- *Capturer*, that allows user interactions with the RIA under test to be recorded for generating user session traces.
- *Test Case Generator*, that is able to transform the execution traces produced either by the Crawler or by the Capturer into executable Test Cases. A test case generator is also responsible for defining the pre-conditions of a test case.
- *Test Suite Reducer*, that is responsible for reducing the size of existing test suites. It

exploits some minimization technique for assuring that the reduced test suites have the same coverage characteristics of the original ones.

- *Replayer*, that is delegated to replay test cases.
- *Execution monitor*, that observes and analyses the execution of the RIA, in order to detect specific types of events (such as the occurrence of JavaScript crashes).
- *Assertion generator*, that is able to generate the assertions on user interfaces that will be evaluated during the testing phase. An Assertion Generator can be:
  - Automatic, if it automatically generates the assertions for the interfaces.
  - Hybrid, if it suggests to the user the assertions to be evaluated for each interface. The user can validate the suggestions and accept or not them.
  - Manual, if it allows the user to set manually the assertions for each interface of the RIA.
- *Assertion verifier*, that is delegated to verify if the expected assertions associated with each interface are verified.

6.3 Tools for RIA testing automation

Several tools and frameworks are now available to support the execution of RIA testing processes. In the following we report the characteristics of both some RIA testing tools that we developed ad hoc in our research laboratory, and some other ones we selected from the Web. All the considered tools belong to the categories presented in the previous subsection and can be freely downloaded from the Web.

6.3.1 CReRIA

CReRIA, already discussed in chapters 4, is an interactive tool for dynamicanalysis that has been designed for supporting comprehension and reverse engineering processes of RIAs implemented in Ajax [4]. In particular the tool supports the semi-automatic reverse engineering of a Finite State Machine (FSM) modelling the behaviour of an Ajax application user interface. The tool offers an integrated Web browser that allows a user to navigate the RIA and to trace and record his user sessions. Hence, this component of the tool actually implements the functionality of a Capturer that can be used in testing

processes too.

As to the recorded information about traced user sessions, both data about the structure and the content of user interfaces, both data about user events that were fired on user interfaces are captured by the tool. More precisely, CReRIA describes each user interface only by a selected sub-set of its widgets (such as buttons, text fields, forms, …) that have specific properties and values of these properties. As to the user events which were fired during the navigation, the user event type (i.e. click, mouseover, mouseout, etc…), and the widget on which the user event was fired are captured by the tool. From the perspective of Ajax testing, this information can be used to produce precise test cases as sequences of user interfaces and events triggered on them.

The CReRIA tool also implements several heuristic criteria for clustering together equivalent interface states, as well as equivalent events.

6.3.2 CrawlRIA

CrawlRIA is a tool belonging to the category of automatic Crawlers. It has been designed for crawling Ajax applications and explores the user interface states by automatically firing events on these interfaces. It fires events by either a depth first or a breadth first visiting strategy. CrawlRIA is also able to extract data about interfaces and triggered events and to cluster them with the same techniques implemented by the CReRIA tool, and to generate the corresponding FSM model. In testing processes, CrawlRIA can be used for automatically generating execution traces (which can be stored in the same format and in the same database used by the CReRIA tool).

6.3.3 TestRIA

TestRIA, already introduced in chapter 5, [5] is a tool designed for Ajax test case generation and management. It implements the functionalities of a *Test Case Generator*, *Test Case Reducer*, *Automatic*, *Hybrid and Manual Assertion Generator*, and of an Assertion Verifier. Moreover, it also provides the functionality of *Test Case Replayer*.

TestRIA generates test cases by translating the execution traces collected either by CReRIA or by CrawlRIA into test cases implemented as Java test classes that use classes

from the Selenium RC library [104]. The tool is an *Assertion Generator* that builds assertions in three distinct ways: if test cases must be used for regression testing of the application, it builds automatically assertions regarding selected properties of the user interfaces of the initial version of the application that were encountered during test case executions. TestRIA is also able to suggest potential assertions about widgets found in the interfaces, or to support the manual definition of assertions. This tool is also an *Assertion Verifier* that exploits the same interface equivalence criteria implemented by CReRIA and CrawlRIA for assessing the equivalence of a given interface to a cluster of similar interfaces. It verifies generic assertions associated with Html validity or Web accessibility requirements by invoking external Web services too. Eventually TestRIA is also able to replay the generated test cases either in the TestRIA context or in a standalone way.

6.3.4 DynaRIA

DynaRIA is a tool supporting the comprehension and the testing of RIAs implemented in Ajax. It is based on dynamic analysis and provides functionalities for recording (acting like a *Capturer*) and analysing user sessions from several perspectives and for producing various types of abstractions and visualizations about the behaviour of the application. In particular it is able to execute the test cases produced by the CReRIA and TestRIA tools (acting as a *Replayer*), and to implement both a Monitor of JS crashes and a Network Monitor of Http errors.

6.3.5 Crawljax and ATUSA

Crawljax [102] is an open source Java tool supporting the automatic crawling and testing of Ajax applications. It has been developed by the SERG group at the Delft University.

Initially designed for the automatic crawling of Ajax Web applications for indexing purposes [58], the most recent releases of Crawljax also supports invariant-based testing, regression testing [105], accessibility validation, security testing, broken links/images/tooltips detection [83]. The last releases of Crawljax include ATUSA, a tool originally designed for supporting AJAX testing.

Crawljax comprehends components that explore the existing application for building a

state-flow graph representing the dynamic DOM states and the transitions between them, and a component (a *Test case generator*) that builds test cases from that model. The tool also provides several functions for generating a set of pre-defined assertions about the user interface states (*Assertion Generator*) and for verifying them (*Assertion Verifier*).

6.3.6 Selenium

Selenium [104] is a framework composed of a set of tools supporting test automation of Web applications. In particular, Selenium-IDE is a Firefox add-on providing an interface for developing test cases starting from information extracted during a user navigation session. Selenium-IDE also provides assertion suggestions during the capture activity, by proposing expressions related to the presence of widgets and attributes in the captured user interfaces. The test cases produced by Selenium IDE can be executed in the context of Selenium IDE itself, or they can be automatically replayed in the context of a program written in a high-level programming language by using the Selenium-RC API.

Table 6.1 summarizes the analysed tools and reports the types of testing tool components they offer.

Table 6.1: Coverage of Tools Categories

	CReRIA	CrawlRIA	DynaRIA	TestRIA	Crawljax	Selenium
Crawler		X			X	
Capturer	X		X			X
Replayer			X	X		X
Execution Monitor			X			X
Test Case Generator				X	X	
Test Case Reducer				X		
Automatic Assertion Generator				X	X	
Hybrid Assertion Generator				X		X
Manual Assertion Generator				X		X
Assertion Verifier				X	X	X

6.4 RIA automated testing processes

In this section, we propose two general RIA automated testing processes that exploit execution traces of the application for generating test cases. They can be instantiated in different ways in order to reach different testing goals: the first process can be used to detect faults with no user-visible effects, such as crashes, the second one aims at detecting

faults with user-visible effects. The processes will be characterized according to the categories proposed in section 6.2

6.4.1 Process #1: Crash Testing Process

The first testing process can be executed with the aim of discovering the occurrence of generic failures of the application, such as run-time crashes of the JavaScript engine, Http errors, etc. JS crashes are frequent during the execution of the JavaScript code of a Web page and are usually due to code faults such as references to non-existing objects, references to out-of-bounds array items, divisions by zero and so on. These types of defect may depend on the fact that JavaScript is a interpreted rather than compiled language, and its code can be dynamically generated at run-time. JS crashes usually do not produce visible effects on the interface.

To discover this type of defect it is not necessary to use assertions. The testing process just requires that a set of execution traces is replayed by a Replayer with the support of an execution Monitor.

In particular, the following types of tools are needed: a Test Case Generator tool that obtains execution traces either by a Crawler or by a Capturer and transforms them into test cases, a Test Case Reducer that operates a possible reduction of the test suite, a Replayer that automatically replays the test cases, and one or more execution Monitors that observe the RIA execution and detect the occurrence of abnormal events. Figure 6.1 shows a possible schema of this process.

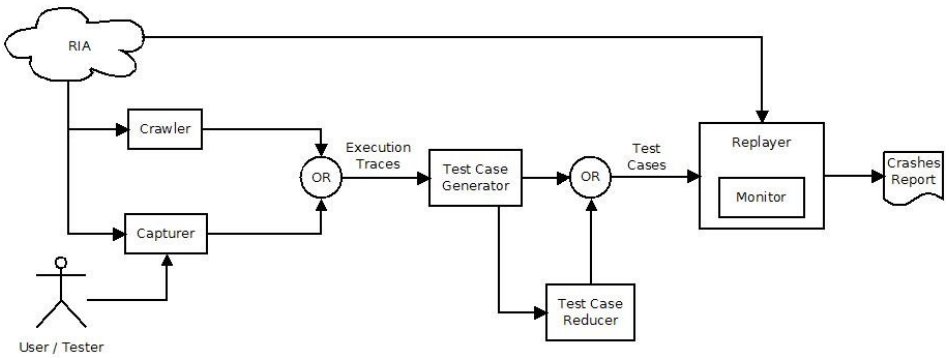

Figure 6.1: An implementation of the Crash Testing process

6.4.2 Process #2: User Visible Fault Testing process

The second testing process is executed with the aim of discovering faults with user-visible effects.

According to the definition given in section 6.2, faults with user-visible effects may be either generic-faults (that cause the violation of generic and implicit requirements of the UI, such as usability, accessibility, security, syntax validity requirements, etc) or application-specific ones. A typical example of generic fault is due to invalid HTML statements, which is very frequent in dynamic Web applications where the HTML code is generated at run-time.

An example of application-specific fault is the one that causes a given user interface to be not correct because it does not comply with the functional specifications of the application.

As an example, the page may miss some widgets (such as a foot note, a disclaimer or a common menu), or include unexpected widgets, or present a layout different from the correct one.

Both types of fault can be detected by analysing the status of the RIA UI and verifying the violation of suitable assertions.

The assertions that are valid for checking generic requirements can be produced once and are applicable to any Web application UI state. Vice-versa, assertions needed for checking application specific requirements must be defined ad-hoc for specific applications or specific UI states of a given application. Hence the generation of this type of assertions usually requires expensive manual processes. However, when the application is submitted to regression testing after the implementation of some changes and a previous version of the application is also available, the invariants can be deduced 'automatically' from the former version executions.

The verification of the assertions can be performed by any Assertion Verifier like the ones offered by TestRIA [5], Selenium [104] and by Atusa [83]. Alternatively, the invariant evaluation may be performed by invocation of external services, such as the ones offered for HTML Validation [106] or Web Accessibility evaluation [107].

Figure 6.2 shows a possible organization of such a testing process that preliminarily requires that execution traces of an RIA are collected (either by a Crawler or by a Capturer) and transformed into executable test cases by adding the assertions to be verified.

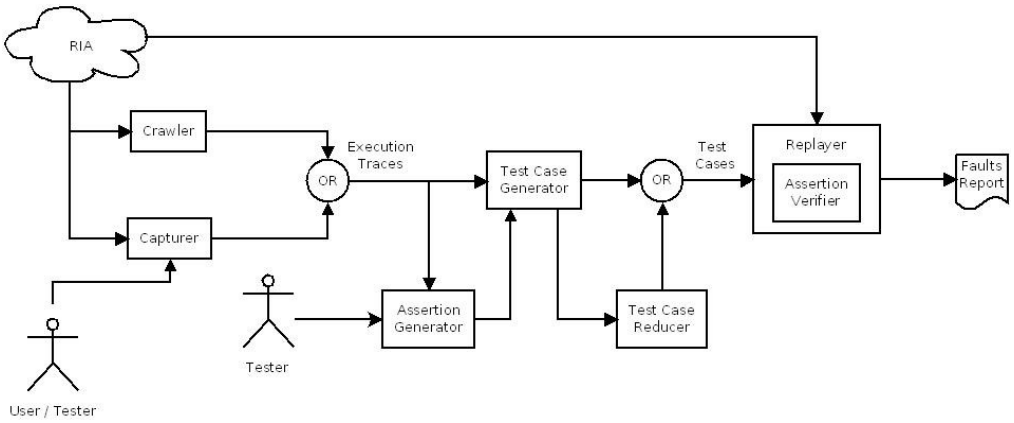

Figure 6.2: An implementation of the User Visible Fault Testing process

The Assertion Generator may produce assertions manually, automatically, or by a hybrid approach. Test cases are then executed by a Replayer that must include an Assertion Verifier component too. Some examples of possible application failures that can be detected by such a process will be illustrated in the next section.

6.5 Examples

In this section we show how the testing processes described in section 6.4 can be implemented by means of available testing tools and how they can be used to test a real Ajax application. We show the usability of the testing approaches by some examples and we discuss some problems and possible solutions that have been adopted to solve some specific testing issues.

The examples that will be presented refer to “Tudu”, an open source application offering ‘todo’ list management facilities (such as adding, deleting, searching for todos, organizing lists of todos, and so on). Tudu is a meaningful example of a simple (but not trivial) RIA whose server side is implemented with JSP pages, while its client side includes typical ‘rich’ pages that modify themselves at run-time on the basis of the user interaction with

the pages; Tudu uses a persistent data source realized with a MySql database. Tudu has been often used in case studies involving RIA reverse engineering and testing [2, 4, 5, 56, 72, 83].

6.5.1 Crash Testing

In order to carry out a crash testing on Tudu, we implemented the process proposed in section 6.4 with the support of some tools presented in section 6.3. The architecture reported in Figure 6.3 shows that the CrawlRIA and CReRIA tools are alternatively used to collect execution traces of the RIA under test, while the DynaRIA tool replays these execution traces, monitors them and reports occurred JS crashes and Http errors.

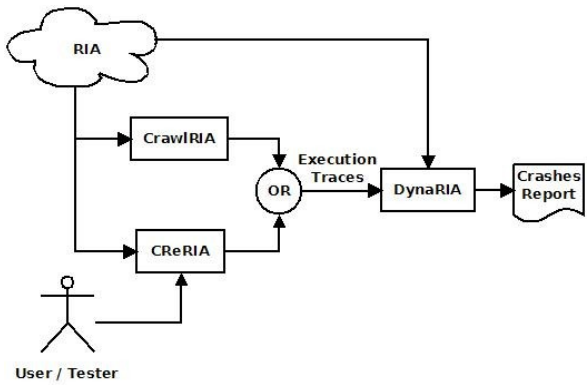

Figure 6.3: An architecture supporting the Crash Testing process

To show the effectiveness of the approach, we injected in Tudu the faults reported in Table 6.2 that were all detected by the proposed testing process. The first row of Table 6.2 shows a piece of code with a fault due to the highlighted line of code (containing the statement *alert(hiAll)*) that makes a reference to the *hiAll* variable, which has never been defined. This fault causes a crash during the function execution that does not

Table 6.2: Examples of Faults Producing JS Crashes

```
function showAddTodoList() {
 hideTodosLayers();
 $("addNewListDiv").style.display="inline";
 document.forms.addNewListForm.name.focus();
alert(hiAll);
}
```

```
function renderTableListId(listId) {
 hideTodosLayers();
 document.forms.todoForms.listId.value = listId;
 todos.forceRenderTodos(listId, replyRenderTable);
 tracker('/ajax/renderTableListId');
}
```

```
function initMenu() {
   var uls = document.getElementsByTagName("ul");
   for (i = 0; i < uls.length+1; i++) {
      if (uls[i].className == "menuList") {
         decorateMenu(uls[i]);
      }
   }
}
```

produce any visible effect on the interface.

The second row of Table 6.2 shows another fault we injected by changing the name of a referred form from *todoForm* to *todoForms* (see the highlighted statement) that does not correspond to any existing page form. The execution of this statement causes a JS crash due to the reference to a non-existing object.

The third injected fault is reported in the third row of Table 6.2 and was obtained by changing the termination value of the *for* iteration from *uls.length* to *uls.length+1*, so that an array out of bound crash occurs when the script tries to access the *uls[uls.length+1]* object.

6.5.2 User Visible Fault Testing process

To carry out a testing process for detecting User Visible Faults, the Process #2 illustrated in section 6.4 was implemented by the architecture shown in Figure 6.4.

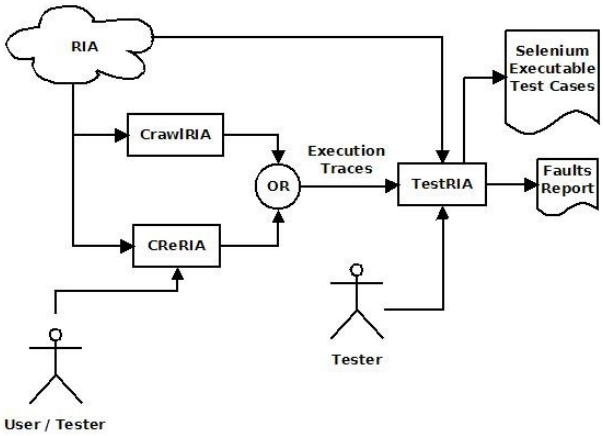

Figure 6.4: An architecture supporting the User Visible Fault Testing process

CrawlRIA and CReRIA tools are used to collect execution traces, while the TestRIA tool is used to support the tester in the generation of assertions, to generate test cases (Selenium executable test cases are also reported as outputs of the process), to replay test cases and to produce a report of the detected faults on the basis of the occurred assertion violations. To carry out this process, it is needed that suitable assertions are defined to test the execution output. Both assertions that are applicable to all the user interfaces of the application and assertions applicable just to any specific interface can be used.

As an example of assertions applicable to all user interfaces of the application, we have considered a requirement of Tudu that states the need for the logo shown in Figure 6.5 to be included in any user interface of Tudu. A possible assertion describing this property of Tudu is given by the following XPath expression:

bool(/html/body/table[1]/tbody/tr/td[1]/a/img)

Figure 6.5: The logo of the 'Tudu' application

This expression is true when an image is present in the precise location of the user interface where the logo usually stays. Of course, this expression does not check if the logo image is the correct one. In order to have a more precise checking, the following XPath expression can be used, which checks the presence of the correct logo anywhere in the page:

bool(//img[@src="http://tudu.sourceforge.net/static/2.2/images/tudu_logo.png"])

As an example of assertions that are applicable just to some specific interface of the application (such as the ones obtained by executing a specific functionality), we consider the Tudu functionality of adding todos to a list, and analyse the execution scenario where three todos were inserted in an empty todo list. To check this functionality, a specific assertion for checking the results of the insertion is needed. Such an assertion contains the following XPath expression that verifies if a list with three todos is shown:

count(//table[@class='list']/tbody/tr/td/div[@style!='display:none;'])=3

In order to test the effectiveness of the test cases including this assertion, we have first executed them on the original application, and obtained the final interface shown in Figure 6.6.

Figure 6.6: An example of interface showing a todo list containing 3 todos

This interface correctly verifies the assertion listed above.

Then we have produced a faulted version of the application, by injecting a fault that disables the code responsible for the insertion of the submitted todos, and we have executed the same test cases on this version. In this case, the user interface shown in Figure 6.7 was obtained, which does not satisfy the assertion, so that the fault was correctly detected.

Figure 6.7: An example of interface showing an empty todo list

6.5.3 Regression Testing Process

The last example regards a Regression Testing process that was executed with the support of the software tools reported in Figure 6.8.

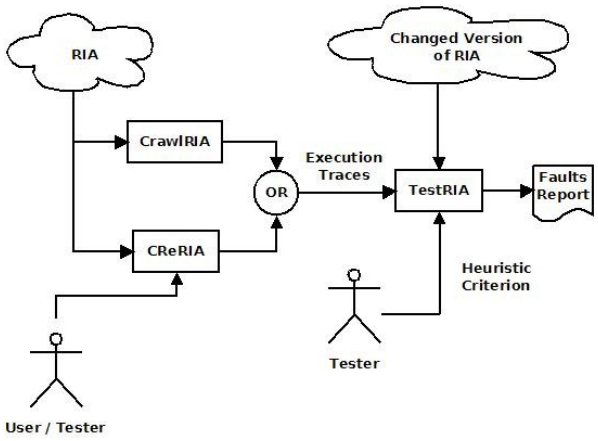

Figure 6.8: An architecture supporting the Regression Testing process

While the CrawlRIA and the CReRIA tools are always used to collect execution traces, the TestRIA tool is now used to generate test cases from the collected execution traces, to automatically generate assertions, to replay the generated test cases on a changed version of the RIA and to produce a report of the detected regressions on the basis of the violated assertions. The assertions in this case were defined to check the equivalence (with respect to a given user interface equivalence criterion) between the homologous interfaces

obtained by executing the same test case on the two different versions of the RIA. The unique intervention of the Tester in the Test Case generation activity consisted of the selection of the heuristic clustering criterion by means of which the equivalence between the interfaces can be assessed. The TestRIA tool is able to evaluate four different equivalence criteria that are described in [2, 4, 5].

As an example of detectable fault, we have considered a scenario consisting in the deletion of all the todos from a todo list. In the changed version of the RIA we inserted a change causing the todos not to be actually deleted.

The C3 clustering criterion presented in [2] was adopted to define the assertions. The criterion is defined as follows:

Two client interfaces I1 and I2 are equivalent if the same active widgets of I1 that are visible and enabled are also included in I2 and vice versa, and they have the same un-indexed path.

Our tests revealed that this criterion was effective in the distinction between the expected interface (that was similar to the one shown in Figure 6.7) and the interface obtained by replaying test cases exercising the deletion scenario in the faulted version of the RIA. In fact, the latter interface contained a non empty list (that was similar to the interface shown in Figure 6.6), that the assertion did not consider equivalent to the expected one.

6.6 Conclusions

Due to the wide diffusion of Rich Internet Applications recorded in the last years, the need for effective and efficient processes, techniques and tools for testing them and assuring their quality has been recorded too. In this chapter we have proposed a classification that distinguishes between RIA testing techniques for finding faults having no effects on the RIA user interface and techniques that are suitable for finding faults with user visible effects. Possible solutions to the problems of generating test cases, defining testing oracles, and automatically evaluating the results of test case executions have been analysed for both types of testing techniques. Moreover, two general testing processes

have been proposed for Ajax applications, where the former is suitable for finding execution crashes, while the latter can be used for finding faults that produce effects visible on the User Interface. Some possible instantiations of these processes that are based on already existing tools for Ajax testing automation have been also discussed in the chapter.

The presented processes are able to discover automatically or semi-automatically different failures in RIA applications. However, further work is needed for extending these processes and the supporting tools in order to address other types of RIA failures. Moreover, empirical studies should be carried out for assessing systematically the fault detection capability and the scalability of these approaches.

Future work will address both this issue and the problem of comparing the proposed approaches with other approaches already proposed in the literature.

Chapter 7 [8]

COMPREHENDING AJAX WEB APPLICATIONS BY THE DYNARIA TOOL

Thanks to Rich Internet Applications (RIAs) with their enhanced interactivity, responsiveness and dynamicity, the user experience in the Web 2.0 is becoming more and more appealing and user-friendly. The dynamic nature of RIAs and the heterogeneous technologies, frameworks, communication models used for implementing them negatively affect their analysability and understandability. Consequently, specific software techniques and tools are needed for supporting RIA comprehension. In this chapter we present DynaRIA, a tool for the comprehension of RIAs implemented in Ajax that is based on dynamic analysis. It provides functionalities for recording and analysing user sessions from several perspectives, and for producing various types of abstractions and visualizations about the run-time behaviour of the application. In order to evaluate this tool, four case studies involving different comprehension tasks of Ajax applications have been executed. The experimental results showed the usefulness and effectiveness of the tool that provided a valid support for Ajax comprehension in reverse engineering, debugging, testing and quality assessment contexts.

7.1 Introduction

As written in previous chapters Ajax Web applications exploit a combination of Web

[8] This chapter was published in the Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC 2010) and was partially published in the Proceedings of the 18th International Conference on Program Comprehension (ICPC 2010).

technologies for obtaining a richer interaction of the user with the application. The user interface of an Ajax application is implemented by one or more Web pages that are composed by individual components, which can be updated, deleted or added at run time independently. The manipulation of the page components is performed by an Ajax engine written in JavaScript (JS) that is loaded by the browser at the start of the session, accesses the page components by the DOM interface [62] and is responsible for communicating with the server on the user's behalf [17]. The execution of the engine's JavaScript functions is driven by user events or other external events (such as server responses or time-out events). Besides synchronous requests of data or elaborations, these functions are also able to send asynchronous requests to the server side of the application that introduce parallelism between the client and the server.

Using Ajax or similar development approaches the user interfaces of Web applications become similar to the ones of desktop applications and provide the same type of user experience. Unfortunately, while the new implementation techniques improve the usability of these applications, there are several factors that negatively impact their internal quality characteristics, such as the comprehensibility and analysability. As an example, RIAs have an heterogeneous nature that make several types of code parsers necessary for their analysis. Moreover, RIAs are dynamically configured systems (as an example, in Ajax applications, JavaScript modules composing the engine can be requested to the server at run-time, as well as new JavaScript functions can be dynamically generated), which make static code analysis not sufficient to gain a deep understanding of the application. Nowadays RIAs are being implemented using a wide variety of frameworks [108], which accelerate development, but lead to opaque application behaviour and make the analysis of generated code and of the interaction among its parts more complicated.

These factors produce a general worsening in the maintainability and testability of RIAs and raise the costs of developing and assuring their quality. As a consequence, there is a great need for effective techniques and tools supporting the efficient execution of analysis tasks involving RIAs. Several open-source and commercial tools offering specific

functionalities of Ajax analysis are now available. Most of them have been designed to support Ajax development and provide functionalities of JavaScript debugging, DOM inspection and network monitoring. Some other ones perform dynamic analysis of the application and record several relevant aspects of a session, such as network requests, JavaScript source code and DOM events. However, the features of these tools have not been designed to support comprehension processes explicitly.

In this chapter we present DynaRIA, already introduced in chapter 5. DynaRIA is a tool for the comprehension of Ajax applications that has been designed to support analysis tasks to be executed in different contexts, such as maintenance, quality assessment, reverse engineering and testing. DynaRIA is based on dynamic analysis and provides functionalities for recording and replaying user sessions, for analysing them from several different perspectives, and for producing several types of abstractions and visualizations about the run-time behaviour of the application. DynaRIA has been implemented in Java using the NetBeans IDE and open source technologies. In order to evaluate this environment, we performed four case studies, where some tasks that were representative of typical comprehension, debugging, quality assessment, and testing activities were executed with the support of the tool. The case studies involved two real Ajax applications and their results showed the actual utility of the tool in supporting the considered types of activity.

7.2 Related Works and Tools for the comprehension of Ajax

In the last years, several approaches for Ajax analysis have been proposed in the literature, both in reverse engineering and testing contexts.

A first proposal is due to Mesbah et al. [58], who presented a technique for crawling Ajax applications through dynamic analysis and obtaining a ‘state-flow graph’ modelling the various navigation paths and states within the applications. This technique was initially proposed for generating a multi-page static version of the original Ajax application that could be used to expose it to general search engine. Later, the same technique has been used to support an invariant-based automatic testing technique of Ajax user interfaces

[83]. Duda et al. proposed in [91] another technique for crawling Ajax applications using hash values of the state content for discovering similar states of the user interface.

In previous chapters we have addressed the problem of obtaining a model of the behaviour of an Ajax application user interface by reverse engineering [1, 2], proposing a technique based on dynamic analysis that exploits some user interface and transition equivalence criteria for abstracting a Finite State Machine (FSM) from user session data. The technique is supported by CReRIA, a tool for the automatic collection and analysis of user sessions. Further techniques for analysing Ajax applications have been proposed to support the execution of testing processes. Marchetto et al. [56, 72] proposed two approaches for testing Ajax applications that exploit a partially automated technique to recover a state graph of the application by analysing its execution.

Due to the growing diffusion of Ajax applications in the last years, several tools supporting their development and run-time analysis have been proposed: most of them are JavaScript debuggers, Ajax profilers, and tools for automated testing. In the following, an overview of the characteristics of some of them is reported. Firebug is a very popular tool for Ajax analysis [109, 110] that is distributed as a Firefox add-on to be executed inside the Mozilla browser. Firebug offers facilities for inspecting and editing Web pages, and highlighting the changes of its nodes at run-time.

It provides a console for editing and executing new JS code, and a JS debugger. Moreover, Firebug offers a useful monitor of the network activity that tracks the progress of both synchronous and asynchronous requests (by the XMLHttpRequest channel) to the server, and a profiler of the JS function executions, reporting for each function the calls it made, minimum, maximum, and average execution time. The Ajax Toolkit Framework ATF [111] has similar features to the ones of Firebug, but it is distributed as a plug-in for the Eclipse IDE.

Besides the functionalities of DOM and CSS inspecting, network monitoring, and JS debugging already proposed by Firebug, ATF has the additional characteristic of offering an integrated Mozilla browser and a framework on which adopters can build advanced and

technology specific tools.

As to the category of debuggers, an example is given by Venkman [112], the Mozilla's JavaScript debugger.

It offers traditional debugging functionalities (such as breakpoint management, call stack inspection, and variable/object inspection) besides an interactive console that also allows the execution of arbitrary JavaScript code.

Another category of tools is that of profilers, such as the Dynatrace AJAX Edition [113] tool for Internet Explorer.

This tool analyses, records and saves several aspects of a session, such as network requests, JavaScript executions, all DOM events, etc., and provides graphical views of the performance of the application (such as page loading time, network request time, amount of resources used, types of resources used, JavaScript execution time, and rendering time).

Other tools support automated testing of Ajax: an example is provided by the Selenium testing framework [90] that was originally designed for capturing and replaying user interactions with traditional Web applications. Recently, Selenium has added specific constructs (such as the waitForText, waitForCondition, etc.) that can be used for correctly replaying the interactions with Ajax applications that exploit asynchronous messaging between client and server.

We observed that most of the analysed tools provided either very detailed views on separate aspects of an RIA (such as its HTML, JS code, or network traffic in specific moments of the execution), or high level views about just the performance of the Ajax application.

None of these tools included the most typical extraction, analysis, cross-referencing, and presentation features that support top-down, bottom-up, or opportunistic software comprehension approaches [114].

As an example, no tool provided abstraction mechanisms for obtaining views documenting the structure, the behaviour, or the run-time interactions between the Ajax application components, such as UML structural or behavioural diagrams.

Table 7.1 summarizes the most relevant features of Ajax analysis offered by the tools we analysed and by the DynaRIA tool. The features include: JS debugging, DOM change inspecting, network monitoring, user session tracing, user session replaying, performance analysis, code coverage and UML diagrams abstraction.

Table 7.1: Ajax Analysis Features offered by the considered tools

	Firebug	Ajax Toolkit Framework	Venkman	DynaTrace	Selenium	DynaRIA
JS debugging	Y	Y	Y	N	N	P
DOM change inspecting	Y	Y	N	N	N	Y
Network Monitor	Y	Y	N	Y	N	Y
User Session Tracing	N	N	N	Y	Y	Y
User Session Replaying	N	N	N	N	Y	Y
Performance Analysis	Y	N	P	Y	N	P
Code Coverage	N	N	N	N	N	Y
UML diagrams abstraction	N	N	N	N	N	Y

In the following, we present the DynaRIA tool whose features were specifically designed for supporting the comprehension of Ajax applications.

7.3 The DynaRIA Tool

As reported by Storey in her review on program comprehension theories, tools and methods [114], tools for program comprehension usually include three categories of features: extraction, analysis and presentation. Extraction tools include parsers and data gathering tools to collect both static and dynamic data. Analysis tools support activities such as clustering, concept assignment, feature identification, slicing, or similar ones. Presentation tools include code editors, browsers, hypertext viewers, and visualizations. The set of features included in software comprehension and reverse engineering environments usually depends on the purposes of these tools, which may vary from aiding top-down or bottom-up comprehension processes, to supporting reverse engineering, maintenance, or testing activities.

The DynaRIA tool has the purpose of supporting comprehension, quality assessment and testing activities involving the client-side of Ajax applications. The tool's features and its architecture are described in the following sub-section.

7.3.1 DynaRIA's program comprehension features

The DynaRIA tool provides the following extraction, analysis and visualization features. As to the extraction, it gathers dynamic data from the run time behaviour of an Ajax application. To this aim, the tool provides an integrated Web browser where a user can interact with a Web application while all relevant data about this user session are captured and stored. Collected data include: the sequence of user events fired on DOM objects of the user interface, the JavaScript functions that are activated by user event handlers, the executed lines of code of JS functions, exceptions and errors occurred at run time. At the same time, the tool keep tracks of the changes (such as add, delete, or change) on DOM objects resulting from a given event management, analyses the network traffic, and monitors message exchanges between client and server.

The class diagram in Figure 7.1 shows the conceptual model of collected data.

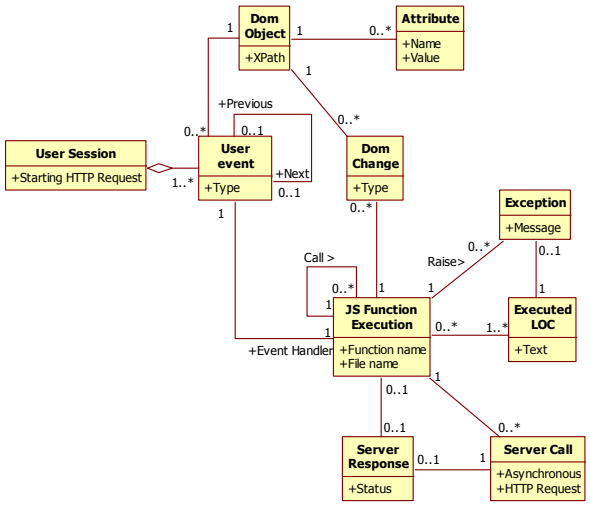

Figure 7.1: The Conceptual Model of collected data

The tool provides some session analysis functions that have been designed to support code artifact discovery, or to display relevant attributes of the retrieved items. As an example the tool analyses the run-time behaviour and provides the sequence of events that were

fired, the associated JS function call-tree, the set and number of executed JS functions (distinguishing among application's functions, functions from development frameworks, and dynamically generated ones), the set of server requests made by a JS function and server callbacks, the set of DOM changes it made and summary data and metrics about JS function executions (such as minimum, maximum, and average execution time, # of server requests it made, percentage of executed lines of code, etc). Moreover, the tool is able to abstract an Event-flow-graph [47] that reports the flow of events fired along a user session and UML sequence diagrams at various levels of detail and abstraction from each user session or from its parts. The high-level sequence diagrams show the observed interactions among three layers of the application, e.g., the browser, the Ajax engine, and the server side. The low-level sequence diagrams report the observed interactions between the Web browser, the single JavaScript modules making up the Ajax engine, and the server side. The DynaRIA tool provides functions for exporting the abstracted diagram in XML format.

Finally, the tool provides several features of software visualization. Multiple views are offered both at the session level and at the JS function level, and cross-referencing functions are provided for switching between views. At the session level, the tool provides both UML sequence diagram visualizations and Event-flow-graph visualizations. At the JS function level, views reporting details about JS function code, JS executed lines of code, JS call tree, DOM changes, network traffic and exceptions are provided. A view on the DOM before and after the management of a given event is also offered by the tool. In next sections some examples of these views and of the cross-reference mechanisms are reported.

7.3.2 DynaRIA's Testing features

As to the testing activity support, the DynaRIA tool provides functionalities for recording user sessions and replaying them automatically. Capture and Replay tools, such as the Selenium IDE, or similar ones, already provide this type of functionality; however the DynaRIA tool offers additional features for test suite error detection and coverage

evaluation. Indeed, during user session replay, the tool traces the JS code execution, keeps track of performed network traffic and detects any JS error or network warning occurred at run-time.

Moreover, with respect to a replayed user session, the tool computes several code coverage metrics such as the percentage of executed JS functions with respect to the defined JS functions and the percentage of executed JS function LOC with respect to the defined JS function LOC. These coverage metrics can be used for evaluating the effectiveness of test suites obtained from user sessions.

7.3.3 DynaRIA's quality assessment features

The dynamic analysis performed by the DynaRIA tool provides insights into the internals of an Ajax application that can be used for expressing a judgment about the application's internal quality too. As an example, the DynaRIA tool is able to compute some complexity and coupling metrics about the JS code that is contained either in HTML pages or in js files of the application (hereafter, JS modules).

These metrics are all computed by the tool based on the data that are retrieved with respect to a given set of executions of the application. The complexity metrics include:

- # JavaScript modules making up the Ajax engine;
- JavaScript module size (in LOC);
- JavaScript module size (in # JS function);
- JavaScript function size (in LOC).
- As to the coupling between modules, the following metrics can be evaluated:
- Fan-in of a JS module (that is the number of distinct calls to JS functions of the subject module);
- Fan-out of a JS module (that is the number of distinct calls to external JS functions made by functions of the subject module);
- Call Coupling between JS modules (that is given by the number of distinct calls to JS functions made by functions of the first module to the second module functions);

- Server Coupling of a module (that is given by the number of distinct HTTP requests to the server that are made by a given module);
- DOM coupling (that is the number of distinct DOM change instructions that are executed by a given JS module).

These metrics can be used for assessing several quality aspects of the RIA, such as its maintainability or testability. As an example, in a maintenance process the complexity metrics can be used to detect the modules whose changes potentially require greater effort (due to their size or coupling to other modules of the application). Analogously, in a testing process, the Server Coupling metric may be exploited for counting the number of different server stubs that at least must be developed for the unit testing of that module; hence it provides a testability indicator for that module.

7.3.4 The architecture of the DynaRIA tool

The DynaRIA tool has been developed using Java technologies. The architecture of the tool includes six main packages and eighteen sub-packages that are illustrated by the UML package diagram reported in Figure 7.2.

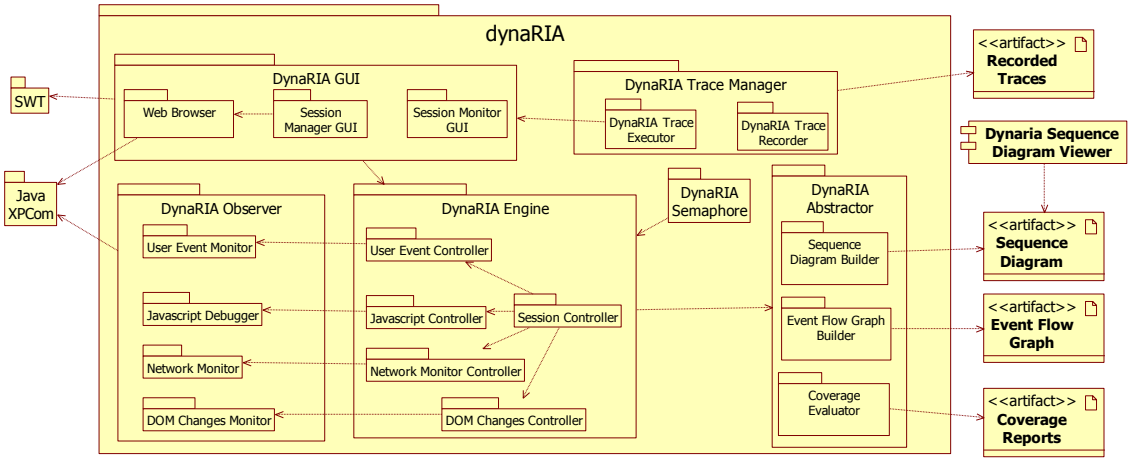

Figure 7.2 The DynaRIA tool Architecture

In particular, the DynaRIA GUI package implements the graphical user interface of the tool and has been developed using the Java SWT libraries [64]. It offers an embedded Mozilla Web Browser, a 'Session Manager' GUI for recording user sessions and analysing or replaying them, and a 'Session Monitor' window providing views on session's events, executed JavaScript code, DOM changes, Client-Server communication details, etc..

The DynaRIA Observer package is responsible for capturing all run-time data such as user events triggered on the interface, the JavaScript code loaded and executed at runtime, DOM changes, and the Client-Server message exchanges. This package has been developed using the APIs of the JavaXPCOM library [66].

The DynaRIA Engine package is the core component of the tool that implements all the business logic of the application and coordinates the execution of all the other tool's packages.

The DynaRIA Abstractor package is responsible for performing the abstraction functions regarding the Web application run-time. The output of these functions is stored in XML files (representing the sequence diagrams), DOT files (providing the EFG graphs in the dotty software format [74]) and text files reporting summary data about the overall Web application execution. Figure 7.2 also reports the DynaRIA Sequence Diagram Viewer tool that is responsible for visualizing the sequence diagrams produced by DynaRIA. This tool has been developed in Visual Basic .NET 2008, and using the Windows Presentation Foundation (WPF) library.

7.4 Case studies

Program comprehension tools are often evaluated by researchers using case studies and evaluation frameworks [114]. The aim of case studies is to evaluate the performance of the tools in a realistic software comprehension scenario, while evaluation frameworks define comprehension tasks that can be used for comparing them [115]. An example of evaluation framework is provided by Pacione et al. [116] who defined a set of both general and specific comprehension tasks for comparing the performance of software visualization tools. This framework has recently been used by Cornelissen et al. to derive some representative tasks for a quantitative evaluation of a tool for the visualization of large execution traces [117].

For evaluating our tool we carried out four case studies, involving two different RIAs where some tasks that were representative of typical analysis activities were executed with

the support of the tool.

The first case study focused on a 'feature comprehension' activity, the second one explored an 'error detection activity' in a testing and debugging context, the third one dealt with a 'testing evaluation activity', and the latter one focused on a 'quality assessment activity'.

7.4.1 First Case Study

In this first case study, we selected a functionality offered by the AjaxFilmDB application available from [68] and considered a comprehension activity whose goal was to understand how the functionality is implemented.

The subject application provides registered users with functionalities for the management of a personal movie archive (including the visualisation of a movie description, the insertion, modification, deletion of a movie, and the search for movies in the archive), the management of movie loans, and so on. The selected functionality consisted in adding a new movie to the personal archive of a registered user of the application.

The comprehension activity was assigned to one of the authors (hereafter, the software engineer) who was a familiar user of the application, but had no knowledge about its internals.

To accomplish this activity, the author used the DynaRIA tool for monitoring several executions of the functionality (which corresponded both to the successful accomplishment of the functionality, both to exceptional scenarios), and thanks to the views and reports produced by the tool he was able to address with success the comprehension tasks reported in Table 7.2.

Table 7.2: Comprehension Tasks in the first case study

Comprehension task descriptions (for the selected functionality)	
T1.1	How do the high-level components of the application interact ?
T1.2	What low-level components of the application interact?
T1.3	How do the low-level components of the application interact?
T1.4	What low-level components exchange messages with the server side of the application?
T1.5	What are the internal elaboration details of the considered functionality?

The T1.1 task was solved using several high-level UML sequence diagrams reporting the

interactions among the browser, the Ajax engine and the server side of the application in both normal and exceptional execution scenarios of the functionality. An excerpt of the diagram associated to the successful function execution is reported in Figure 7.3.

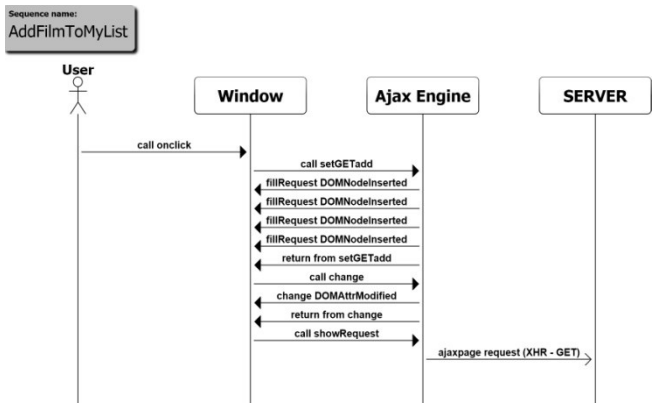

Figure 7.3: An excerpt of an high level UML sequence diagram for an AjaxFilmDB functionality

Using this diagram, the software engineer obtained a comprehension of the set of fired user events, DOM changes produced by the Ajax engine, synchronous and asynchronous requests to the server, as well as server responses. In particular, this execution involved 8 user events, 46 messages from the Ajax engine (including 42 messages representing DOM change requests and 4 requests to the server), 4 server responses and 0 exceptions.

The tasks T1.2, T1.3, T1.4 were accomplished using low-level UML sequence diagrams. An excerpt of one of them is shown in Figure 7.4.

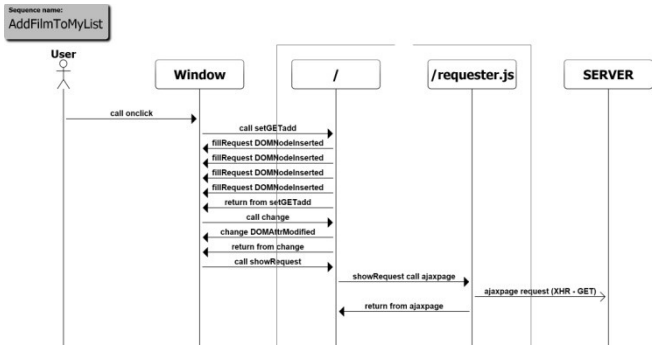

Figure 7.4 An excerpt of a detailed UML sequence diagram for an AjaxFilmDB functionality

This diagram provides a more detailed view on the Ajax engine internals, where the interacting objects above the lifelines represent the JS modules containing the executed JS functions.

Table 7.3 reports summary data about these interactions.

Table 7.3: Summary data about the traced execution

# user events	8
# JS modules	6
# JS function calls	97
# server requests	25
# server responses	25
# DOM change requests	42
# exceptions	0

The T1.5 task required the comprehension of internal details of the elaboration. In an Ajax application, the implementation of a functionality can be analysed from several distinct perspectives, e.g., the one of the *events* that are fired on the UI and trigger the elaboration, the one of the *JS functions* that carry out the elaboration, the perspective of the *server* that provides data or elaboration by communicating with the client, and the perspective of the *User Interface* where the effects of the elaboration are shown.

The software engineer analysed the execution of the selected Ajax functionality from these four perspectives using the 'Session Monitor' view offered by the tool. This view is composed of several panels showing data and details that are relevant for each considered perspective. An example of this view is reported in Figure 7.5.

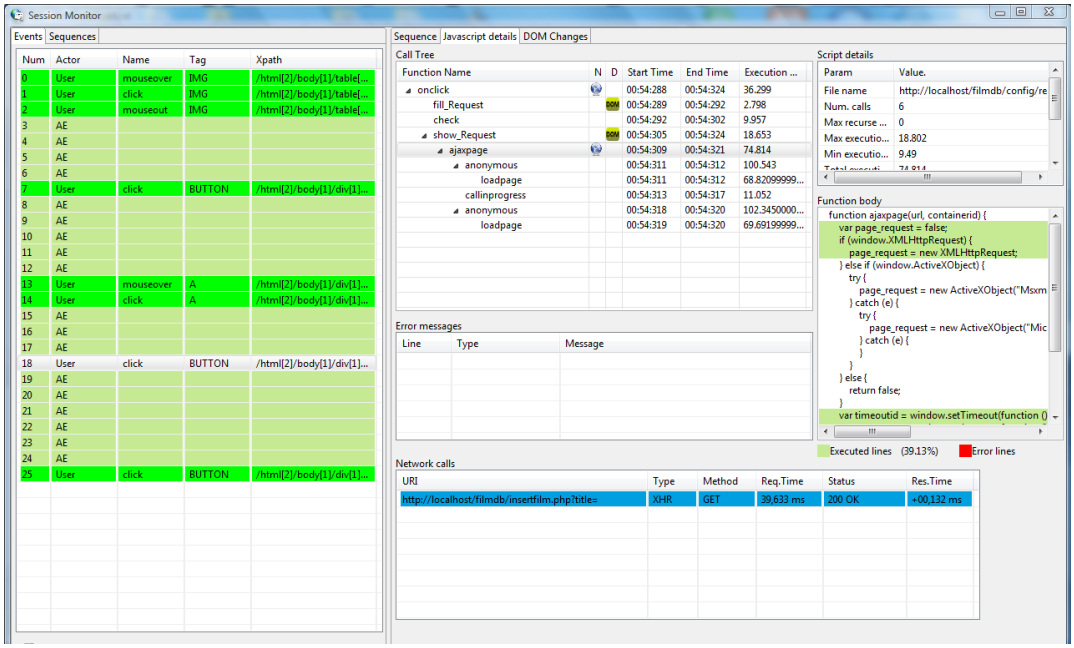

Figure 7.5: The Session Monitor view provided by DynaRIA

The left side panel corresponds to the 'fired event' analysis perspective and reports the

sequence of fired events and, for each event, the actor who fired it, the name of the event, and the DOM object on which the event was fired (by its tag and XPath).

Clicking on a given event, the central panel will report a view on the 'JS function' perspective that includes: the call-tree of executed JS functions and details about each function execution, such as Start, End, and execution times, and an indication about network requests or DOM changes performed by the function. This last information is represented by graphic icons reported in the 'N' and 'D' labelled columns of the panel. Further details belonging to this same perspective are provided by two right side panels showing a summary of script function details and the script function body (where executed lines of code are highlighted with a different colour), respectively.

The third analysis perspective is about the 'interactions with the server' and it is offered by the lower central panel that shows network calls made by a selected JS function. The Error Message panel finally shows details about occurred exceptions. Eventually, a given elaboration can be analysed from the 'User Interface' perspective that is offered by another view of the tool (that is obtainable by selecting the 'DOM changes' tab of the central panel), which reports details about the DOM changes produced by a JS function execution. As an example, Figure 7.6 depicts an instance of this view that includes three distinct panels reporting details about added, deleted, or modified DOM nodes, while the low right panel shows the rendering of one of the modified nodes that was selected in one of these panels.

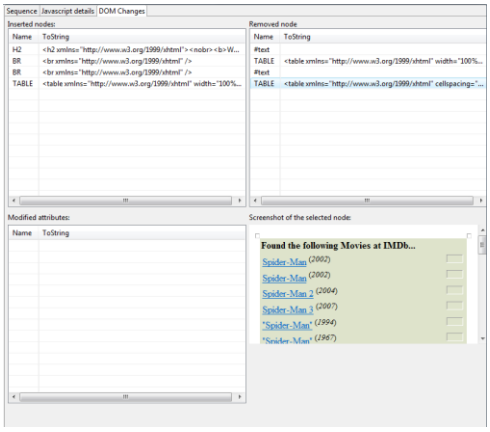

Figure 7.6: The DOM Changes panel of DynaRIA

In conclusion, all the considered comprehension tasks were accomplished thanks to the

high-level views and the lower-level ones offered by the tool, and thanks to the opportunity for a user of DynaRIA of navigating through different views. These views provided useful insights about the implementation of the selected functionality.

Other Ajax dynamic analysers (like the DynaTrace Ajax edition tool presented in section 8.2) lack of high-level views produced by our tool, but often provide just low-level ones that focus on the run-time performance of the application rather than on its implementation details. Hence, we concluded that the features of the DynaRIA tool supported the comprehension activity more effectively than other tools.

7.4.2 Second Case Study

In this case study, we analysed the support offered by DynaRIA in testing and debugging contexts.

The goal of the considered activity was of finding exceptions of an application's functionality execution and comprehending what JS components were responsible for them. To this aim, one of the authors injected faults of different types in the JS code implementing the functionality of 'adding a movie to the user's archive' of the Ajax FilmDB application, and asked another author for addressing the comprehension tasks reported in Table 7.4.

Table 7.4: Comprehension Tasks in the second case study

Comprehension task descriptions (for the selected functionality)	
T2.1	What run-time exceptions do occur during the functionality execution?
T2.2	What JS functions (and lines of code) are responsible for run-time exceptions?

The T2.1 task was completed with success thanks to the functionality provided by the tool of detecting JS exceptions at run-time. In particular, the types of JS exception that are detectable by the DynaRIA tool include the ones caused by references to not defined objects/methods/attributes, JS function call instructions with undefined, incorrect, or missing parameters, JS syntax errors, array out of bound errors, server requests of missing resources or JS Files. The T2.2 task was solved by the tool's feature of detecting the

components that are involved in the exceptional execution. In particular, the 'Session Monitor' view reports in different panels: the event that triggers an exceptional execution, the sequence of called JS functions, the JS function and its line of code that caused the exception, the type of the exception, and the corresponding message error.

As an example, one of the exceptions that was detected in this case study was due to a '*check*' script function that is executed during the handling of the '*click-_3*' user event.

As the 'Session Monitor' view in Figure 7.7 reports, the occurrence of this exception is signalled in its central panel by the highlighted colour of the '*check*' function row.

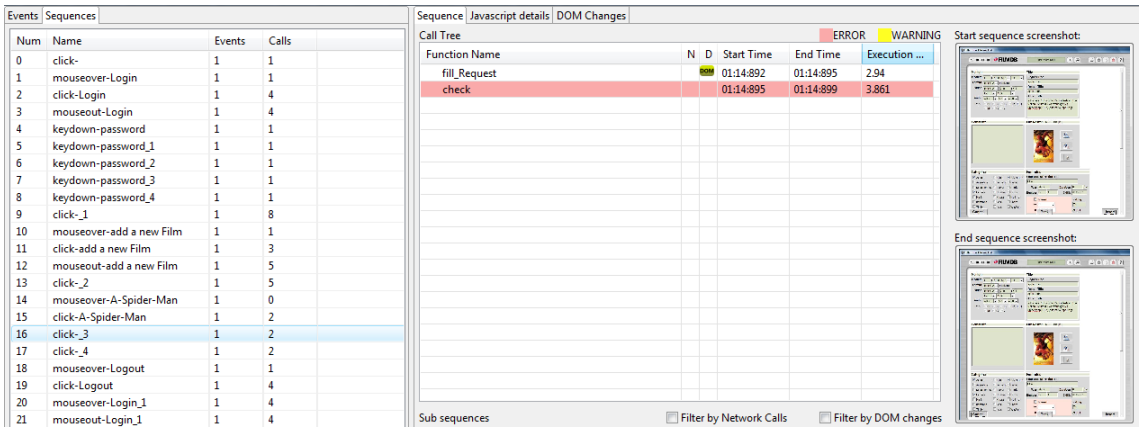

Figure 7.7: The Session Monitor View showing the exception caused by the 'check' script function executed during the handling of the 'click-_3' user event

Figure 7.8 reports the error messages panel showing details about this exception, while Figure 7.9 shows the body of the *'check'* function, and the faulty line of code.

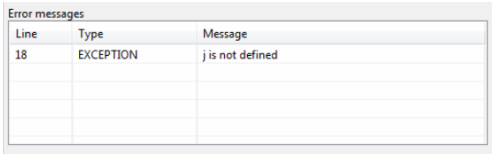

Figure 7.8: The Error message panel showing the detected exception

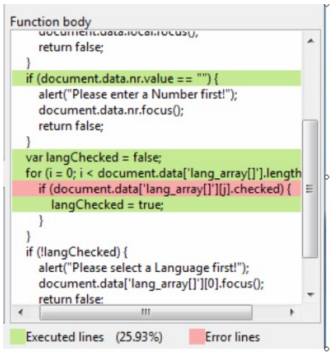

Figure 7.9: An excerpt of the 'check' function body causing the exception

The right side of Figure 7.7 depicts user interface screenshots of the analysed Ajax

application at the start and the end of the executed event sequence, respectively. This view can be used to visually track the user interface evolution and it may show the effects of the occurred exception.

In conclusion, thanks to the tool, the software engineer was able to find the exceptional executions and to locate their causes in the JS code effectively. Moreover, the tool provided such a detailed knowledge about the exception leading elaboration that it could be used to support debugging activities. Debugging tools such as Firebug or Venkman are able to signal the instruction leading to the exception and its call stack, whereas DynaRIA reports the complete elaboration sequence triggered by the user event too.

7.4.3 Third Case Study

In this case study we explored the contribution of DynaRIA in a testing context. In particular, we considered the problem of assessing the effectiveness of test suites by evaluating their code coverage and fault detection capability.

We considered a user-session based testing technique proposed in the 5th chapter [5] and decided to use it for testing an open source Ajax application called 'Tudu' [118] that offers *'todo'* list management facilities (such as adding, deleting, searching for todos, organizing lists of todos, and so on). This technique is based on the sequential tasks reported in Table 7.5. One of the authors executed these tasks with the support of the DynaRIA tool.

Table 7.5: Tasks in the Third case study

Testing task descriptions	
T3.1	Generation of a test suite from user sessions
T3.2	Test suite coverage assessment
T3.3	Generation of several application faulty versions by fault injection
T3.4	Replay of test suites on the faulty versions of the application
T3.5	Test suite fault detection capability assessment.

As to the T3.1 task, the software engineer recorded 21 user sessions by the tool, triggering 518 events and navigating 539 interfaces. The corresponding test suite TS consisted of 21 test cases. To accomplish the T3.2 task, the test suite's code coverage was automatically computed by the tool and consisted of 172/1018 (16.9%) (distinct JS functions that were executed / distinct JS functions that were defined in the JS code).

This small coverage of JS functions could be explained because the Tudu application largely includes library functions from frameworks, but just a little part of these library functions are actually used by it. As to the coverage of function LOC, the value of 1016/6150 (16.5%) was obtained.

The T3.3 task was executed manually by another author who injected 19 faults and obtained 19 faulty versions of Tudu, each one containing just one fault. Table 7.6 shows a summary of the typologies of the injected faults. The faults were all able to generate exceptions possibly detectable by the DynaRIA tool.

Table 7.6: Summary data about the injected faults

Fault Type	Number of injected faults
References to not defined objects/ methods/ attributes	7
JS function call instructions with undefined, incorrect, or missing parameters	5
JS syntax errors	2
Array out of bound errors	2
Server requests of missing resources or JS files.	3

The T3.4 task was automatically performed by our tool, and the fault detection capability (T3.5 task) was found to be 100%. Thanks to this case study, we assessed that the tool provides a valid aid for client-side automated testing of Ajax applications. It offered the same functionality of other Capture and Replay tool for Web applications, such as Selenium [90], but also the additional features of code coverage computation.

7.4.4 Fourth Case Study

In this case study we analysed the contribution of the DynaRIA tool in carrying out tasks of internal quality assessment of an Ajax application's JS code. To this aim, we considered the 'Tudu' Ajax application and used the complexity and coupling metrics computed by the tool as possible indicators of its internal quality.

One of the authors exercised the application through the DynaRIA's browser and executed a user task that included the following sequence of actions: User registration- Login- Adding a todo list- Adding a todo- Logout. By means of the tool, the set of JS modules, which had been loaded at run-time, was obtained.

The modules were characterized with respect to their complexity and coupling levels: indeed, the tool computed the size (in LOC and number of JS functions) of the involved JavaScript modules, as well as their Fan-in, Fan-out, and Coupling values.

The values of some of these metrics are reported in Table 7.7.

Table 7.7: Some complexity and coupling metrics about the Tudu application

Module	# JS func.	LOC	Fan-in	Fan-out	# Serv. Requ.	# DOM Changes
logout.action	2	2	1	1	0	0
scriptaculous.js	395	2693	27	15	5	0
util.js	65	1321	17	28	4	140
showTodos.action	54	338	17	17	1	3
todos.js	45	90	4	6	0	0
welcome.action	2	2	1	1	0	0
register.action	3	3	2	2	0	0
scriptaculous/ effects.js	143	1134	21	12	0	0
engine.js	62	908	21	25	5	0
tabs.js	9	92	5	8	0	3
Todo_lists.js	35	70	1	2	0	0
prototype.js	328	1961	34	34	0	0

The data in the Table indicate that the ‘scriptaculous.js’ and ‘prototype.js’ modules are the most complex ones (due to their size in LOC and # JS functions) and are characterized by the higher Fan-in values. Moreover, the ‘scriptaculous.js’ and ‘engine.js’ modules are the mostly coupled to the server modules (they make 5 Http requests), while prototype.js does not make any request to the server, or DOM changes. Eventually, the ‘util.js’ module is the one making the most changes to the DOM of the application. These metrics provide a useful starting point for making hypotheses about the quality of the modules involved in given executions of the applications, such as their maintainability or testability.

Of course, the data reported in the Table do not definitely characterize the size and complexity of the modules, but are just valid with respect to the considered execution of the application and they will change each time the application is exercised in a different way. However, when dealing with Ajax applications whose source code can be dynamically loaded at run-time, this one is the only feasible approach for obtaining the code of the application and analysing it. In this perspective, the DynaRIA tool provides a valid support to the activity of source code quality assessment, too.

7.5 Conclusions

Program comprehension tools based on dynamic analysis provide a formidable support for the analysis of software systems with an event-based and dynamic nature. Several works presented in the literature have shown the utility of these tools for the comprehension and analysis of Java, C++, desktop, Web-based applications, and so on, and [117] reports a comprehensive survey of papers tackling this topic. We believe that this type of tools will receive greater interest in the software engineering community thanks to the growing diffusion and request for Rich Internet Applications not only designed for PC platforms, but also for mobile devices, such as PDAs or smartphones. In This chapter we presented the DynaRIA tool that provides a user-friendly environment for analysing the dynamic behaviour of Rich Internet applications implemented in Ajax. The features of this tool have been designed to address the analysability issues that are typical of Ajax applications, such as their heterogeneous nature and the dynamically built configuration of the source code. In the chapter some case studies showed how this tool can be used to carry out program comprehension, testing, debugging and quality assessment activities. The considered activities, which are typical of RIA life-cycle processes, were accomplished with success thanks to the tool. However, further experiments are necessary for evaluating the actual cognitive support provided by the tool and for comparing it against other analysis tools by empirical studies [119]. These topics will be addressed by future work. In future work we also plan to improve the analysis and visualization features offered by the tool by means of techniques for detecting recurrent interaction patterns in the reconstructed sequence diagrams, and techniques for the horizontal and vertical compression of the diagrams. In addition, we aim to extend the analysis techniques of DynaRIA for abstracting architectural diagrams of an RIA by integrating static and dynamic data, and to improve its error detection capability by considering further types of run-time exceptions (such as errors due to the violation of specific invariant conditions).

Chapter 8 [9]

USING DYNAMIC ANALYSIS FOR GENERATING USER DOCUMENTATION FOR WEB 2.0 APPLICATIONS

The relevance of end user documentation for improving usability, learnability and operability of software applications is well known. However, software processes often devote little effort to the production of end user documentation due to budget and time constraints, or leave it not up-to-date as new versions of the application are produced. In particular, in the field of Web applications, due to their quick release time and the rapid evolution, end user documentation is often lacking, or it is incomplete and of poor quality. In this chapter a semi-automatic approach for user documentation generation of Web 2.0 applications is presented. The approach exploits dynamic analysis techniques for capturing the user visible behaviour of a web application and, hence, producing end user documentation compliant with known standards and guidelines for software user documentation. A suite of tools support the approach by providing facilities for collecting user session traces associated with use case scenarios offered by the Web application, for abstracting a Navigation Graph of the application, and for generating tutorials and procedure descriptions. The obtained documentation is provided in textual and hypertextual formats. In order to show the feasibility and usefulness of the approach, an example of generating the user documentation for an existing Web application is presented in the chapter.

[9] This chapter was published in the Proceedings of the 13th International Symposium on Web Systems Evolution (WSE 2011).

8.1 Introduction

According to the ISO/IEC 9126 Standard on Software Quality [126], software usability depends on several sub-characteristics of software, such as its learnability, i.e. the capability of the software product to enable the user to learn its application, and its operability, i.e. the product capability to enable the user to operate and control it.

The relevance of end user documentation for improving learnability and operability of software applications is well known [127]. Differently from technical software documentation that is intended to software developers, testers or maintainers and describes software from its internals, end user documentation shows how to use a software application and may include user guides, reference guides, help files, tutorials and walkthroughs which explain how to accomplish certain tasks. The IEEE Standard 1063 for Software User Documentation [128] describes minimum requirements for the structure, information content, and format of user documentation.

According to the IEEE 1063 Standard, user documentation should be complete and describe all the critical use cases offered by the application, as well as all the associated interaction scenarios. At the same time, documentation shall be accurate and reflect the functions and results of the applicable software version. Moreover, the standard recommends including explanations about all the known problems in using the software in sufficient detail such that the users can either recover from the problems themselves or clearly report the problem to technical support personnel. Moreover, reference mode documentation shall include each error message with an identification of the problem, probable cause, and corrective actions that the user should take. Eventually, the standard provides possible structures and format of user documentation.

It can be deduced that obtaining complete, accurate, and effective user documentation is not a trivial task. Real software development processes often devote little effort to the production of end user documentation due to budget and time constraints, or leave it not up-to-date as new versions of the application are produced. In particular, in the field of Web applications, due to their quick release time and the rapid evolution, end user

documentation is often lacking, or it is incomplete and of poor quality. Nowadays, due to the fast and growing diffusion of the Web 2.0, this problem is particularly true with Rich Internet Applications (RIAs). RIAs, indeed, with the enhanced dynamicity, responsiveness, and interactivity of their user interfaces are more and more similar to desktop applications, being able to offer more complex and richer functionalities [17]. As a consequence, richer and accurate end user documentation is absolutely needed for RIAs too. To save on time and costs for developing this documentation, end user documentation tools that provide facilities to automate some or all of the often laborious tasks associated with creating an application's documentation can be used. End user documentation tools include screen casting software and authoring tools. Screen casting tools [129] are used to record activities on the computer screen, mouse movement and are suitable for recording demonstrations, remote technical assistance, sales presentations, and training. Authoring tools are computer based systems that allow a general group (including non-programmers) to create (i.e., author) content for intelligent tutoring systems.

Unfortunately, these tools are just able to record the workflow needed for accomplishing given tasks (presenting the sequence of screens shown to users and the actions that must be performed by users on these screens) and to transform it into procedure descriptions by means of editing functions, but are not able to provide any other information about the overall application behaviour. As an example, with respect to rich Web applications, these tools have no facilities for generating site maps, navigational trees of the site, overview descriptions, or any other information that vice-versa may be obtained by dynamic analysis of the application. Reverse engineering techniques based on dynamic analysis of RIAs have been recently proposed in the literature for obtaining a Finite State Machine model of the user interface of the Web application [1, 2, 4]. These techniques have been exploited in the context of comprehension [6, 8] and testing processes [5, 7], but the FSM model they produce can be considered a suitable model for describing the navigation of the application too.

In this chapter, we propose of using the reverse engineering techniques proposed in [4]

with the aim of generating end-user documentation for RIAs. In particular, we present a documentation process of RIAs that and is based on a reverse engineering process and a tool that provide facilities for collecting user session traces associated with use case scenarios offered by the Web application, for abstracting a Navigation Graph of the application, and for generating tutorials and procedure descriptions. The obtained end user documentation is provided in textual and hypertextual formats and is compliant with some indications provided by the IEEE Standard for Software User Documentation [128].

8.2 Related Works and Tools for the software re-documentation

Software Documentation is a relevant part of a software product [130] but it is often neglected in software development processes. Usually, software engineers operate under the pressure of strict schedules and deadlines and do not devote much time to the production of documentation. In these conditions, tools for the automatic generation of technical documentation about the code, like JavaDoc [131] or Doxygen [132] that create online documents by extracting text from specially formatted comments can be used, or reverse engineering [133] techniques and tools can be exploited for post-generating technical documentation after the development.

End user documentation is usually even more overlooked, being usually produced manually or at least using some tool of user documentation generation based on screen-casting or authoring techniques. As an example, Zhang et al. [134] propose SmartTutor an environment for creating IDE-based interactive tutorials via editable replay. This tool is proposed to support programmers in the task of learning software IDEs and its features are similar to the ones offered by Jtutor [135], a tool designed to create and replay code-based tutorials in Eclipse. Other similar solutions are provided by DocWizards [136], a follow-me documentation wizard system in which the procedures are authored through demonstration as well as by manual editing, and by EpiDocx [137], a commercial tool for tutorial generation and maintenance of Windows-based applications.

However, all these tools just limit themselves to capture and record the procedures needed

for accomplishing user tasks, but they are not able to do any data mining from the observed application behaviour. As an example, they have no feature for automatic classification of shown user interfaces (or user events) based on their similarity and for associating them with a unique meaningful description, and have no feature for cross-referencing similar interfaces or events belonging to different user tasks.

As to the field of traditional Web applications, while several reverse engineering techniques and tools have been proposed for obtaining technical documentation about the applications [22, 138, 139, 140, 141], no specific reverse engineering solution has been defined for obtaining end-user documentation.

With respect to Web 2.0 applications, feasible solutions for obtaining Finite State Machine based models of user interactions with rich internet applications have been proposed in the literature [2, 56, 58]. At the moment, these models have been used for the aims of comprehending the behaviour of a RIA from the user point of view [1], crawling the application [24], or testing it [5, 72]. However, these models may provide a suitable starting point for obtaining end user documentation of the Web application too.

8.3 End User Documentation of Web 2.0 Applications

Sommerville states that end user documentation should be prepared for different classes of user and different levels of user expertise, and suggests five types of documents (or five chapters) with different audience and levels of detail to be delivered with the software system. These documents include: Functional descriptions of services provided, Installation document, Introductory manual for getting started with the system, Reference manual including details of all system facilities and System administrators guide [127].

Analogously, IEEE Standard 1063 for Software User Documentation [128] recommends including both instructional mode (to learn about software) and reference mode documentation (to refresh the user memory about it). Instructional mode documentation should include procedures structured according to user's task, while Reference mode documentation should be arranged to facilitate random access to individual units of

information. The Standard also suggests ways of organizing chapters and topics to facilitate learning.

According to these suggestions, we have decided to organize the end user documentation of a Rich Internet Application in three main parts*: 1) an Introductory manual for getting started with the system, that provides an overall description of user tasks; 2) a Tutorial showing detailed descriptions of single user tasks; 3) a Reference guide showing explicative materials about screens shown to users and user actions to be performed during task executions.*

In particular, the Introductory manual will be based on a Navigation Model showing how the application allows its user to access all its functions. The Tutorial is vice-versa composed of a set of more detailed views and descriptions about each user function execution. There will be also traceability relationships between the various parts of the documentation that will be implemented by means of hyper-textual files.

In the following we provide further details about the documentation items.

8.3.1 Introductory Manual

In traditional Web applications the navigation model is one of the most important aspects of the application to be communicated to application users. This model is usually given by a navigation graph with nodes representing Web pages and edges representing direct transitions between Web pages. A navigation graph of static Web applications can be obtained in a straightforward manner by means of spiders or link checkers, while obtaining this graph for dynamic Web applications requires more sophisticated approaches for dealing with the problem of page explosion and the request generation problem [141].

With respect to Rich Internet Applications that can be considered as a hybrid between a Web application and a desktop application [17], obtaining a navigational model is even more complicate. Indeed, the user interface of RIAs is not implemented by traditional Web pages having different URIs and interconnected by hyperlinks, but it is usually associated with a single Web page whose state changes depending both on events triggered on the user interface of the application and on various types of external or asynchronous event.

As a consequence, a suitable navigational model of the application is given by Finite State Machines (FSMs) that represent the various states of the user interface and the transitions between them.

In particular, in the proposed Navigation Graph we assume that user interface states with similar structure are represented as a single node, while edges between nodes represent transitions due to user events that caused the user interface state to change. Moreover, selected paths belonging to this graph will show possible execution scenarios of use cases offered by the RIA.

Figure 8.1 reports an excerpt of Navigation graph for an example Web application. The graph shows five nodes and nine edges associated with transitions between user interface states.

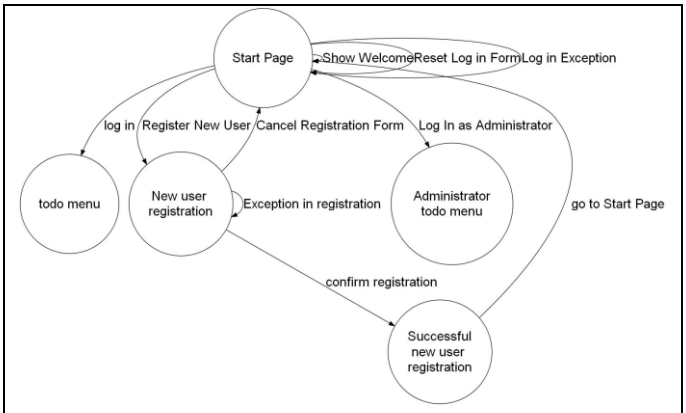

Figure 8.1: An example of Navigation Graph corresponding to the Login and Registration use cases of a Web Application

We propose the Introductory Manual of end user documentation of a RIA to include the Navigation Graph and an index of all the user functions offered to various classes of actors of the application. Moreover, each user function will be cross-referenced to the Navigation Graph paths that describe the corresponding user interactions with the application, representing both normal and exceptional execution scenarios.

To obtain this graph, the reverse engineering technique proposed in [4] will be used. The technique is based on dynamic analysis of the application and is supported by the CreRIA tool that exploits data collected from user sessions for abstracting this model. Further details about the technique are provided in section 8.4.

8.3.2 Tutorial Documentation and Reference Guide

To show the procedures needed for accomplishing user tasks, the proposed end user documentation will include a Tutorial section reporting operational descriptions of the

scenarios of each use case offered by the Web application. The descriptions will be grouped on the basis of the actors involved in the use cases. For each actor's use case, both textual descriptions of the scenarios and the Navigation Graph paths associated with them will be reported, as well as the sequence of screen shots shown by the application during the scenarios execution will be illustrated. Screen shots will have labels reporting the meaning of the corresponding state of the execution and there will be explanations of input values and user events that need to be fired on the corresponding user interface.

As to the Reference Guide, it will be composed of detailed descriptions of all user interactions and user interfaces encountered during the execution of user tasks.

8.4 The Documentation Generation Approach

The documentation generation process that we propose in this chapter exploits both user knowledge about the application, both information extracted and abstracted about the Web application by reverse engineering. The process will be based on three main steps: 1) Dynamic analysis of the Web application, 2) Navigation Graph Generation, 3) User Documentation Generation. The proposed process is supported by a reverse engineering tool and relies on a repository that stores both information obtained by reverse engineering and data annotations provided manually by the software engineer during the generation process. The process can be used to generate new user documentation incrementally, as well as to up-to-date existing documentation as the Web application evolves.

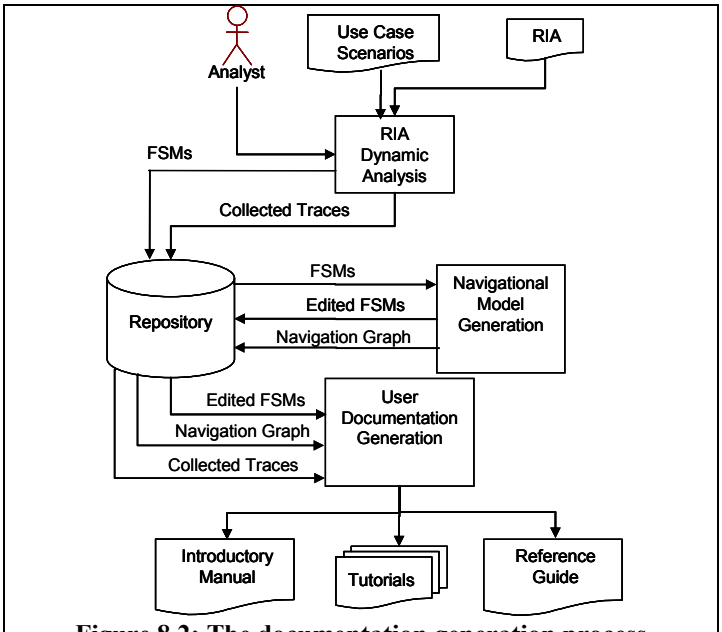

Figure 8.2: The documentation generation process

Figure 8.2 shows the proposed process, while the process steps are illustrated in the following.

8.4.1 Web application Dynamic Analysis

The purpose of this activity is to record user sessions devoted to exercising the single use cases offered by the Web application. The software engineer who is in charge of generating the end user documentation will carry out this activity having the attention of exercising all the use case scenarios of the application that will have to be included in the user manual. This activity must be performed to reach two aims: a) obtaining user session traces that will be transformed into walkthroughs showing how given user tasks can be accomplished, and b) abstracting an FSM describing the behaviour of the User Interface for each user session.

To record the execution traces and to obtain the corresponding FSMs, the RIA comprehension process proposed in [4] can be used, which assumes that an FSM can be obtained incrementally through the iterative steps of User Interaction, Extraction, Abstraction and Concept Assignment. This process is supported by the CReRIA reverse engineering tool and is illustrated in Figure 8.3.

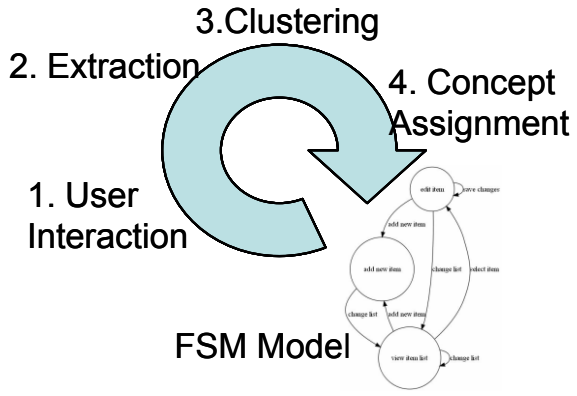

Figure 8.3: The iterative comprehension process of a RIA

The process starts with the *User Interaction* step where the user interacts with the RIA and fires an event on its current user interface: this interaction is performed in the controlled navigation environment offered by the CReRIA tool that observes and registers all the interactions and the needed information about them.

In the successive *Extraction step*, information about current interface, fired user event and

user interface obtained after the event processing must be retrieved and persistently stored. Hereafter, we call 'Interface Instance' each user interface captured during the dynamic analysis, and 'Transition Instance' each transition recorded during the interaction. Interface Instances and Transition Instances are captured and stored in the CReRIA tool repository.

The *Clustering step* is performed using some heuristic clustering criteria that evaluate the degree of similarity of the current user interface instance with the previously produced ones, as well as the similarity among occurred transitions. Each criterion creates a different clustering of similar interfaces and similar transitions. Hereafter we call 'Interface Class' each clustering of similar interface instances, and 'Transition Class' each clustering of similar transition instances.

The *Concept Assignment* is actually a comprehension [73] and validation step where the software engineer has to validate the clustering proposed by the heuristic criteria and accepts or refuses it. If an interface (transition) clustering is refused, he has to propose the correct clustering of the interface instance (transition). In this way, the expert incrementally reconstructs a FSM modelling the behaviour of the RIA GUI, since he either associates the current interface with a new interfaces class (and a new FSM state), or with an already existing interface class (and FSM state). Analogously, he associates the current transition either with a new class of transitions, or with an already existing one.

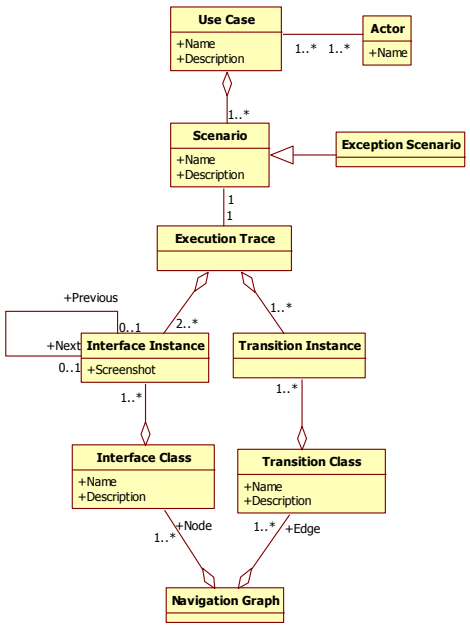

Figure 8.4: Information Model

Figure 8.4 shows the conceptual model of the information that is captured during the dynamic analysis step and that is used to build the Navigation Graph.

8.4.2 Generation of the Navigational Model

During this step, the single FSMs obtained by dynamic analysis are merged and transformed into the Navigational Model of the Web application that will be included in the Introductory Manual of the Web application. This step will be performed using the CreRIA tool: indeed, the software engineer will have to select the set of traces that he wants to include in the final user documentation and the tool will produce the overall Navigation Graph. In this graph, each node will represent a user interface class and will be labelled with the textual string provided to it in the Concept Assignment step. Analogously, each edge will be associated with a transition between states and will be labelled with the corresponding transition description.

The user will be able to edit this graph and associate each node or edge with additional textual annotations, or to correct the labels, if he considers them incorrect due to wrong past interpretations of Interfaces and Transitions.

At the end of this step, the navigation graph data will be stored in the tool repository.

8.4.3 End User Documentation Generation

This step will be devoted to the automatic generation of the end user documentation of the application and will be carried out by the CReRIA tool using the data stored in its repository. The documentation will include the Introductory Manual, the Tutorial Guide and a Reference Guide and will be provided both in textual and in hypertextual format.

The hypertextual documentation can be seen as an interactive site map and can be accessed both online (by publishing it on a Web server and linking it to the Web application) and offline, as a downloaded hypertext on a client machine. The hypertextual format provides links and shortcuts between the different pages in which the same items are cited. As an example, Interfaces and Transitions in the Navigation graph are clickable and the link reaches the page containing the detailed description of the Interface or of the Transition.

8.5 The CReRIA Tool

The CReRIA tool provides an integrated environment for dynamic analysis of Rich Internet Applications implemented with Ajax-based technologies whose main functionalities include:

- *incorporating* a Web browser (implemented with JavaXPCOM technology) for navigating the Rich Internet Application;
- *extracting* and *storing* in a Mysql database the relevant information about traced user sessions, such as user interfaces, events and transitions that occurred during the navigation;
- *capturing* and *storing* the screen shots of the navigated interfaces;
- *proposing* clustering of interfaces and transitions according to heuristic clustering criteria;
- *supporting* the Concept Assignment task on the basis of information collected or abstracted in the previous steps of the process;
- *supporting* the interactive navigation and editing of the Navigational model information, such as the collected scenario executions, screen shots and details of collected Interfaces and Transitions;
- g*enerating* user documentation in textual (rtf) or hypertextual (html, css, and JavaScript) format.

The original basic version of the CReRIA tool was born to exclusively support the FSM abstraction from RIAs and provided the former five functionalities of the above list as we have presented it in chapter 4.

A more recent version of the tool has been tailored for software re-documentation processes and implements the latter two functionalities too.

Using this tool, several versions of the user documentation can be generated by selecting different subsets of execution traces.

As an example, it is possible to generate the user documentation only for the subset of use cases scenarios related to a given actor, or to generate documentation for a new release of

the application by selecting only recently updated use cases and scenarios.

8.6 An Example

In this section an example of using the proposed document generation approach will be described with the purpose to show the feasibility of the process and to present some details about the RIA documentation that it produced.

The involved application is an Ajax-based open source Web application called Tudu [118], available at http://www.julien-dubois.com/tudu-lists and offering functionalities for the management of lists of tasks (the so-called 'todos') such as adding, deleting, searching for todos, organizing lists of todos, and so on.

Tudu provides an exemplar Rich Internet Application that has been frequently used for experimenting reverse engineering and testing techniques of RIAs [4, 5, 56, 72, 83]. Indeed, Tudu is a simple (but not trivial) RIA composed of about 10 KLOC, whose server side is implemented with Java/JSP technology, while its client side includes typical 'rich' pages that modify themselves at run-time on the basis of the user interaction with the pages.

As a consequence, for comprehending which are the user functionalities offered by Tudu and how they can be replayed, static analysis of its server pages does not suffice, while dynamic analysis must be carried out.

Moreover, the user documentation retrievable on the Tudu website is very poor.

Only a brief list of the 8 main use cases and 4 screen shots are reported in the Web page http://www.julien-dubois.com/tudu-lists/user-documentation.

Instead, on the basis of our past knowledge about Tudu, we know that it offers a wider set of use cases, including 23 use cases and 119 scenarios offered to three different actors, i.e. a Generic User "GU" (not yet logged in) (involved in 2 use cases), the Logged User "LU" (with 16 use cases) and the Administrator "A" (with 20 use cases, 16 of which are shared with the Logged User).

The overall list of use cases, involved actors and # of related scenarios is reported in Table 8.1.

In order to re-document the user functions provided by Tudu, we followed the process proposed in the chapter. In the first step of the process, one author performed Dynamic Analysis of Tudu and collected 119 Execution Traces exactly, corresponding to the known use case scenarios of the application.

These execution traces included 425 Interface Instances and 306 Transition Instances.

During the dynamic analysis, the Concept Assignment activity was performed and these instances were grouped into 42 Interface classes and 138 Transition classes.

Figure 8.5 reports the screen shown by the CReRIA tool during the execution of the clustering and concept assignment steps, where the clustering suggestions provided by tool can be accepted or refused by the software engineer.

Table 8.1: Use Cases of Tudu Lists

Use Case	Actor	Scenarios #
User Login	GU	5
Register a New User	GU	12
Quick Add of a Todo	LU & A	2
Advanced Add of a Todo	LU & A	11
Manage Completed Todos	LU & A	3
Filter Listed Todos	LU & A	6
Edit a Todo	LU & A	11
Delete a Todo	LU & A	2
Backup	LU & A	1
Restore	LU & A	4
Order Listed Todos	LU & A	3
Refresh	LU & A	1
Add New Todo List	LU & A	7
Open a Todo List	LU & A	2
Edit a Todo List	LU & A	5
Share a Todo List	LU & A	6
Delete a Todo List	LU & A	5
User Logout	LU & A	2
Show User Info	LU & A	16
User Monitoring	A	6
Configuration	A	4
Manage Users	A	5
Dump Database	A	1

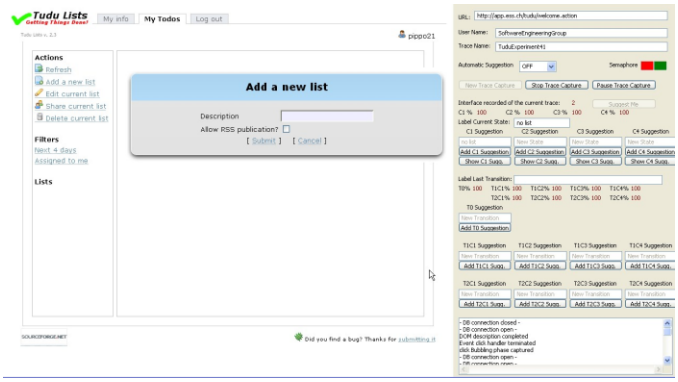

Figure 8.5: An Example of CReRIA GUI during the execution of the comprehension process

In this case, the clustering suggestions were accepted 420 out of 425 times for the Interface Instances, and 278 out of 306 times for the Transition Instances. The overall dynamic analysis activity was accomplished in about ten hours. At the end of this step, 119 FSM models associated with the execution traces were obtained and stored in the tool repository.

In the second process step, the overall Navigation Graph of Tudu was generated by merging the available FSMs. This task was performed using the features of CReRIA of selecting the set of execution traces, navigating them possibly refining the concepts assigned with the related FSMs states and transitions, and finally adding extra-information to be included in the final documentation. Figure 8.6 shows some snapshots of the CReRIA tool during the execution of the tasks of trace selection, concept refinement, and extra-information editing.

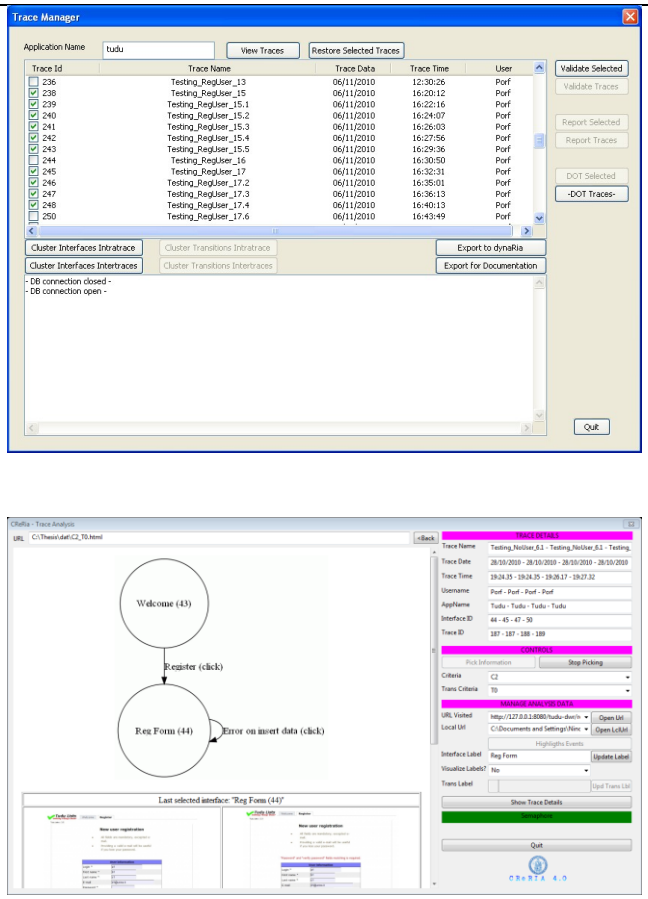

Figure 8.6: Two examples of CReRIA GUI showing Execution Traces selection (on the upper side) and concept refinement (on the lower side)

As the lower part of Figure 8.6 shows, to support the Concept refinement task the CReRIA tool allows a user to select a FSM node and to get a view about the original associated screens of the RIA, as well as to get further data captured during dynamic analysis (see the right panel in the Figure).

The Navigation Graph of Tudu obtained at the end of this second step is reported in Figure 8.7.

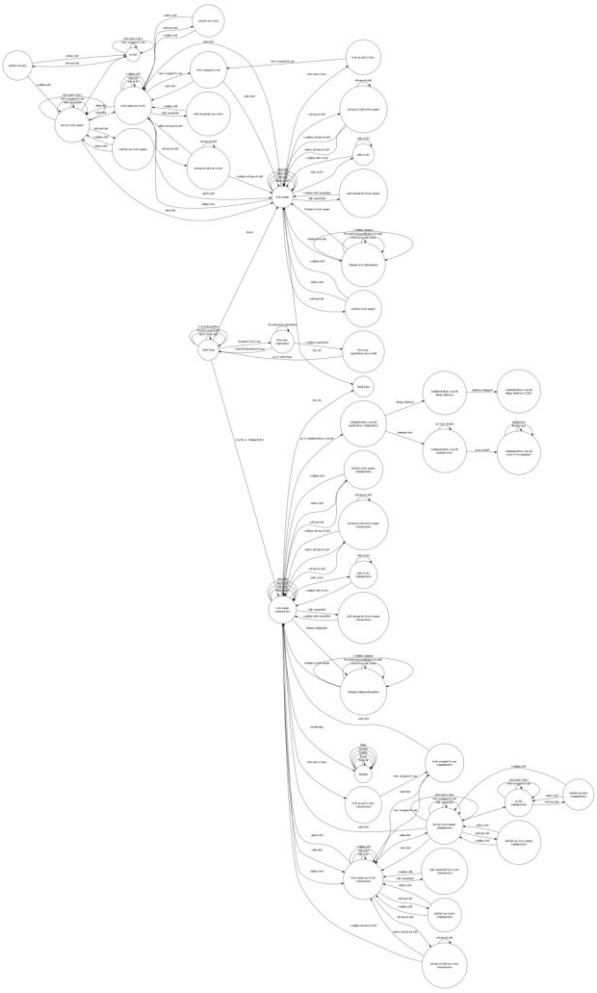

Figure 8.7: The overall Navigation Graph of Tudu

In the third step of the process, the final user documentation was obtained. In particular, the overall Navigation Graph of Tudu was generated both in textual format and in hypertextual one. In the hypertextual format, the Navigation graph was implemented by a HTML clickable map that allows the reader to click on graph nodes and edges and to jump to related pages describing additional details about each Interface and Transition.

As to the Tutorial Guide, it had to include separate descriptions of how each use case scenario can be executed. Each scenario description comprised:

- An explicative text of the use case scenario;
- the list of Actors involved in that scenario;

- the part of Navigation Graph comprehending only Interface nodes and Transitions involved in the scenario;
- the sequence of Interface screen shots and the description of the user events needed in order to replicate the scenario execution.

In the following, we show the part of Tutorial Guide documenting just a use case related to the insertion of a new todo. In particular, we considered the use case labelled "Advanced Add" that allows a todo to be inserted by specifying several parameters of it (such as todo description, priority, due date, assigned user, notes). The latter use case has 11 different scenarios, 7 of which correspond to correct insertions of a todo (with different valid combinations of input data) and 4 of which correspond to exceptional scenarios (due to incorrect input data). Two different views of the Navigation Graph for the "Advanced Add" use case are reported in Figure 8.8. The view on the left is a typical graph visualization, while in the right view graph nodes have been substituted by an interface instance of the Interface Class. Both the views can be included in the textual and in the hypertextual versions of the documentation (as a HTML map). The last one is a more intuitive view that can be used to have an immediate exemplification of the real appearance of the Interfaces shown by the application.

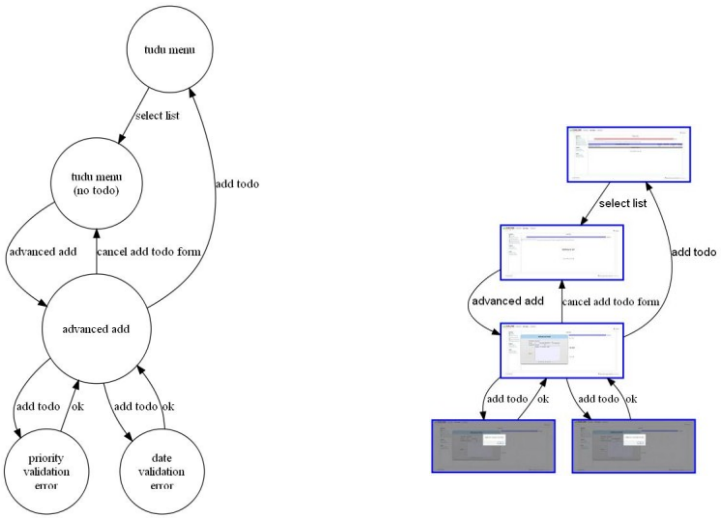

Figure 8.8: The Navigation Graph (in its classical view on the left, with thumbnails on the right) for the Advanced Add use case

Moreover, the Tutorial guide includes a walkthrough description of user tasks needed for accomplishing each "Advanced Add" scenario. Figure 8.9 shows the walkthrough generated for the first scenario of Advanced Add. This scenario explains the procedure for inserting a todo in a todo list by specifying its parameters, and reports a synthetic description of the use case scenario, the sequence of screen shots of the obtained Interface Instances (labelled with the names given to the corresponding Interface classes) and the description of the events causing the Transitions between the Interface Instances (with the values set in the input fields). In conclusion, the produced documentation was more complete and detailed with respect to the one obtainable by using the most part of user documentation generation tools described in the literature and commercially available, such as SmartTutor [134], DocWizards [136], JTutor [135], EpiDocX [137], that essentially produce only

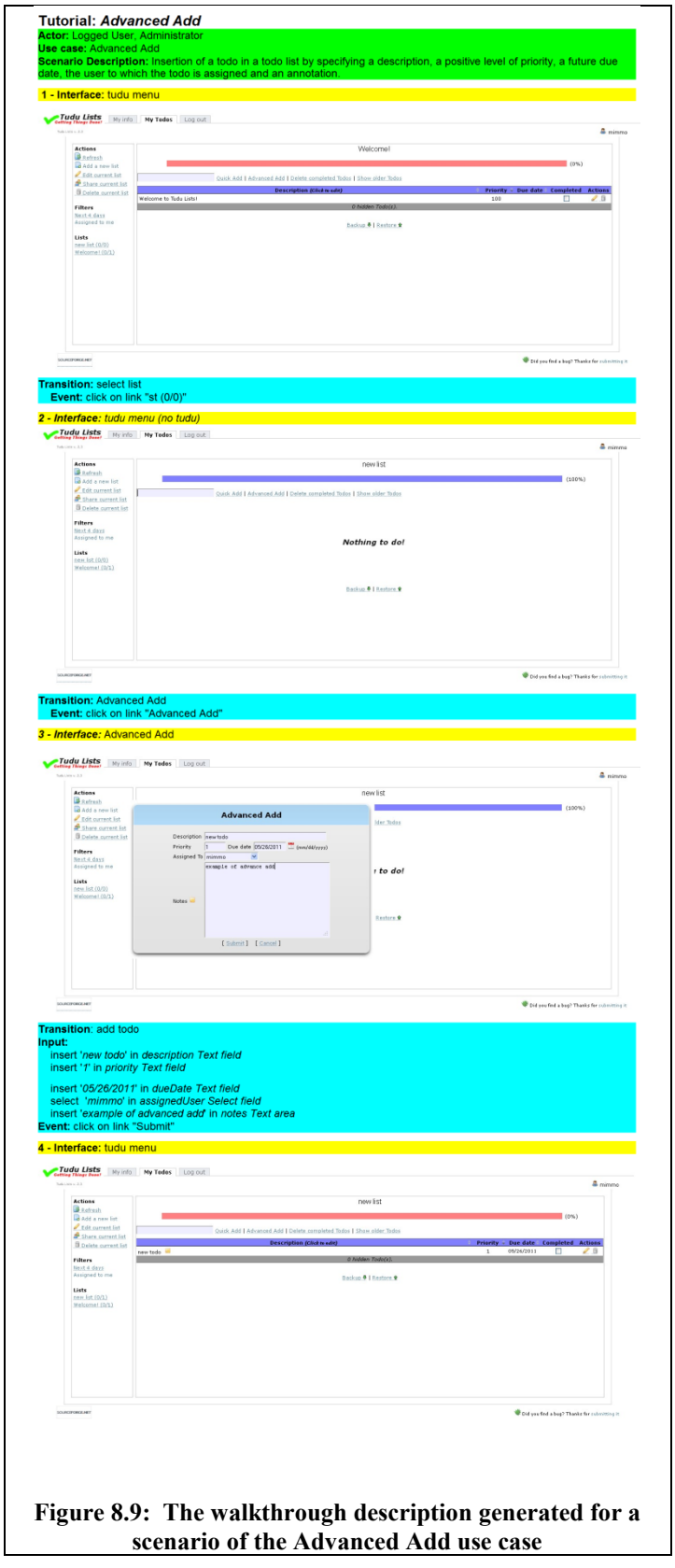

Figure 8.9: The walkthrough description generated for a scenario of the Advanced Add use case

walkthroughs of user tasks.

Of course, our approach was more expensive as to the time needed for accomplishing the dynamic analysis of the RIA. However, this extra-effort was valuable for obtaining more abstractions and details about the application usage, that could be transformed automatically into user documentation by our tool.

8.7 Conclusions

According to the survey presented by [142], software documentation generation processes should rely on technologies that improve automation of the documentation process, as well as facilitating documentation maintenance.

In this chapter we have proposed a technique and a tool for semi-automatic generation of end user documentation about Web 2.0 applications. The technique is innovative since it exploits reverse engineering processes and tools for generating the documentation, differently from most existing solutions supporting user documentation production. With respect to other competing tools, ours is able to generate a more flexible, complete and accurate documentation.

The chapter presented the features of the tool that we designed to support the proposed process and showed an example of using it for re-documenting an existing application implemented using Ajax technology. The resulting documentation provides both overview and more detailed descriptions of the user functions offered by the application, and was obtained effectively thanks to the tool support. In future work, we will extend the features of our documentation generation tool in order to allow the generation of other types of contents suggested by the IEEE Standard on User Documentation, such as documentation about error management.

Of course, in order to demonstrate the validity of the proposed approach, experiments are needed for showing both the effectiveness of the documentation process and of the obtained documentation. In future work we will carry out experiments aiming at comparing effectiveness and scalability of our approach against other ones, and at

obtaining a systematic evaluation of the actual contribution given to software usability by the produced documentation.

Chapter 9 [10]

A GUI CRAWLING-BASED TECHNIQUE FOR ANDROID MOBILE APPLICATION TESTING

As mobile applications become more complex, specific development tools and frameworks as well as cost-effective testing techniques and tools will be essential to assure the development of secure, high-quality mobile applications.

In this chapter we address the problem of automatic testing of mobile applications developed for the Google Android platform, and present a technique for rapid crash testing and regression testing of Android applications. The technique is based on a crawler that automatically builds a model of the application GUI and obtains test cases that can be automatically executed. The technique is supported by a tool for both crawling the application and generating the test cases. In the chapter we present an example of using the technique and the tool for testing a real small size Android application that preliminary shows the effectiveness and usability of the proposed testing approach.

9.1 Introduction

With about three billion people using mobile phones worldwide and the number of devices that can access the net climbing rapidly, the future of the Web is definitely mobile.

Bridging the gap between desktop computers and hand-held devices is the main challenge that research in mobile applications is addressing for the next future: according to Andy Rubin, Guru for Google's Android, *"There should be nothing that users can access on*

[10] This chapter was published in the Proceedings of the 4th International Conference Software Testing, Verification, and Validation Workshops (ICSTW 2011).

their desktop that they can't access on their cell phone".

Thanks to the advancement of hardware industry, modern mobile phones have now faster processors, growing memories, faster Internet connections, and much richer sensors, and are able to host more demanding applications. Moreover, the current applications programming platforms and development tools used to develop applications for mobile devices (such as Java ME, .NET Compact Framework, Flash Lite, Android) provide options to create highly functional mobile multimedia applications [152], allowing the use of various technologies, like Java, Open C, Objective C, Python, Flash Lite or Web technologies.

In such a scenario, the complexity, variety and functional richness of mobile applications are growing and the request for mobile software applications offering even more complex, rich, and usable functionalities is going to grow more and more in the next future.

Unfortunately, the quality of applications for mobile devices is often poor. This lack of quality is mostly due to very fast development processes where the testing activity is neglected or carried out in a superficial way since it is considered too complex, difficult to automate, expensive and time-consuming. Indeed, testing a mobile device application is not a trivial task due to several factors: a first factor consists of the variety of input that normally solicit a mobile application (such as user input, context and environment inputs) which makes it hard to find the right test cases that expose faults. A second factor is the heterogeneity of the technologies used by the devices, so that multiple tests on multiple platforms should be performed.

In order to obtain higher quality mobile applications, greater attention should be devoted to the testing activity throughout the development process and effective models, methods, techniques and tools for testing should be available for testers. In particular, cost-effective, rapid, and automated testing processes should be executed when possible, in order to cope with the fundamental necessity of the rapid delivery of these applications.

This chapter focuses on the problem of automatic testing of mobile applications developed for the Google Android platform. Among the currently available mobile platforms (such as

Symbian, Android, Research In Motion and Apple iOS), Android is predicted to become the second largest mobile Operating System by 2012 [153], thanks to the open-source nature and the programmability features: Android is indeed based on open source Linux software that allows developers to access to the underlying code. This feature will certainly increase Android diffusion in the market of mobile devices.

Android applications can be actually considered Event Driven Software (EDS) whose behaviour is driven by several types of events. Hence, a major issue in Android application testing is that of assessing which testing approaches usable for traditional EDS systems (such as GUIs, Rich Internet Applications, embedded software, etc.) are also applicable for Android based mobile applications and which tuning and technological adaptations are needed for them.

In particular, in the chapter we focus on GUI testing techniques already adopted for traditional applications and propose a GUI crawling based technique for crash testing and regression testing of Android applications. The technique is supported by a tool for producing test cases that can be automatically executed.

9.2 Related Works

As mobile applications become more complex, specific development tools and frameworks as well as software engineering processes will be essential to assure the development of secure, high-quality mobile applications. According to Wasserman [144], there are important areas for mobile software engineering research, and defining testing methods for product families, such as Android devices, is one of the areas requiring further efforts and investigations.

In the literature, recent works in testing mobile applications have mostly focused on the definition of frameworks, environments and tools supporting testing processes in specific development contexts. Other works have addressed specific issues of functional or non-functional requirements testing, like performance, reliability or security testing of mobile applications.

As an example, She at al. [149] have proposed a tool for testing J2ME mobile device applications that comprises a framework for writing tests using XML and a distributed run-time for executing tests automatically on the actual device, rather than on device emulators. Satoh [147, 148] presented a framework providing an application-level emulator for mobile computing devices that enables application-level software to be executed and tested with the services and resources provided through its current network.

As to the performance testing, Kim et al. [154] describe a method and a tool based on JUnit for performance testing at the unit level of mobile applications implemented in the J2ME environment.

As to the techniques for testing the correctness of a mobile application, Delamaro et al. [146] proposed a white-box testing technique that derives test cases using structural testing criteria based on the program Control-Flow-Graph. This technique is supported by a test environment that provides facilities for generating, running the tests and collecting the trace data of a test case execution from the mobile device.

More recently, a black-box testing technique for GUI Adaptive Random Testing has been presented in [155]. This technique considers two types of input events to a mobile application, namely user events fired on the application GUI, and environmental events produced by the mobile device equipments like GPS, bluetooth chips, network, etc. or by the other applications. Test cases are defined as event sequences composed by pools of randomly selected events. The technique has been experimented with six real-life applications running on Android 1.5 Mobile OS.

In the Android development platform, several tools, APIs and frameworks have been recently proposed for supporting application testing.

The Android Testing framework, besides native JUnit classes and API, includes an API that extends the JUnit API with an instrumentation framework and Android-specific testing classes. As an example, the extensions to the JUnit classes include Assertion classes (that contain specific assertions about Views and Regular Expressions), MockObject classes (that can be used to isolate tests from the rest of the system and to

facilitate dependency injection for testing), and specific TestCase classes that allow peculiar components of the Android application (such as Activity, Content Provider, and Intent) to be tested in an effective manner.

Among the available tools, Monkey [156] is a built-in application that can send random event sequences targeted at a specific application and can be used for stress testing. However, pure random testing, although simple and fully automatic, may not be effective for detecting a fault. The Monkey Runner tool [157] vice-versa provides an API for writing programs (written in Python) that control an Android device or emulator from outside of Android code. Monkey Runner can be used both for functional testing, where the tester provides input values with keystrokes or touch events, and view the results as screenshots, and for regression testing (Monkey Runner can test application stability by running an application and comparing its output screenshots to a set of screenshots that are known to be correct).

The Google Code site presents the Robotium framework [158] based on JUnit that can be used to write automatic black-box test cases for testing Android applications at function, system and acceptance level. Using Robotium, test case results can be checked by means of GUI assertions like in Web application testing using the Selenium framework.

9.3 Background

The Android Developers Web site [159] defines Android as a software stack for mobile devices that includes a Linux-based operating system, middleware and core applications. Using the tools and the APIs provided by the Android SDK, programmers can access the stack resources and develop their own applications on the Android platform using the Java programming language. Although based on well-known open source technologies like Linux and Java, Android applications own remarkable peculiar features that must be correctly taken into account when developing and testing them. In the following, we present an insight into Android application internals and focus on the technological approaches adopted for developing user interfaces and event handling in user oriented

applications.

9.3.1 Implementing the GUI of an Android Application

The Android operating system is often installed on smartphone devices that may have limited hardware resources (like CPU or memory) and a small-sized screen, but are usually equipped with a large number of sensors and communication devices such as a microphone, wi-fi and Bluetooth chips, GPS receiver, single or multi touch screen, inclination sensors, camera and so on. In order to optimize the management of all these resources and to cope with the intrinsic hardware limitations, the Android applications implement a multi-thread process model in which only a single thread can access to user interface resources, while other threads contemporarily run in background. Moreover, each application runs in its own virtual machine (the Dalvik one) that is a virtual machine optimized for Android mobile devices.

An Android application is composed of several types of Java components instantiated at run-time (namely, Activities, Services, Broadcast Receivers, and Content Providers) where the Activity components are crucial for developing the user interface of an application [159]. The Activity component, indeed, is responsible for presenting a visual user interface for each focused task the user can undertake. An application usually includes one or several Activity classes that extend the base Activity class provided by the Android development framework. The user interface shown by each activity on the screen is built using other framework classes such as View, ViewGroup, Widget, Menu, Dialog, etc.

In its lifecycle, an Activity instance passes through three main states, namely *running*, *paused* and *stopped*. At run-time just one activity instance at the time will in the *running* state and will have the complete and exclusive control of the screen of the device. An Activity instance can make dynamic calls to other activity instances, and this causes the calling activity to pass to the *paused* state. When a *running* activity becomes *paused* then it has lost focus but is still visible to the user. Moreover, an activity can enter the *stopped* state when it becomes completely obscured by another activity.

In Android applications, processing is event-driven and there are two types of events that

can be fired (e.g., user events, and events due to external input sources). The user events (such as Click, MouseOver, etc.) that can be fired on the user interface items (like Buttons, Menu, etc.) are handled by handlers whose definition belong either to the respective interface object, or to the related Activity class instance (using the Event Delegation design pattern). As to the events that are triggered by other input sources, such as GPS receiver, phone, network, etc., their handling is always delegated to an Activity class instance.

9.3.2 Open Issues with Android Application Testing

Since the behaviour of an Android application is actually event-driven, most of the approaches already available for EDS testing are still applicable to Android applications. However, it is necessary to assess how these techniques can be adopted to carry out cost-effective testing processes in the Android platform.

Most of the EDS testing techniques described in the literature are based on suitable models of the system or sub-system to be tested like Event-Flow Graphs, Event-Interaction-Graphs, or Finite State Machines [51, 56, 87], exploit the analysis of user session traces for deriving test cases [5], or are based on GUI rippers [44] or Web application crawlers [83] that automatically deduce possible sequences of events that can be translated into test cases.

Using such techniques for the aims of Android testing will firstly require an adaptation of the considered models and strategies in order to take into account the peculiar types of event and input source that are typical of Android devices.

As a consequence, new reverse engineering and GUI ripping techniques will have to be designed for obtaining the necessary models, as well as platforms and tools aiding user session analysis, will have to be developed.

From the point of view of the supporting technologies, the Android development environment [159] provides an integrated testing framework based on JUnit [160] to test the applications. At the moment, the framework has been mostly proposed to carry out assertion based unit testing and random testing of activities. A further issue consists of

assessing what support it is able to offer to the implementation of other automatic testing techniques too.

9.4 A Technique for Testing Android Applications

Like the crawler-based technique presented by [83] for testing Ajax applications, the automatic testing technique we propose for Android applications is based on a crawler that simulates real user events on the user interface and infers a GUI model automatically. The GUI model is hence used for deriving test cases that can be automatically executed for different aims, such as crash testing and regression testing. The model produced by the crawler is actually a GUI Tree, the nodes of which represent the user interfaces of the Android application, while edges describe event-based transitions between them.

For obtaining this model, while the crawler fires events on the application user interface, it also captures data about interfaces and events that will be also used to decide the further events to be fired. The data analysed by the crawler at run time belong to the conceptual model of an Android GUI that is represented by the class diagram shown in Figure 9.1.

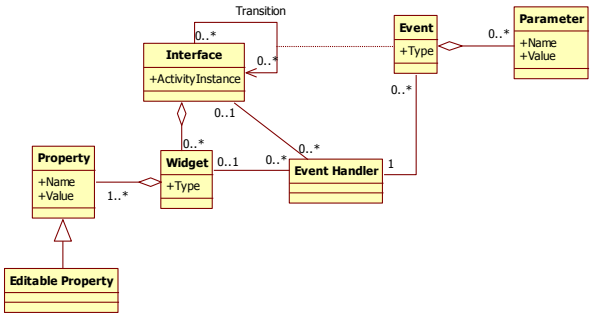

Figure 9.1: Conceptual Model of an Android Application GUI

The model shows that a GUI is made up of *interfaces* linked to each other by a *Transition* relationship. Each interface is characterized by the *Activity instance* that is responsible for drawing it and is composed by a set of Widgets. We define a *Widget* as a visual item of the Interface. A Widget can be implemented in the Android framework by an instance of a View class, a Dialog class or a Menu Item class.

Any Widget is characterized by a set of *Properties* with related *Values* (such as size, color,

position, caption and so on). Some Widget Properties are *Editable*: in this case their values are provided as user input at run time (as an example, we can consider the text field of a TextView object).

Events can cause transitions between Interfaces. In Android applications there can be both user events and events associated with interrupt messages sent from any component making up the device equipment (such as GPS, phone, wireless connections, inclination sensors, etc.).

Event Handlers code can be either defined in the context of a Widget of the interface, or in the context of an Activity, depending on the type of Event. Events may have zero or more Parameters and each Parameter has a Name and a Value.

The GUI crawler builds the GUI tree using an iterative algorithm that relies on two main temporary lists (Event list and Interface list, respectively) and executes the steps reported in Figure 9.2.

0) Describe the starting interface (associated with the first interface shown by the application at its launch) in terms of its activity instance, widgets, properties and event handlers, and store this description into the Interface list;

1) Detect all the interface fireable events having an explicitly defined Event Handler and, for each event, define a possible way of firing it by choosing the random values that will be set into the widget Editable Properties and to the Event Parameter Values (if they are present). Save this information into an Event description and store this description into the Event List[11].

repeat
2) Choose one fireable event *E* from the Event List, set the needed preconditions and fire it, according to its description.

3) Catch the current interface and add a node representing that interface to the GUI tree; then add an edge between the nodes associated with the consecutively visited interfaces.

4) Describe the current interface in terms of all its properties, store the interface description in the Interface List, and check whether the current interface is 'equivalent' to any previously visited one, or it is a 'new' one. If it is equivalent to any interface or it does not include fireable events, the corresponding GUI node will be a leaf of the tree, otherwise the new interface fireable events will be detected and a description of each event will be defined and added to the Event List. In both cases, the E Event that caused that interface to be reached will be removed from the Event List.

Until the fireable Event list is empty

Figure 9.2: The Crawling algorithm

[11] In the Event List, the description of each event will include the sequence of events that need to be fired before firing that event. This sequence actually represents the pre-conditions for firing the event.

A critical aspect of any GUI crawling algorithm consists of the criterion used for understanding when two interfaces are equivalent. Several approaches have been proposed in the literature to solve this problem [56, 83, 159]. Our algorithm assumes two interfaces to be equivalent if they have the same Activity Instance attribute (see the model in Figure 9.1) and they have the same set of Widgets, with the same Properties and the same Event Handlers.

Another critical aspect of this algorithm consists of the approach it uses for defining the values of widgets' properties and event parameters that must be set before firing a given event. At the moment, the crawler assigns them with random values.

9.4.1 Test Case Definition

The GUI tree generated by the crawler is the starting point for obtaining test cases that can be run both for automatic crash testing and for regression testing of the application.

According to Memon et al. [87], crash testing is a testing activity that aims at revealing application faults due to uncaught exceptions.

To detect crashes in the subject Android application, we have implemented a technique based on a preliminary instrumentation of the application code that automatically detects uncaught exceptions at run-time. In this way, during the GUI exploration performed by the crawler we are able to perform a first crash testing. Indeed, test cases used for crash testing are given by the sequences of events associated with GUI tree paths that link the root node to the leaves of the tree.

As to the regression testing activity that must be executed after changes to a given application have been made, it is usually performed by rerunning previously run tests and checking whether program behaviour has changed and whether new faults have emerged.

In the regression testing of an Android application, we propose to use the same test cases used for crash testing, and we had to define a suitable solution to check possible differences between the application behaviours.

A possible way of detecting differences is by comparing the sequences of user interfaces obtained in both the test runs. The interface comparison can be made using test oracles

having different degrees of detail or granularity [42]. As an example, the Monkey Runner tool [157] executes regression testing of Android applications but it checks results just by comparing the output screenshots to a set of screenshots that are known to be correct.

We propose to check whether all the intermediate and final Interfaces obtained during test case rerunning coincide with the ones obtained in the previous test execution, and their Activity, Event Handlers, and Widgets' Properties and Values are the same. To do this checking, we add specific assertions to the original test cases that will be verified when tests are run.

A test will reveal a failure if any assertion is not verified, or some event triggering is not applicable

9.5 The Testing Tool

In this section a tool supporting the testing technique proposed in the previous section will be presented.

The tool, named A^2T^2 (Android Automatic Testing Tool), has been developed in Java and is composed of three main components: a Java code instrumentation component, the GUI Crawler and the Test Case Generator.

The Java code instrumentation component is responsible for instrumenting the Java code automatically, in order to allow Java crashes to be detected at run-time.

The GUI crawler component is responsible for executing the Android crawling process proposed in section 9.3. It produces a repository describing the obtained GUI Tree, comprehending the description of the found Interfaces and of the triggered Events. Moreover, it produces a report of the experienced crashes, with the event sequences producing them.

The GUI crawler exploits Robotium [158], a framework originally designed for supporting testing of Android applications. Robotium provides facilities for the analysis of the components of a running Android application.

The GUI crawler extracts information about the running Activity, the Event Handlers that

the Activity implements and the Widgets that it contains (with related Properties, Values and Event Handlers). Moreover, the GUI crawler is able to emulate the triggering of Events and to intercept application crashes.

The current prototype of A^2T^2 manages only a subset of the possible Widgets of an Android application, comprehending, TextView labels, TextEdit fields, Buttons and Dialogs, while, in the future, we plan to extend the support to a larger number of Widgets and Event typologies.

The Test Case Generator component is responsible for the abstraction of executable test cases supporting crash testing and regression testing from the GUI Tree produced by the GUI Crawler component.

Test cases produced by the Test Case Generator are Java test methods that are able to replay event sequences, to verify the presence of crashes (for crash testing) and to verify assertions regarding the equivalence between the Interfaces obtained during the replay and the original ones obtained in the exploration process (for regression testing).

Generated Test Cases exploit the functionalities offered by the Robotium framework both for events triggering and for the extraction of information about the obtained Interfaces.

Both the Crawler and the Generated Test Cases can be executed in the context of the Android Emulator provided by the Android SDK [161].

9.6 An Example

In this section we show an example of using the proposed technique and tool for testing a simple Android application. The subject application implements a simple mathematic calculator that can operate either in a basic mode, providing the possibility of executing the basic arithmetic operations between numeric input values, or in a scientific mode, providing trigonometric functions, inverse trigonometric functions and other ones.

The application was developed for the Android 2.2 platform by using the libraries provided by the corresponding SDK. It consists of five Java classes contained in one package, for a total of 557 Java LOCs. Two of the implemented classes extend the

Android Activity class and contain, in total, 36 different Widgets, comprehending Buttons, EditText and TextView Widgets.

After a preliminary automatic instrumentation of the application - that was needed for detecting runtime crashes - the application crawling was automatically executed by the tool and a GUI tree of the application was obtained. During crawling, 19 Events were triggered, 19 Interfaces were obtained, and an exception causing an application crash occurred. Using the equivalence criterion presented in section 3.1, the 19 Interfaces we obtained were grouped into the following three equivalence classes:

- Class IC1 that comprehends the Interfaces I1, I2, I3, I4, I5, I9, I16 corresponding to instances of the BaseCalculator Activity, by means of which the basic arithmetic operations can be performed (an example of an Interface belonging to IC1 is reported in Figure 9.3-a);
- Class IC2, comprehending the Interfaces I6, I7, I8, I10, I11, I12 and I19, corresponding to instances of the ScientificCalculator Activity, by means of which the trigonometric functions, the reciprocal function and the square root function can be computed (an example of Interface belonging to IC2 is reported in Figure 9.3-b);
- Class IC3, comprehending the Interfaces I13, I14, 15, I17 and I18, corresponding to instances of the ScientificCalculator Activity by means of which inverse trigonometric functions, the reciprocal function and the square function can be computed (an example of Interface belonging to IC3 is reported in Figure 9.3-c).

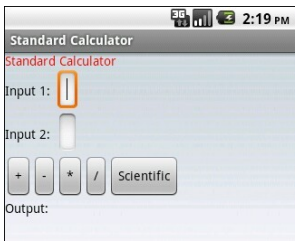

Figure 9.3-a: IC1 Interface

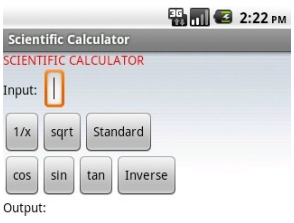

Figure 9.3-b: IC2 Interface

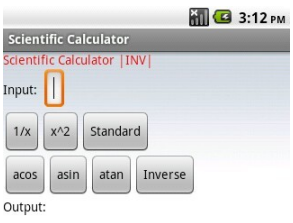

Figure 9.3-c: IC3 Interface

Figure 9.3: Screenshots of Interfaces of the example Android application

Figure 9.4 shows the GUI Tree we obtained, where each node reports the screenshot and the label associated to the corresponding interface, and edges are labeled by the event that caused the transition between the interfaces. The leaves of the tree correspond always to interfaces that were equivalent to at least another interface previously explored by the crawler (the number in the Interface label represents, too, the order in which the Interface was found by the crawler).

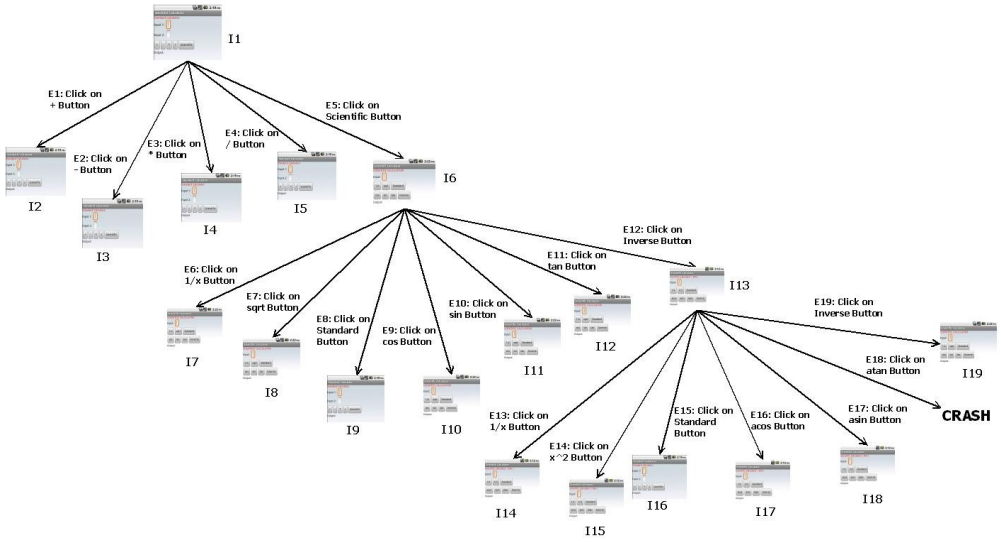

Figure 9.4: The GUI Tree obtained by crawling the example Android application

As an example, our crawling technique was able to distinguish automatically the instances of Interfaces belonging to IC1 from interfaces of the other groups because they were associated with instances of different Activity classes. Moreover, it was able to distinguish between instances of Interfaces belonging to IC2 and IC3, because they included different sets of Buttons.

While exploring the GUI interfaces via the crawler some crashes of the application were discovered, too. As an example, a crash occurred after firing the E18 Event that corresponds to the click on the 'atan' Button on the Interface I13.

The cause of this crash was the lack of the try/catch code block for handling the exception due to the input of a non-numeric value in the Input TextEdit widget. This caused a java.lang.NumberFormatException when the application tries to convert the string in the input field into a Double value before computing the arctangent function. After correcting

this defect, we run the crawler again and obtained a new GUI tree where another instance of Interface (belonging to IC3 group) was correctly associated with the right node.

After obtained the GUI Tree, the Test Case Generator produced 17 test cases for crash testing that corresponded to the 17 different paths from the root to the leaves of the tree. The Test Case Generator tool developed 17 test cases for regression testing, too.

In order to assess the effectiveness of our test cases for the aims of regression testing, we injected two faults in the Android application and run the 17 regression test cases to find these faults.

The first injected fault was due to a change of the code of the Scientific Calculator Activity causing an interface Button (namely the one that makes it possible to return to the Base Calculator) to be no more drawn on the screen window.

One of the regression test cases (namely the test case corresponding to the execution of the event sequence E5- E12-E13) revealed an assertion violation. The assertion violation was due to a layout difference between the obtained Interface I13 and the corresponding one collected during the previous crawling process, since the new Interface did not contain the Button that was included in the original one.

Figure 9.5 shows the Java code of the test case corresponding to the execution of the event sequence E5-E12-E13 that detected the fault.

```
public void testSequence11() throws Exception {
        InterfaceComparator.compare("I1");
        solo.clickOnButton("Scientific");
        InterfaceComparator.compare("I6");
        solo.clickOnButton("Inverse");
        InterfaceComparator.compare("I13");
        solo.enterText("Input", "dfghfdjg");
        solo.clickOnButton("1/x");
        InterfaceComparator.compare("I14");


}
```

Figure 9.5: Java code of the test case firing the E5-E12-E13 event sequence

In Figure 9.5, `'solo'` is one of the classes that Robotium provides for automatically firing events onto the application, while `'InterfaceComparator'` is a class that we developed, having a method `'compare'` that is used to check the coincidence between interfaces.

The second fault we injected consisted of associating an incorrect event handler to the click event on the cosine Button (e.g., the `'calculateSin'` function) instead of the correct one (e.g., the `'calculateCos'` function). This fault is explained by the code fragment shown in Figure 9.6, where in the last line of code, `calculateCos` should be written instead of `calculateSin`.

```
View.OnClickListener calculateSin = new View.OnClickListener() {
  public void onClick(View v) { … }
};

View.OnClickListener calculateCos = new View.OnClickListener() {
  public void onClick(View v) { … }
};

sinButton.setOnClickListener(calculateSin);
cosButton.setOnClickListener(calculateSin);
```

Figure 9.6: Code fragment associated with an injected fault

The execution of the test case corresponding to the event sequence E5-E10 revealed an assertion violation and allowed the injected fault to be discovered. The violation was due to the difference between the obtained Interface and the one collected during the crawling process, since they contained different methods associated to the onClickListener attribute of cosButton widget.

We explicitly remark that, thanks to the type of assertion checked by our regression test cases, we were able to find a fault whose effects were not visible on the GUI. Other regression testing tools like Monkey Runner could not discover it, since it just limits itself to check screenshots.

However, the fault detection effectiveness of the technique depends considerably on the strategy used by the crawler for defining the input values needed for firing the events. As an example, a possible fault in the reciprocal function due to an unmanaged exception of a division by zero might be revealed only by a test case with a zero value in the input field. This value may not be used in any test case, due to the random strategy used by the crawler for generating input. Other input generation techniques should be considered in order to solve this problem.

Moreover, in the example we assumed that the replay of the same event sequence with the

same input values produced always the same effects. In general, instead, the problems related to the management of preconditions and postconditions related to persistent data sources (such as files, databases, Shared Preferences objects, remote data sources) must be considered, too.

In conclusion, this example showed the usability of the technique for running crash testing and regression testing, and its effectiveness in detecting some types of fault in a completely automatic manner.

9.7 Conclusions

In this chapter we propose a technique for automatic testing of Android mobile applications. The technique is inspired to other EDS testing techniques proposed in the literature and relies on a GUI crawler that is used to obtain test cases that reveal application faults like run-time crashes, or that can be used in regression testing. Test cases consist of event sequences that can be fired on the application user interface.

At the moment, we have not considered other types of events that may solicit a mobile application (such as external events produced by hardware sensors, chips, network, or other applications running on the same mobile device) and just focused on user events produced through the GUI. In future work, we intend to propose a strategy for considering other types of events, too, in the test case definition process.

The proposed testing technique aims at finding runtime crashes or user-visible faults on modified versions of the application.

In the chapter we just discussed an example of using the technique for testing a small size Android application, and showed the usability and effectiveness of the technique and supporting tool.

In future work, we plan to carry out an empirical validation of the technique by experiments involving several real world applications with larger size and complexity, with the aim of assessing its cost-effectiveness and scalability in a real testing context.

Moreover, in order to increase the effectiveness of the obtained test suites we intend to investigate further and more accurate techniques for the crawler to generate several kinds

of input values, including both random and specific input values depending on the considered type of widget. In addition, solutions for managing test case preconditions and postconditions related to persistent data sources (such as files, databases, Shared Preferences objects, remote data sources) will be looked for.

10. Conclusions

The Web is steadily increasing its reach beyond the desktop to devices ranging from mobile phones to domestic appliances. This rapidly expanding accessibility is largely due to the Web's foundation in open protocols and markup languages, which offer the most widely implemented global infrastructure for content and application access [162].

New technologies, frameworks, tools and environments that allow to develop Web and mobile applications with the least effort and in very short time have been introduced in the last years. As consequence both client software for the Web and mobile applications have quickly evolved.

This evolution has been accompanied by some drawbacks that are mostly due to the lack of applying well-known software engineering practices and approaches. As a consequence, new research questions and challenges have emerged in the field of web and mobile applications maintenance and testing. Some of these questions have been addressed in the research activity described in this thesis.

The thesis describes the main results of this activity. In particular, in the first chapters we presented the results of a reverse engineering research that aimed at defining and validating effective reverse engineering processes and techniques for reconstructing suitable representation models of Rich Internet Applications.

Our research preliminarily addressed the problem of modelling the client-side behaviour of a RIA and we choose Finite State Machines to represent the event-driven nature of this behaviour. FSMs are one of the most popular models used in literature for representing the behaviour of a software system, providing an abstract view of a system in terms of states

and transitions among them.

We have presented a reverse engineering process for obtaining the FSM model of a RIA. The process is based on the dynamic analysis of the application. Dynamic analysis is a mandatory technique for reconstructing all possible behaviours exhibited by event-driven applications, but it exposes to several problems in particular to the potential explosion both of states and transitions of the FSM. To solve the explosion problem we have proposed clustering techniques that exploit equivalence criteria for recognizing and classifying equivalent states and transitions.

Afterwards we have presented the results of a validation experiment involving real Web applications that showed the cost-effectiveness of the proposed reverse engineering technique. Moreover the experiment showed how the equivalence criteria are able to influence the effectiveness of the technique, as well as its cost-effectiveness.

Successively, we have presented an 'Agile' process for the reverse engineering of Rich Internet Applications that is iterative, based on the continuous feedback of the process executor, and allows the incremental reconstruction of a Finite State Machine modelling the behaviour of RIA. The approach has been experimented with some case studies, which involved the dynamic analysis of existing RIAs. The results of the experiment showed the effectiveness of the approach and how it simplifies the concept assignment tasks needed for software comprehension that was the weakness point of the first proposed Reverse Engineering process.

As for the Rich Internet Applications testing we have proposed a technique that transforms execution traces of an existing application into executable test cases. To achieve the technique scalability, a test suite selection technique is employed that reduces the size of obtained test suites. For exploring the feasibility and effectiveness of this technique, we carried out an experiment involving an open-source RIA application, where different approaches (both human-based, and automatic) for execution trace collection and several criteria for reducing the test suites were analysed.

In the field of RIA testing we have proposed a classification framework that distinguishes

different RIA testing techniques, including techniques for finding faults having no effects on the RIA user interface and techniques that are suitable for finding faults with user visible effects. The classification is made on the basis of the following categories: testing goal, technique used for generating test cases, testing oracle and types of tool supporting the testing process. Possible solutions to the problems of generating test cases, defining testing oracles, and automatically evaluating the results of test case executions have been analysed for both types of testing techniques.

Another topic we addressed in the thesis is the comprehension of existing RIAs developed in Ajax. About this topic we presented the DynaRIA tool that provides a user-friendly environment for analysing the dynamic behaviour of Rich Internet applications implemented in Ajax. The features of this tool have been designed to address the analysability issues that are typical of Ajax applications, such as their heterogeneous nature and the dynamically built configuration of the source code.

We performed some case studies that showed how this tool can be used to carry out program comprehension, testing, debugging and quality assessment activities. In the case studied the considered activities, which are typical of RIA life-cycle processes, were accomplished with success thanks to the tool.

Finally, we've addressed the problem of re-documenting existing RIAs. In this thesis we have proposed a technique and a tool for semi-automatic generation of end user documentation about Web 2.0 applications. The technique is innovative since it exploits reverse engineering processes and tools for generating the documentation, differently from most existing solutions supporting user documentation production. With respect to other competing tools, ours is able to generate a more flexible, complete and accurate documentation.

In the final part of the thesis, we have described the preliminary results of a research activity we performed in the field Android application testing. Likewise RIAs, these mobile applications have characteristics of event-driven software systems, so we decided to transfer the research finding we obtained in the field of RIA testing to this specific

sector.

In regard to this argument, in this thesis we have described a technique for automatic testing of Android mobile applications. The technique is inspired to other EDS testing techniques proposed in the literature and relies on a GUI crawler that is used to obtain test cases that reveal application faults like run-time crashes, or that can be used in regression testing. Test cases consist of event sequences that can be fired on the application user interface. Moreover we have just discussed an example of using the technique for testing a small size Android application, and showed the usability and effectiveness of the technique and of the supporting tool we developed.

All the techniques we've presented in the thesis are supported by prototype tools that aim at automating the proposed processes, such as ReRIA and CReRIA, DynaRIA, TestRIA, CrawlRIA and A^2T^2. Thanks to these tools, we were able to carry out experiments for validating the proposed techniques and for deducing further research topics and questions that will need to be investigated in future works.

References

[1] Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; , "Reverse Engineering Finite State Machines from Rich Internet Applications," Reverse Engineering, 2008. WCRE '08. 15th Working Conference on , vol., no., pp.69-73, 15-18 Oct. 2008

[2] Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; , "Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications," Software Maintenance, 2009. ICSM 2009. IEEE International Conference on , vol., no., pp.571-574, 20-26 Sept. 2009

[3] Amalfitano, D.; Fasolino, A.R.; Tramontana, P.;, "A Tool-supported Process for Reliable Classification of Web Pages," International Conference on Advanced Software Engineering & Its Applications (ASEA 2009), Volume 59, 338-345, Springer.

[4] Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; , "An Iterative Approach for the Reverse Engineering of Rich Internet Application User Interfaces," Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on, pp.401-410, 9-15 May 2010.

[5] Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; , "Rich Internet Application Testing Using Execution Trace Data," Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on , vol., no., pp.274-283, 6-10 April 2010

[6] Amalfitano, D.; Fasolino, A.R.; Polcaro, A.; Tramontana, P.; , "DynaRIA: A Tool for Ajax Web Application Comprehension," Program Comprehension (ICPC), 2010 IEEE 18th International Conference on , vol., no., pp.46-47, June 30 2010-July 2 2010

[7] Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; , "Techniques and tools for Rich Internet Applications testing," Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on , vol., no., pp.63-72, 17-18 Sept. 2010

[8] Amalfitano, D.; Fasolino, A.R.; Polcaro, A.; Tramontana, P.; , "Comprehending Ajax Web Applications by the DynaRIA Tool," Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the , vol., no., pp.122-131, Sept. 29 2010-Oct. 2 2010

[9] Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; , "A GUI Crawling-Based Technique for Android Mobile Application Testing," Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on , vol., no., pp.252-261, 21-25 March 2011

[10] Amalfitano, Domenico; Fasolino, Anna Rita; Tramontana, Porfirio; , "Using dynamic analysis for generating end user documentation for Web 2.0 applications," Web Systems Evolution (WSE), 2011 13th IEEE International Symposium on , vol., no., pp.11-20, 30-30 Sept. 2011

[11] Murugesan, S.; , "Understanding Web 2.0," IT Professional , vol.9, no.4, pp.34-41, July-Aug. 2007

[12] Leon Shklar, Rich Rosen, "Web Application Architecture: Principles, Protocols and Practices, 2nd Edition", Wiley

[13] Allaire, J; , "Macromedia Flash MX—A next-generation rich client", Macromedia White Paper (March, 2002)

[14] Fraternali, Piero; Rossi, Gustavo; Sánchez-Figueroa, Fernando; , "Rich Internet Applications," Internet Computing, IEEE , vol.14, no.3, pp.9-12, May-June 2010

[15] Meliá, S.; Gómez, J.; Pérez, S.; Díaz, O.; , "Architectural and Technological Variability in Rich Internet Applications," Internet Computing, IEEE , vol.14, no.3, pp.24-32, May-June 2010

[16] Atkins, D.L.; Ball, T.; Bruns, G.; Cox, K.; , "Mawl: a domain-specific language for form-based services," Software Engineering, IEEE Transactions on , vol.25, no.3, pp.334-346, May/Jun 1999

[17] J. Garrett, "AJAX: A new approach to Web applications", Adaptive Path, 2005

[18] Paulson, L.D.; , "Building rich web applications with Ajax," Computer , vol.38, no.10, pp. 14-17, Oct. 2005

[19] Preciado, J.C.; Linaje, M.; Sanchez, F.; Comai, S.; , "Necessity of methodologies to model rich Internet applications," Web Site Evolution, 2005. (WSE 2005). Seventh IEEE International Symposium on , vol., no., pp. 7- 13, 26 Sept. 2005

[20] Preciado, J.C.; Linaje, M.; Comai, S.; Sanchez-Figueroa, F.; , "Designing Rich Internet Applications with Web Engineering Methodologies," Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on , vol., no., pp.23-30, 5-6 Oct. 2007

[21] G.A. Di Lucca, A.R. Fasolino, "Testing Web-Based Applications: the State of the Art and Future Trends", Information and Software Technology Journal, Vol. 48, Issue 12, Pages: 1172-1186 (December 2006), Elsevier inc.

[22] G.A. Di Lucca, A.R. Fasolino, P. Tramontana, "Reverse Engineering Web Application: the WARE approach", Journal of Software Maintenance and Evolution: Research and Practice, Volume 16, Issue 1-2, John Wiley & Sons, Ltd, Chichester, West Sussex, UK. Date: January - April 2004, Pages: 71-101

[23] T. Isakowitz, E. A. Stohr, and P. Balasubramanian, "RMM: a methodology for structured hypermedia design", Communications of the ACM, August 1995

[24] Isakowitz, T.; Kamis, A.; Koufaris, M.; , "Extending the capabilities of RMM: Russian dolls and hypertext," System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on , vol.6, no., pp.177-186 vol.6, 7-10 Jan 1997

[25] Garzotto F., Paolini P. and Schwabe, D., "HDM: a model-based approach to hypertext application design", ACM Transactions on Information Systems, ACM Press, 1993, vol. 11 is. 1 pp. 1 - 26

[26] G. Rossi, D. Schwabe, F. Lyardet, "Web application models are more than conceptual models", Proceedings of the First International Workshop on Conceptual Modeling and the WWW, Paris, France, November 1999

[27] C. Gnaho and F. Larcher, "A user centered methodology for complex and customizable web applications engineering", 1st ICSE Workshop on Web Engineering, Los Angeles, May 1999

[28] D. Jones and T. Lynch, "A model for the design of web-based systems that supports adoption, appropriation and evolution", 1st ICSE Workshop on Web Engineering, Los Angeles, May 1999

[29] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginlge, "Web engineering: A new discipline for development of web-based systems", 1st ICSE Workshop on Web Engineering, Los Angeles, May 1999

[30] M. H. Cloyd, "Designing user-centered web applications in web time", IEEE Software, IEEE Computer Society Press, 18:62–69, Jan/Feb 2001

[31] Ceri, S., Fraternali, P. and Bongio, A., "Web Modeling Language (WebML): a Modeling Language for Designing Web Sites", 9th International WWW Conference, Amsterdam, 2000, pp. 137 - 157

[32] Ceri S., Fraternali P., Bongio A., Brambilla M., Comai S., and Matera M., "Designing Data-Intensive Web Applications", Morgan Kauffmann, 2002

[33] Conallen, J. 1999. "Modeling Web application architectures with UML" Commun. ACM 42, 10 (Oct. 1999), 63-70

[34] Conallen, J. 1999. "Building Web Applications with UML", Addison-Wesley Publishing Company, Reading, MA

[35] Koch N., Kraus A., Cachero C. and Meliá S., "Integration of Business Processes in Web Application Models", Journal of Web Engineering, Rinton Press, vol. 3 is.1 pp. 22 - 49

[36] Melia, S.; Gomez, J.; Perez, S.; Diaz, O.; , "A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA," Web Engineering, 2008. ICWE '08. Eighth International Conference on , vol., no., pp.13-23, 14-18 July 2008

[37] Francisco Valverde and Oscar Pastor. 2009. Facing the Technological Challenges of Web 2.0: A RIA Model-Driven Engineering Approach. In Proceedings of the 10th International Conference on Web Information Systems Engineering (WISE '09), Gottfried Vossen, Darrell D. Long, and Jeffrey Xu Yu (Eds.). Springer-Verlag, Berlin, Heidelberg, 131-144.

[38] Fons, J., Pelechano, V., Albert, M., Pastor, O.: "Development of Web Applications from Web Enhanced Conceptual Schemas", ER 2003, Vol. 2813. LNCS. Springer (2003) 232-245

[39] Pastor, O., Molina, J.C.: "Model-Driven Architecture in Practice. A Software Production Environment Based on Conceptual Modelling", Springer-Verlag, Berlin Heildeberg (2007)

[40] Mesbah, A.; van Deursen, A.; , "An Architectural Style for Ajax," Software Architecture,

2007. WICSA '07. The Working IEEE/IFIP Conference on , vol., no., pp.9, 6-9 Jan. 2007

[41] Ali Mesbah and Arie van Deursen. 2008. A component- and push-based architectural style for ajax applications. J. Syst. Softw. 81, 12 (December 2008), 2194-2209

[42] Xie, Q. and Memon, A. M. 2007. "Designing and comparing automated test oracles for GUI-based software applications", ACM Trans. Softw. Eng. Methodol. 16, 1 (Feb. 2007), 4

[43] Memon, A. M.. "An Event-Flow Model to Test EDS". In Software Engineering and Development, (Enrique A. Belini, ed.), 2009

[44] Memon, I. BanerJee, A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing", Proc. of 10th Working Conference on Reverse Engineering (WCRE '03), 2003

[45] Memon, A. M., Soffa, M. L., Pollack, M. E. "Coverage Criteria for GUI Testing", Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, Vienna, Austria. Pages: 256 – 267. 2001

[46] Xie, Q. and Memon, A. M. 2006. "Automated model-based testing of community-driven open source GUI applications" In ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, 145-154

[47] Memon, A. M. and Xie, Q. "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software" IEEE Transactions on Software Engineering 31, 10 (Oct.), 2005, 884–896

[48] Zhu, H., Wong, W. E., Belli, F. "Advancing test automation technology to meet the challenges of model-driven software development" report on the 3rd workshop on automation of software test, ICSE, 2008

[49] Belli, F. "Finite-State Testing and Analysis of Graphical User Interfaces", ISSRE, 2001

[50] Yuan Miao; Xuebing Yang; , "An FSM based GUI test automation model," Control Automation Robotics & Vision (ICARCV), 2010 11th International Conference on , vol., no., pp.120-126, 7-10 Dec. 2010

[51] F. Belli, C. J. Budnik, L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study", Software Testing Verification and Reliability, J. Wiley & Sons, Ltd., 2006, 16: 3-32.

[52] Stroulia E., El-Ramly M., Kong L., Sorenson P., Matichuk B., 1999. Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach. In:Proceedings of the Sixth IEEE Working Conference on Reverse Engineering, IEEE CS Press, 292-302.

[53] Stroulia E., El-Ramly M., Sorenson P., 2002. From Legacy to Web through Interaction Modeling. Proc. of the IEEE International Conference on Software Maintenance, IEEE CS Press, pp.320-329

[54] G. Canfora, A.R.Fasolino, G. Frattolillo, P.Tramontana, "Migrating Interactive Legacy Systems To Web Services", 10th IEEE European Conference on Software Maintenance and Reengineering, CSMR 2006, pp. 23-32.

[55] G. Di Lorenzo, A. R. Fasolino, L. Melcarne, P. Tramontana, V. Vittorini, "Turning Web Applications into Web Services by Wrapping Techniques", 14th Working Conference on Reverse Engineering, WCRE 2007, pp. 199- 208

[56] Marchetto, A.; Tonella, P.; Ricca, F.; , "State-Based Testing of Ajax Web Applications," Software Testing, Verification, and Validation, 2008 1st International Conference on , vol., no., pp.121-130, 9-11 April 2008

[57] Mesbah, A.; van Deursen, A.; Roest, D.; , "Invariant-Based Automatic Testing of Modern Web Applications," Software Engineering, IEEE Transactions on , vol.PP, no.99, pp.1, 0

[58] Mesbah, A.; Bozdag, E.; van Deursen, A.; , "Crawling AJAX by Inferring User Interface State Changes," Web Engineering, 2008. ICWE '08. Eighth International Conference on , vol., no., pp.122-134, 14-18 July 2008

[59] Matthijssen, N.; Zaidman, A.; Storey, M.-A.; Bull, I.; van Deursen, A.; , "Connecting Traces: Understanding Client-Server Interactions in Ajax Applications," Program Comprehension (ICPC), 2010 IEEE 18th International Conference on , vol., no., pp.216-225, June 30 2010-July 2 2010

[60] A. E. Hassan and R. C. Holt, "Architecture recovery of web applications," in Proceedings of the International Conference on Software Engineering (ICSE). ACM, 2002, pp. 349–359

[61] Draheim D., Weber G., 2005, Form-Oriented Analysis. A New Methodology to Model Form-Based Applications. Springer-Verlag

[62] Document Object Model (DOM), W3C, available at: http://www.w3.org/DOM/

[63] A. Andrews, J. Offutt, R. Alexander, "Testing Web applications by modeling with FSM", Software and System Modeling, vol.4, no.3, pp. 326-345, 2005
[64] Standard Widget Toolkit (SWT), available at: http://www.eclipse.org/swt/
[65] Gecko: https://wiki.mozilla.org/Gecko
[66] JavaXPCOM, available at: http://developer.mozilla.org/en/docs/JavaXPCOM
[67] B. Hohrmann, P. Le Hègaret, T. Pixley and D. Schepers, J. Rossi"Document Object Model Events", available at: http://www.w3.org/TR/DOM-Level-3-Events/
[68] Ajax FilmDB, available at: http://sourceforge.net/projects/ajaxfilmdb/
[69] D.A. Konovalov, B. Litow and N. Bajema, "Partition-distance via the assignment problem", Bioinformatics 21(10), 2005, pp. 2463-2468
[70] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis,", IEEE Transactions on Software Engineering, vol. 99, pp. 684-702, 2009
[71] B. Cornelissen, L. Moonen and A. Zaydman, "An Assessment Methodology for Trace Reduction Techniques", IEEE Int. Conf. on Software Maintenance, 2008, IEEE CS Press, pp. 107-116
[72] A. Marchetto and P. Tonella, "Search-Based Testing of Ajax Web Applications", Proc. Of 1st International Symposium on Search Based Software Engineering, IEEE CS Press, 2009, pp. 3 - 12
[73] T.J. Biggerstaff, B.G. Mitbander and D. Webster, "Program understanding and the concept assignment problem", Communications of the ACM, vol. 37 (5), pp. 72- 83
[74] Graphviz - Graph Visualization Software, available at http://www.graphviz.org
[75] Scott Amber: Agile Modeling: Effective Practices for Extreme Programming and the Unified Process, Wiley 2002
[76] Kent Beck: Extreme Programming Explained - Embrace Change, Addison Wesley 2000
[77] M. Smith, J.Miller, L. Huang and A. Tran, "A More Agile Approach to Embedded System Development", IEEE Software, May/June 2009, pp. 50- 57
[78] S. Cohan, "Successful Integration of Agile Development Techniques within DISA", Agile 2007, IEEE CS Press, pp. 255-261
[79] M.I: Cagnin, J.C. Maldonado and R.D. Penteado, "PARFAIT: towards a framework-based agile reengineering process", Proc. of the Agile Development Conference, ADC 2003. 2003 Page(s):22 – 31
[80] Naresh Jain, "Offshore Agile Maintenance", Proceedings of AGILE 2006 Conference (AGILE'06), IEEE CS Press, 2006
[81] S. Elbaum, G. Rothermel, S. Karre, M. Fisher, "Leveraging User-Session Data to support Web Application Testing", IEEE Transactions on Software Engineering, 2005, 31 (3):187- 202
[82] F. Ricca , P. Tonella, "Analysis and Testing of Web Applications" Proc. of International Conference on Software Engineering, IEEE Computer Society Press, 2001, pp. 25-34
[83] Mesbah, A.; van Deursen, A.; , "Invariant-based automatic testing of AJAX user interfaces," Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on , vol., no., pp.210-220, 16-24 May 2009
[84] S. Sampath, V. Mihaylov, A. Souter, L. Pollock, "A Scalable approach to user-session based testing of Web applications through Concept Analysis", Proc. of 19th Int. Conf. on Automated Software Engineering, IEEE CS Press, 2004, pp. 132- 141
[85] P. A. Brooks, A. M. Memon, "Automated GUI Testing guided by Usage Profiles", Proceedings of ASE'07, ACM , 2007, pp. 333- 342
[86] S. Elbaum, S. Karre, G. Rothermel, "Improving Web Application Testing with User Session Data", Proceedings of International Conference on Software Engineering, IEEE Comp. Society Press, 2003, pp. 49-59
[87] A. M. Memon, Q. Xie, "Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software", IEEE Transaction on Software Engineering, 2005, Vol. 31, No. 10, pp. 884-896
[88] X. Yuan, A. M. Memon, "Using GUI Run-Time State as Feedback to Generate Test Cases", ICSE 2007, IEEE CS Press, pp. 396-405
[89] X. Yuan, A. M. Memon, "Generating Event Sequence-Based Test Cases using GUI Run-Time State Feedback", IEEE Transaction on Software Engineering, 2010, Vol. 36, No. 1, pp. 81-95
[90] Selenium: http://seleniumhq.org/
[91] C. Duda, G. Frey, D. Kossmann, R. Matter, C. Zhou, "AJAX Crawl: Making AJAX Applications searchable", Proc. of IEEE Int. Conf. on Data Engineering, 2009, IEEE CS Press, pp. 78-89

[92] M. J. Harrold, R. Gupta, M. L. Soffa, "A methodology for controlling the size of a test suite", ACM Transactions on Software Engineering and Methodology, 1993, 2 (3): 270- 285
[93] S. McMaster, A. M. Memon, "Call-Stack Coverage for GUI Test Suite Reduction", IEEE Trans. on Software Engineering, Vol. 34, No. 1, Jan. 2008, pp. 99- 115
[94] S. Sampath, S. Sprenkle, E. Gibson, A. Souter, L. Pollock, "Applying concept analysis to user-session based testing of Web applications", IEEE Trans. on Software Engineering, v. 33, n. 10, Oct.2007, pp. 643- 658
[95] G. A. Di Lucca, A. R. Fasolino, P. Tramontana, "A Technique for Reducing User Session Data Sets in Web Application Testing", Proc. of IEEE Workshop on Web Site Evolution, WSE 2006, IEEE CS Press, pp. 7-13
[96] Cem Kaner, "What is a good test case?", Software Testing Analysis & Review Conference (STAR) East, Orlando, FL, May 12-16, 2003
[97] Y. Guo and S. Sampath, "Web application fault classification - an exploratory study", in Proceedings of the Second ACM-IEEE international Symposium on Empirical Software Engineering and Measurement (Kaiserslautern, Germany, October 09 - 10, 2008). ESEM '08. ACM, New York, NY, 303-305, 2008
[98] A. Marchetto, F. Ricca and P. Tonella, "Empirical Validation of a Web Fault Taxonomy and its usage for Fault Seeding", In the IEEE Int. Symposium on Web Site Evolution, pp.31-38, 2007
[99] A. Marchetto, F. Ricca and P. Tonella, "An Empirical Validation of a Web Fault Taxonomy and its usage for Web Testing", In Journal of Web Engineering, vol..8, n. 4, pp. 316-345, 2009
[100] D. Delgado, A. Quiroz Gates and S. Roach, "A taxonomy and catalog of Runtime software-fault monitoring tools", IEEE Trans. On Software Engineering, 2004, Vol. 30, No. 12, pp. 859-872
[101] R. V. Binder, "Testing Object-Oriented Systems. Models, Patterns, and Tools", 1999, Addison Wesley
[102] Crawljax, available from http://crawljax.com
[103] Giuseppe A. Di Lucca, Anna Rita Fasolino and Porfirio Tramontana: "Web Pages Classification using Concept Analysis", IEEE International Conference on Software Maintenance, ICSM 2007, IEEE CS Press, pp. 385-394, 2007
[104] Selenium: available from http://seleniumhq.org/
[105] D. Roest, A. Mesbah and A. van Deursen, "Regression Testing Ajax Applications: Coping with Dynamism". In Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10), 2010 IEEE Computer Society, pp. 127-136
[106] Markup Validation Service, available from http://validator.w3.org/
[107] Complete List of Web Accessibility Evaluation Tools, W3C WAI, available from http://www.w3.org/WAI/ER/tools/complete.html
[108] Ajax Frameworks, available at: http://ajaxpatterns.org/Ajax_Frameworks
[109] John J. Barton and Jan Odvarko. 2010. Dynamic and graphical web page breakpoints. In Proceedings of the 19th international conference on World wide web (WWW '10). ACM, New York, NY, USA, 81-90.
[110] Firebug, available at: http://getfirebug.com/
[111] Ajax Toolkit Framework Project, available at http://www.eclipse.org/atf/
[112] Venkman JavaScript Debugger Project, available at: http://www.mozilla.org/projects/venkman/
[113] dynaTrace Ajax Edition, available at: http://ajax.dynatrace.com/pages/
[114] M.A. Storey, "Theories, tools and research methods in program comprehension: past, present, future", Software Quality Journal, 2006, Springer, Vol. 14: pp. 187-208
[115] S. R. Tilley, D. B. Smith, S. Paul "Towards a Framework for Program Understanding", Proceedings of the 4th International Workshop on Program Comprehension, WPC 1996
[116] M. J. Pacione, M. Roper, and M. Wood, "A Novel Software Visualisation Model to Support Program Comprehension", Proc. of 11th Work. Conf. on Reverse Engineering (WCRE '04), 2004, IEEE CS Press, pp. 70- 79
[117] B. Cornelissen, A. Zaidman, A. Van Deursen, "Trace Visualization for Program Comprehension: a Controlled Experiment", Proc. of Int. Conf. on Program Comprehension (ICPC '09), 2009, IEEE CS Press, pp. 100-109
[118] Tudu Lists, available at: http://sourceforge.net/projects/tudu/
[119] M. Di Penta, R.E.K. Stirewalt, and E. Kraemer, "Designing your Next Empirical Study on Program Comprehension", Proc. of Int. Conf. on Program Comprehension (ICPC), 2007, IEEE C.S. Press, pp. 281- 285
[120] GWT, Google Web Toolkit, available at: http://code.google.com/intl/it-IT/webtoolkit/
[121] ASP.NET Ajax: http://www.asp.net/ajax

[122] Microsoft Silverlight, available at: http://silverlight.net/
[123] Adobe AIR, available at: http://www.adobe.com/products/air.html
[124] Adobe Flex, available at: http://www.adobe.com/products/flex.html
[125] JavaFx, available at: http://javafx.com/
[126] ISO, Software Product Evaluation - Quality Characteristics and Guidelines for Their Use (ISO/IEC IS 9126). Geneva, Switzerland: International Organization for Standardization 1991
[127] I. Sommerville, Software Engineering: Nineth edition, 2011, Addison-Wesley
[128] IEEE Standard for Software User Documentation, IEEE Std 1063-2001, 2001
[129] Wikipedia, Comparison of Screencasting Tools, available at: http://en.wikipedia.org/wiki/Comparison_of_screencasting_software
[130] Institute of Electrical and Electronics Engineers. Glossary of Software Engineering Terminology. IEEE, New York, 1990. IEEE Standard 610.12-1990
[131] Javadoc Tool, available at http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[132] Doxygen, Generate documentation from source code. available at: http://www.stack.nl/~dimitri/doxygen/
[133] E.J. Chikofsky and J.H. Cross II. "Reverse engineering and design recovery: a taxonomy", IEEE Software, vol.7, no.1, pp.13-17, Jan 1990
[134] Y. Zhang, G. Huang, N. Zhang, and H. Mei. "SmartTutor: Creating IDE-based interactive tutorials via editable replay", In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society Press, pp. 559-562
[135] C. Kojouharov, A. Solodovnik, and G. Naumovich, "JTutor: an Eclipse plug-in suite for creation and replay of code-based tutorials", In Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange (eclipse '04). ACM, New York, NY, USA, 27-31
[136] L. Bergman, V. Castelli, T. Lau and D. Oblinger, "DocWizards: a system for authoring follow-me documentation wizards", In Proceedings of the 18th annual ACM symposium on User interface software and technology (UIST '05). ACM, New York, NY, USA, 191-200
[137] Epiance Software, epiDOCX, http://www.epiplex500.com/ (also available at http://download.cnet.com/windows/epiance-software/3260-20_4-6311213.html)
[138] G. Antoniol, M. Di Penta, M. Zazzara: "Understanding Web Applications through Dynamic Analysis", Proceedings of the International Workshop on Program Comprehension, IWPC 2004, IEEE CS Press, pp. 120-131
[139] M. L. Bernardi, G. A. Di Lucca and D. Distante. "The RE-UWA approach to recover user centered conceptual models from Web applications", International Journal on Software Tools for Technology Transfer, STTT 11(6): 485-501 (2009)
[140] F. Ricca and P. Tonella, "Understanding and Restructuring Web Sites with ReWeb", IEEE Multimedia magazine, special issue on Web Engineering, pp. 40-51, April-June 2001, Vol 8, N. 2
[141] W. Wang, Yu Lei, S. Sampath, R. Kacker, R. Kuhn and J. Lawrence, "A Combinatorial Approach to Building Navigation Graphs for Dynamic Web Applications", Proceedings of the 2009International Conference on Software Maintenance (ICSM '09). IEEE Computer Society Press, pp. 211- 220
[142] A. Forward and T. Lethbridge, "The relevance of software documentation, tools and technologies: a survey", In Proceedings of the 2002 ACM symposium on Document engineering (DocEng '02). ACM, New York, NY, USA, 26-33
[143] Murugesan, San; Rossi, Gustavo; Wilbanks, Linda; Djavanshir, Reza; , "The Future of Web Apps," IT Professional, vol.13, no.5, pp.12-14, Sept.-Oct. 2011
[144] A.Wasserman, "Software Engineering Issues for Mobile Application Development", Proc. of the FSE/SDP workshop on Future of software engineering research, FOSER 2010, IEEE Comp. Soc. Press, pp. 397- 400
[145] J. Bo, L. Xiang, and G. Xiaopeng,. "Mobiletest. A Tool Supporting Automatic Black Box Testing for Software on Smart Mobile Devices". In AST '07: Proceedings of the Second International Workshop on Automation of Software Test. Washington, DC, USA: IEEE Computer Society, 2007, p. 8-14.
[146] M. E. Delamaro, A. M. R. Vincenzi, and J. C. Maldonado. "A strategy to perform coverage testing of mobile applications". In Proceedings of the 2006 international workshop on Automation of software test (AST '06). ACM, New York, NY, USA, 118-124
[147] I. Satoh. "A Testing Framework for Mobile Computing Software". IEEE Trans. Softw. Eng. 29, 12 (December 2003), 2003, pp. 1112-1121
[148] I. Satoh. Software testing for wireless mobile application. IEEE Wireless Communications,

Oct. 2004 pp. 58-64

[149] S. She, S. Sivapalan, I. Warren. Hermes: A Tool for Testing Mobile Device Applications. Proc. of 2009 Australian Software Engineering Conference, IEEE Comp. Soc. Press., pp. 123-130

[150] S. Srirama, R. Kakumani, A. Aggarwal, and P. Pawar, "Effective Testing Principles for the Mobile Data Services Applications", First International Conference on Communication System Software and Middleware, Comsware 2006, IEEE Comp. Soc. Press, pp. 1-5

[151] J. L. Wesson and D. F. van der Walt, "Implementing Mobile Services: Does the Platform Really Make a Difference?" in SAICSIT '05: Proc. of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries. South African Institute for Computer Scientists and Information Technologists, 2005, pp. 208–216

[152] D. Gavalas and D. Economou. "Development Platforms for Mobile Applications: Status and Trends". IEEE Software, Volume: 28, Issue: 1 , 2011, pag. 77- 86

[153] Gartner Newsroom. Gartner Says Android to Become No. 2 Worldwide Mobile Operating System in 2010 and Challenge Symbian for No. 1 Position by 2014.

[154] H. Kim, B. Choi, W. Eric Wong. "Performance Testing of Mobile Applications at the Unit Test Level". Proc. of 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement, IEEE Comp. Soc. Press, pp. 171- 181

[155] Z. Liu, X. Gao, X.Long. "Adaptive Random Testing of Mobile Application". Proc. of 2010 2nd International Conference on Computer Engineering and Technology (ICCET), IEEE Comp. Soc. Press, pp. 297-301

[156] Android Developers. UI Application Exerciser Monkey Available at: http://developer.android.com/guide/developing/tools/monkey.html

[157] Android Developers. Monkeyrunner. Available at: http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html

[158] Google Code. Robotium. Available at: http://code.google.com/p/robotium

[159] Android Developers. The Developer's Guide. Available at: http://developer.android.com

[160] Junit. Resources for Test Driven Development. Available at: http://www.junit.org

[161] Android Emulator, available at: http://developer.android.com/guide/developing/tools/emulator.html

[162] Butler, M.; Giannetti, F.; Gimson, R.; Wiley, T.; , "Device independence and the Web," Internet Computing, IEEE , vol.6, no.5, pp. 81- 86, Sep/Oct 2002