Analysis and Testing of AJAX-based Single-page Web Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 19 juni 2009 om 14:00 uur

door

Ali MESBAH

ingenieur informatica
geboren te Karaj, Iran

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. A. van Deursen

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. A. van Deursen	Technische Universiteit Delft, promotor
Prof. dr. A. Orso	Georgia Institute of Technology
Prof. dr. A.L. Wolf	Imperial College London
Prof. dr. P.M.E. De Bra	Technische Universiteit Eindhoven
Prof. dr. P. Klint	Universiteit van Amsterdam & CWI
Prof. dr. ir. G.J.P.M. Houben	Technische Universiteit Delft
Prof. dr. ir. F.W. Jansen	Technische Universiteit Delft

The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was partially supported by the Dutch Ministry of Economic Affairs under the SenterNovem program, project Single Page Computer Interaction (SPCI).

IPA Dissertation Series 2009-08

ISBN 978-90-79982-02-8

Cover design by Azin Nader and Mohamad Yekta.

This work has been typeset by the author using LaTeX.

Printed by Wöhrmann Print Service, Zutphen.

Contents

Preface

Four years have passed since I started my PhD research project (SPCI). Now that I look back, I can say with great confidence that it is has been a very instructive and pleasant experience. I wish to thank all those who have made it possible.

First of all, I would like to extend my heartfelt gratitude to Arie van Deursen for giving me the opportunity to work under his excellent supervision. Arie has been a great source of inspiration and I am very thankful for his continuous encouragement and support, enthusiasm, and kindness. His extensive knowledge and insight along with his open and positive attitude make him a great mentor for every student.

I would like to thank the members of my defense committee: prof. dr. Alessandro Orso, prof. dr. Alex Wolf, prof. dr. Paul de Bra, prof. dr. Paul Klint, prof. dr. Geert-Jan Houben, and prof. dr. Erik Jansen, for providing me with valuable feedback on this thesis.

I would like to thank Engin Bozdag for his feedback on Chapter 2, our collaboration in writing Chapter 4, and more importantly, for the good times on conference trips.

Many thanks to the (current and former) members of the Software Engineering Research Group (SERG) at Delft University of Technology: Andy Zaidman (for proof reading parts of the thesis), Bas Cornelissen (for correcting the 'Samenvatting'), Cathal Boogerd, Marius Marin, Rui Abreu, Eelco Visser and his 'gang', Martin Pinzger, Gerd Gross, Marco Lormans, Bas Graaf, Alberto Gonzalez, Michaela Greiler, Eric Piel, Hans Geers, Frans Ververs, Teemu Kanstren, Eric Bouwers, and Adam Nasr.

The first year of my PhD work took place in the software engineering research group (SEN1) at the Centrum Wiskunde & Informatica (CWI) in Amsterdam. I thank Paul Klint and the members of SEN1 for facilitating my stay at CWI.

During my MSc studies I started working at West Consulting BV (West), a software company composed of highly motivated and disciplined software engineers. I have gained most of my real-world industrial software engineering experience at West, for which I am very thankful to all the colleagues. In particular, I am grateful to Rob Westermann, managing director of West, for his continuous support and sponsoring through all these years.

Within the SPCI project, I had the privilege to work with a number of talented MSc students: Engin, Maikel, Justin, Vahid, Cor-Paul, and Danny. Thanks for the great collaborations and results.

I really appreciate spending time with my dear friends Payman, Parham, Behnam, Vahid, and Samrad. Thanks for the fun times as well as the valuable conservations on all aspects of life, politics, and technology.

My sincere thanks go to Azin Nader and my dear friend Mohamad Yekta for designing the cover of this thesis.

I am very grateful to my parents, my two sisters, and my parents-in-law, for their unconditional support and kindness. I thank my father for encouraging me to follow his steps in doing a PhD.

Last, but certainly not least, I am indebted to my best friend and wife, Negin, who tolerated my negligence during the paper submission deadlines. Thanks for everything!

Ali Mesbah
April 24, 2009
Delft

List of Acronyms

ASP Active Server Pages

API Application Programming Interface

ATUSA Automatically Testing User-interface States of Ajax

CGI Common Gateway Interface

Crawljax CRAWLing aJAX

CSS Cascading Style Sheets

DOM Document Object Model

DSL Domain Specific Language

GUI Graphical User Interface

GWT Google Web Toolkit

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

JSF JavaServer Faces

JSP JavaServer Pages

JSON JavaScript Object Notation

Ajax Asynchronous JavaScript and XML

REST REpresentational State Transfer

Retjax Reverse Engineering Tool for Ajax

RIA Rich Internet Application

SPIAR Single Page Internet Application aRchitecture

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

XHTML Extensible HyperText Markup Language

WWW World Wide Web

Introduction

Chapter 1

According to recent statistics,[1] there are approximately 1.46 billion people using the Internet, with a penetration of 21.9%, based on the world population estimate of 6.67 billion persons for mid-year 2008. This figure represents an incredible increase of 305% in 2008, compared to the year 2000.

The World Wide Web (WWW) has been growing at a very fast pace (see Figure 1.1). In July 2008, Google engineers announced[2] that the search engine had discovered one trillion unique URLs on the Internet.

Current wide user participation in accessing, creating, and distributing digital content is mainly driven by two factors: wider world wide broadband access and new web technologies providing user-friendly software tools.

The web has had a significant impact on business, industry, finance, education, government, and entertainment sectors, as well as our personal lives. Many existing software systems have been and continue to be migrated to the web, and many new domains are being developed, thanks to the ubiquitous nature of the web.

In this thesis, we focus on understanding, analyzing, and testing interactive standards-based web applications, and the consequences of moving from the classical multi-page model to a single-page style.

In this chapter, we first take a brief look at the evolution of the web from its infancy stages up until today, with respect to the level of offered user interactivity and responsiveness. We discuss the advantages and challenges that new web technologies bring with them and outline the main questions that are addressed in this research.

1.1 Web Evolution

1.1.1 Static Hypertext Documents

The WWW was created in 1989 by Tim Berners-Lee, and released in 1992. The web was initially based on the following four concepts (Berners-Lee, 1996):

- Independence of specifications, to achieve the ultimate goal of flexibility through as few and independent specifications as possible;
- The Uniform Resource Identifier (URI), a sequence of characters used to identify or name a resource, such as a web page, uniquely on the web space. The web requires only unidirectional links, enabling users to link to external resources without any action required from the external

[1] `http://www.internetworldstats.com`, retrieved 20 October 2008.

[2] `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`

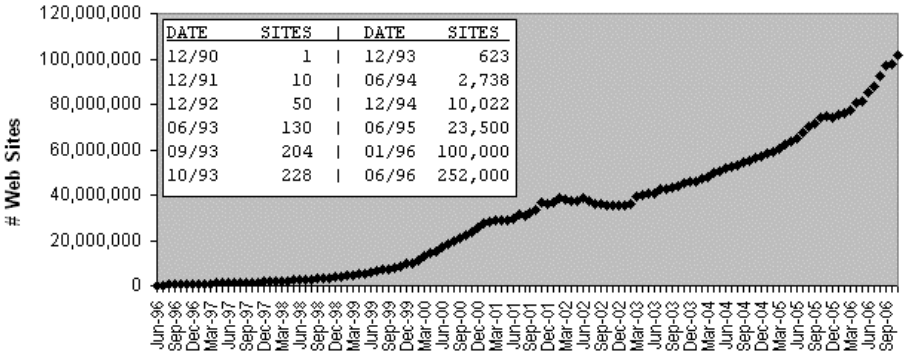

Figure 1.1 The WWW growth. Note: Sites = number of web servers (one host may have multiple sites by using different domains or port numbers). Taken from (Zakon, 2006).

party. The Uniform Resource Locator (URL) is a URI, which also specifies the location of the identified resource and the protocol for accessing it (Berners-Lee et al., 1994);

- The HyperText Markup Language (HTML), to format data in *hypertext* documents. Hypertext refers to text on a computer that makes a dynamic organization of information through connections (called *hyperlinks*) possible;
- The HyperText Transfer Protocol (HTTP), an application-level, stateless request-response protocol for distributed, hypermedia information systems (Fielding et al., 1999). It is used for transporting data on the network between the client (e.g., browser) and server. The protocol supports eight operations: GET, POST, HEAD, PUT, OPTIONS, DELETE, TRACE, and CONNECT. In practice, mostly the fist two are used to GET pages from, and POST user data to, servers.

Using these concepts, a simple but powerful client/server architecture was developed in which resources could be linked together and easily accessed through web browsers. In the early nineties, the web was merely composed of linked simple static hypertext documents. Upon sending a request to the server, the server would simply locate and retrieve the corresponding web page on the file-system, and send it back to the client browser. The browser would then use the new web page to refresh the entire interface. Figure 1.2 shows one of the first web browsers called Mosaic, credited with popularizing the web because of its user friendly interface.

1.1.2 Dynamically Generated Pages

After the wide adoption of the web, more complex web applications began to flourish, moving from static pages on the file-system to dynamically assem-

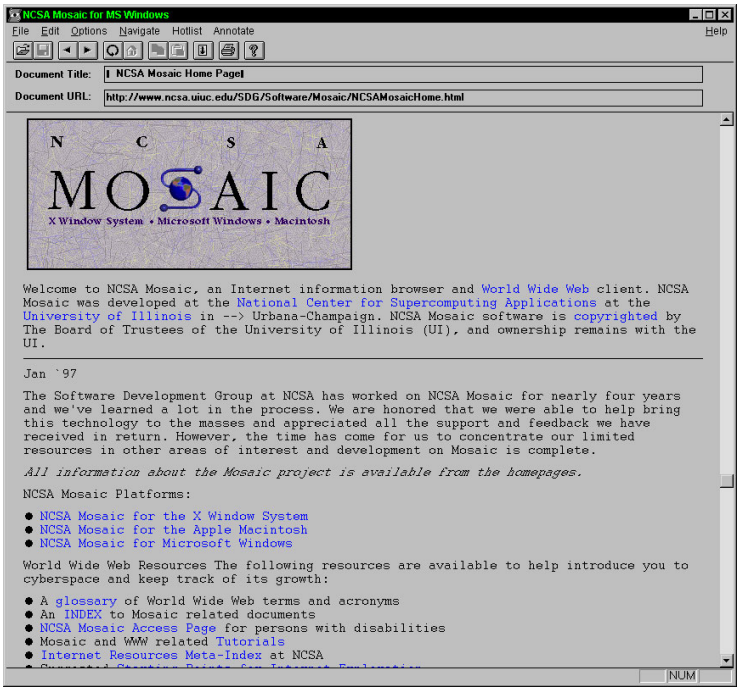

Figure 1.2 Screen shot of the Mosaic web browser interface, late 1993, designed at NCSA.

bled pages served by smarter web servers. Figure 1.3 shows the classical web client/server interaction style in which a typical scenario would be as follows:

- the user clicks on a hypertext link (URL),
- the browser sends a (HTTP GET) request to the server,
- (static) if the request points to a file stored on disk, the server retrieves the contents of that file,
- (dynamic) if the request cannot be associated with a file on the disk, then based on the request, the parameters, and the server-side state, the server assembles a new web page,
- the server sends the page to the browser as a response, and
- the browser refreshes the entire page.

The first dynamic web pages were often created with the help of server-side languages, such as Perl, typically though the Common Gateway Interface (CGI), a standard (W3C, 1995) for interfacing external applications, such as databases, with web servers to assemble dynamic web pages.

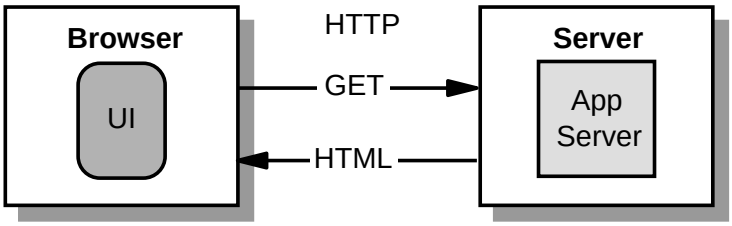

Figure 1.3 Classical web client/server interaction pattern.

As the web matured, more server-side scripting languages appeared, examples of which include PHP, Python, Ruby, JavaServer Pages (JSP), and Active Server Pages (ASP). Such languages typically run on the server, have access to various resources, and are capable of creating and returning web pages upon request.

The ability to generate web pages contributed to the separation of concerns (presentation, business logic, data) and realization of multi-tier architectures for web applications.

1.1.3 Web Architecture

By the year 2000, many of the initial concepts forming the backbone of the web (e.g., HTTP, URI, HTML) and additional recommendations such as Cascading Style Sheets (CSS) (level 2) and Document Object Model (DOM) (W3C, a) (level 2) were standardized through the World Wide Web Consortium (W3C). In addition, an architectural style of the web called REpresentational State Transfer (REST) was proposed by Fielding (2000), capturing the essence of the main features of the web architecture, through architectural constraints and properties. REST specifies a layered client-stateless-server architecture in which each request is independent of the previous ones, inducing the property of scalability. In practice, however, not many web implementations can be found that abide by the restrictions set by REST. In particular, many developers have ignored the *stateless* constraint by allowing the server to keep track of relevant state changes. Chapter 2 discusses the architecture of the web in more detail.

1.1.4 Rich Internet Applications

It soon became apparent that HTML was not designed for creating an interactive Graphical User Interface (GUI). Classical web applications are, inherently, based on a *multi-page user interface* model, in which interactions are based on a synchronous page-sequence paradigm. While simple and elegant in design for exchanging documents, this model has many limitations for developing modern web applications with user friendly human-computer interaction. The main limitations can be summarized as follows:

- Low lever of user interactivity;

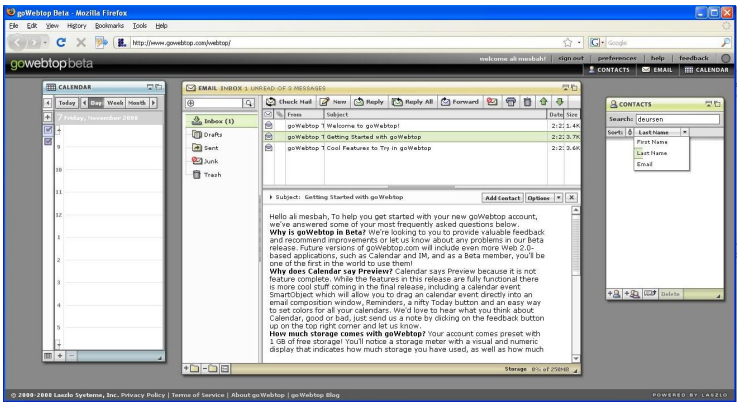

Figure 1.4 Screen shot of OpenLaszlo Webtop, a RIA providing functionality for email, address book, and calendars.

- Redundant data transfer between the client/server;
- High user-perceived latency;
- The browser is passive: there is hardly any application-specific client-side processing.

The concept of a Rich Internet Application (RIA) was proposed (Allaire, 2002) as a response, to describe a new class of web applications that could provide richer user interface components on the browser.

The common ground for all RIAs is an intermediate layer of code introduced between the user and the server, which acts as an extension of the browser, usually taking over responsibility of server communication and rendering the web user interface. An example of a RIA technology is a Java Applet, which extends the web browser, using a Java Virtual Machine (JVM). Other well-known examples include Adobe Flex (based on Flash), OpenLaszlo, and Microsoft Silverlight. Figure 1.4 depicts a screen shot of a RIA desktop, built on OpenLaszlo.

One of the main issues with such technologies is their non-standard (proprietary) plugin-based nature. Users need to install specific plugins for each of the RIA technologies mentioned above.

1.1.5 Web 2.0

Web 2.0 (O'Reilly, 2005) is a term often used describing changing trends in the use of web technology, i.e., evolution from a hypertext read-only system into a dynamic medium of user-created content and rich interaction.

Even though the term is ambiguously defined, it revolves around web technologies that promote:

- strong participation of web users as a source of content (e.g., the online free encyclopedia Wikipedia, the photo sharing site Flickr),

- user collaboration and information sharing (e.g., the social networking web application Facebook),
- rich but simple to use web user interfaces (e.g., Google Maps),
- and software as a service through the web (e.g., the online office application Google Docs).

Many Web 2.0 applications rely heavily on a prominent enabling technology called Asynchronous JavaScript and XML (Ajax) (Garrett, 2005), which is the key topic of this thesis.

1.2 Ajax

1.2.1 JavaScript and the Document Object Model

In 1995, Netscape 2 introduced a simple API called the Document Object Model (DOM), and a new client-side scripting language into the browser called JavaScript. JavaScript is a weakly typed, prototype-based language with first-class functions. JavaScript 1.1 was submitted to Ecma International resulting in the standardized version named ECMAScript. Microsoft followed Netscape by introducing its dialect of the language JScript into Internet Explorer in 1996.

The Document Object Model (DOM) (W3C, a) is a platform- and language-neutral standard object model for representing HTML and XML documents. It provides an API for dynamically accessing, traversing, and updating the content, structure, and style of such documents.

In the Web browser, a DOM instance can be seen as the run-time representation of an HTML page; The DOM object is a tree-based model of the relationships between various HTML elements (e.g., images, paragraphs, forms, tables) present in the document. The first instance of the object is created after an HTML page is loaded into the browser and parsed. This object can be further traversed and modified through JavaScript and the results of modifications are incorporated back into the presented page. There is generally a one-to-one relation between DOM elements and user interface elements (See Appendix A for an example of how the DOM is manipulated in JavaScript).

Netscape 2 and 3 supported a simple DOM that offered access to a limited set of document elements such as images, links, and form elements. This DOM API was adopted by most browser vendors and incorporated into the HTML specification as DOM *Level 0* by the W3C. later on, Internet Explorer 4 improved the DOM by allowing access to and modification of all document elements. By 1998, the first specification of DOM (Level 1) was released, defining the core DOM interfaces, such as Document, Node, Element, and Attr. DOM Level 2 followed in 2000, as an extension on the previous level, to define API's for working with DOM events and stylesheets.

JavaScript was initially used, primarily, for performing simple computations on the browser and modifying the browser User Interface (UI) through

DOM APIs, a technique that was called Dynamic HTML (DHTML), to add some degree of dynamism to the static browser interface. Significant browser compatibility issues regarding DOM implementations, however, discouraged web developers to make much use of DHTML beyond styling menus and simple form manipulations.

The latest version of the W3C DOM specification is Level 3, released in 2004. Currently, web browsers support many features of the W3C DOM standard (mostly Level 2).

1.2.2 Cascading Style Sheets

Cascading Style Sheets (CSS) is a standard for specifying the presentation of HTML (or XML) documents. While HTML is used to define the structure of a document, CSS is used to specify how the structured elements should be displayed. The first version of CSS (Level 1) was adopted in 1996 by the WC3, to define attributes for specifying styling properties such as colors, margins, and fonts. The second version, CSS Level 2, was released in 1998, to define a number of advanced features, such as relative, absolute, and fixed positioning of elements. CSS Level 2 is supported by most modern browsers. A new recommendation by W3C, CSS Level 3, is currently under development.

1.2.3 The XMLHttpRequest Object

In 1998, Microsoft introduced Remote Scripting, a technology that allowed scripts running inside a browser, e.g., through a Java Applet, to exchange information with a server. Shortly after, in 1999, the XMLHttpRequest object was created as an ActiveX control in Internet Explorer 5. Other browsers (e.g., Mozilla, Safari) followed with their own implementation of this object soon. The XMLHttpRequest object can be accessed in JavaScript to transfer text in various formats, such as XML, HTML, plain text, JavaScript Object Notation (JSON), and JavaScript, between the server and the browser (a)synchronously at the background without the need of a page refresh.

1.2.4 A New Approach to Web Applications

The term Ajax was coined, in February 2005, in an article called 'Ajax: A New Approach to Web Applications' by Garrett (2005), and defined as:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using DOM;
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- and JavaScript binding everything together.

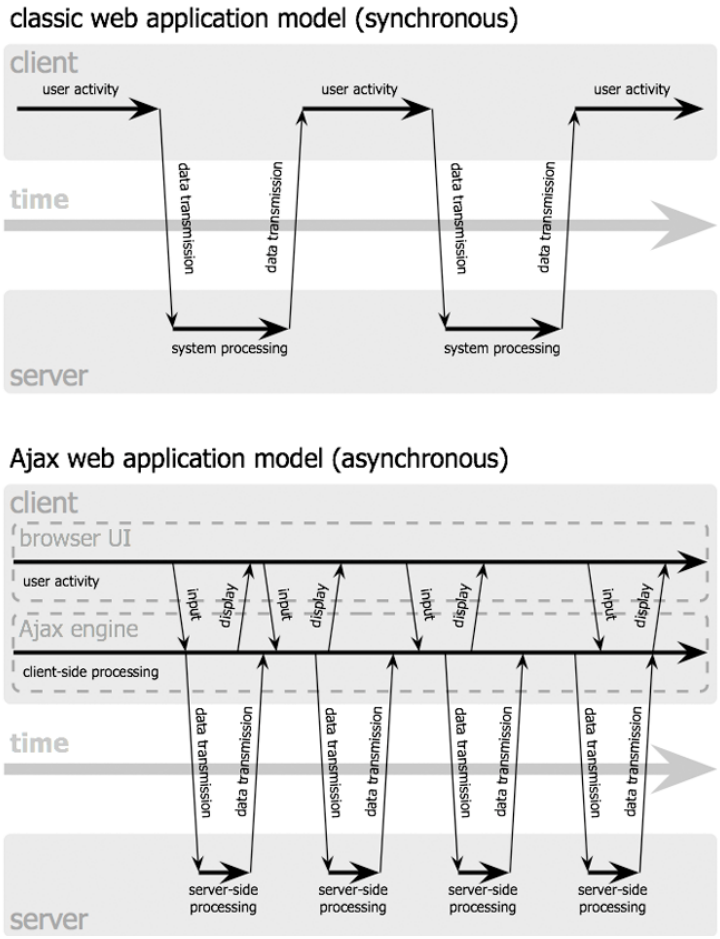

Figure 1.5 The synchronous interaction pattern of a classical web application (top) compared with the asynchronous pattern of an AJAX application (bottom). Taken from Garrett (2005).

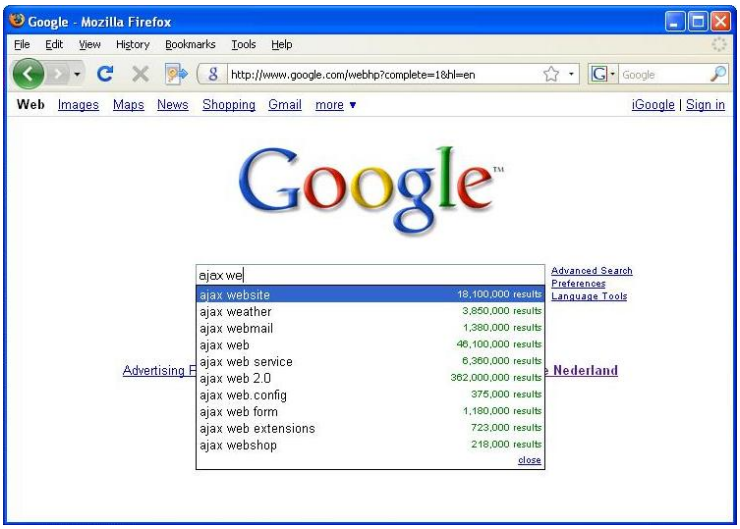

Figure 1.6 Screen shot of Google Suggest. As a web user types search keywords, the application retrieves suggestions from the server at the background, and shows them to the user without having to refresh the whole page.

The term Ajax served to highlight and give a name to a new breed of web applications that could be seen as a further evolution of the classical web. Although many of the technologies Ajax is based on are not new, this naming by Garrett made web developers aware of the possibilities for adopting a new way of developing applications on the web. Figure 1.5 depicts the classical synchronous interaction pattern compared with the asynchronous pattern of an Ajax application.

For more technical details we refer to Appendix A, which describes a simple single-page Ajax-based example application.

Although there have been some disagreements (Koch, 2005) on what Ajax is exactly, how new it is, and which essential components the technology is constituted from, a general acceptance of the fundamental concepts has been achieved within the web community. The acceptance has mainly been driven by the many concrete Ajax examples Google has been working on (even before the term Ajax was coined) from Google Suggest (Figure 1.6), Google Maps, Gmail, to porting desktop applications like Word to web-based versions like Google Documents (Figure 1.7).

When we take a look at such applications from a user's perspective, the main difference, with respect to classical web applications, is the increased responsiveness and interactivity. A refresh of the entire page is generally not seen any longer for each user action and the interaction takes place at a much finer granularity level.

When compared to RIA technologies such as OpenLaszlo, the main difference and advantage is that no plugin is required, since Ajax is based on web standards that modern browsers support already.

Building robust AJAX web applications has been made possible thanks to the evolution of the major browsers (e.g., Firefox) and the way they have supported web standards such as DOM and JAVASCRIPT. In 2006, the W3C released the first draft specification of XMLHttpRequest (W3C, b) to create an official web standard for this invaluable component of AJAX technology.

Adopting AJAX for developing web applications has a number of key advantages that are briefly discussed next.

1.2.5 Multi-page versus Single-page Web Applications

AJAX potentially brings an end to the classical *click-and-wait* style of web navigation, enabling us to provide the responsiveness and interactivity end users expect from desktop applications. In a classical web application, the user has to wait for the entire page to reload to see the response of the server. With AJAX, however, small delta messages are requested from the server, behind the scenes, by the AJAX engine and updated on the current page through modification to the corresponding DOM-tree. This in sharp contrast to the classical *multi-page* style, in which after each state change a completely new DOM-tree is created from a full page reload.

AJAX gives us a vehicle to build web applications with a *single-page web interface*, in which all interactions take place on one page. Single-page web interfaces can improve complex, non-linear user workflows (Willemsen, 2006) by decreasing the number of click trails and the time needed (White, 2006) to perform a certain task, when compared to classical multi-page variants.

Another important aspect of AJAX is that of enriching the web user interface with interactive components and widgets. Examples of widgets, which can all co-exist on the single-page web interface, include auto-completion for input fields, in-line editing, slider-based filtering, drag and drop, rich tables with within-page sorting, shiny photo albums and calendars, to name a few. These are all web UI components that are made possible through extensive DOM programming by means of JAVASCRIPT and delta client/server communication.

In most classical web applications, a great deal of identical content is present in page sequences. For each request, the response contains all the redundant content and layout, even for very marginal updates. Using AJAX to update only the relevant parts of the page results, as expected, in a decrease in the bandwidth usage. Experimental results have shown a performance increase of 55 to 73% (Smullen III and Smullen, 2008; Merrill, 2006; White, 2006) for data transferred over the network, when AJAX is used to conduct partial updates.

1.2.6 Reverse Ajax: Comet

The classical model of the web requires all communication between the browser and the server to be initiated by the client, i.e., the end user clicks on a button or link, and thereby requests a new page from the server. No *permanent*

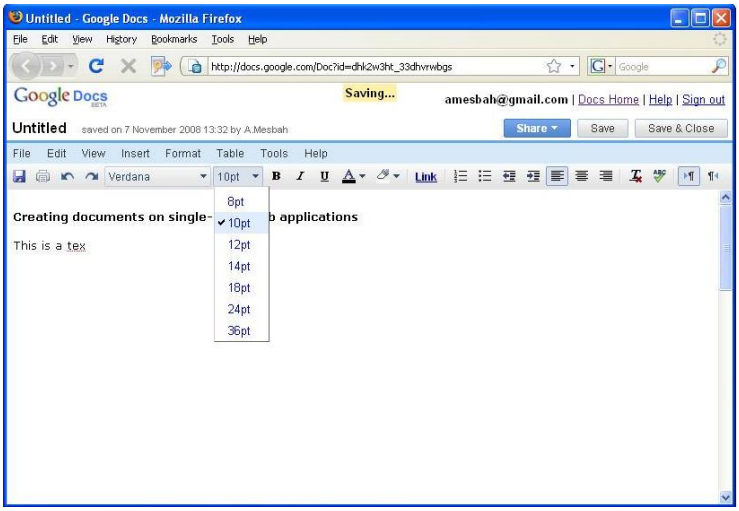

Figure 1.7 Screen shot of Google Documents, a web-based word processor, spreadsheet, presentation, and form application.

connection is established between client/server and the server is not required to maintain any state information from the clients.

This pull-based (polling) style of interaction, although scalable, has limitations for applications that require fast data delivery to the clients. Examples of such applications include auction web sites where the users need to be informed about higher bids, web-based stock tickers where stock prices are frequently updated, multi-user collaboration applications, web-based chat applications, or news portals.

An alternative to the traditional pull-based approach is the push-based style, where the clients subscribe to their topic of interest, and the server publishes the changes to the clients asynchronously every time its state changes.

In 1995, Netscape introduced a method (Netscape, 1995) for pushing data on the web through HTTP Streaming, by using a special MIME type called `multipart/x-mixed-replace`. This method simply consists of streaming server data in the response of a long-lived HTTP connection that is kept open by server side programming.

The push-based approach has recently gained much attention, thanks to many advancements in client and server web technologies that make pushing data from the server, in a seamless manner, possible.

Comet (Russell, 2006) is a neologism to describe this new model of web data delivery. Although Comet provides multiple techniques for achieving high data delivery on the web, the common ground for all of them is relying on standard technologies supported natively by browsers, rather than on proprietary plugins. Generally, Comet (also known as Reverse Ajax) applications rely on Ajax with long polling (see Chapter 4) to deliver state changes to the clients, as fast and reliable as possible. Well-known examples include

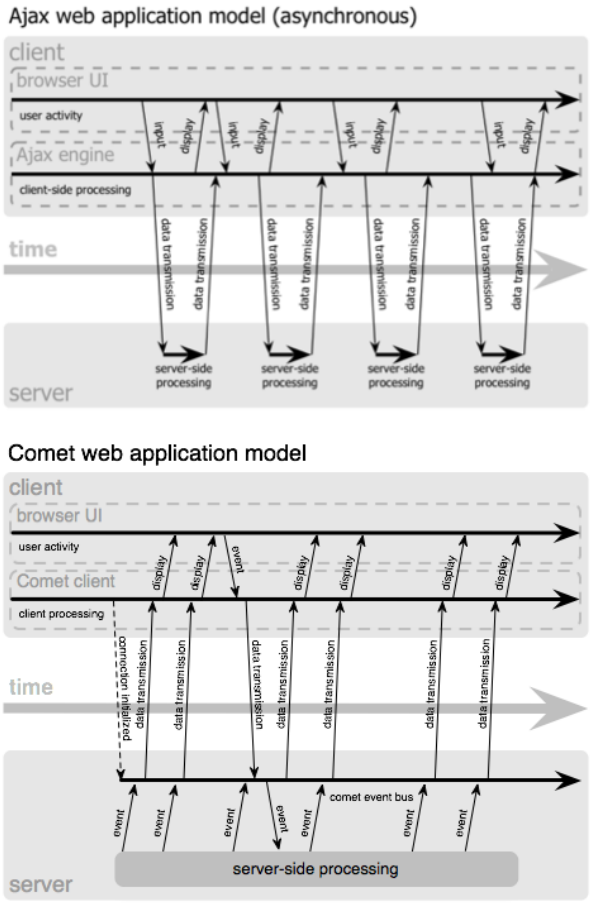

Figure 1.8 The classical poll-based AJAX interaction pattern (top) compared with the push-based style of a COMET application (bottom). Taken from Russell (2006).

Google's web-based chat application in Gmail and the in-browser instant messaging application Meebo[3]. Figure 1.8 taken from (Russell, 2006), shows from the perspective of network activity, the difference between the poll-based and push-based interaction patterns.

1.3 Challenges and Research Questions

The new changes in the web bring not only advantages but also come with a whole set of new challenges.

[3] Meebo, `http://www.meebo.com`

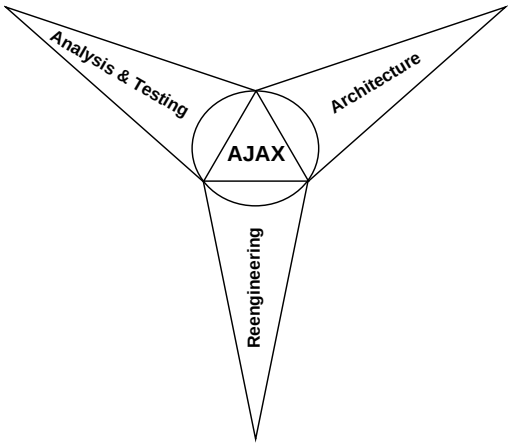

Figure 1.9 Our research viewpoints on AJAX web applications.

Web sites and applications have been deployed at a fast pace not only by experts but also by individuals who lack the required training and knowledge to implement structured systems. Although this phenomenon has helped the fast adoption of the web itself, it has also drastically decreased the quality of software that is produced as a result. The manner in which such systems have been developed, deployed, and managed has raised serious concerns (Ginige and Murugesan, 2001). In addition, the use of multiple software languages to develop a single web application has contributed to the complexity of web systems. As a result, the field has been characterized by a lack of well-defined design methodologies and development processes (Coda et al., 1998). Hence, the need many researchers have felt for a new discipline, called Web Engineering, for web-based systems (Murugesan et al., 2001; Deshpande and Hansen, 2001).

Although developing web applications is different from traditional software development and poses additional challenges due to the heterogeneous and distributed nature, we believe that web engineering can adopt and encompass many software engineering principles, that have been proven useful over the years. In this thesis, we examine challenges of developing interactive AJAX-based web systems from a *software engineering* perspective.

Lehman and Belady's laws of software evolution (Lehman and Belady, 1985) have taught us that software programs require change to remain useful. Over time, enabling software technologies and our understanding of software programs, in terms of their models (Jazayeri, 2005), evolve, while the expectations of the surrounding environment change.

Software that is not modified to meet changing needs (e.g., end users' expectations) becomes old (Parnas, 1994). Coping with aging software and evolving technologies is a challenging task for software engineers.

The web is an excellent example of an evolving technology that causes its applications to age. It started as a simple static page-sequence client/server

system. Web applications based on the classical model of the web and the technologies available in the early nineties, have aged and become out-dated. Over the course of the past 15 years, many web technologies (e.g., browsers, servers, web standards) have evolved. These technological advancements have made it possible to develop web systems that meet up with current user expectations, i.e., a satisfactory degree of responsiveness and interactivity similar to desktop applications. However, what we witness is that web applications built with the new models and technologies have issues with existing tools (e.g., web crawlers) and techniques (e.g., web testing) that are still focused on the old models (multi-page).

This leads developers to a dilemma: on the one hand, sticking to the classical web model means missing on the advantages of the technological innovations and failing to meet today's expectations. On the other hand, adopting the new model means renovating the system to meet expectations and, at the same time, facing many challenges the new model has with existing web technologies. The challenges are mainly due to the fact that AJAX shatters the metaphor of a web 'page' (i.e., a sequence of web pages connected through hyperlinks) upon which many web technologies are based. Hence, the AJAX-based web model will only be widely adopted with success, if the supporting technologies also evolve and support the new model.

Figure 1.9 shows our software engineering research viewpoints on modern AJAX web application. Our focus in this work has been on three main research themes:

Software Architecture to gain an abstract understanding of the new AJAX-based web model;

Software Reengineering to understand the implications of reengineering classical multi-page web systems to single-page AJAX variants;

Software Analysis and Testing to explore strategies for analyzing and testing this new breed of web application.

Below, we discuss these viewpoints and formulate the research questions that drive the work presented in this thesis.

1.3.1 Architecture

After the neologism AJAX was introduced in 2005, numerous frameworks and libraries appeared, and many web developers started adopting one or more of the ideas underpinning AJAX. However, despite all the attention AJAX has been receiving from the web community, the field is characterized by a lack of coherent and precisely defined architectural descriptions.

AJAX provides us with a set of techniques to design a new style of web client/server interaction that was not possible in classical web systems. An interesting question is whether concepts and principles as developed in the software architecture research community, and specifically those related to

client/server and network-based environments, can be of help in understanding the essential components and architectural properties of AJAX-based web applications. In such a context, a software architecture is defined (Perry and Wolf, 1992; Fielding, 2000) by a configuration of architectural elements – processing, connecting, and data – constrained in their relationships in order to achieve a desired set of architectural properties.

Through such an understanding, we could gain a more abstract perspective on the actual differences between the classical web model and the modern AJAX-based settings. An abstract model would also enable us to anticipate the tradeoffs between, e.g., interactivity and scalability, of adopting AJAX techniques.

Thus, our first main research question can be formulated as follows:

Research Question 1

> *What are the fundamental architectural differences and tradeoffs between designing a classical and an* AJAX*-based web application? Can current architectural styles describe* AJAX*? If not, can we propose an architectural style taylored for* AJAX*?*

An architectural style is defined by a coordinated set of architectural constraints that restricts the roles of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style (Fielding, 2000). Our hypothesis is that AJAX changes the web interaction model so significantly that it cannot be fit into the existing architectural styles, and hence requires its own architectural style.

1.3.2 Reengineering

Once an abstract perspective on the target model is gained, we focus on ways classical web applications could be reengineered to AJAX settings.

Many organizations are beginning to consider migration, also known as *Ajaxification*, possibilities of their web-based systems to this new paradigm. Usually, significant investments have been made for classical web-based systems and most organizations are not likely to throw these valuable assets away to adopt a new emerging technology.

As a result, the well-known problems of software legacy renovation (van Deursen et al., 1999) and migration (Brodie and Stonebraker, 1995) are becoming increasingly important for web applications. If until a few years ago, the problem revolved solely around migrating legacy desktop systems to web-based settings, today we have a new challenge of reengineering classic web applications to AJAX applications as well.

Our assumption is that a multi-page web application already exists, and our intention is to explore techniques to support its comprehension, analysis, and restructuring by recovering abstract models from the current implementation.

Research Question 2

> *Is it possible to support the migration process (Ajaxification) of multi-page web applications to single-page* Ajax *interfaces? Can reverse engineering techniques help in automating this process?*

Our hypothesis is that *reverse engineering* (Chikofsky and Cross II, 1990) techniques can assist us in reconstructing abstract models of the source application, by automating (Arnold, 1993) all or parts of the process. Since the user interface interaction models of the source (page-sequence) and the target (single-page with UI components, see Chapter 2) systems are substantially different, user interface reverse engineering (Stroulia et al., 2003) will play an important role in our quest. Automatically reconstructing an abstract user interface model of the source multi-page web application is a first, but also a key step in the migration process.

1.3.3 Analysis and Testing

Our final main question deals with the dependability (Sommerville, 2007) of Ajax applications. In principle, we are interested in appropriate ways to analyze and test Ajax systems.

For traditional software, analysis and testing is still largely ad hoc (Bertolino, 2007) and already a notoriously time-consuming and expensive process (Beizer, 1990). Classical web applications present even more challenges (Di Lucca and Fasolino, 2006; Andrews et al., 2005) due to their distributed, heterogeneous nature. In addition, web applications have the ability to dynamically generate different UIs in response to user inputs and server state (Andrews et al., 2005).

The highly dynamic nature of Ajax user interfaces and their client/server delta communication adds an extra level of complexity to the classical web analysis and testing challenges. Therefore, we formulate our third main question as:

Research Question 3

> *What are the challenges for analyzing and testing* Ajax *applications in an automatic approach?*

which, in turn, is composed of three sub-questions focusing on data delivery performance, automatic crawling, and user interface testing.

Performance Testing

One of the challenges related to client/server interactions is to ensure data coherence, i.e., ensuring that the data (state changes) on the server and the client are synchronized. The hypothesis is that a push-based implementation offers a higher degree of data coherence when compared to a pull-based one. But at the same time, it is generally believed that a push-based solution that keeps open connections for all clients causes scalability problems on the web.

To the best of our knowledge, at the time of writing no study had been conducted to explore the actual tradeoffs in terms of data coherence, server performance and scalability, network performance, and data delivery reliability, involved in applying a push- versus pull-based approach to web-based settings.

Web applications are distributed systems, and distributed systems are inherently more difficult to engineer (Wang et al., 2005) and test than sequential systems (Alager and Venkatsean, 1993). Controllability, observability (Chen et al., 2006), and reproducibility are all challenging issues in distributed testing environments. In order to conduct a comparison of different web data delivery techniques, first an automated, controllable, and repeatable test environment has to be set up, to obtain accurate empirical data for each approach. This leads us to our next research question:

Research Question 3.1

> *What are the tradeoffs of applying pull- and push-based data delivery techniques on the web? Can we set up an automated distributed test environment to obtain empirical data for comparison?*

Automatic Crawling

General web search engines, such as Google and Yahoo!, cover only a portion of the web called the *publicly indexable web*, which consists of the set of web pages reachable purely by following hyperlinks. Dynamic content behind web forms and client-side scripting is generally ignored and referred to as the *hidden web* (Raghavan and Garcia-Molina, 2001).

Although there has been extensive research on finding and exposing the hidden web behind forms (Barbosa and Freire, 2007; de Carvalho and Silva, 2004; Lage et al., 2004; Ntoulas et al., 2005; Raghavan and Garcia-Molina, 2001; Madhavan et al., 2008), the hidden web induced as a result of client-side scripting in general and Ajax in particular has gained very little attention so far.

Consequently, while Ajax techniques are very promising in terms of improving rich interactivity and responsiveness, Ajax-based applications may very well end up in the hidden web. We believe this is one of the main reasons people hesitate to use Ajax on their public web sites.

Crawling Ajax-based web interfaces is fundamentally more challenging than crawling classical multi-page applications. The main reason is that in the classical web model, all states are explicit, and each one corresponds to a unique URL. In Ajax-based applications, however, the state of the user interface is determined dynamically, through event-driven changes in the run-time DOM-tree. This means that simply extracting and following hyperlinks does not suffice any longer. New methods and techniques are required to dynamically analyze the complex user interface elements, events, and state changes, which leads us to our next question:

Research Question 3.2

Can Ajax*-based web applications be crawled automatically?*

Being able to crawl Ajax automatically opens up many analysis opportunities, since we gain access to different dynamic states.

User Interface Testing

Ajax-based web applications rely on stateful asynchronous client/server communication, and client-side run-time manipulation of the DOM-tree, which not only makes them fundamentally different from classical web applications, but also more error-prone and harder to test.

Traditional web testing techniques (Ricca and Tonella, 2001; Andrews et al., 2005; Di Lucca et al., 2002a; Elbaum et al., 2003) have serious limitations in testing modern Ajax-based web applications (Marchetto et al., 2008a).

Therefore, new techniques and tools are needed to test this new class of software. Whether Ajax applications can be tested automatically, is the subject of our last research question in this thesis:

Research Question 3.3

Can Ajax*-based web user interfaces be tested automatically?*

1.4 Research Method and Evaluation

Proposing new concepts, techniques, and tools to support the ideas, forms the core of our research method. There is a strong emphasis on tools and automation in this thesis. In fact, this research has resulted in the development of four tools (Retjax, Chiron, Crawljax, and Atusa), all built in Java and three already made open source.[4]

Validity of our methods and tools are assessed by extensive empirical evaluation (Wohlin et al., 2005). We use descriptive case studies as suggested by Yin (2003) and Kitchenham et al. (1995) to investigate the applicability of our techniques. Representative industrial and open source Ajax applications are used as subject systems to validate or discover limitations of the techniques.

In Chapter 4, in order to conduct a comparison of a number of data delivery techniques as accurately as possible and with minimal manual errors, we set up a controlled experiment (Wohlin et al., 2000).

Critical discussions on the findings are used to achieve analytical generalizations of the experimental results.

[4] http://spci.st.ewi.tudelft.nl/content/software/

RQ \ Chapter	2	3	4	5	6
1	✓	✓	✓		
2	✓	✓			
3	✓		✓	✓	✓
3.1	✓		✓		
3.2	✓			✓	✓
3.3	✓			✓	✓

Table 1.1 Overview of the research questions and the covering chapters.

1.5 Thesis Outline

Table 1.1 provides an overview of the questions that were investigated in this research and the corresponding chapters.

In Chapter 2, we examine a number of Ajax frameworks to understand their common architectural properties. The current state of existing client/server architectures is investigated to explore whether Ajax-based interaction models can be captured within those styles. Our analysis reveals the limitations of the classical web architectural style Rest for capturing modern Ajax-based client/server interactions. Based on these findings, a new component- and push-based architectural style is proposed, called Spiar, constituting the architectural - processing, connecting, and data - elements of Ajax applications, and the constraints that should hold between the elements to meet the desired properties. Chapter 2 sets the foundation for the rest of this thesis by describing the key characteristics of Ajax-based web architectures.

In Chapter 3, we propose a user interface migration process consisting of five major steps. First, our ajaxification approach starts by reconstructing the paths that users can follow when navigating through web pages. We propose a schema-based clustering technique to group pages that are structurally similar along the navigational path. Once a simplified model of the navigational path has been extracted, we can focus on extrapolating candidate user interface components. For this purpose, we use a differencing technique to calculate the fragment changes of browsing from one page to another. After candidate components have been identified, we can derive an Ajax representation for them. An intermediate single-page model can be opted for, from which specific Ajax implementations can be derived. Finally, through a model-driven engineering (Schmidt, 2006) approach, a meta-model can be created for each target system and the corresponding transformation between the single-page meta-model and the platform-specific language defined (Gharavi et al., 2008). The first three steps of this migration process have been implemented in a tool called Retjax, discussed in detail along with the proposed process in Chapter 3.

In Chapter 4, we investigate the challenges of setting up an automated testing environment for measuring performance data. We present Chiron, our

open source, distributed, automated testing framework and how it helps to obtain reliable data from different web application settings. Using CHIRON, Chapter 4 discusses an experiment that reveals the differences between traditional pull and COMET-based push solutions in terms of data coherence, scalability, network usage, and reliability. Such a study helps software engineers to make rational decisions concerning key parameters such as publish and pull intervals, in relation to, for instance, the anticipated number of web clients.

Chapter 5 discusses the challenges of crawling AJAX automatically, and proposes a new method for crawling AJAX through dynamic analysis of the user interface. The method, implemented in an open source tool called CRAWLJAX, automatically infers a state-flow graph of the application by, simulating user events on interface elements, and analyzing the internal DOM state, modeling the various navigational paths and states within an AJAX application. Such a crawling technique has various applications. First, it can serve as a starting point for adoption by general search engines. Second, such a crawler can be used to expose the AJAX induced hidden content on the web by automatically generating a static version of the dynamic user interface (Mesbah and van Deursen, 2008b). In addition, the ability to automatically exercise all the executable elements of an AJAX user interface gives us a powerful test mechanism, which brings us to Chapter 6.

In Chapter 6, we present an automatic testing method that can dynamically make a full pass over an AJAX application, and examine the user interface behavior. To that end, the crawler from Chapter 5 is extended with data entry point handling to trigger faults through input values. With access to different dynamic DOM states we can check the user interface against different constraints. We propose to express those as invariants on the DOM tree, and the inferred state machine, which can be checked automatically in any state. Such invariants are used as oracles, to deal with the well-known oracle problem. We present our open source testing tool called ATUSA, implementing the approach, offering generic invariant checking components, as well as a plugin-mechanism to add application-specific state validators, and test suite generation from the inferred state machine. This tool is used in a number of case studies to investigate the actual fault revealing capabilities and automation level of the tool.

1.6 Origin of Chapters and Acknowledgments

Each main chapter in this thesis is directly based on (at least) one peer reviewed publication. While all chapters have distinct core contributions, there is some redundancy, in the introduction of background material and motivation, to ensure each chapter is self-contained and can be read independent of the others.

The author of this thesis is the main contributor of all chapters except Chapter 4, in which the first two authors, Engin Bozdag and the author, contributed

equally and we chose to use an alphabetical ordering. All publications have been co-authored by Arie van Deursen.

Chapter 2 This chapter was published in the Journal of Systems and Software (JSS), in December 2008, as Mesbah and van Deursen (2008a).

An earlier version of this chapter appeared in the Proceedings of the 6^{th} Working IEEE/IFIP Conference on Software Architecture (WICSA 2007) (Mesbah and van Deursen, 2007a). Thanks to Engin Bozdag for his feedback on this chapter.

Chapter 3 This chapter was published in the Proceedings of the 11^{th} European Conference on Software Maintenance and Reengineering (CSMR 2007) as Mesbah and van Deursen (2007b).

A short version also appeared in the Proceedings of the 29^{th} International Conference on Software Engineering (ICSE 2007), Doctoral Symposium (Mesbah, 2007).

Chapter 4 This chapter has been accepted for publication in the Journal of Web Engineering (JWE), in September 2008, as Bozdag et al. (2009).

An earlier version of this chapter appeared in the Proceedings of the 9^{th} IEEE International Symposium on Web Site Evolution (WSE 2007) (Bozdag et al., 2007).

Chapter 5 This chapter was published in the Proceedings of the 8^{th} International Conference on Web Engineering (ICWE 2008) as Mesbah et al. (2008).

Chapter 6 This chapter has been accepted for publication in the Proceedings of the 31^{st} International Conference on Software Engineering (ICSE 2009), Research Papers, as Mesbah and van Deursen (2009).

This publication has won one of five ACM SIGSOFT Distinguished Paper awards, which puts it in the top 10% of the accepted papers at ICSE 2009, a conference that had an acceptance rate of 12% to begin with.

Furthermore, our research has resulted in the following publications that are not directly included in this thesis:

- Modelling and Generating Ajax Applications: A Model-Driven Approach. In Proceedings of the 7^{th} International Workshop on Web-Oriented Software Technologies (IWWOST 2008) (Gharavi et al., 2008).
- Crosscutting Concerns in J2EE Applications. In Proceedings of the 7^{th} IEEE International Symposium on Web Site Evolution (WSE 2005) (Mesbah and van Deursen, 2005).

A Component- and Push-based Architectural Style for AJAX Applications*

Chapter 2

A new breed of web application, dubbed AJAX*, is emerging in response to a limited degree of interactivity in large-grain stateless Web interactions. At the heart of this new approach lies a single page interaction model that facilitates rich interactivity. Also push-based solutions from the distributed systems are being adopted on the web for* AJAX *applications. The field is, however, characterized by the lack of a coherent and precisely described set of architectural concepts. As a consequence, it is rather difficult to understand, assess, and compare the existing approaches. We have studied and experimented with several* AJAX *frameworks trying to understand their architectural properties. In this chapter, we summarize four of these frameworks and examine their properties and introduce the* SPIAR *architectural style which captures the essence of* AJAX *applications. We describe the guiding software engineering principles and the constraints chosen to induce the desired properties. The style emphasizes user interface component development, intermediary delta-communication between client/server components, and push-based event notification of state changes through the components, to improve a number of properties such as user interactivity, user-perceived latency, data coherence, and ease of development. In addition, we use the concepts and principles to discuss various open issues in* AJAX *frameworks and application development.*

2.1 Introduction

Over the course of the past decade, the move from desktop applications towards web applications has gained much attention and acceptance. Within this movement, however, a great deal of user interactiveness has been lost. Classical web applications are based on a *multi page interface* model, in which interactions are based on a page-sequence paradigm. While simple and elegant in design for exchanging documents, this model has many limitations for developing modern web applications with user friendly human-computer interaction.

Recently, there has been a shift in the direction of web development towards the new generation of WEB 2.0 applications. A new breed of web application, dubbed AJAX (Asynchronous JavaScript And XML) (Garrett, 2005), has

*This chapter was published in the Journal of Systems and Software in December 2008 (Mesbah and van Deursen, 2008a).

been emerging in response to the limited degree of interactivity in large-grain stateless web interactions. AJAX utilizes a set of existing web technologies, previously known as *Dynamic HTML (DHTML)* and *remote scripting* (Crane et al., 2005), to provide a more interactive web-based user interface.

At the heart of this new approach lies a *single page interface* model that facilitates rich interactivity. In this model, changes are made to individual user interface components contained in a web page, as opposed to (refreshing) the entire page.

Another recent development, under the same umbrella, is applying the push-based concepts from the distributed systems to the web (Khare, 2005). For applications that require *real-time event notifications*, the client-initiated pull model is very inefficient and might lead to network congestion. The push-based style, where the server broadcasts the state changes to the clients asynchronously every time its state changes, is emerging as an alternative on the web, which is known as COMET (Russell, 2006) or Reverse AJAX (Direct Web Remoting, 2007). Each of these options has its own architectural trade-offs.

Thanks to the momentum of AJAX, the technology has attracted a strong interest in the web application development community. After the name AJAX was coined in February 2005 (Garrett, 2005), numerous frameworks[1] and libraries have appeared, many web applications have adopted one or more of the ideas underpinning AJAX, and an overwhelming number of articles in developer sites and professional magazines have appeared.

Adopting AJAX-based techniques is a serious option not only for newly developed applications, but also for ajaxifying (Mesbah and van Deursen, 2007b) existing web sites if their user friendliness is inadequate.

A software engineer considering adopting AJAX, however, is faced with a number of challenges. What are the fundamental architectural trade-offs between designing a legacy web application and an AJAX web application? How would introducing a push-based style affect the scalability of web applications? What are the different characteristics of AJAX frameworks? What do these frameworks hide? Is there enough support for designing such applications? What problems can one expect during the development phase? Will there be some sort of convergence between the many different technologies? Which architectural elements will remain, and which ones will be replaced by more elegant or more powerful solutions? Addressing these questions calls for a more abstract perspective on AJAX web applications. However, despite all the attention the technology is receiving in the web community, there is a lack of a coherent and precisely described set of architectural formalisms for AJAX enabled web applications. In this chapter we explore whether concepts and principles as developed in the software architecture research community can be of help to answer such questions.

To gain an abstract perspective, we have studied a number of AJAX frameworks, abstracted their features, and documented their common architectural elements and desired properties. In particular, we propose SPIAR, the Single

[1] At the time of writing more than 150 frameworks are listed at `http://ajaxpatterns.org/Frameworks`.

Page Internet Application aRchitectural style, which emphasizes user interface component-based development, intermediary delta-communication between client/server components, and push-based event notification of state changes through the components, to improve a number of properties such as user interactivity, user-perceived latency, data coherence, and ease of development. The style can be used when high user interaction and responsiveness are desired in web applications.

One of the challenges of proposing an architectural style is the difficulty of evaluating the success of the style. As also observed by Fielding (Fielding, 2000), the success of an architecture is ultimately determined by the question whether a system built using the style actually meets the stated requirements. Since this is impossible to determine in advance, we will evaluate our style in the following ways:

1. We investigate how well existing Ajax frameworks such as GWT or Echo2 are covered by the style;
2. We discuss how a number of typical Ajax architectures (client-centric, server centric, push-based) are covered by the style;
3. We show how the style can be used to discuss various tradeoffs in the design of Ajax applications, related to, architectural properties such as scalability and adaptability.

This chapter is organized as follows. We start out, in Section 2.2 by exploring Ajax, studying four frameworks (Google's GWT, Backbase, Echo2, and the push-based Dojo/Cometd framework) that have made substantially different design choices. Then, in Section 2.3, we survey existing architectural styles (such as the Representational State Transfer architectural style Rest on which the World Wide Web is based (Fielding and Taylor, 2002)), and analyze their suitability for characterizing Ajax. In Sections 2.4–2.7, we introduce Spiar, describing the architectural properties, elements, views of this style, and the constraints. Given Spiar, in Section 2.8 we use its concepts and principles to discuss various open issues in Ajax frameworks and application development and evaluate the style itself. We conclude with a summary of related work, contributions, and an outlook to future work.

2.2 Ajax Frameworks

Web application developers have struggled constantly with the limits of the HTML page-sequence experience, and the complexities of client-side JavaScript programming to add some degree of dynamism to the user interface. Issues regarding cross-browser compatibility are, for instance, known to everyone who has built a real-world web application. The rich user interface (UI) experience Ajax promises comes at the price of facing all such problems. Developers are required to have advanced skills in a variety of Web technologies, if they are to build robust Ajax applications. Also, much effort has to be

spent on testing these applications before going in production. This is where frameworks come to the rescue. At least many of them claim to.

Because of the momentum AJAX has gained, a vast number of frameworks are being developed. The importance of bringing order to this competitive chaotic world becomes evident when we learn that 'almost one new framework per day' is being added to the list of known frameworks.[2]

We have studied and experimented with several AJAX frameworks trying to understand their architectural properties. We summarize four of these frameworks in this section. Our selection includes a widely used open source framework called Echo2, the web framework offered by Google called GWT, the commercial package delivered by Backbase and the Dojo/Cometd push-based comet framework. All four frameworks are major players in the AJAX market, and their underlying technologies differ substantially.

2.2.1 Echo2

Echo2[3] is an open-source AJAX framework which allows the developer to create web applications using an object-oriented, UI component-based, and event-driven paradigm for Web development. Its Java *Application Framework* provides the APIs (for UI components, property objects, and event/listeners) to represent and manage the state of an application and its user interface. Echo applications can be created entirely in server-side Java code using a component-oriented and event-driven API. Server-side components are rendered into client-side code automatically, and both the client and server-side code are kept in sync.

All functionality for rendering a component or for communicating with the client browser is specifically assembled in a separate module called the *Web Rendering Engine*. The engine consists of a server-side portion (written in Java/J2EE) and a client-side portion (JAVASCRIPT). The client/server interaction protocol is hidden behind this module and as such, it is entirely decoupled from other modules. Echo2 has an *Update Manager* which is responsible for tracking updates to the user interface component model, and for processing input received from the rendering agent and communicating it to the components.

The *Echo2 Client Engine* runs in the client browser and provides a remote user interface to the server-side application. Its main activity is to synchronize client/server state when user operations occur on the interface.

A *ClientMessage* in XML format is used to transfer the client state changes to the server by explicitly stating the nature of the change and the component ID affected. The server processes the ClientMessage, updating the component model to reflect the user's actions. Events are fired on interested listeners, possibly resulting in further changes to the server-side state of the application. The server responds by rendering a *ServerMessage* which is again an

[2] `http://ajaxpatterns.org/wiki/index.php?title=AJAXFrameworks`

[3] Echo2 2.0.0, `www.nextapp.com/platform/echo2/echo/`.

XML message containing directives to perform partial updates to the DOM representation on the client.

2.2.2 GWT

Google has a novel approach to implementing its AJAX framework, the Google Web Toolkit (GWT)[4]. Just like Echo2, GWT facilitates the development of UIs in a fashion similar to AWT or Swing and comes with a library of widgets that can be used. The unique character of GWT lies in the way it renders the client-side UI. Instead of keeping the UI components on the server and communicating the state changes, GWT compiles all the Java UI components to JavaScript code (compile-time). Within the components the developer is allowed to use a subset of the Java 1.4's API to implement needed functionality.

GWT uses a small generic client engine and, using the compiler, all the UI functionality becomes available to the user on the client. This approach decreases round-trips to the server drastically. The server is only consulted if raw data is needed to populate the client-side UI components. This is carried out by making server calls to defined services in an RPC-based style. The services (which are not the same as Web Services) are implemented in Java and data is passed both ways over the network, in JSON format, using serialization techniques.

2.2.3 Backbase

Backbase[5] is an Amsterdam-based company that provided one of the first commercial AJAX frameworks. The framework is still in continuous development, and in use by numerous customers world wide.

A key element of the Backbase framework is the Backbase Client Run-time (BCR). This is a standards-based AJAX engine written in JavaScript that runs in the web browser. It can be programmed via a declarative user interface language called XEL. XEL provides an application-level alternative to JavaScript and manages asynchronous operations that might be tedious to program and manage using JavaScript.

BCR's main functionality is to:

- create a single page interface and manage the widget tree (view tree),
- interpret JavaScript as well as the XEL language,
- take care of the synchronization and state management with the server by using delta-communication, and asynchronous interaction with the user through the manipulation of the representational model.

The Backbase framework provides a markup language called Backbase Tag Library (BTL). BTL offers a library of widgets, UI controls, a mechanism for

[4] http://code.google.com/webtoolkit/

[5] `http://www.backbase.com`

attaching actions to them, as well as facilities for connecting to the server asynchronously.

The server side of the Backbase framework is formed by BJS, the Backbase Jsf Server. It is built on top of JavaServer Faces (Jsf)[6], the new J2EE presentation architecture. Jsf provides a user interface component-based framework following the model-view-controller pattern. Backbase Jsf Server utilizes all standard Jsf mechanisms such as validation, conversion and event processing through the JSF life-cycle phases. The interaction in Jsf is, however, based on the classical page sequence model, making integration in a single page framework non trivial. Backbase extends the JSF request life-cycle to work in a single-page interface environment. It also manages the server-side event handlers and the server-side control tree.

Any Java class that offers getters and setters for its properties can be directly assigned to a UI component property. Developers can use the components declaratively (web-scripting) to build an Ajax application. The framework renders each declared server-side UI component to a corresponding client-side XEL UI component, and keeps track of changes on both component trees for synchronization.

The state changes on the client are sent to the server on certain defined events. Backbase uses DOM events to delegate user actions to BCR which handles the events asynchronously. The events can initiate a client-side (local) change in the representational model but at the same time these events can serve as triggers for server-side event listeners. The server translates these state changes and identifies the corresponding component(s) in the server component tree. After the required action, the server renders the changes to be responded to the engine, again in XEL format.

2.2.4 Dojo and Cometd

The final framework we consider is the combination of the client-side Dojo and the server-side Cometd frameworks, which together support a push-based client-server communication. The framework is based on the Bayeux protocol which the Cometd group[7] has recently released, as a response to the lack of communication standards. For more details see Chapter 4 and (Bozdag, 2007).

The Bayeux message format is defined in JSON (JavaScript Object Notation),[8] which is a data-interchange format based on a subset of the JavaScript Programming Language. The protocol has recently been implemented and included in a number of web servers including Jetty[9] and IBM Websphere.[10]

The frameworks that implement Bayeux currently provide a connection type called *Long Polling* for HTTP push. In Long Polling, the server holds on to the client request, until data becomes available. If an event occurs, the

[6] JavaServer Faces Specification v1.1, http://java.sun.com/j2ee/javaserverfaces/

[7] `http://www.cometd.com`

[8] `http://www.json.org`

[9] `http://www.mortbay.org`

[10] `http://www-306.ibm.com/software/websphere/`

server sends the data to the client and the client has to reconnect. Otherwise, the server holds on to the connection for a finite period of time, after which it asks the client to reconnect again. If the data publish interval is low, the system will act like a pure pull, because the clients will have to reconnect (make a request) often. If the data publish interval is high, then the system will act like a pure push.

Bayeux defines the following phases in order to establish a Comet connection. The client:

1. performs a handshake with the server and receives a client ID,
2. sends a connection request with its ID,
3. subscribes to a channel and receives updates.

Bayeux is supported by the client-side Ajax framework called Dojo.[11] It is currently written entirely in JavaScript and there are plans to adopt a markup language in the near future. Dojo provides a number of ready-to-use UI widgets which are prepackaged components of JavaScript code, as well as an abstracted wrapper (`dojo.io.bind`) around various browsers' implementations of the XMLHttpRequest object to communicate with the server. Dojo facilitates the `dojo.io.cometd` library, to make the connection handshake and subscribe to a particular channel.

On the server-side, Bayeux is supported by Cometd.[12] This is an HTTP-based event routing framework that implements the Comet style of interaction. It is currently implemented as a module in Jetty.

2.2.5 Features

While different in many ways, these frameworks share some common architectural characteristics. Generally, the goals of these frameworks can be summarized as follows:

- Hide the complexity of developing Ajax applications - which is a tedious, difficult, and error-prone task,
- Hide the incompatibilities between different web browsers and platforms,
- Hide the client/server communication complexities,
- All this to achieve rich interactivity and portability for end users, and ease of development for developers.

The frameworks achieve these goals by providing a library of user interface components and a development environment to create reusable custom

[11] `http://dojotoolkit.org`
[12] `http://www.cometd.com`

components. The architectures have a well defined protocol for small interactions among known client/server components. Data needed to be transferred over the network is significantly reduced. This can result in faster response data transfers. Their architecture takes advantage of client side processing resulting in improved user interactivity, smaller number of round-trips, and a reduced web server load.

The architectural decisions behind these frameworks change the way we develop web applications. Instead of thinking in terms of sequences of Web pages, Web developers can now program their applications in the more intuitive (single page) component- and event-based fashion along the lines of, e.g., AWT and Swing.

2.3 Architectural Styles

In this section, we first introduce the architectural terminology used in this chapter and explore whether styles and principles as developed in the software architecture research community, and specifically those related to network-based environments, can be of help in formalizing the architectural properties of AJAX applications.

2.3.1 Terminology

In this chapter we use the software architectural concepts and terminology as used by Fielding (Fielding, 2000) which in turn is based on the work of Perry and Wolf (Perry and Wolf, 1992). Thus, a software architecture is defined (Perry and Wolf, 1992) as a configuration of architectural elements — processing, connectors, and data — constrained in their relationships in order to achieve a desired set of architectural properties.

An architectural style, in turn, (Fielding, 2000) is a coordinated set of architectural constraints that restricts the roles of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style. An architectural style constrains both the design elements and the relationships among them (Perry and Wolf, 1992) in such a way as to result in software systems with certain desired properties. Clements et al. (Clements et al., 2002) define an architectural style as a specialization of element and relation types, together with a set of constraints on how they can be used. A style can also be seen as an abstraction of recurring composition and interaction characteristics in a set of architectures.

An architectural system can be composed of multiple styles and a style can be a hybrid of other styles (Shaw and Garlan, 1996). Styles can be seen as reusable (Monroe and Garlan, 1996) common architectural patterns within different system architectures and hence the term *architectural pattern* is also used to describe the same concept (Bass et al., 2003).

The benefits of using styles can be summarized as follows:

- Design reuse: well-understood solutions applied to new problems

- Code reuse: shared implementations of invariant aspects of a style
- Understandability and ease of communication: phrases such as 'client-server' or 'REST' make use of a vocabulary conveying a wealth of implicit information.
- Interoperability: supported by style standardization
- Specific trade-off analysis: enabled by the constrained design space
- Visualizations: specific depictions matching mental models

In our view, being able to understand the tradeoffs inherent in the architectures (Kazman et al., 1998) of AJAX systems is the essence of using architectural styles. An architectural style enables us to pin-point relevant tradeoffs in different instantiated architectures.

2.3.2 Existing Styles

Client/server (Sinha, 1992), n-tier (Umar, 1997), and Mobile Code (Carzaniga et al., 1997; Fuggetta et al., 1998), are all different network-based architectural styles (Fielding, 2000), which are relevant when considering the characteristics of AJAX applications.

In addition, user interface applications generally make use of popular styles such as Module/View/Controller (Krasner and Pope, 1988) to describe large scale architecture and, in more specific cases, styles like C2 (Taylor et al., 1996) to rely on asynchronous notification of state changes and request messages between independent components.

There are also a number of interactional styles, such as event observation and notification (Rosenblum and Wolf, 1997), publish/subscribe (Eugster et al., 2003), the component and communication model (Hauswirth and Jazayeri, 1999), and ARRESTED (Khare and Taylor, 2004), which model the client/server push paradigm for distributed systems.

In our view, the most complete and appropriate style for the Web, thus far, is the REpresentational State Transfer (REST) (Fielding and Taylor, 2002). REST emphasizes the abstraction of data and services as resources that can be requested by clients using the resource's name and address, specified as a Uniform Resource Locator (URL) (Berners-Lee et al., 1994). The style inherits characteristics from a number of other styles such as client/server, pipe-and-filter, and distributed objects.

The style is a description of the main features of the Web architecture through architectural constraints which have contributed significantly to the success of the Web.

It revolves around five fundamental notions: a *resource* which can be anything that has identity, e.g., a document or image, the *representation of a resource* which is in the form of a media type, *synchronous request-response interaction* over HTTP to obtain or modify representations, a *web page* as an instance of the application state, and *engines* (e.g., browser, crawler) to move from one state to the next.

REST specifies a client-stateless-server architecture in which a series of proxies, caches, and filters can be used and each request is independent of the previous ones, inducing the property of scalability. It also emphasizes a uniform interface between components constraining information to be transferred in a standardized form.

2.3.3 A Style for Ajax

AJAX applications can be seen as a hybrid of desktop and web applications, inheriting characteristics from both worlds. Table 2.1 summarizes the differences between what REST provides and what modern AJAX (with COMET) applications demand. AJAX frameworks provide back-end services through UI components to the client in an event-driven or push style. Such architectures are not so easily captured in REST, due to the following differences:

- While REST is suited for large-grain hypermedia data transfers, because of its uniform interface constraint it is not optimal for small data interactions required in AJAX applications.
- REST focuses on a hyper-linked resource-based interaction in which the client requests a specific *resource*. In contrast, in AJAX applications the user interacts with the system much like in a desktop application, requesting a response to a specific *action*.
- All interactions for obtaining a resource's representation are performed through a synchronous request-response pair in REST. AJAX applications, however, require a model for asynchronous communication.
- REST explicitly constrains the server to be stateless, i.e., each request from the client must contain all the information necessary for the server to understand the request. While this constraint can improve scalability, the tradeoffs with respect to network performance and user interactivity are of greater importance when designing an AJAX architecture.
- REST is cache-based while AJAX facilitates real-time data retrieval.
- Every request must be initiated by a client, and every response must be generated immediately; every request can only generate a single response (Khare and Taylor, 2004). COMET requires a model which enables pushing data from the server to the client.

These requirement mismatches call for a new architectural style capable of meeting the desired properties.

2.4 Architectural Properties

The architectural properties of a software architecture include both the functional properties achieved by the system and non-functional properties, often

REST provides	AJAX demands
Large-grain hypermedia data transfers	Small data interactions
Resource-based	UI component-based
Hyper-linked	Action- Event-based
Synchronous request-response	Asynchronous interaction
Stateless	Stateful
Cache-based	Real-time data retrieval
Poll-based	Poll and Push

Table 2.1 What REST provides versus what AJAX demands.

referred to as quality attributes (Bass et al., 2003; Offutt, 2002). The properties could also be seen as requirements since architecting a system requires an understanding of its requirements.

Below we discuss a number of architectural properties that relate to the essence of AJAX. Other properties, such as extensibility or security, that may be desirable for any system but are less directly affected by a decision to adopt AJAX, are not taken into account. Note that some of the properties discussed below are related to each other: for instance, user interactivity is influenced by user-perceived latency, which in turn is affected by network performance.

2.4.1 User Interactivity

The Human-computer interaction literature defines interactivity as the degree to which participants in a communication process have control over, and can exchange roles in their mutual discourse. User interactivity is closely related to *usability* (Folmer, 2005), the term used in software architecture literature. Teo *et al.* (Teo et al., 2003) provide a thorough study of user interactivity on commercial web applications. Their results suggest that an increased level of interactivity has positive effects on user's perceived satisfaction, effectiveness, efficiency, value, and overall attitude towards a Web site. Improving this property on the Web has been the main motivating force behind the AJAX movement.

2.4.2 User-perceived Latency

User-perceived latency is defined as the period between the moment a user issues a request and the first indication of a response from the system. Generally, there are two primary ways to improve user-perceived performance. First, by reducing the round-trip time, defined as time elapsed for a message from the browser to a server and back again, and second, by allowing the user to interact asynchronously with the system. This is an important property in all distributed applications with a front-end to the user.

2.4.3 Network Performance

Network performance is influenced by *throughput* which is the rate of data transmitted on the network and *bandwidth*, i.e., a measure of the maximum available throughput. Network performance can be improved by means of reducing the amount and the granularity of transmitted data.

2.4.4 Simplicity

Simplicity or development effort is defined as the effort that is needed to understand, design, implement, maintain and evolve a web application. It is an important factor for the usage and acceptance of any new approach.

2.4.5 Scalability

In distributed environments scalability is defined by the degree of a systems ability to handle a growing number of components. In Web engineering, a system's scalability is determined, for instance, by the degree to which a client can be served by different servers without affecting the results. A scalable Web architecture can be easily configured to serve a growing number of client requests.

2.4.6 Portability

Software that can be used in different environments is said to be portable. On the Web, being able to use the Web browser without the need for any extra actions required from the user, e.g., downloading plug-ins, induces the property of portability.

2.4.7 Visibility

Visibility (Fielding, 2000) is determined by the degree to which an external mediator is able to understand the interactions between two components, i.e., the easier it is for the mediator to understand the interactions, the more visible the interaction between the two components will be. Looking at the current implementations of AJAX frameworks, visibility in the client/server interactions is low, as they are based on proprietary protocols. Although a high level of visibility makes the interaction more comprehensible, the corresponding high observability can also have negative effects on security issues. Thus low visibility is not per se an inferior characteristic, depending on the desired system property and tradeoffs made.

2.4.8 Reliability

Reliability is defined as the continuity of correct service (Avizienis et al., 2004). The success of any software system depends greatly on its reliability. On the Internet, web applications that depend on unreliable software and do not

work well, will lose customers (Offutt, 2002). Testing (test automation, unit and regression testing) resources can improve the reliability level of an application. However, web applications are generally known to be poorly tested compared to traditional desktop applications. In addition to the short time-to-market pressure, the multi-page interaction style of the web makes it difficult to test. Adopting a server-side component-based style of web application development can improve the testability of the system and as a consequence its reliability. Note that the dynamic nature of AJAX makes the client-side actually harder to test (see Chapter 6); However, if the server-side code responsible for the user interface is implemented in a component-based style, the functionality is easier to test as separate modules.

2.4.9 Data Coherence

An important aspect of real-time event notification of web data that need to be available and communicated to the user as soon as they happen, e.g., stock prices, is the maintenance of data coherence (Bhide et al., 2002). A piece of data is defined as coherent, if the data on the server and the client is synchronized. In web applications adhering to the HTTP protocol, clients need to frequently pull the data based on a pre-defined interval. In contrast, servers that adopt push capability maintain state information pertaining to clients and stream the changes to users as they happen. These two techniques have different properties with respect to the data coherence achieved (Bozdag et al., 2007).

2.4.10 Adaptability

Adaptability is defined as the ease with which a system or parts of the system may be adapted to the changing environment. In web applications, an architecture that allows changes on the server to be propagated to the clients is called adaptable. We use the notion of *code mobility* (Fuggetta et al., 1998) to compare the dynamic behavior of different AJAX architectures in terms of changeability and adaptability. Mobile code is, generally, software code obtained from remote servers, transferred across a network, and then downloaded and executed on the client without explicit installation or execution by the recipient.

2.5 Spiar Architectural Elements

Following (Fielding, 2000; Perry and Wolf, 1992), the key architectural elements of SPIAR are divided into three categories, namely processing (components), data, and connecting elements. An overview of the elements is shown in Figure 2.1. In this section we explain the elements themselves, while in the next section we discuss their interaction.

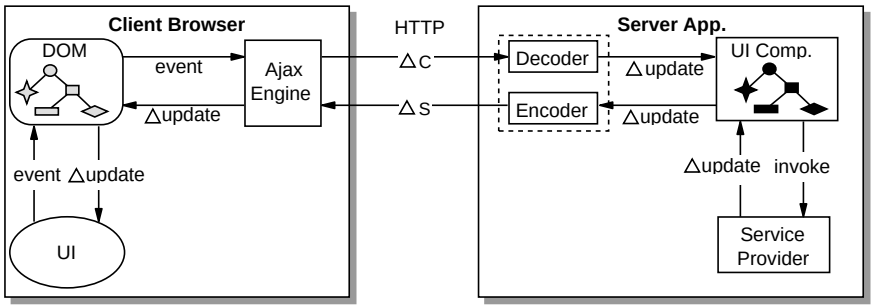

Figure 2.1 Processing View of a SPIAR-based architecture.

2.5.1 Processing Elements

The processing elements are defined as those components that supply the transformation on the data elements.

The *Client Browser* offers support for a set of standards such as HTTP, HTML, Cascading Style Sheets, JavaScript, and Document Object Model. It processes the representational model of a web page to produce the user interface. The user interaction can be based on a single page user interface model. All the visual transitions and effects are presented to the user through this interface. Just like a desktop client application, it consists of a single main page with a set of identifiable widgets. The properties of widgets can be manipulated individually while changes are made in-place without requiring a page refresh.

The AJAX *Engine* is a client engine that loads and runs in the client browser. There is no need for a plug-in for the web application to function. However, downloading the engine does introduce an initial latency for the user which can be compensated by the smaller data transfers once the engine is in place. The engine is responsible for the initialization and manipulation of the representational model. As can be seen in Figure 2.1, the engine handles the events initiated by the user, communicates with the server, and has the ability to perform client-side processing.

The *Server Application* resides on the server and operates by accepting HTTP-based requests from the network, and providing responses to the requester. All server-side functionality resides in the server application processing element.

The *Service Provider* represents the logic engine of the server and processes state changes and user requested actions. It is capable of accessing any resource (e.g., database, Web Services) needed to carry out its action. A Service Provider's functionality is invoked by event listeners, attached to components, initiated by incoming requests.

The *Delta Encoder/Decoder* processes outgoing/incoming delta messages. It is at this point that the communication protocol between the client and the server is defined and hidden behind an interface. This element supports delta

communication between client and server which improves user-perceived latency and network performance.

UI Components consist of a set of server-side UI components. The component model on the server is capable of rendering the representational model on the client. Each server-side component contains the data and behavior of that part of the corresponding client-side widget which is relevant for state changes; There are different approaches as when and how to render the client-side UI code. GWT, for instance, renders the entire client-side UI code compile-time from the server-side Java components. Echo2 which has a real component-based architecture, on the other hand, renders the components at run-time and keeps a tree of components on both client and server side. These UI components have event listeners that can be attached to client-side user initiated events such as clicking on a button. This element enhances simplicity by providing off-the-shelf components to build web applications.

A *Push Server* resides as a separate module on the server application. This processing element has the ability to keep an HTTP connection open to push data from the server to the client. The Service Provider can publish new data (state changes) to this element.

A *Push Client* element resides within the client. It can be a separate module, or a part of the AJAX Engine. This element can subscribe to a particular channel on the Push Server element and receive real-time publication data from the server.

2.5.2 Data Elements

The data elements contain the information that is used and transformed by the processing elements.

The *Representation* element consists of any media type just like in REST. HTML, CSS, and images are all members of this data element.

The *Representational Model* is a run-time abstraction of how a UI is represented on the client browser. The Document Object Model inside the browser has gained a very important role in AJAX applications. It is through dynamically manipulating this representational model that rich effects have been made possible. Some frameworks such as Backbase use a domain-specific language to declaratively define the structure and behavior of the representational model. Others like GWT use a direct approach by utilizing JavaScript.

Delta communicating messages form the means of the delta communication protocol between client and server. SPIAR makes a distinction between the client delta data (DELTA-CLIENT) and the server delta data (DELTA-SERVER). The former is created by the client to represent the client-side state changes and the corresponding actions causing those changes, while the latter is the response of the server as a result of those actions on the server components. The delta communicating data are found in a variety of formats in the current frameworks, e.g., XML, JavaScript Object Notation (JSON), or pure JavaScript. The client delta messages contain the needed information for the server to know for instance which action on which component has to be carried out.

```
REQUEST (DELTA-CLIENT):

POST http://demo.nextapp.com/Demo/app/Demo/app?serviceId=Echo.Synchronize
Content-Type: text/xml; charset=UTF-8
<client-message trans-id="1">
 <message-part processor="EchoAction">
  <action component-id="c_7" name="click"/>
 </message-part>
</client-message>

RESPONSE (DELTA-SERVER):

<?xml version="1.0" encoding="UTF-8"?>
<server-message
 xmlns="http://www.nextapp.com/products/echo2/svrmsg/servermessage"
 trans-id="2">
 <message-part-group id="update">
  ...
  <message-part processor="EchoDomUpdate.MessageProcessor">
   <dom-add>
    <content parent-id="c_35_content">
     <div id="c_36_cell_c_37" style="padding:0px;">
      <span id="c_37" style="font-size:10pt;font-weight:bold;">
        Welcome to the Echo2 Demonstration Application.
      </span>
     </div>
    </content>
   </dom-add>
  </message-part>
  ...
 </message-part-group>
</server-message>
```

Figure 2.2 An example of Echo2 delta-communication.

As an example, Figure 2.2 illustrates delta-communication in Echo2. After the user clicks on a component (button) with ID c_7, the client-side engine detects this `click` event and creates the DELTA-CLIENT (in Echo2 called `client-message`) and posts it to the server as shown in the REQUEST part of Figure 2.2. The server then, using the information in the DELTA-CLIENT which, in this case, is composed of the action, component ID, and the event type, responds with a DELTA-SERVER in XML format. As can be seen, the DELTA-SERVER tells the client-side engine exactly how to update the DOM state with new style and textual content on a particular parent component with ID `c_35_content`.

We distinguish between three types of code that can change the state of the client: *presentational code*, *functional code*, and *textual data*.

Presentational code as its name suggests has influence on the visual style and presentation of the application, e.g., CSS, or HTML. Textual data is simply pure data. The functional code can be executed on the client, e.g., JavaScript code, or commands in XML format (e.g., `dom-add` in Figure 2.2). The DELTA-SERVER can be composed of any of these three types of code. These three types of code can influence the Representational model (DOM) of the client application which is the run-time abstraction of the presentational code, executed functional code and textual data.

GWT uses an RPC style of calling services in which the DELTA-SERVER is mainly composed of textual data, while in Backbase and Echo2 a component-based approach is implemented to invoke event listeners, in a mixture of presentational and functional code.

2.5.3 Connecting Elements

The connecting elements serve as the glue that holds the components together by enabling them to communicate.

Events form the basis of the interaction model in SPIAR. An event is initiated by each action of the user on the interface, which propagates to the engine. Depending on the type of the event, a request to the server, or a partial update of the interface might be needed. The event can be handled asynchronously, if desired, in which case the control is immediately returned to the user.

On the server, the request initiated by an event *invokes* a service. The service can be either invoked directly or through the corresponding UI component's event listeners.

Delta connectors are light-weight communication media connecting the engine and the server using a request/response mechanism over HTTP.

Delta updates are used to update the representational model on the client and the component model on the server to reflect the state changes. While a delta update of the representational model results in a direct apparent result on the user interface, an update of the component model invokes the appropriate listeners. These updates are usually through procedural invocations of methods.

Channels are the connecting elements between the push consumer and producer. A consumer (receiver) *subscribes* to a channel, through a handshake procedure, and receives any information that is sent on the channel by the producer (information source) as *delta push server*.

2.6 Architectural Views

Given the processing, data, and connecting elements, we can use different architectural views to describe how the elements work together to form an architecture. Here we make use of two processing views, which concentrate on the data flow and some aspects of the connections among the processing elements with respect to the data (Fielding, 2000). Such views fit in the Components and Connectors viewtype as discussed by Clements et al. (Clements et al., 2002). We discuss one processing view for a pure component-based AJAX solution, one for an RPC-based Ajax application, and one view for the push-based variant.

2.6.1 Ajax view

Figure 2.1 depicts the processing view of an SPIAR-based architecture based on run-time components rendering as in, e.g., Echo2. The view shows the

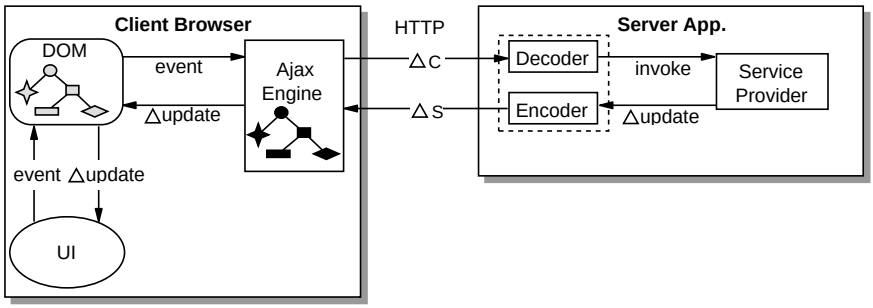

Figure 2.3 Processing View of an RPC-based GWT architecture.

interaction of the different components 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 run-time processing view of the GWT framework is depicted in Figure 2.3. As can be seen, GWT does not maintain a server-side component tree. Instead the server-side UI components are transformed into client-side components at compile-time. The client engine knows the set and location of all available components at run-time. The RPC-based interaction with the server is however still conducted in a delta-communication style. Here, the encoder and decoder talk directly to the Service Provider without going through the server-side component model.

Note that each framework uses a different set of SPIAR's architectural elements to present the run-time architectural processing view. See also Section 2.8.1 for a discussion on how each approach fits in SPIAR.

2.6.2 Comet view

In the majority of the current COMET frameworks the data is pushed directly to the client as shown in Figure 2.4. This direct approach is fine for implementations that are not component-based. However, for UI component-based frameworks, if the push data is directly sent to the client, the client has to

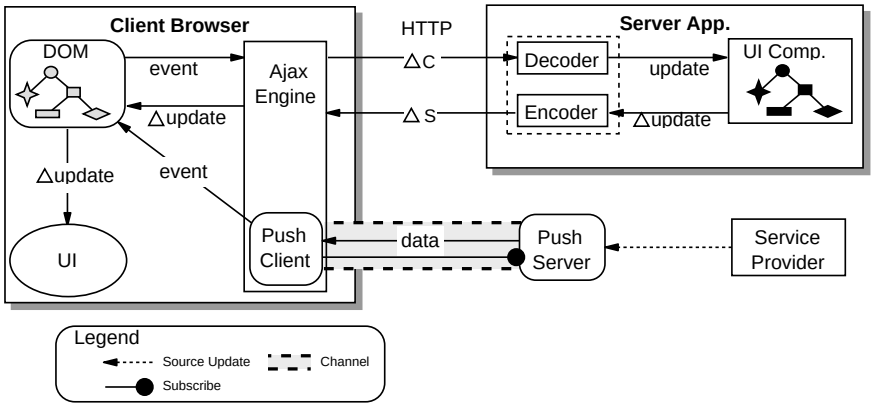

Figure 2.4 Processing View of a push-based SPIAR architecture.

handle this data itself and update its components locally. To notify UI components on the server, the client has to send a client delta back to the server. This is inefficient, since in many cases, the push server and the application server are in the same machine or network. The SPIAR architectural style thus reveals an interesting tension between the UI component-based and the push-based constraint.

A possible solution (Bozdag, 2007) would be to take a short-cut for this synchronization process. Whenever an event arrives from the Service Provider (state change, new data), instead of publishing the new data directly to the client, first the state changes are announced to the UI Components for all the subscribed clients. The changes are then passed through the encoder to the Push Server and then passed as push delta-server to the push client.

This approach makes sure that the state on the server is synchronized with the state on the client for each notification. Figure 2.5 depicts our proposed push-based view. The push client subscribes to a particular channel through the push server, and the changes are passed, through the component model, as push delta server, real-time to the client.

Note that the normal interaction of the client/server as depicted on Figure 2.1 can continue unhindered by the introduction of this push module. There are two advantages of this solution. First of all, it allows ΔS to be sent directly to the user in one step.

Second advantage is the simplicity for the application programmer. Without this solution, the programmer has to write explicit JavaScript functions in order to process the incoming push data. In the proposed solution, no such function is needed, since the response will be in an expected ΔS format, which will be processed by the AJAX Engine automatically.

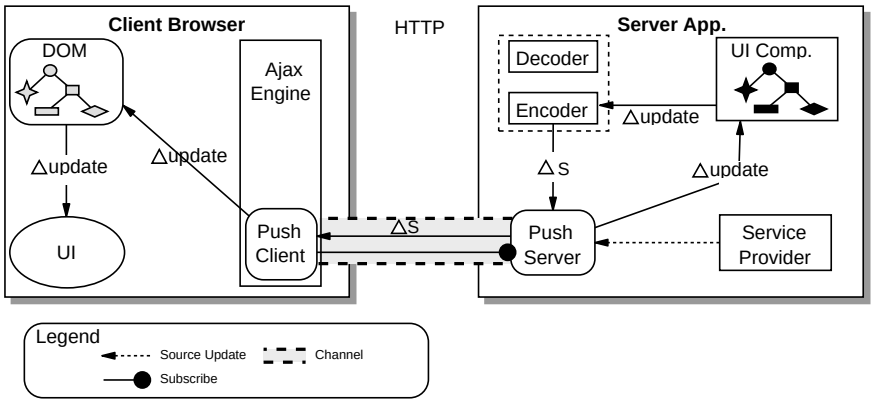

Figure 2.5 Proposed push-based integration.

2.7 Architectural Constraints

Architectural constraints can be used as restrictions on the roles of the architectural elements to induce the architectural properties desired of a system. Table 2.2 presents an overview of the constraints and induced properties. A "+" marks a direct positive effect, whereas a "–" indicates a direct negative effect.

SPIAR rests upon the following constraints chosen to retain the properties identified previously in this chapter.

2.7.1 Single Page Interface

SPIAR is based on the client-server style which is presumably the best known architecture for distributed applications, taking advantage of the separation of concerns principle in a network environment. The main constraint that distinguishes this style from the traditional Web architecture is its emphasis on a single page interface instead of the page-sequence model. This constraint induces the property of user interactivity. User interactivity is improved because the interaction is on a component level and the user does not have to wait for the entire page to be rendered again as a result of each action. Figure 2.6 and Figure 2.7 show the interaction style in a traditional web and in a single-page client-centric AJAX application respectively.

2.7.2 Asynchronous Interaction

AJAX applications are designed to have a high user interactivity and a low user-perceived latency. *Asynchronous interaction* allows the user to, subsequently, initiate a request to the server at any time, and receive the control back from the client instantly. The requests are handled by the client at the background and the interface is updated according to server responses. This

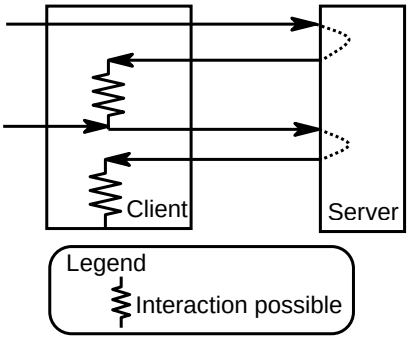

Figure 2.6 Traditional multi-page Web Interaction.

model of interaction is substantially different from the classic synchronous request, wait for response, and continue model.

2.7.3 Delta-communication

Redundant data transfer which is mainly attributed to retransmissions of unchanged pages is one of the limitations of classic web applications. Many techniques such as caching, proxy servers and fragment-based resource change estimation and reduction (Bouras and Konidaris, 2005), have been adopted in order to reduce data redundancy. Delta-encoding (Mogul et al., 1997) uses caching techniques to reduce network traffic. However, it does not reduce the computational load since the server still needs to generate the entire page for each request (Naaman et al., 2004).

Spiar goes one step further, and uses a *delta-communication* style of interaction. Here merely the state changes are interchanged between the client and the server as opposed to the full-page retrieval approach in classic web applications. Delta-communication is based on delta-encoding architectural principles but is different: delta-communication does not rely on caching and as a result, the client only needs to process the deltas. All Ajax frameworks hide the delta-communication details from the developers.

This constraint induces the properties of network performance directly and as a consequence user-perceived latency and user interactivity. Network performance is improved because there are less redundant data (merely the delta) being transported. Data coherence is also improved because of the fine-grain nature of the data which can be transferred to the user faster than when dealing with data contained in large-grain web pages.

2.7.4 User Interface Component-based

Spiar relies on a single page user interface with components similar to that of desktop applications, e.g., AWT's UI component model. This model defines the state and behavior of UI components and the way they can interact.

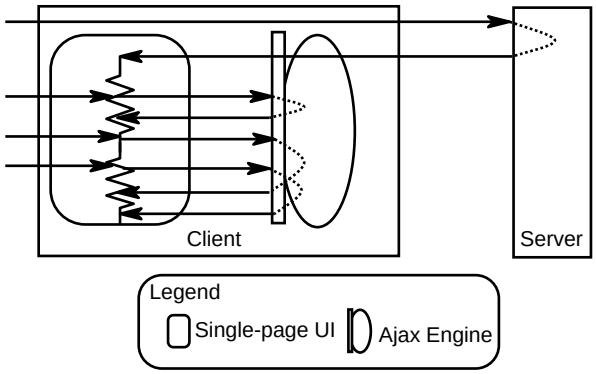

Figure 2.7 Client-centric AJAX Interaction.

UI component programming improves simplicity because developers can use reusable components to assemble a Web page either declaratively or programmatically. User interactivity is improved because the user can interact with the application on a component level, similar to desktop applications. In addition, testing component-based software is inherently easier than testing traditional page-based web applications, which induces the property of reliability.

Frameworks adhering to this constraint are very adaptable in terms of code mobility since state changes in the three code types (2.5.2) can be propagated to the client.

2.7.5 Web standards-based

Constraining the Web elements to a set of standardized formats is one way of inducing portability on the Web. This constraint excludes approaches that need extra functionality (e.g., plug-ins, virtual machine) to run on the Web browser, such as Flash and Java applets, and makes the client cross-browser compatible. This constraint limits the nature of the data elements to those that are supported by web browsers. Also using web standards, web browsers that abide by standards are easily supported and hence some degree of reliability is induced (Avizienis et al., 2004).

2.7.6 Client-side Processing

Client-side processing improves user interactivity and user-perceived latency through round-trip reduction. For instance, client-side form validation reduces unnecessary server-side error reports and reentry messages. Additionally, some server-side processing (e.g., sorting items) can be off-loaded to clients using mobile code that will improve server performance and increase the availability to more simultaneous connections. As a tradeoff, client performance can become an issue if many widgets need processing resources

	User Interactivity	User-perceived Latency	Network Performance	Simplicity	Scalability	Portability	Visibility	Data Coherence	Reliability	Adaptability
Single-page Interface	+									
Asynchronous Interaction	+	+								
Delta Communication	+	+	+		−		−	+		
Client-side processing	+	+	+							
UI Component-based	+			+					+	+
Web standards-based				+		+			+	
Stateful	+	+	+		−		−			
Push-based Publish/Subscribe		+	+		−		−	+		+

Table 2.2 Constraints and induced properties

on the client. GWT takes advantage of client-side processing to the fullest, by generating all the UI client-side code as JavaScript and run it on the client.

2.7.7 Stateful

A stateless server is one which treats each request as an independent transaction, unrelated to any previous request, i.e., each request must contain all of the information necessary to understand it, and cannot take advantage of any stored context on the server (Fielding and Taylor, 2002). Even though the Web architecture and HTTP are designed to be stateless, it is difficult to think of stateless Web applications. Within a Web application, the order of interactions is relevant, making interactions depend on each other, which requires an awareness of the overall component topology. The statefulness is imitated by a combination of HTTP, client-side cookies, and server-side session management.

Unlike Rest, Spiar does not constrain the nature of the state explicitly. Nevertheless, since a stateless approach may decrease network performance (by increasing the repetitive data), and because of the component-based nature of the user interactions, a stateful solution might become favorable at the cost of scalability and visibility.

2.7.8 Push-based Publish/Subscribe

The client-server interaction can be realized in both a push- or pull-based style. In a push-based style (Hauswirth and Jazayeri, 1999), the server broadcasts the state changes to the clients asynchronously every time its state changes. Event-based Integration (Barrett et al., 1996) and Asynchronous Rest (Khare and Taylor, 2004) are event-based styles allowing asynchronous notifi-

cation of state changes by the server. This style of interaction has mainly been supported in peer-to-peer architectural environments.

In a pull-based style, client components actively request state changes. Event-driven (Newman and Sproull, 1979) architectures are found in distributed applications that require asynchronous communication, for instance, a desktop application, where user initiated UI inputs serve as the events that activate a process.

COMET enables us to mimic a push-based publish/subscribe (Eugster et al., 2003) style of interaction on the web. This ability improves the network performance (see Chapter 4) because unnecessary poll requests are avoided. User-perceived latency, and adaptability are also improved by allowing a real-time event notification of state changes to clients. The results of our empirical study in Chapter 4 show that data coherence is improved significantly by this constraint, but at the same time the server application performance and reliability can be deteriorated and as a result scalability can be negatively influenced.

2.8 Discussion and Evaluation

In this section we evaluate SPIAR by investigating how well existing AJAX frameworks and typical AJAX architectures are covered by the style, and discuss various decisions and tradeoffs in the design of AJAX applications in terms of the architectural properties.

2.8.1 Retrofitting Frameworks onto Spiar

Each framework presented in Section 2.2 can be an architectural instance of SPIAR, even if not fully complying with all the architectural constraints of SPIAR. *Echo2* is the best representative of SPIAR because of its fully event-driven and component-based architecture. The JSF-based *Backbase* architecture is also well covered by SPIAR even though JSF is not a real event-based approach. *GWT*, on the other hand, is an interesting architecture. Although the architecture uses UI components during the development phase, these components are compiled to client-side code. GWT does not rely on a server-side component-based architecture and hence, does not fully comply with SPIAR. None of these three frameworks has push-based elements. While the push-based constraint is well represented in the *Dojo and Cometd* framework, the component-based constraint is missing here.

SPIAR abstracts and combines the component- and push-based styles of these AJAX frameworks into a new style.

2.8.2 Typical Ajax Configurations

Many industrial frameworks have started supporting the AJAX style of interaction on the web. However, because of the multitude of these systems it is difficult to capture their commonalities and draw sharp lines between their

	User Interactivity	User-perceived Latency	Network Performance	Simplicity	Scalability	Portability	Visibility	Data Coherence	Reliability	Adaptability
REST-based Classic Web	−	−	−	+	+	+	+	−	+−	−
Client-centric AJAX	+	+		−		+				−
ARPC AJAX	+	+	+	+			−			−
Push-based AJAX		+	+		−		−	+		+
SPIAR-based AJAX	+	+	+	+	+−	+	−	+	+	+

Table 2.3 AJAX configurations and properties.

main variations. Using SPIAR as a reference point, commonalities and divergences can be identified.

Table 2.3 shows a number of AJAX configurations along with the induced architectural properties. The first entry is the REST-based classic Web configuration. While simple and scalable in design, it has, a very low degree of responsiveness, high user-perceived latency, and there is a huge amount of redundant data transferred over the network.

The second configuration is the Client-centric AJAX. Most AJAX frameworks started by focusing on the client-side features. Frameworks such as Dojo, Ext,[13] and jQuery[14] all provide rich UI widgets on the client, facilitating a client-centric style of interaction in which most of the functionality is off-loaded to the browser. Generally, an interaction between components that share the same location is considered to have a negligible cost when compared to interaction that is carried out through a network (Carzaniga et al., 1997). This variant provides a high degree of user interactivity and very low user-perceived latency. There is, however, no support for adaptability as all the code is off-loaded to the client and that makes this variant static in terms of code changes from the server.

Frameworks such as GWT, DWR,[15] and JSON-RPC-Java[16] support the *Asynchronous Remote Procedure Call* (ARPC) style of interaction. In this configuration, all the presentational and functional code is off-loaded to the browser and the server is only asynchronously contacted in case of a change in terms of raw textual data. Low user-perceived latency, high user interactivity and reduced server round-trips are the characteristics of this configuration. Even though the textual data can be dynamically requested from the server, there is a limited degree of adaptability for the presentational and functional code.

The fourth configuration is a pure push-based interaction in which the state changes are streamed to the client (by keeping a connection alive), with-

[13] http://extjs.com
[14] http://jquery.com
[15] http://getahead.org/dwr
[16] http://oss.metaparadigm.com/jsonrpc/

out any explicit request from the client. High level of data coherence and improved network performance compared to the traditional pull style on the web are the main positive properties of this variant. High server load and scalability issues are mainly due to the fact that the server has to maintain state information about the clients and the corresponding connections.

For the sake of comparison, the last entry in Table 2.3 presents the component and push-based SPIAR style itself.

2.8.3 Issues with push Ajax

Scalability is the main issue in a push model with a traditional server model. COMET uses persistent connections, so a TCP connection between the server and the client is kept alive until an explicit disconnect, timeout or network error. So the server has to cope with many connections if the event occurs infrequently, since it needs to have one or more threads for every client. This will bring problems on scaling to thousands of simultaneous users. There is a need for better event-based tools on the server. According to our latest findings (Bozdag et al., 2009), push can handle a higher number of clients if new techniques, such as the continuations (Jetty, 2006) mechanism, are adopted by server applications. However, when the number of users increases, the reliability in receiving messages decreases.

The results of our empirical study (Bozdag et al., 2009) show that push provides high data coherence and high network performance, but at the same time a COMET server application consumes more CPU cycles as in pull.

A stateful server is more resistant to failures, because the server can save the state at any given time and recreate it when a client comes back. A push model, however, due to its list of subscribers is less resilient to failures. The server has to keep the state, so when the state changes, it will broadcast the necessary updates. The amount of state that needs to be maintained can be large, especially for popular data items (Bhide et al., 2002). This extra cost of maintaining a state and a list of subscribers will also have a negative effect on scalability.

These scalability issues are also inherited by SPIAR as can be seen in Table 2.3.

2.8.4 Resource-based versus Component-based

The architecture of the World Wide Web (W3C Technical Architecture Group, 2004) is based on resources identified by Uniform Resource Identifiers (URI), and on the protocols that support the interaction between agents and resources. Using a generic interface and providing identification that is common across the Web for resources has been one of the key success factors of the Web.

The nature of Web architecture which deals with Web pages as resources causes redundant data transfers (Bouras and Konidaris, 2005). The delta-communication way of interaction in SPIAR is based on the component level

and does not comply with the Resource/URI constraint of the Web architecture. The question is whether this choice is justifiable. To be able to answer this question we need to take a look at the nature of interactions within single page applications: safe versus unsafe interactions.

2.8.5 Safe versus Unsafe Interactions

Generally, client/server interactions in a Web application can be divided into two categories of *Safe* and *Unsafe* interactions (W3C, 2004). A safe interaction is one where the user is not to be held accountable for the result of the interaction, e.g., simple queries with GET, in which the state of the resources (on the server) is not changed. An unsafe interaction is one where a user request has the potential to change the state of the resources, such as a POST with parameters to change the database.

The web architecture proposes to have unique resource-based addressing (URL) for safe interactions, while the unsafe ones do not necessarily have to correspond to one. One of the issues concerning Ajax applications is that browser history and bookmarks of classic web applications are broken if not implemented specifically. In Ajax applications, where interaction becomes more and more desktop-like, where eventually *Undo/Redo* replaces *Back/Forward*, the safe interactions can remain using specific addressing while the unsafe ones (POST requests) can be carried out at the background. Both variants use delta-communication, however, the safe interactions should have unique addressing and the unsafe one do not necessarily correspond to any Rest-based resource identified by a URL.

To provide the means of linking to the safe operations in Ajax, the URI's *fragment identifier* (the part after # in the URL) can be adopted. Interpretation of the fragment identifier is then performed by the engine that dereferences a URI to identify and represent a state of the application. Libraries such as the jQuery history/remote plugin[17] or the Really Simple History[18] support ways of programatically registering state changes with the browser history through the fragment identifier.

2.8.6 Client- or server-side processing

Within the current frameworks it is not possible for developers to choose whether some certain functionality should be processed on the client or on the server. How the computation is distributed can be an important factor in tuning a web application. Ajax frameworks architectures should provide the means for the developer to decide if and to what extent computation should be done on the client. Also adopting adaptive techniques to choose between the server or client for processing purposes needs more attention.

[17] http://stilbuero.de/jquery/history/

[18] http://code.google.com/p/reallysimplehistory/

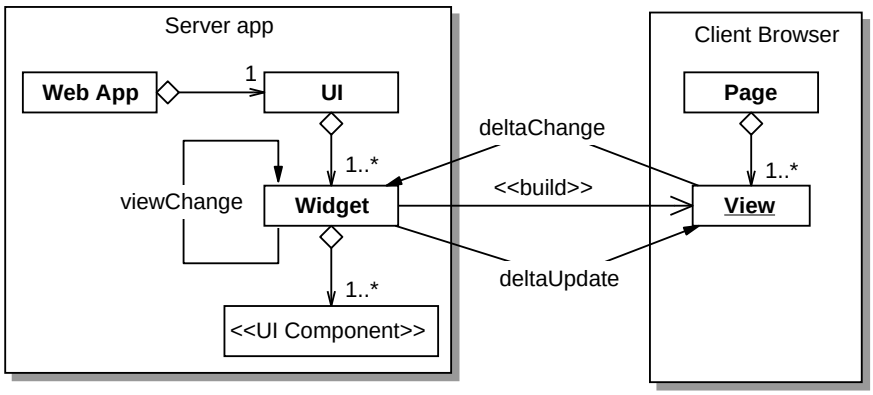

Figure 2.8 A single page web application composed of UI components.

2.8.7 Asynchronous Synchronization

The asynchronous interaction in AJAX applications may cause race conditions if not implemented with care. The user can send a request to the server before a previous one has been responded. In a server processor that handles the requests in parallel, the second request can potentially be processed before the first one. This behavior could have drastic effects on the synchronization and state of the entire application. A possible solution would be handling the event-triggered requests for each client sequentially at the cost of server performance.

2.8.8 Communication Protocol

As we have seen, currently each AJAX framework has implemented its own specific communication protocol. This makes the visibility of client/server interactions poor as one must know the exact protocol to be able to make sense of the delta messages. It also results in a low level of portability for these applications. For a client to be able to communicate with an AJAX server, again it needs to know the protocol of that server application. These two properties can be improved by defining a standard protocol specification for the communication by and for the AJAX community.

If we look at the current push approaches, we see different techniques on achieving the push solution itself, but also different measures to deal with portability. Without a standard here, it will be difficult for a mediator to understand the interactions between system components, therefore the system itself will be less visible. The definition and adoption of the BAYEUX protocol is a first attempt in the right direction which will improve both visibility and portability.

2.8.9 Design Models

Figure 2.8 shows a meta-model of an AJAX web application. The UI is composed of widgets of UI components. The client single page is built by the server-side widgets. Delta changes as well as view changes occur on the widget level. A view change, can be seen as navigating through the available widgets. AJAX frameworks should provide clear navigational models for developers. Research is needed to propose design models for AJAX developers by for instance extending the UML language to model user interaction, navigation through components, asynchronous/synchronous actions and client versus server side processing.

2.8.10 Scope of Spiar

The essential requirements for an AJAX application are speed of execution and improved user experience, small size of client/server data, and very specific interaction behavior. SPIAR is a coordinated set of architectural constraints that attempts to minimize user-perceived latency and network usage, and improve data coherence and ultimately user experience. Because of these properties, the components of AJAX frameworks are tightly coupled. Loose coupling is thus not a property included in SPIAR. This inherent tight coupling also encompasses some scalability tradeoffs.

The style focuses on the front-end of the new breed of web applications, i.e., the Service Provider is an abstract component that could be composed of middle en back-end software. Service-oriented architecture solutions could therefore easily be combined with SPIAR, e.g., by replacing the Service Provider with a SOAP server. SPIAR elaborates only those parts of the architecture that are considered indispensable for AJAX interaction.

2.9 Related Work

While the attention for rich Internet applications in general and AJAX in particular in professional magazines and Internet technology related web sites has been overwhelming, few research papers have been published on the topic so far.

A number of technical books have appeared on the subject of developing AJAX applications. Asleson and Schutta (2005), for instance, focus primarily on the client side aspects of the technology and remain 'pretty agnostic' to the server side. Crane et al. (2005) provide an in-depth presentation of AJAX web programming techniques and prescriptions for best practices with detailed discussions of relevant design patterns. They also mention improved user experience and reduced network latency by introducing asynchronous interactions as the main features of such applications. While these books focus mainly on the implementation issues, our work examines the architectural design decisions and properties from an abstraction level by focusing on the interactions between the different client/server components.

The push-based style has received extensive attention within the distributed systems research community. However, most of the work focuses on client/server distributed systems and non-HTTP multimedia streaming or multi-casting with a single publisher (Franklin and Zdonik, 1998; Hauswirth and Jazayeri, 1999). The only work that currently focuses on AJAX is the white-paper of Khare (2005). Khare discusses the limits of the pull approach and proposes a push-based approach for AJAX. However, the white-paper does not evaluate possible issues with this push approach, such as scalability and performance. Their work on the `mod_pubsub` event router over HTTP (Khare et al., 2002) is highly related to the concepts of AJAX push implementations.

The page-sequence model of the traditional web architecture makes it difficult to treat portions of web pages (fragments), independently. Fragment-based research (Bouras and Konidaris, 2005; Brodie et al., 2005; Challenger et al., 2005) aims at providing mechanisms to efficiently assemble a web page from different parts to be able to cache the fragments. Recently proposed approaches include several server-side and cache-side mechanisms. Server-side techniques aim at reducing the load on the server by allowing reuse of previously generated content to serve user requests. Cache-based techniques attempt to reduce the latency by moving some functionality to the edge of the network. These fragment-based techniques can improve network and server performance, and user-perceived latency by allowing only the modified or new fragments to be retrieved. Although the fragments can be retrieved independently, these techniques lack the user interface component interactivity required in interactive applications. The UI component-based model of the SPIAR style in conjunction with its delta-communication provides a means for a client/server interaction based on state changes that does not rely on caching.

The SPIAR style itself draws from many existing styles (Khare and Taylor, 2004; Newman and Sproull, 1979; Sinha, 1992; Taylor et al., 1996) and software fields (Fielding, 2000; Mogul et al., 1997; Perry and Wolf, 1992), discussed and referenced in the chapter. Our work relates closely to the software engineering principles of the REST style (Fielding and Taylor, 2002). While REST deals with the architecture of the Web (W3C Technical Architecture Group, 2004) as a whole, SPIAR focuses on the specific architectural decisions of AJAX frameworks.

Parsons (2007) provides an overview of the current state of the web by exploring the evolving web architectural patterns. After the literature on the core patterns of traditional web application architectures is presented, the paper discusses some new emerging patterns, by focusing on the recent literature on WEB 2.0 in general and AJAX in particular.

On the architectural styles front the following styles can be summarized: Pace (Suryanarayana et al., 2004) an event- based architectural style for trust management in decentralized applications, TIGRA (Emmerich et al., 2001) a distributed system style for integrating front-office systems with middle- and back-office applications, and Aura (Sousa and Garlan, 2002) an architectural framework for user mobility in ubiquitous environments which uses models

of user tasks as first class entities to set up, monitor and adapt computing environments.

Khare and Taylor (2004) also evaluate and extend REST for decentralized settings and represent an event-based architectural style called ARRESTED. The asynchronous extension of REST, called A+REST, permits a server to broadcast notifications of its state changes to 'watchers'.

Recently, Erenkrantz et al. (2007) have re-evaluated the REST style for new emerging web architectures. They have also come to the conclusion that REST is silent on the area that AJAX expands. They recognize the importance of the AJAX engine which is seen as the interpretation environment for delivered content. They also notice, that REST's goal was to reduce server-side state load, while AJAX reduces server-side computational load by adopting client-side processing, and increases responsivity. Their new style extends REST and is called Computational REST (CREST). CREST requires a transparent exchange of computation so that the client no longer is seen as merely a presentation agent for delivered content; 'it is now an execution environment explicitly supporting computation'. In other words, CREST much like SPIAR recognizes the significance of the AJAX engine as a processing component. On the other hand, CREST ignores other important architectural characteristics of AJAX applications, such as the the delta-communication and asynchronous interaction covered in SPIAR.

2.10 Concluding Remarks

AJAX is a promising solution for the design and implementation of responsive rich web applications, since it overcomes many of the limitations of the classical client-server approach. However, most efforts in this field have been focused on the implementation of different AJAX tools and frameworks, with little attention to the formulation of a conceptual architecture for the technology.

In this chapter we have discussed SPIAR, an architectural style for AJAX. The contributions of this chapter are in two research fields: web application development and software architecture.

From a software architecture perspective, our contribution consists of the use of concepts and methodologies obtained from software architecture research in the setting of AJAX web applications. This chapter further illustrates how the architectural concepts such as properties, constraints, and different types of architectural elements can help to organize and understand a complex and dynamic field such as single page AJAX development. In order to do this, this chapter builds upon the foundations offered by the REST style, and offers a further analysis of this style for the purpose of building web applications with rich user interactivity.

From a web engineering perspective, our contribution consists of an evaluation of different variants of AJAX client/server interactions, the SPIAR style itself, which captures the guiding software engineering principles that practi-

tioners can use when constructing and analyzing Ajax applications and evaluating the tradeoffs of different properties of the architecture. We further propose a component- push-based architecture capable of synchronizing the events both on the server and the client efficiently.

The style is based on an analysis of various Ajax frameworks and configurations, and we have used it to address various design tradeoffs and open issues in Ajax applications.

Ajax development field is young, dynamic and changing rapidly. Certainly, the work presented in this chapter needs to be incrementally enriched and revised, taking into account experiences, results, and innovations as they emerge from the web community.

Future work encompasses the use of Spiar to analyze and influence Ajax developments. One route we foresee is the extension of Spiar to incorporate additional models for representing, e.g., navigation or UI components, thus making it possible to adopt a model-driven approach to Ajax development. At the time of writing, we are using Spiar in the context of enriching existing web applications with Ajax capabilities.

Migrating Multi-page Web Applications to Single-page AJAX Interfaces*

Chapter 3

Recently, a new web development technique for creating interactive web applications, dubbed AJAX*, has emerged. In this new model, the single-page web interface is composed of individual components which can be updated/replaced independently. If until a year ago, the concern revolved around migrating legacy systems to web-based settings, today we have a new challenge of migrating web applications to single-page* AJAX *applications. Gaining an understanding of the navigational model and user interface structure of the source application is the first step in the migration process. In this chapter, we explore how reverse engineering techniques can help analyze classic web applications for this purpose. Our approach, using a schema-based clustering technique, extracts a navigational model of web applications, and identifies candidate user interface components to be migrated to a single-page* AJAX *interface. Additionally, results of a case study, conducted to evaluate our tool, are presented.*

3.1 Introduction

Despite their enormous popularity, web applications have suffered from poor interactivity and responsiveness towards end users. Interaction in classic web applications is based on a multi-page interface model, in which for every request the entire interface is refreshed.

Recently, a new web development technique for creating interactive web applications, dubbed AJAX (Asynchronous JAVASCRIPT And XML) (Garrett, 2005), has emerged. In this new model, the single-page web interface is composed of individual components which can be updated/replaced independently, so that the entire page does not need to be reloaded on each user action. This, in turn, helps to increase the levels of interactivity, responsiveness and user satisfaction.

Adopting AJAX-based techniques is a serious option not only for newly developed applications, but also for existing web sites if their user friendliness is inadequate. Many organizations are beginning to consider migration (ajaxification) possibilities to this new paradigm which promises rich interactivity and satisfaction for their clients. As a result, the well-known problem of legacy migration is becoming increasingly important for web applications.

*This chapter was published in the Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007) (Mesbah and van Deursen, 2007b).

If until a year ago, the problem revolved around migrating legacy systems to web applications, today we have a new challenge of migrating classic web applications to single-page web applications.

The main question addressed in this chapter is how to identify appropriate candidate single-page components from a page sequence interface web application. Obtaining a clear understanding of the navigational model and user interface structure of the source application is an essential step in the migration process.

In this chapter, we present a reverse engineering technique for classification of web pages. We use a schema-based clustering technique to classify web pages with similar structures. These clusters are further analyzed to suggest candidate user interface components for the target AJAX application.

The rest of this chapter is organized as follows. We start out, in Section 3.2 by exploring AJAX and focusing on its characteristics. Section 3.3 presents the overall picture of the migration process. Section 3.4 describes our page classification notion and proposes a schema-based clustering approach. Section 3.5 outlines how we identify candidate user interface components. The implementation details of our tool, called RETJAX, are explained in Section 3.6. Section 3.7 evaluates a sample web application and its recovered navigational and component model by applying RETJAX. Section 3.8 discusses the results and open issues. Section 3.9 covers related work. Finally, Section 3.10 draws conclusions and presents future work.

3.2 Single-page Meta-model

Figure 3.1 shows a meta-model of a single-page AJAX web application which is composed of widgets. Each widget, in turn, consists of a set of user interface components. The specific part of the meta-model is target specific, i.e., each AJAX framework provides a specific set of UI components at different levels of granularity. The client side page is composed of client-side views, which are generated by the server-side widgets/components. Navigation is through view changes. For each view change, merely the state changes are interchanged between the client and the server, as opposed to the full-page retrieval approach in multi-page web applications.

The architectural decisions behind AJAX change the way we develop web applications. Instead of thinking in terms of sequences of Web pages, Web developers can now program their applications in the more intuitive single-page user interface (UI) component-based fashion along the lines of, for instance, Java AWT and Swing.

3.3 Migration Process

What we would like to achieve is support in migration from a multi-page web application to a single-page AJAX interface. In this section we describe the steps needed in such a process.

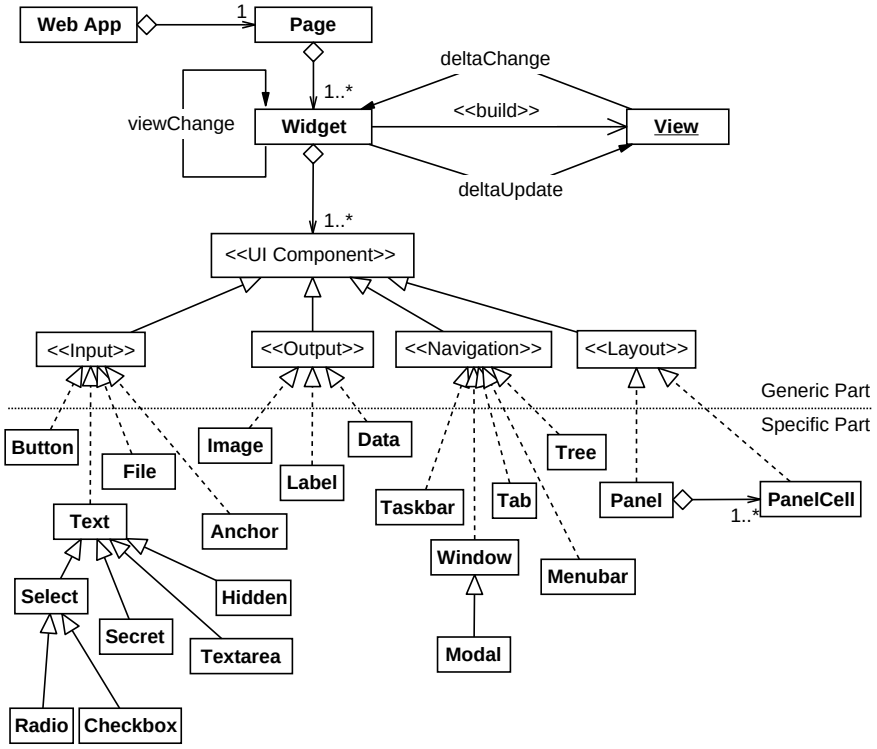

Figure 3.1 The meta-model of a single-page AJAX application composed of UI components.

Figure 3.2 depicts an overall view of the migration process. Note that we are primarily focusing on the user interface and not on the server-side code (which is also an essential part of a migration process). The user interface migration process consists of five major steps:

1. Retrieving Pages
2. Navigational Path Extraction
3. UI Component Model Identification
4. Single-page UI Model Definition
5. Target UI Model Transformation

Below we briefly discuss each of these steps. The main focus of this chapter is on steps two and three, i.e., finding candidate user interface components to be able to define a single-page user interface model. Nevertheless, we will shortly present how we envision the other steps which are currently part of our ongoing research.

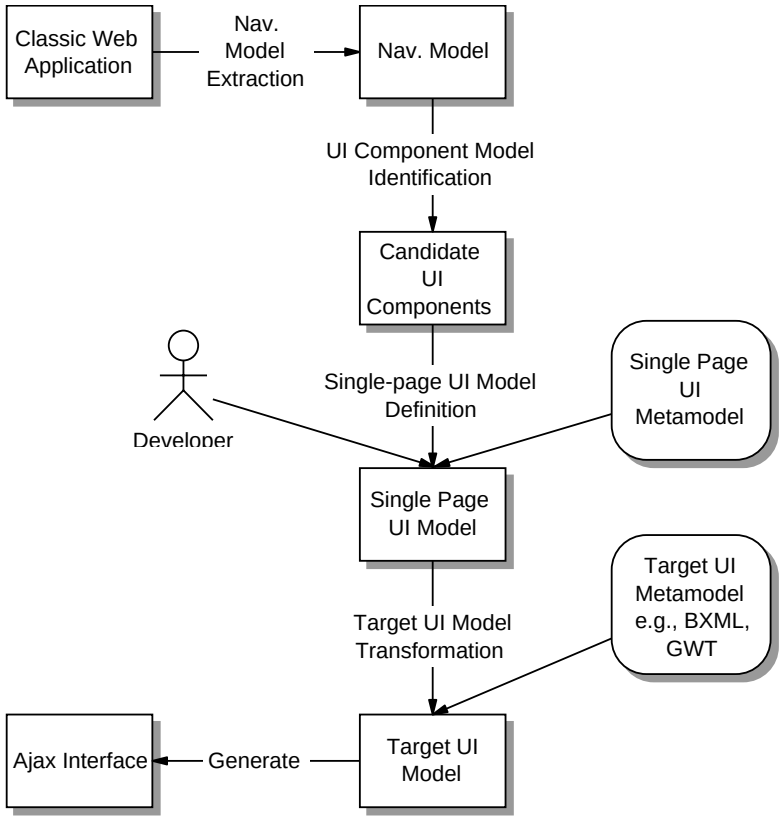

Figure 3.2 Reverse Engineering Classic Web Applications to Ajax Interfaces.

3.3.1 Retrieving Pages

Looking at dynamic web applications from an end-user's perspective enables us to gain an understanding of the application without having to cope with the many different server-side web programming languages. Building a run-time mirror-copy of the web pages can be carried out by applying static as well as dynamic analysis techniques. Static analysis can examine the pages and find `href` links to other internal pages. Dynamic analysis can help up to retrieve pages which require specific request parameters (form-based), for instance, through scenario-based test cases or collecting traces of user actions (e.g., sessions, input data) interacting with the web application. It is clear that the more our retrieved mirror-copy resembles the actual web application, the better our navigational path and UI component identification will be.

3.3.2 Navigational Path Extraction

In order to migrate from a classic web application (source) to a single-page AJAX interface (target), we first need to gain an understanding of the navi-

gational and structural model of the source application. A *navigational path* is the route a user can take while browsing a web application, following links on the pages. For ajaxification, gaining an understanding of this navigational path is essential to be able to model the navigation in the single-page user interface model. For instance, knowing that Category pages link with Product Item List pages, implies that in our single-page model, we need a Category UI component which can navigate to the Product Item List UI component.

While browsing a web application, the structural changes, for certain pages, are so minor that we can instantly notice we are browsing pages belonging to a certain category e.g., Product List. Classifying these similar pages into a group, simplifies our navigational model. Our hypothesis is that such a classification also provides a better model to search for candidate user interface components.

3.3.3 UI Component Model Identification

Within web applications, navigating from one page (A) to another (B) usually means small changes in the interface. In terms of HTML source code, this means a great deal of the code in A and B is the same and only a fraction of the code is new in B. It is this new fraction that we humans distinguish as change while browsing the application.

Speaking in terms of Ajax components, this would mean that instead of going from page A to B to see the interface change, we can simply update that part of A that needs to be replaced with the new fraction from B. Thus, this *fraction* of code from page B, becomes a UI *component* on its own in our target system.

The identified list of candidate components along with the navigational model will help us define a single-page user interface model.

3.3.4 Single-page UI Model Definition

Once we have identified candidate components, we can derive an Ajax representation for them. We have opted for an intermediate single page model, from which specific Ajax implementations can be derived.

A starting point for such a model could be formed by user interface languages such as XUL,[1] XIML (Puerta and Eisenstein, 2002), and UIML (Abrams et al., 1999). However, most such languages are designed for static user interfaces with a fixed number of UI components and are less suited for modeling dynamic interfaces as required in Ajax.

We are currently working on designing an abstract single-page user interface meta-model for Ajax applications. This abstract model should be capable of capturing dynamic changes, navigational paths as needed in such applications, and abstract general Ajax components, e.g., Button, Window, Modal, as depicted in Figure 3.1.

[1] `http://www.mozilla.org/projects/xul/`

3.3.5 Target UI Model Transformation

For each target system, a meta-model has to be created and the corresponding transformation between the single-page meta-model language and the platform-specific language defined. The advantage of having an abstract user interface model is that we can transform it to different AJAX settings. We have explored a number of AJAX frameworks such as Backbase, Echo2, and GWT, and have started conducting research to adopt a model-driven approach to AJAX (Gharavi et al., 2008).

3.4 Navigational Path Extraction

Our ajaxification approach starts by reconstructing the paths that users can follow when navigating between web pages. This requires that we group pages that are sufficiently similar and directly reachable from a given page. For example, a web page *A* could contain 7 links, 3 of which are similar. We cluster those 3 pages, and look if the links contained in those 3 pages, together, could be clustered, and so on. This way we build clusters along with the navigational paths.

In this section, we discuss our specific notion of web page similarity, and the steps that we follow to compute the clusters.

3.4.1 Page Classification

Web pages can be classified in many different ways depending on the model needed for the target view. Draheim et al. (Draheim et al., 2005) list some of possible classification notions. In this chapter, our target view focuses on *structural* classification. Tonella and Ricca (Ricca and Tonella, 2001; Tonella and Ricca, 2004) present three relevant notions of classification:

- *Textual Identity* considers two pages the same if they have exactly the same HTML code,
- *Syntactical Identity* groups pages with exactly same structure, ignoring the text between tags, according to a comparison of the syntax trees,
- *Syntactical Similarity* classifies pages with similar structure, according to a similarity metric, computed on the syntax trees of the pages.

Textual and Syntactical Identity classification notions have limited capabilities in finding pages that belong to a certain category as they look for exact matches. Syntactical Similarity is the notion that can help us cluster pages into useful groups by defining a similarity threshold under which two pages are considered clones. We propose a new approach based on *schema-based similarity*.

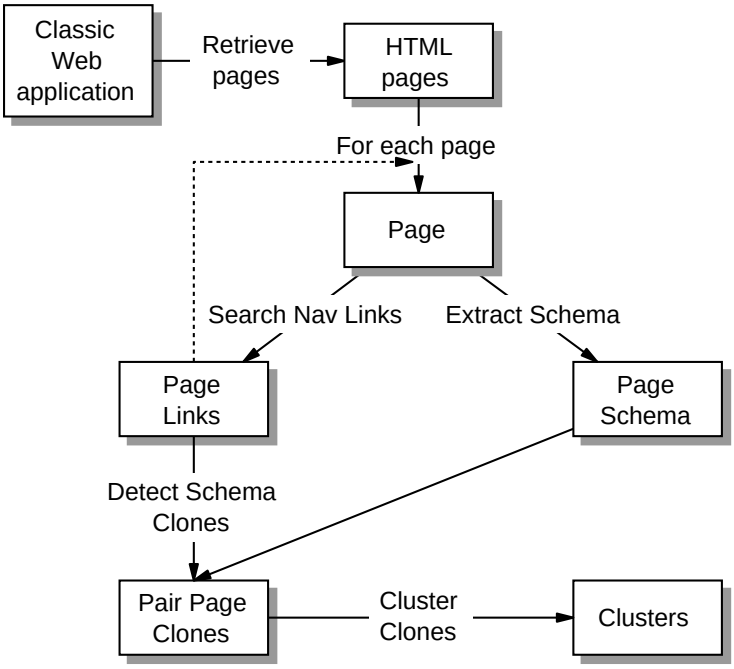

Figure 3.3 Schema-based clustering process.

3.4.2 Schema-based Similarity

Many web clustering approaches (Di Lucca et al., 2002b; De Lucia et al., 2004a; Ricca and Tonella, 2003) base their similarity degree on the computation of the edit distance between the syntax trees of web pages. This approach, although useful, has a limited capability to group HTML pages that have similar presentational structure.

For instance, consider two pages, the first page having two table rows and the second seven rows with the same structure and number of cells. On the screen, we instantly classify these two under one category, but the edit distance of these two pages could be quite high and as thus the classification metric would not classify them in one cluster. Increasing the metric threshold is not an option because that results in an increase in the number of incorrectly combined pages.

To overcome this issue, our approach relies on a comparison of the explicit schemas of pages. This means, instead of comparing the syntax trees of pages, we first reverse engineer the schemas of the pages and then compute the edit distance of the corresponding schemas. Two pages are considered clones if their schemas are similar. Going back to our example, the two pages can now be clustered correctly as a table with two row elements has the same schema as a table with seven row elements.

3.4.3 Schema-based Clustering

Given the schema-based similarity metric, we can create a schema-based clustering of web pages. We take a tree-based approach for recovering and presenting the navigational paths. In a tree structure with a set of linked nodes, each node represents a web page with zero or more child nodes and edges represent web links to other pages. We believe tree structures can provide a simple but clear and comprehensible abstract view of the navigational path for web applications.

Figure 3.3 shows our schema-based clustering process. Starting from a given root node (e.g., index.html), the goal is to extract the navigational path by clustering similar pages on each navigational level.

It is important to note that we do not conduct clustering of all pages at once. Instead we walk along the navigational path and for each node we cluster the pages that are linked with that node. It is important to cluster along with the navigational path, because we would like to recover the changes in the interface and later identify candidate UI components. If we cluster all pages as if they were on a single level, the navigational information will be lost and that is what we try to avoid.

Algorithm 1 shows how on each level the schemas of linked pages are compared. The search is *depth-first*. For each page on the navigational path, recursively, first the internal links (i.e., links to pages within the web application) are extracted. Afterwards, for each found link, the corresponding page is retrieved and converted to XHTML. The XHTML instance is then examined to extract an explicit schema. The variables used in the algorithm are local variables belonging to the page being processed at that level.

After the schemas are extracted, we conduct a pairwise comparison of the schemas to find similar structures. The structural edit distance between two schemas is calculated using the Levenshtein (Levenshtein, 1996) method. After this step, the `connected` set contains a list of cloned pair pages (e.g., `{(a-b), (b-c), (d-e)}`).

To perform the classification of pages, we provide a practical way in which the actual computation of the clusters, given a set of clone pairs, i.e., `connected`, is simply done by taking the *transitive closure* of the clone relation (De Lucia et al., 2005). In this approach, there is no need to define the number of clusters in advance. The result of calling the `transclos` function on our given example would be `{(a-b-c), (d-e)}`.

Our tool also supports an agglomerative hierarchical manner of classification. Hierarchical clustering algorithms, however, require the desired number of clusters to be defined in advance.

3.4.4 Cluster Refinement/Reduction

Because our search is depth-first, after the first page classification phase, we can further refine the clusters in order to obtain what we call the *simplified navigational path* (SNP).

Algorithm 1

```
1: procedure start (Page p)
2: Set L ← extractLinks(p)
3: for i = 0 to L.size − 1 do
4:    pl[i] ← retrievePage(L(i))
5:    px[i] ← convertToXHTML(pl[i])
6:    ps[i] ← extractSchema(px[i])
7:    start(pl[i])
8: end for
9: Set connected ← ∅
10: for i = 0 to L.size − 1 do
11:    for j = i + 1 to L.size − 1 do
12:       if distance(ps[i], ps[j]) < threshold then
13:          connected ← connected ∪ clone(pl[i], pl[j])
14:       end if
15:    end for
16: end for
17: Set clusters ← transclos(connected)
18: write(p, clusters)
19: end procedure
```

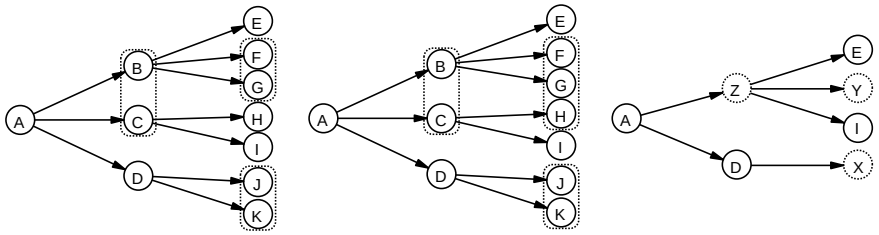

Figure 3.4 Refinement and reduction of clusters.

Beginning at the root node, for each cluster c that contains two or more pages, we examine all outgoing linked pages (from all pages in c) to determine whether further refinement of the classification is possible on the next levels of the navigational path. This is done by applying the same classification technique as explained in 3.4.3.

To simplify the navigational path, we reduce each c to a node containing only one page. For that, we presently use the simple but effective approach to choose the largest page as the reduced cluster page. A more elegant (but more expensive) solution would be to replace the cluster c by a page that is composed by extracting all common elements of the pages in c. These common elements can be computed using the shortest common supersequence algorithm (Barone et al., 2001).

From left to right, Figure 3.4 presents, the initial classification, the refined classification in which pages F, G, and H are classified into a cluster, and the

simplified navigational path (SNP) in which $Z = B \cup C$, $Y = F \cup G \cup H$, and $X = J \cup K$. Note that this refinement computation is performed until no more clusters are identified.

3.5 UI Component Identification

As mentioned before, our goal is to determine which parts of the web interface change as we browse from one page to another. These changes in the interface, along the navigational path, form the list of candidate components.

3.5.1 Differencing

Once a simplified model of the navigational path has been extracted, we can focus on extrapolating candidate user interface components. Using the SNP obtained in the previous step, Algorithm 2 describes how we calculate the fragment changes using a differencing approach.

Algorithm 2

```
1:  procedure begin (Page p)
2:  pr ← removeTexualContent(p)
3:  pp ← prettyPrint(pr)
4:  compare(pp)
5:  end procedure
6:
7:  procedure compare (Page p)
8:  Set L ← getLinksOnSNP(p)
9:  for i = 0 to L.size − 1 do
10:    prl ← removeTexualContent(L(i))
11:    ppl ← prettyPrint(prl)
12:    candidate[i] ← Diff (p ↦ ppl)
13:    compare(ppl)
14: end for
15: end procedure
```

Starting from the root node, we compare the current page (A) with all the pages on the next level on the SNP to find changes between A and those linked pages.

We use a modified version of the *Diff* method. This method only returns changes of going from A to B that are found on B, ignoring the changes on A.

To be able to conduct proper comparisons we remove all textual content, i.e., all text between the HTML tags in both pages A and B. We also pretty print both pages by writing each opening and closing tag on a separate line.

The result is a list of candidate components in HTML code along the navigational path.

3.5.2 Identifying Elements

The list of candidate user interface components can be used, for instance, to gain a visual understanding of the changes while browsing.

The code for a candidate user interface component can also provide us useful information as what sort of HTML elements it is composed of. The elements used in the candidate components can lead us towards our choice of single-page UI components.

To that end, the content of each candidate component is examined by parsing and searching for elements of interest, (e.g., Button, Text, Textarea, Select) which can be converted to the corresponding single-page instances.

Thus, the end result of this step is a mapping between legacy HTML elements and candidate single-page user interface components.

3.6 Tool Implementation: Retjax

We have implemented the navigational path extraction and Ajax component identification approach as just described in a prototype tool called Retjax (Reverse Engineer To Ajax). Retjax is written entirely in Java 5 and is based on a number of open-source libraries. A beta version of the tool will be made available from our software engineering site `swerl.tudelft.nl`.

Retjax implements the following steps:

Parsing & Extracting Links.
The first step consists of parsing HTML pages and extracting internal links. *HTML Parser*[2] is used for his purpose. It is also modified to pretty-print pages which is a prerequisite for the differencing step.

Cleaning up.
For cleaning up faulty HTML pages and converting them to well-formed XHTML instances, *JTidy*[3], a Java port of the HTML syntax checker *HTML Tidy*, is used. A well-formed XHTML page is required by the schema extractor step.

Schema Extraction.
EXTRACT (Garofalakis et al., 2000) and DTDGenerator[4] are tools that can be used to automatically detect and generate a Document Type Definition (DTD) from a set of well-formed XML document instances. We have chosen and modified DTDGenerator to extract the schema of the XHTML version of the pages. DTDGenerator takes an XML document and infers the corresponding DTD. It creates an internal list of all the elements and attributes that appear in the page, noting how they are nested, and which elements contain character data. This list is used to generate the corresponding DTD according to some pattern matching rules.

[2] http://htmlparser.sourceforge.net/
[3] http://jtidy.sourceforge.net/
[4] `http://saxon.sourceforge.net/dtdgen.html`

Distance Computation & Clustering.
The Levenshtein edit distance method is implemented in Java and used to compare the schemas pairwise. Clustering is implemented using an algorithm which finds the transitive closure of a set of clone pair.

Simplifying Navigational Model.
After clusters have been identified, we simplify the navigational model by refining the clusters on the next levels and reducing each cluster to a single node. In the current implementation, we choose the largest page as the candidate node.

Differencing.
The *Diff* algorithm has been implemented extending a Java version[5] of the GNU Diff algorithm. The extended version has the ability to calculate and print page specific changes between two pages. For instance, the method diff(A, B, true) returns changes in B ignoring all changes in A.

Presentation.
The tool takes three input parameters namely, location (URI) of the initial page to start from, a similarity threshold, and the link depth level. Given these inputs, it automatically produces clusters along the extracted navigational path in DOT (Visualization) and in XML format. Also a list of connected found candidate components in XML and HTML format is produced.

3.7 Case Study

3.7.1 JPetStore

We have chosen JPetStore[6] as our migration case study, which is a publicly available dynamic web application based on Sun's original J2EE PetStore. The primary differences are that JPetStore, is vendor independent, has a standard-based multi-page web interface, and is Struts-based, which make it a typical modern web application.

3.7.2 Reference Classification

The idea of an automatic way of supporting the migration process from multi-page to single-page web applications came to us when we initially conducted a manual re-engineering of the JPetStore web application a few months ago. Our goal was to ajaxify the application using the Backbase[7] framework.

Backbase provides a set of server-side UI components, based on the *JavaServer Faces* technology. It became immediately evident to us that the fist step one needs to take in order to conduct such a migration process, is to figure

[5] `http://www.bmsi.com/java/#diff`
[6] `http://ibatis.apache.org/petstore.html`
[7] `http://www.backbase.com`

Classification	# of pages
Home (Index)	1
Product Categories	5
Product Item Lists	14
Product Items	23
Checkout	1
New Account	1
Sing On	1
View Cart	1
Add Item to Cart	24
Remove Item From Cart	24
Help	1

Table 3.1 JPetstore Reference Page Classification.

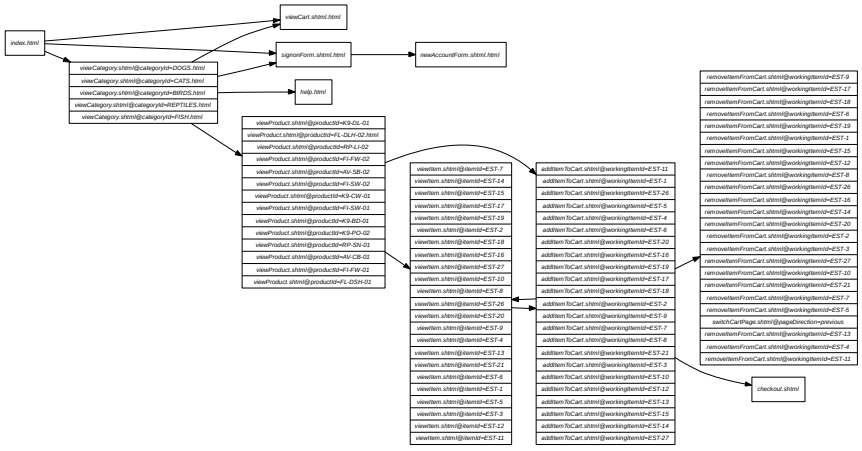

Figure 3.5 Retrieved Clusters Along The Navigational Path.

out the navigational model and UI components of the current implementation of JPetStore.

Our first step was to create a mirror copy of the web application interface by retrieving as many pages as possible. A total of 96 pages were retrieved. The pages were manually examined to document a reference classification in advance. This reference classification was used for comparing candidate clusters found by the tool to evaluate the results. The reference contains 11 classifications as shown in Table 3.1.

3.7.3 Automatic Classification

The aim of the case study is to determine to what extent we can use JPetStore's web interface to automatically find a list of candidate user interface components along with their navigational path.

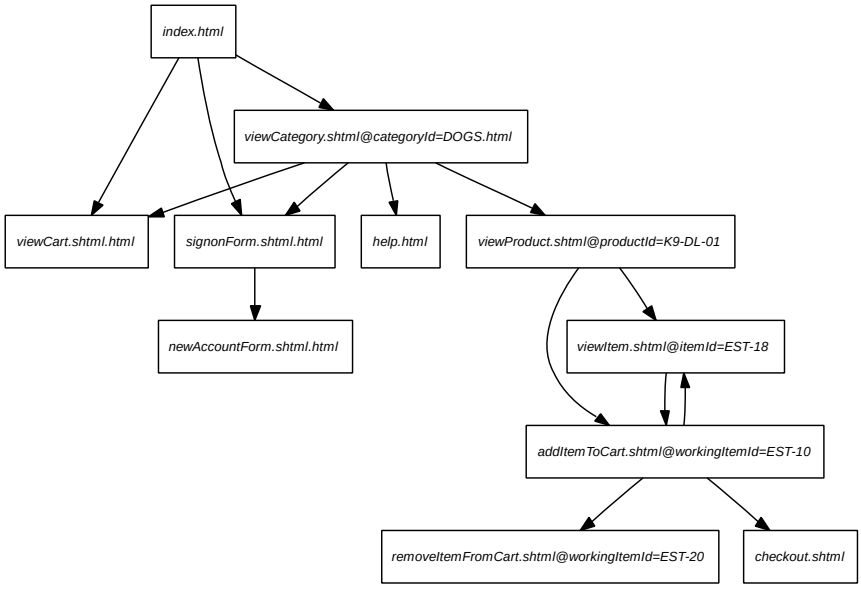

Figure 3.6 Reduced Clusters.

In order to conduct a preliminary evaluation of the described reverse engineering process, we used two different methods, namely, our own schema-based similarity approach (MMS), and our own implementation of a syntax tree similarity (STS) approach as proposed by, e.g., (De Lucia et al., 2004b). We also used different thresholds to find out the best achievable results.

In the first step of the reverse engineering process, pages were clustered along the navigational path (tree-based) and the navigational path was reduced by refining the clusters, as shown in Figure 3.5. Subsequently, as illustrated in Figure 3.6, in the second step, found pages in each cluster were reduced to a single node using the method described in 3.4.4.

Afterwards, candidate UI components were calculated by applying the *Differencing* algorithm as described in Section 3.5.

Figure 3.7 depicts viewing a candidate UI component (HTML code) in a browser, which is the result of going from the index page to the (dogs) category page. As a result, only that fraction of the category page that is unique with respect to the index page is reported. This way, we are able to visualize the delta changes (candidate single-page components) of the web interface by browsing the navigational path.

The list of candidate components and the simplified navigational path help us model our target single-page interface in the Conallen UML extension (Conallen, 2003), which is shown in Figure 3.8. Our single-page (called Page), contains three UI components namely, Category, Cart, and SignOn. Navigation takes place by changing the view from one component to another. For instance, from the Category component we can change our view to go to the

Figure 3.7 A candidate UI component (Product Category).

Product component. This is a delta change, meaning only that part of the Page that contained the Category component will be updated to view the new Product component.

3.7.4 Evaluation

Two well known metrics namely *precision* and *recall* were used to evaluate the results. Precision represents how accurately the clusters from the algorithm represent the reference classification. Recall measures how many pages in the reference classification are covered by clusters from the algorithm. We count only exact matches against the reference classification in the *Relevant Clusters Retrieved* (RCR) group. This means, if the algorithm finds a cluster which contains one or more extra (or one or more less) pages than the corresponding reference cluster, it is counted in the *Irrelevant Clusters Retrieved* (ICR).

Other comparison techniques, such as the ones introduced by Koschke and Eisenbarth (Koschke and Eisenbarth, 2000) and Tzerpos and Holt (Tzerpos and Holt, 1999) could also have been chosen. However, we would expect similar results from these techniques as well.

Table 3.2 shows the results. With the syntax tree similarity (STS) approach, the best recall value obtained was 82 % with a precision of 69 %, using a similarity threshold of 91 %.

The meta-based similarity (MMS) approach, however, was able to find all 11 documented reference clusters with a recall and precision of 100 % using a similarity threshold of 98 %. Note that by increasing the threshold to 99 %, the precision and recall drop to 82 %. This behavior can be explained because the algorithm expects the schemas to be almost identical, and as a result very little difference in the corresponding pages is tolerated. This increases the number of false positives.

Method	Threshold	RCR	ICR	Precision (%)	Recall (%)
STS	0.89	6	3	66	54
STS	0.91	9	4	69	82
STS	0.93	7	8	46	63
MMS	0.97	7	1	87	63
MMS	0.98	11	0	100	100
MMS	0.99	9	2	82	82

Table 3.2 Results of Clustering JPetstore Web Interface.

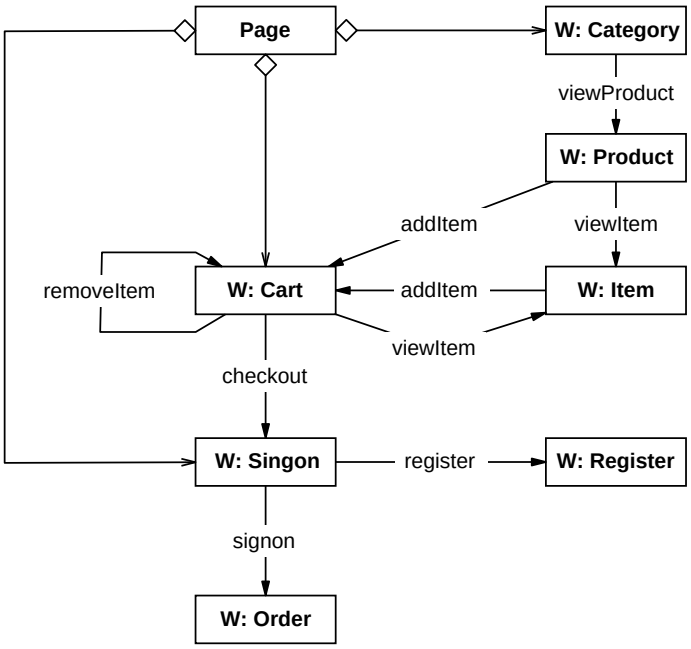

Figure 3.8 Target JPetstore Single-page Interface.

3.8 Discussion

As mentioned before, we came to the idea of a tool support for ajaxification process when we first conducted a manual migration.

The required knowledge for ajaxification was obtained by manually browsing the interface, from one page to the other, noting the differences, and building a map of the interaction model. This was when we realized that reverse engineering techniques should be able to provide some degree of support. Having a tool which provides us with information about the UI components needed and their positions on the navigational paths, can be of great value.

Applying the techniques described in this chapter to our case study, we were able to find all reference classifications. Additionally, with some degree of manual intervention, we were able to create a single-page model of the

target system.

Even though the techniques introduced in this chapter have only been applied to one case study, considering the results obtained, we believe the applications can span real-world web application migration cases. Although the JPetStore interface is very simple, it is representative of dynamic transactional web applications, and this class of web applications is exactly what we aim for. Our approach is not meant for web sites that are composed of long pages such as news, article, or forum sites. We will need to conduct more case studies to find strengths and weaknesses of our techniques and improve the tool.

We take a client-side analysis approach. While having the benefit of being server-code independent, the information that can be inferred from the server-side, such as scripting languages as JSP, is also essential for conducting a real migration process.

One of the problems we encountered while carrying out the case study, was that some HTML pages contained elements that were not well-formed or were not recognized by the formatter. Even JTidy was not able to fix the problems and no conversion to XHTML could be conducted. For instance in a few pages, instead of `<img ...>` element a `<image ...>` was used. Manual intervention was required to fix the problem. This sort of problems are inherent in web applications and can cause real problems in real-world migration cases, where standard guidelines are neglected and faulty HTML code is written/generated.

For a more detailed discussion of the limitations of the proposed approach see 7.2.

3.9 Related Work

Reverse engineering techniques have been applied to web application settings primarily to gain a comprehensible view of the systems.

Hassan and Holt (2002) present an approach to recover the architectural model of a web application by extracting relations between the various components and visualizing those relations.

Di Lucca et al. (2002c,d) propose WARE which is a tool for reverse engineering Web applications to the Conallen extension (Conallen, 2003) of UML models. Draheim et al. (2005), present Revengie to reconstruct form-oriented analysis models for web applications.

Ricca and Tonella (2001) propose ReWeb, a tool to analyze source code to recover a navigational model of a web site. They use the models obtained by ReWeb for testing (Tonella and Ricca, 2004) web applications. Supporting the migration of static to dynamic web pages is illustrated in (Ricca and Tonella, 2003) by applying an agglomerative hierarchical clustering approach.

De Lucia et al. (2005, 2004b) present a program comprehension approach to identify duplicated HTML and JSP pages based on a similarity threshold using Levenshtein string edit distance method. They use three notions of

similarity namely, structure, content, and scripting code. In (De Lucia et al., 2004a), the authors apply the techniques in a re-engineering case study.

WANDA (Antoniol et al., 2004) is a tool for dynamic analysis of web applications. It instruments web pages and collects information during the execution. This information is used to extract diagrams, such as component, deployment, sequence and class diagrams according to Conallen UML extensions.

Cordy et al. (2004) use an island grammar to identify syntactic constructs in HTML pages. The extracted constructs are then pretty-printed to isolate potential differences between clones to as few lines as possible and compared to find candidate clones using the UNIX diff tool.

A study of cloning in 17 web applications is presented by Rajapakse and Jarzabek (2005), aiming at understanding the nature of web clones and their sources. Lanubile and Mallardo (2003) discuss a pattern matching algorithm to compare scripting code fragments in HTML pages.

Stroulia et al. (2003) analyze traces of the system-user interaction to model the behavior of the user interface for migrating the user interface from a legacy application to a web-based one. GUI Ripping (Memon et al., 2003) creates a model from a graphical user interface for testing purposes, i.e., it generates test cases to detect abnormalities in user interfaces. Vanderdonckt et al. (2001) propose Vaquista, a XIML-based tool for static analysis of HTML pages. Its goal is to reverse engineer the user interface model from individual HTML pages to make them device independent.

Our classification approach is in two ways different from work conducted earlier on this topic. First, while others have based their structural similarity notion on the edit distance calculated on the syntax trees of pages, we propose a meta-model similarity notion and implement a schema-based clustering approach which, in the case of HTML pages, provides very promising results. Second, we try to find the clusters along the navigational path (different levels), as opposed to classifying all pages at once (one level) in order to identify candidate UI components along with their navigational model.

3.10 Concluding Remarks

In this chapter, we emphasized the rise of single-page Ajax applications and the need for support in migrating classical multi-page web applications to this new paradigm.

The main contributions of this chapter can be summarized as follows. First, we proposed a migration process, consisting of five steps: retrieving pages, navigational path extraction, user interface component model identification, single-page user interface model definition, and target model transformation. The second and third steps were described in full detail.

Second, we introduced a novel meta-model similarity metric for web page classification, which in our case studies achieves a higher recall and precision than approaches based directly on the HTML syntax trees.

Third, we proposed a schema-based clustering technique that operates per navigational level, instead of on the full set of web pages. Furthermore, we provide a mechanism for simplifying navigational paths, allowing us to find candidate user interface components through a differencing mechanism.

Future work encompasses the in-depth application of our approach in other case studies and more focus on the last two steps of the proposed migration process and study how a model-driven approach can be adopted in Ajax development. Furthermore, we will investigate how we can take advantage of dynamic analysis concepts to support the retrieving pages step of the migration process.

Finally, more research is needed to understand to what extent the server-side code should be adapted while migrating from a multi-page web application to a single-page Ajax interface.

Performance Testing of Data Delivery Techniques for Ajax Applications*

Chapter 4

Ajax applications are designed to have high user interactivity and low user-perceived latency. Real-time dynamic web data such as news headlines, stock tickers, and auction updates need to be propagated to the users as soon as possible. However, Ajax still suffers from the limitations of the Web's request/response architecture which prevents servers from pushing real-time dynamic web data. Such applications usually use a pull style to obtain the latest updates, where the client actively requests the changes based on a predefined interval. It is possible to overcome this limitation by adopting a push style of interaction where the server broadcasts data when a change occurs on the server side. Both these options have their own trade-offs. This chapter first introduces the characteristics of both pull and push approaches. It then presents the design and implementation of our distributed test framework, called Chiron, where different Ajax applications based on each approach can be automatically tested on. Finally, we present and discuss the results of our empirical study comparing different web-based data delivery approaches.

4.1 Introduction

Recently, there has been a shift in the direction of web development. A new breed of web application, dubbed Ajax (Asynchronous JavaScript and XML) (Garrett, 2005) is emerging in response to the limited degree of interactivity in large-grain stateless Web interactions. The intent is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes and making changes to individual user interface components. This way, the entire web page does not have to be reloaded each time the user makes a change.

The term Ajax spread rapidly from a Weblog to the Wall Street Journal within weeks. The new web applications under the Ajax banner have re-defined end users' expectations of what is possible within a Web browser. However, Ajax still suffers from the limitations of the Web's request/response architecture. The classical model of the web called Rest (Fielding and Taylor, 2002) requires all communication between the browser and the server to be initiated by the client, i.e., the end user clicks on a button or link and thereby

*This chapter has been accepted for publication in the Journal of Web Engineering, in 2009 (Bozdag et al., 2009).

requests a new page from the server. No 'permanent' connection is established between client/ server and the server is required to maintain no state information about the clients. This "pull" scheme helps scalability (Fielding and Taylor, 2002), but precludes servers from sending asynchronous notifications. There are many cases where it is important to update the client user interface in response to server-side changes. For example:

- An auction web site, where the users need to be averted that another bidder has made a higher bid. Figure 4.1 shows a screen-shot taken from eBay. In a site such as eBay, the user has to continuously press the 'refresh' button of his or her browser, to see if somebody has made a higher bid.

- A stock ticker, where stock prices are frequently updated. Figure 4.2 shows a screen-shot taken from MSN's MoneyCentral site.[1] The right column contains a stock ticker. The site currently uses a *pull*-based mechanism to update the stock data.

- A chat application, where new sent messages are delivered to all the subscribers.

- A news portal, where news items are pushed to the subscriber's browser when they are published.

Today, such web applications requiring real-time *event notification* and *data delivery* are usually implemented using a *pull* style, where the client component actively requests the state changes using client-side timeouts. An alternative to this approach is the *push*-based style, where the clients subscribe to their topic of interest, and the server publishes the changes to the clients asynchronously every time its state changes.

However, implementing such a push solution for web applications is not trivial, mainly due to the limitations of the HTTP protocol. It is generally accepted that a push solution that keeps an open connection for all clients will cause scalability problems. However, as far as we know, no empirical study has been conducted into the actual trade-offs involved in applying a push- versus pull-based approach to browser-based or Ajax applications. Such a study will answer questions about data coherence, scalability, network usage and latency. It will also allow web engineers to make rational decisions concerning key parameters such as publish and pull intervals, in relation to, e.g., the anticipated number of clients.

In this chapter, which is an extension of our previous work (Bozdag et al., 2007), we focus on the following challenges:

- How can we set up an automated, controllable, repeatable, and distributed test environment, so that we can obtain empirical data with high accuracy for comparing data delivery approaches for Ajax applications?

[1] `http://moneycentral.msn.com`

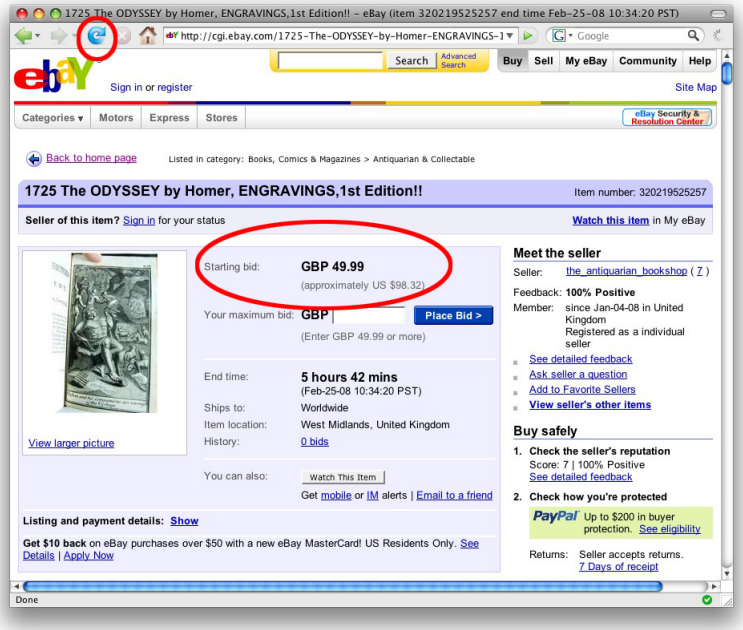

Figure 4.1 A screenshot taken from eBay. The user has to constantly click the "Refresh" button to see any updates.

- How does a push-based web data delivery approach compare to a pull-based one, in terms of data coherence, scalability, network performance, and reliability?

This chapter is further organized as follows.

We start out, in Section 4.2, by exploring current techniques for real-time HTTP-based data delivery on the web. Subsequently, in Section 4.3, we discuss the push-based Bayeux protocol and the DWR library, the two open source push implementations that we will use in our experiments. In Section 4.4, we present the experimental design by articulating our research questions and outlining the proposed approach. The independent and dependent variables of our experiment are also discussed in this section. A detailed presentation of our distributed testing framework called Chiron [2] as well as the environment and applications that we use to conduct our experiments, is shown in Section 4.5. In Section 4.6 the results of our empirical study involving push and pull data delivery techniques are covered, followed by a discussion of the findings of the study and threats to validity in Section 4.7. Finally, in Section 4.8, we survey related work on this area, after which we conclude our

[2] In Greek mythology, Chiron, was the only immortal centaur. He became the tutor for a number of heroes, including Ajax.

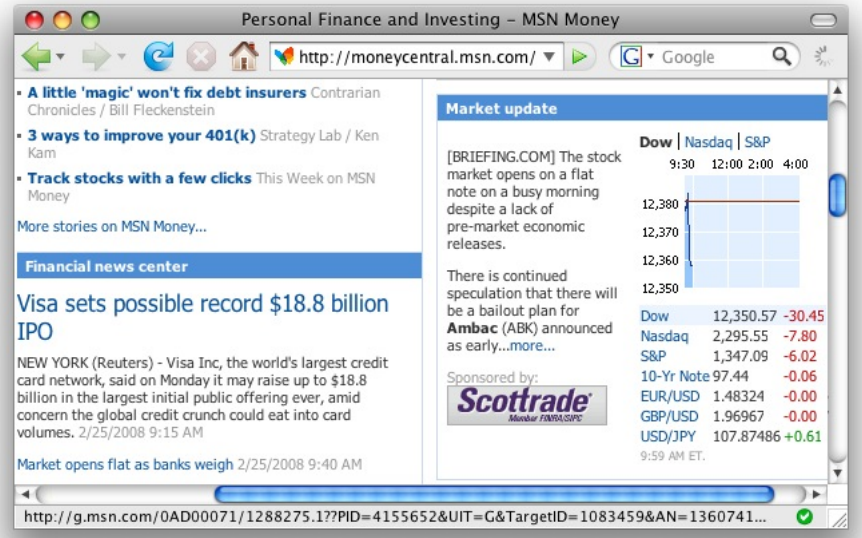

Figure 4.2 Stock ticker from MSN MoneyCentral

chapter in Section 4.9 with a summary of our key contributions and suggestions for future work.

4.2 Web-based Real-time Notification

The classical page-sequence web, based on the REST style, makes a server-initiated HTTP request impossible. Every request has to be initiated by a client, precluding servers from sending asynchronous notifications without a request from the client (Khare and Taylor, 2004). There are several solutions used in practice that still allow the client to receive (near) real-time updates from the server. In this section we analyze some of these solutions.

4.2.1 HTTP Pull

Most web applications check with the server at regular user-definable intervals known as *Time to Refresh* (TTR). This check occurs blindly regardless of whether the state of the application has changed.

In order to achieve high data accuracy and data freshness, the pulling frequency has to be high. This, in turn, induces high network traffic and possibly unnecessary messages. The application also wastes some time querying for the completion of the event, thereby directly impacting the responsiveness to

the user. Ideally, the pulling interval should be equal to the *Publish Rate* (PR), i.e., rate at which the state changes. If the frequency is too low, the client can miss some updates.

This scheme is frequently used in web systems, since it is robust, simple to implement, allows for offline operation, and scales well to high number of subscribers (Hauswirth and Jazayeri, 1999). Mechanisms such as Adaptive TTR (Bhide et al., 2002) allow the server to change the TTR, so that the client can pull on different frequencies, depending on the change rate of the data. This dynamic TTR approach in turn provides better results than a static TTR model (Srinivasan et al., 1998). However, it will never reach complete data accuracy, and it will create unnecessary traffic.

4.2.2 HTTP Streaming

HTTP Streaming is a basic and old method that was introduced on the web first in 1995 by Netscape, under the name 'dynamic document' (Netscape, 1995). HTTP Streaming comes in two forms namely, Page and Service Streaming.

Page Streaming

This method simply consists of streaming server data in the response of a long-lived HTTP connection. Most web servers do some processing, send back a response, and immediately exit. But in this pattern, the connection is kept open by running a long loop. The server script uses event registration or some other technique to detect any state changes. As soon as a state change occurs, it streams the new data and flushes it, but does not actually close the connection. Meanwhile, the browser must ensure the user-interface reflects the new data, while still waiting for response from the server to finish.

Service Streaming

Service Streaming relies on the XMLHttpRequest object. This time, it is an XMLHttpRequest connection that is long-lived in the background, instead of the initial page load. This brings some flexibility regarding the length and frequency of connections. The page will be loaded normally (once), and streaming can be performed with a predefined lifetime for connection. The server will loop indefinitely just like in page streaming, and the browser has to read the latest response (*responseText*) to update its state on the DOM.

4.2.3 Comet or Reverse Ajax

Currently, major Ajax push tools support Service Streaming. The application of the Service Streaming scheme under Ajax is now known as Reverse Ajax or Comet (Russell, 2006). Comet enables the server to send a message to the client when the event occurs, without the client having to explicitly request it. Such a client can continue with other work while expecting new data from

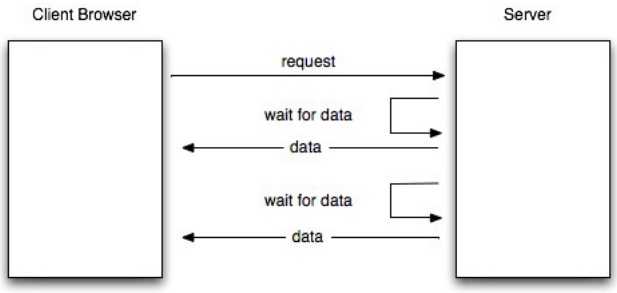

Figure 4.3 Streaming mode for COMET

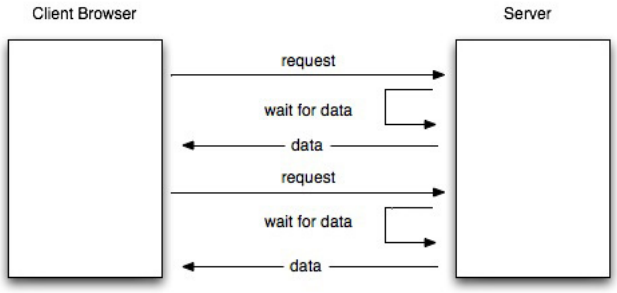

Figure 4.4 Long polling mode for COMET

the server. The goal is to achieve a real-time update of the state changes and offer a solution to the problems mentioned in Section 4.2.1.

The COMET scheme is available thanks to the 'persistent connection' feature brought by HTTP/1.1. With HTTP/1.1, unless specified otherwise, the TCP connection between the server and the browser is kept alive, until an explicit 'close connection' message is sent by one of the parties, or a timeout/network error occurs. Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. With persistent connections, fewer TCP connections are opened and closed, leading to savings both in CPU time for routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), as well as in memory usage for TCP protocol control blocks in hosts.

HTTP/1.1 reduces the total number of TCP connections in use. However, it still states that the protocol must follow the request/response scheme, where the client makes a request and the server returns a response for this particular request. Thus, once a complete response is returned, there is no further way for the server to send data back to the client browser.

COMET implementations have mainly adopted the following techniques to overcome this problem:

Streaming Figure 4.3 shows how the streaming mode operates. After the initial request, the server does not close the connection, nor does it give a full response. As the new data becomes available, the server returns it to the client in HTTP chunked mode (W3C, 1999), using the same request and the connection. Typical Comet server implementations implement this feature by opening a hidden *iframe* element in the browser after page load, establishing a long-lived connection inside the hidden iframe. Data is pushed incrementally from the server to the client over this connection, and rendered incrementally by the web browser (Schiemann, 2007).

The problem with this approach is that some web servers might identify this open-request as idle and close the connection. HTTP chunked mode might also be not supported in every router that is located on the path.

Long polling Figure 4.4 shows the operation of the long polling mode. In this mode, once again the server holds on to the client request, however this time until data becomes available. If an event occurs, the server sends the data to the client and the client has to reconnect. Otherwise, the server holds on to the connection for a finite period of time, after which it asks the client to reconnect again. Long polling (also known as Asynchronous-Polling) is a mixture of pure server push and client pull. If the publish interval (or the timeout value) is low, the system acts more like a pure pull-based style. If the publish interval is high, it will act more like a pure push approach.

4.3 Comet Implementations

Cometd and DWR are currently two actively developed open source libraries that bring Comet support to Ajax applications. In the following subsections we take a closer look at these two libraries.

4.3.1 Cometd Framework and the Bayeux Protocol

As a response to the lack of communication standards for Ajax applications, the Cometd group[3] released a Comet protocol draft called Bayeux (Russell et al., 2007). The Bayeux message format is defined in JSON (JavaScript Object Notation), which is a data-interchange format based on a subset of the JavaScript Programming Language. The protocol has recently been implemented and included in a number of web servers including Jetty and IBM Websphere.

This protocol follows the 'topic-based' (Eugster et al., 2003) publish-subscribe scheme, which groups events according to their topic (name) and maps individual topics to distinct communication channels. Participants subscribe to individual topics, which are identified by keywords. Like many modern topic-based engines, Bayeux offers a form of *hierarchical addressing*, which permits

[3] `http://www.cometd.com`

programmers to organize topics according to containment relationships. It also allows topic names to contain *wildcards*, which offers the possibility to subscribe and publish to several topics whose names match a given set of keywords. Bayeux defines the following phases in order to establish a Comet connection:

1. The client performs a handshake with the server, receives a client ID and list of supported connection types, such as IFrame or long-polling (See Section 4.2.3).

2. The client sends a connection request with its ID and its preferred connection type.

3. The client later subscribes to a channel and receives updates

Although the Bayeux specification supports both streaming and long polling modes, the Cometd framework currently only implements the long polling approach. Please note that, although there are other frameworks that support the Bayeux protocol, in this chapter we will use the terms Bayeux and Cometd interchangeably.

4.3.2 Direct Web Remoting (DWR)

Direct Web Remoting (DWR)[4] is a Java open source library which allows scripting code in a browser to use Java functions running on a web server just as if they were in the browser. DWR works by dynamically generating JavaScript based on Java classes. To the user it feels like the execution is taking place on the browser, but in reality the server is executing the code and DWR is marshalling the data back and forwards. DWR works similar to the RPC mechanism (e.g., Java RMI), but without requiring any plugins. It consists of two main parts:

- A Java Servlet running on the server that processes requests and sends responses back to the browser.

- A JavaScript engine running in the browser that sends requests and can dynamically update the DOM with received responses from the server.

From version 2.0 and above DWR supports Comet and calls this type of communication "Active Reverse Ajax" (Direct Web Remoting, 2007). Currently, DWR does not support Bayeux, and has adopted its own protocol to exchange data. DWR supports the *long polling* as well as the *streaming* mode. Because Cometd has no streaming implementation, we will only use the long polling mode of both libraries in our experiment, in order to be able to make a comparison.

[4] `http://getahead.org/dwr`

4.4 Experimental Design

In this section we first present our research questions. Later we describe our proposed approach, and the independent and dependent variables which we will use to conduct the experiment and come to answers to our questions.

4.4.1 Goal and Research Questions

We set up our experiments in order to evaluate several dependent variables using the GQM/MEDEA[5] framework proposed by Briand et al. (Briand et al., 2002). First, we describe the goal, perspective and environment of our experiment:

Goal. To obtain a rigorous understanding of the actual performance trade offs between a push-based and a pull-based approach to Ajax data delivery.

Perspective:. In particular, we aim at an automated, repeatable experiment, in which as many (combinations) of the factors that influence performance (such as the number of users, the number of published messages, the intervals between messages, etc.) can be taken into account.

Environment. The experiments are targeted at distributed environments, particularly those with UNIX/Linux nodes. Further relevant factors of these systems will be described in Section 4.5.2.

We have formulated a number of questions that we would like to find answers for in each approach. Our research questions can be summarized as:

RQ1 How fast are state changes (new messages) on the server propagated to the clients?

RQ2 What is the scalability and overall performance of the server?

RQ3 To what extent is the network traffic generated between the server and clients influenced?

RQ4 How reliable is each approach? Are there messages that are missed or abundantly received on the clients?

4.4.2 Outline of the Proposed Approach

In order to obtain an answer to our research questions, we propose the following steps:

1. creating two separate web applications, having the same functionality, using each push library (one for Cometd and one for DWR), consisting of the client and the server parts,

[5]Goal Question Metric/MEtric DEfinition Approach

2. creating a pull-based web application with the same functionality as the other two,

3. implementing an application, called *Service Provider*, which publishes a variable number of data items at certain intervals,

4. simulating a variable number of concurrent web users operating on each application, by creating *virtual users*,

5. gathering data and measuring: the mean time it takes for clients to receive a new published message, the load on the server, number of messages sent or retrieved, and the effects of changing the data publish rate and number of users,

6. coordinating all these tools automatically in order to have consistent test runs for each combination of the variables,

7. reporting, analyzing, and discussing the measurements found.

4.4.3 Independent Variables

To see how the application server reacts to different conditions, we use different combinations of the following independent variables:

Number of concurrent users 100, 500, 1000, 2000, 5000 and 10000; the variation helps to find a maximum number of users the server can handle simultaneously.

Publish interval 1, 5, 15, 30, and 50 seconds; the frequency of the publishing updates is also important. Because of the *long polling* implementation in CometD and DWR (See Section 4.2), the system should act more like pure pull when the publish interval is small, and more like pure push when the publish interval increases. This is because a smaller publish interval causes many reconnects (See Section 4.2.3).

Pull interval 1, 5, 15, 30, and 50 seconds; when a pull approach is used, the pulling interval will also have an effect on the measurements.

Application mode CometD, DWR, and pull; we also made an option in our framework that allows us to switch between different application modes.

Total number of messages 10; to constrain the period needed to conduct the experiment, for each test run, we generate a total of 10 publish messages.

4.4.4 Dependent Variables

In order to be able to answer the research questions we measure the following dependent variables:

Mean Publish Trip-time (MPT) We define trip-time as follows:

Trip-time = | *Data Creation Date* − *Data Receipt Date* |

Data Creation Date is the date on the publishing server the moment it creates a message, and *Data Receipt Date* is the date on the client the moment it receives the message. Trip-time shows how long it takes for a publish message to reach the client and can be used to find out how fast the client is notified with the latest events.For each combination of the independent variables, we calculate the mean of the publish trip-time for the total number of clients.

We define a piece of data as *coherent*, if the data on the server and the client is synchronized. We check the data coherence of each approach by measuring the trip-time. Accordingly, a data item with a low trip-time leads to a high coherence degree.

Server Performance (SP) Since push is stateful, we expect it to have more administration costs on the server side, using more resources. In order to compare this with pull, we measure the CPU usage for the push- and pull-based approaches.

Received Publish Messages (RPM) To see the message overhead, we publish a total of 10 messages and count the total number of (non unique) messages received by the clients that could make a connection to the server. This shows us if a client receives an item multiple times and causes unnecessary network traffic.

Received Unique Publish Messages (RUPM) It is also interesting to see if all the 10 messages we have published reach the clients that could make a connection to the server. This shows us if a client misses any items.

Received Message Percentage (RMP) It is quite possible that not all the clients could make a connection, or receive all the messages. Therefore, for each run, we divide the total number of messages received, by the total number of messages published.

Network Traffic (NT) In order to see the difference in network traffic, we record the number of TCP packets coming to and going from the server to the clients.

4.5 Distributed Testing

In order to measure the impact the various combinations of independent variables have on the dependent variables, we need a distributed testing infrastructure. This infrastructure must make it possible to control independent variables such as the number of concurrent users and total number of messages, and observe the dependent variables, such as the trip-time and the network traffic.

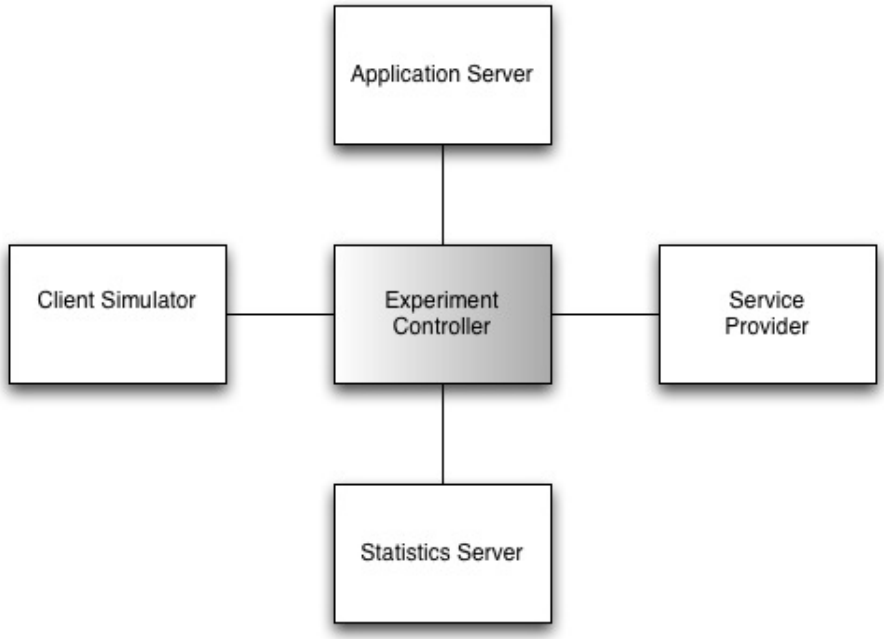

Figure 4.5 Design of the Distributed Automated Testing Framework CHIRON

Unfortunately, distributed systems are inherently more difficult to design, program, and test than sequential systems (Alager and Venkatsean, 1993). They consist of a varying number of processes executing in parallel. A process may also update its variables independently or in response to the actions of another process. Testing distributed programs is a challenging task of great significance. *Controllability*, *observability* (Chen et al., 2006), and *reproducibility* problems might occur in distributed testing environments. In this section we will present our distributed, automated testing framework called CHIRON and how it helps to decrease the magnitude of these problems. We have released our testing framework under an open source license. More information about CHIRON can be obtained from the following URL: `http://spci.st.ewi.tudelft.nl/chiron/`

4.5.1 The Chiron Distributed Testing Framework

As mentioned in Section 4.4.3, we defined several independent variables in order to measure several dependent variables (see Section 4.4.4). The combination of the independent variables (i.e. pull intervals, publish intervals, and the number of users) is huge and that makes performing the tests manually an error-prone, and time consuming task.

In addition, the tools and components we use are located in different machines to simulate real-world environments. This distributed nature con-

tributes to the complexity of controlling the experiment manually. This all makes it difficult to repeat the experiment at later stages with high validity.

In order to overcome these challenges, we have created an integrated performance testing framework called Chiron that automates the whole testing process. As depicted in Figure 4.5, the controller has direct access to different servers and components (Application server, client simulation server, the statistic server and the Service Provider). By automating each test run, the controller coordinates the whole experiment. This way we can repeat the experiment many times without difficulty and reduce the non-determinism, which is inherent in distributed systems (Alager and Venkatsean, 1993). Since no user input is needed during a test run, observability and controllability problems (Chen et al., 2006) are minimized.

We have implemented Chiron using a number of open source packages. In particular we use *Grinder*,[6] which seemed to be a good option, providing an internal TCPProxy, allowing to record and replay events sent by the browser. It also provides scripting support, which allows us to create a script that simulates a browser connecting to the push server, subscribing to a particular stock channel and receiving push data continuously. In addition, Grinder has a built-in feature that allows us to create multiple threads of a simulating script.

Figure 4.6 shows the steps Chiron follows for each test iteration. The middle column denotes the task Chiron performs, the left column denotes the step number and the right column shows which machine Chiron interacts with in order to perform the task. For each combination of the independent variables, Chiron carries out the tasks 3–10:

1. Read the input from the configuration file. This input consists of the independent variables, but also local and remote folders on different servers, path to the tools, publish channels, number of nodes that are used to simulate clients, etc,
2. Start the statistics server to listen for and receive experimental data from the clients,
3. Generate a Grinder properties file for this iteration with a combination of independent variables (pull interval, publish interval, number of users, etc.),
4. Upload the created properties file to the client simulation server,
5. Start the application server that hosts the three different versions (See Section 4.5.4) of the Stock Ticker web application,
6. Start performance and network traffic measurement tools on the application server,
7. Start simulating clients using the generated and uploaded properties file,

[6] `http://grinder.sourceforge.net`

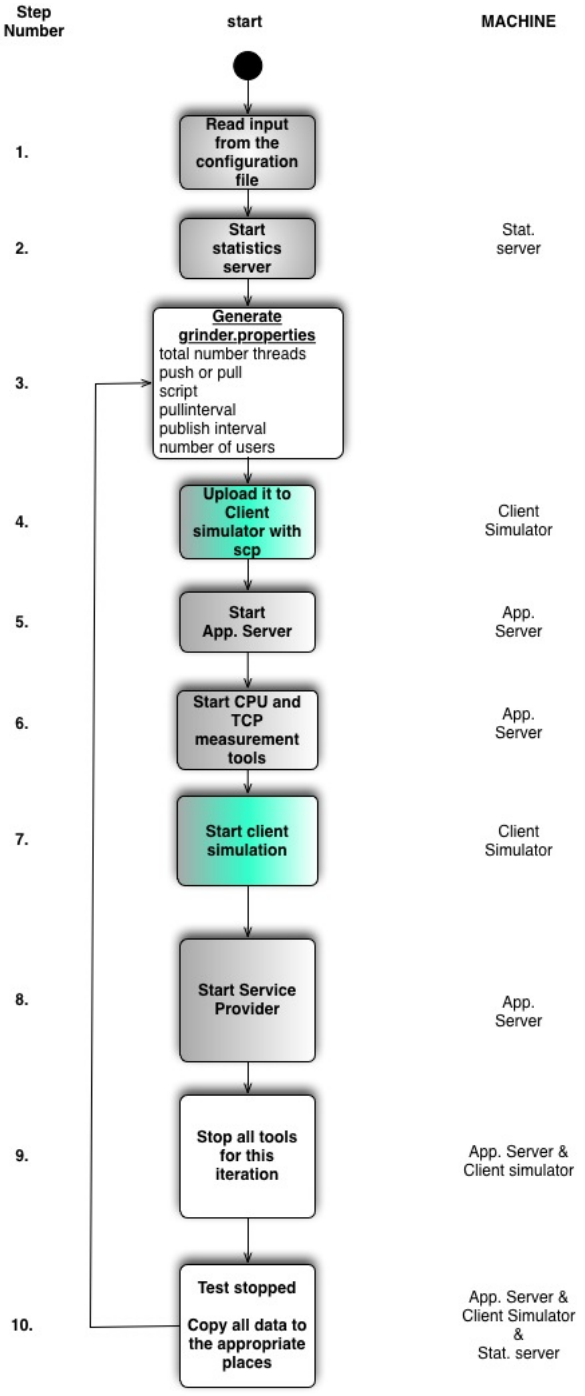

Figure 4.6 The steps CHIRON follows for each iteration

8. Start the publisher and begin publishing data to the server,

9. When the publisher is done or a time-out has occurred, stop all the components,

10. Gather and copy the data to the statistics server and get ready for the next iteration by going to the Grinder properties generation step.

Because of the distributed nature of the simulated clients, we use Log4J's SocketServer[7] to set up a logging server that listens for incoming log messages. The clients then send the log messages using the `SocketAppender`.

We use *TCPDump*[8] to record the number of TCP (HTTP) packets sent to and from the server on a specific port. Note that we only record packets coming/going to the simulated clients. We also have created a script that uses the UNIX *top*[9] utility to record the CPU usage of the application server every second. This is necessary to observe the scalability and performance of each approach.

Finally, we use Trilead's SSH2 library to manage (start, stop, etc) all the tools mentioned above. Trilead SSH-2 for Java[10] is an open source library which implements the SSH-2 protocol in Java. Since the tools are distributed on different machines, SSH2 library allows us to automatically issue commands from one single machine, gather the results on different machines and insert them into our statistics server.

In addition, we have created a *Data Analyzer* component that parses log files of the different tools and serializes all the data into a database using Hibernate[11] and MySQL Connector/J[12]. This way, different views on the data can be obtained easily using queries to the database.

4.5.2 Testing Environment

We use the Distributed ASCI Supercomputer 3 (DAS3)[13] to simulate the virtual users on different distributed nodes. The DAS3 cluster at Delft University of Technology consists of 68 dual-CPU 2.4 GHz AMD Opteron DP 250 compute nodes, each having 4 GB of memory. The cluster is equipped with 1 and 10 Gigabit/s Ethernet, and runs Scientific Linux 4. It is worth noting that we use a combination of the 64 DAS3 nodes and Grinder threads to simulate different numbers of users.

The application server runs on a AMD 2x Dual Core Opteron 2212 machine with 8 GB RAM. The server has Ubuntu 7.10 server edition installed. For the *push* version we use the Cometd *6.1.7* and DWR *2.0.2* libraries. Both libraries run on Jetty, an open source web server implemented entirely in Java. Jetty

[7] `http://logging.apache.org/log4j/docs/`
[8] `http://www.tcpdump.org/`
[9] `http://www.unixtop.org/`
[10] `http://www.trilead.com/Products/Trilead-SSH-2-Java/`
[11] `http://www.hibernate.org`
[12] `http://www.mysql.com/products/connector/j/`
[13] `http://www.cs.vu.nl/das3/overview.shtml`

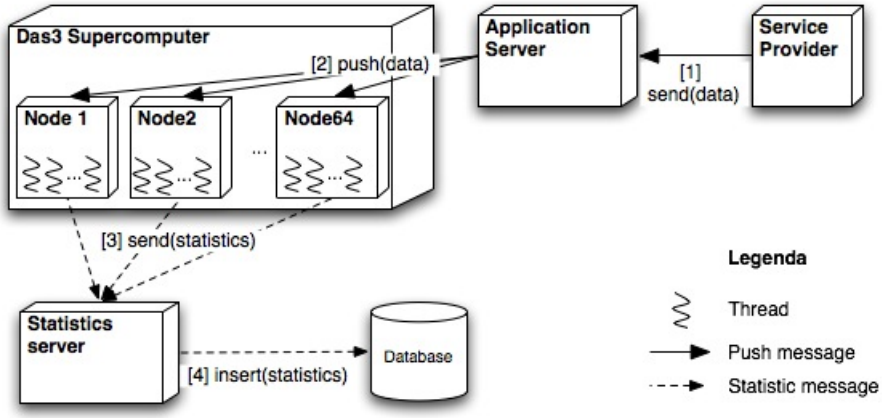

Figure 4.7 Sequence of Events in the Experimental Environment.

uses Java's new IO package (NIO). The NIO package follows the event-driven design, which allows the processing of each task as a finite state machine (FSM). As the number of tasks reach a certain limit, the excess tasks are absorbed in the server's event queue. The throughput remains constant and the latency shows a linear increase. The Event-driven design is supposed to perform significantly better than thread-concurrency model (Welsh et al., 2001; Welsh and Culler, 2003). Jetty also contains a mechanism called Continuations (Jetty, 2006). This mechanism allows an HTTP request to be suspended and restarted after a timeout or after an asynchronous event has occurred. This way less threads are occupied on the server. Note that no other process was running in this application server other than TCPDump and UNIX top. Their load on the server and their effect on the results are negligible, since we only measure the PID of Jetty. Besides, as mentioned, the test machine has 2x Dual Core processors, and the server never had 100% load, which can be seen in Section 4.6.2.

The connectivity between the server and DAS3 nodes is through a 100 Mbps ethernet connection.

4.5.3 Example Scenario

Figure 4.7 shows the sequence of events of an example test scenario from the perspective of the servers:

1. The Service Provider publishes the stock data to the application server via an HTTP POST request, in which the creation date, the stock ID, and the stock message are specified.

2. For push: The application server *pushes* the data to all the subscribers of that particular stock. For pull: the application server updates the

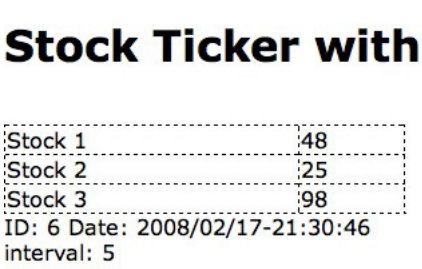

Figure 4.8 Sample Stock Ticker Application

internal stock object, so that when clients send pull requests, they get the latest data.

3. Each simulated client logs the responses (after some calculation) and sends it to the statistics server.

4.5.4 Sample Application: Stock Ticker

We have developed a Stock Ticker web application as depicted in Figure 4.8. As new stock messages come in, the fields are updated in the browser accordingly. As mentioned before, the Stock Ticker has been implemented in three variants, namely, Cometd, DWR, and pull.

The Cometd version. consists of a JSP page which uses Dojo's Cometd library to subscribe to a channel on the server and receive the Stock data. For the server side, we developed a Java Servlet (`PushServlet`) that pushes the new data into the clients' browsers using the Cometd library.

The DWR version. consists of the same JSP page as the Cometd version, but uses the DWR library instead of Dojo on the client side. For the server side, we again have a separate Java Servlet (`PushServlet`) that pushes the data into the browsers, but this time it uses the DWR's Comet servlet.

The pull version. also consists of a JSP page, but instead of Cometd or DWR, it uses the normal *bind* method of Dojo to request data from the server. The pull nature is set using the standard `setInterval` JavaScript method. The interval at which the client should request/pull for new updates is configurable. On the server, a `PullServlet` is created which updates and keeps an internal stock object (the most recent one) and simply handles and responds to every incoming request the classical way.

The Service Provider. uses the HTTPClient library[14] to publish stock data to the `Servlets`. The number of publish messages as well as the interval at which the messages are published are configurable.

[14] `http://jakarta.apache.org/commons/httpclient/`

Concurrent clients. are simulated by using the Grinder TCPProxy to record the actions of the JSP client pages for push and pull and create scripts for each in Jython.[15] Jython is an implementation of the high-level, dynamic, object-oriented language Python, integrated with the Java platform. It allows the usage of Java objects in a Python script and is used by Grinder to simulate web users. In our framework, we created Jython scripts that are actually imitating the different versions of the client pages.

4.6 Results and Evaluation

In the following subsections, we present and discuss the results which we obtained using the combination of variables mentioned in Section 4.4.3. Figures 4.9–4.14 depict the results. For each number of clients on the *x-axis*, the five publish intervals in seconds (1, 5, 15, 30, 50) are presented as colored bars. Note that the scale on the *y-axis* might not be the same for all the graphics.

4.6.1 Publish Trip-time and Data Coherence

Figure 4.9 shows the mean publish trip-time versus the total number of clients for each publish interval, with Cometd (shown with Bayeux label), DWR and *pull* techniques.

As we can see in Figure 4.9, Mean Publish Trip-time (MPT) is, at most, 1200 milliseconds with Cometd, which is significantly better than the DWR and *pull* approaches. Surprisingly, DWR's performance is worse than *pull* with a pull interval of 1. However, with a pull interval of 5 or higher, DWR performs better (with the exception of: clients = 2000, publish interval = 30) than *pull*. Also note that MPT is only calculated with the trip-time data of the clients that actually received the items. As we will see in Section 4.6.4, pull clients may miss many items, depending on the pull interval. From these results we can say that pull has a lower degree of data coherence compared to Cometd, even with a very small pull interval.

4.6.2 Server Performance

Figure 4.10 shows the mean percentage of the server CPU usage. In all scenarios, we see that the server load increases as the number of users increases. With pull, even with a pull interval of 1, the CPU usage is lower than Cometd and DWR.

However, with push, we notice that even with 10000 users, the server is not saturated: CPU usage reaches 45% with Cometd, and 55% with DWR. We assume that this is partly due to the Jetty's continuation mechanism (See Section 4.5.2).

[15] http://www.jython.org

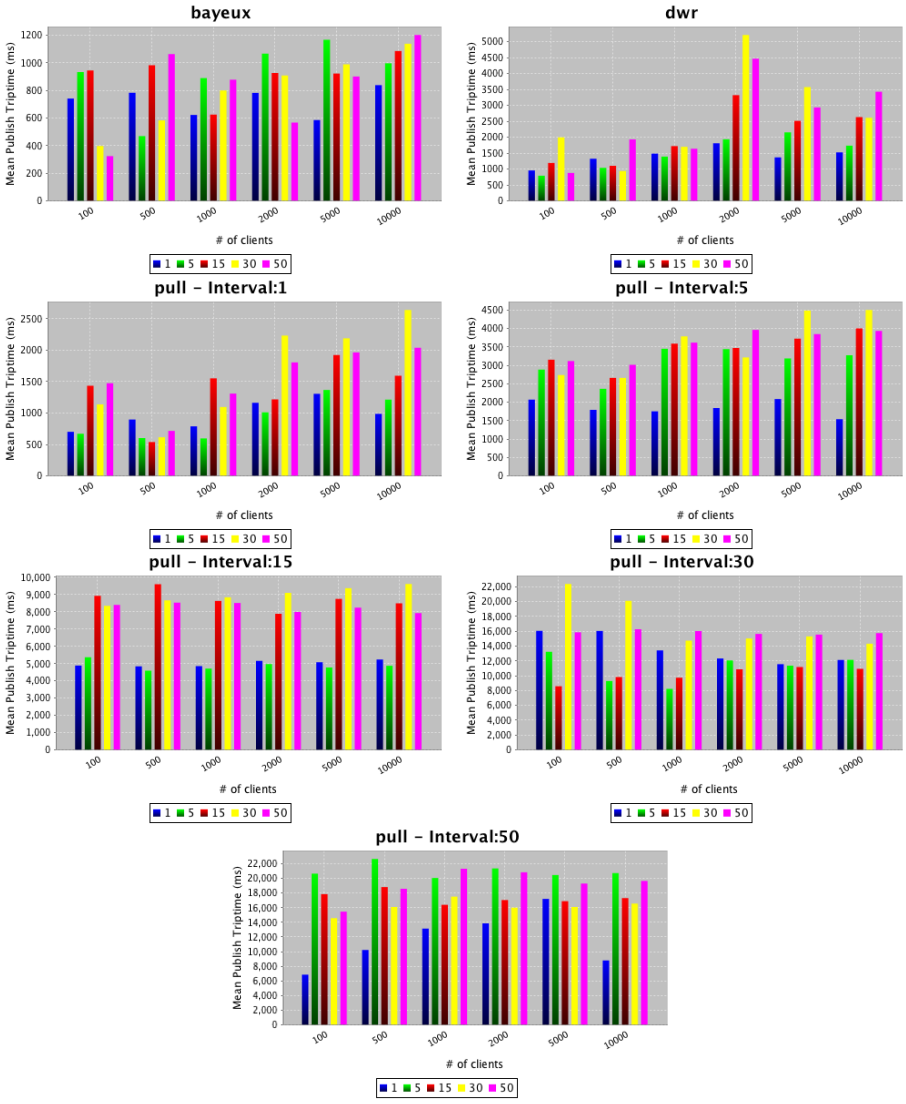

Figure 4.9 Mean Publish Trip-time

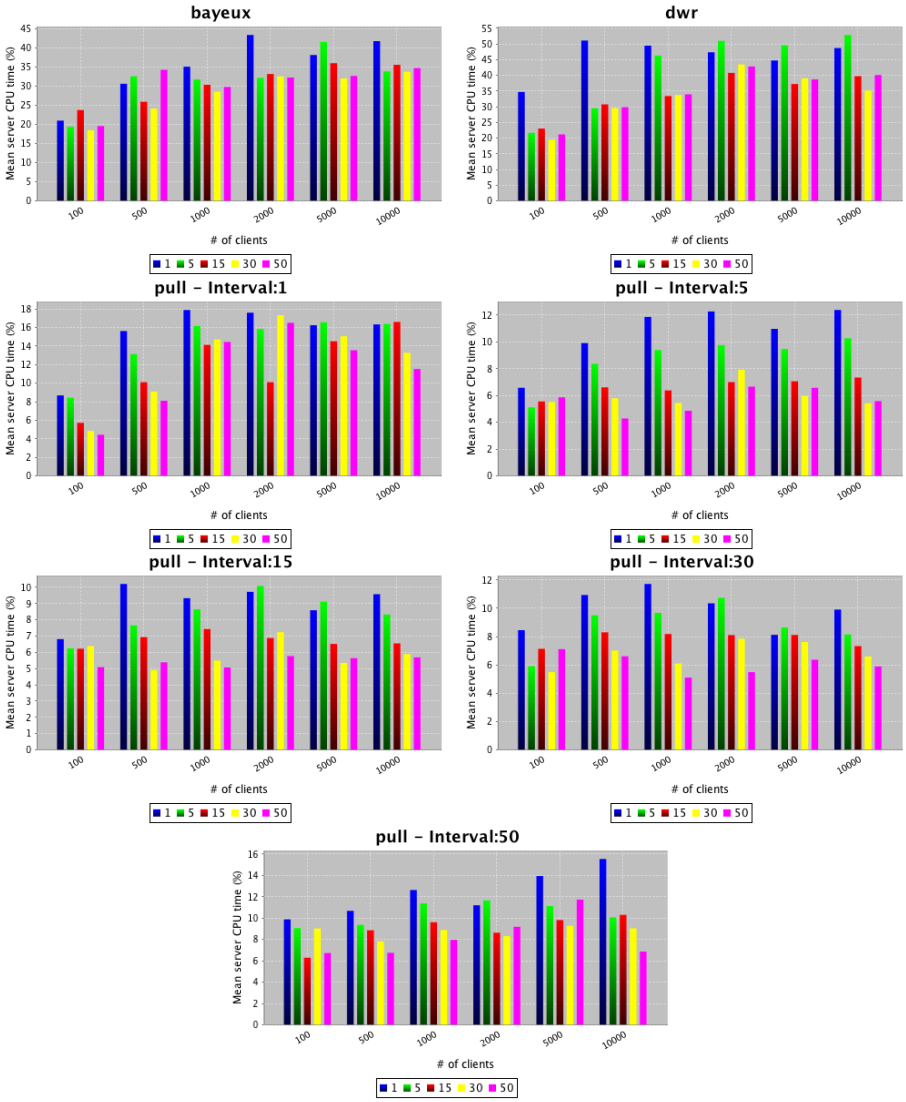

Figure 4.10 Server application CPU usage.

4.6.3 Received Publish Messages

Figure 4.11 shows the mean number of received non-unique publish items versus the total number of clients. Note that this shows the mean of Received Publish Messages (RPM) for only those clients that could make a connection to the server and receive data. Also note that if a pull client makes a request while there is no new data, it will receive the same item again and this pattern can happen multiple times. This way a client might receive more than 10 messages. As we mentioned in Section 4.2.1, in a pure pull system, the pulling frequency has to be high to achieve high data coherence. If the frequency is higher than the data publish interval, the pulling client will pull the same data more than once, leading to redundant data and overhead.

We notice that with a pull interval of 1, the clients can receive up to approximately 250 non-unique messages, while we published only 10. In the same figure we see that push clients (both CometD and DWR) received approximately a maximum of 10 messages. This means that, in the worst-case (`pull interval = 1, publish interval = 50`) 96% of the total number of pull requests were unnecessary. In the CometD and DWR graphics, we see that, with up to 2000 users the server is very stable; almost all the clients receive a maximum of 10 published messages. What is interesting, however, as the number of clients becomes 2000 or more, some data miss begins to occur for both push approaches. Pull clients also begin to miss some data with 2000 users or more, but the decrease in received data items is much less.

4.6.4 Received Unique Publish Messages

Figure 4.12 shows the mean number of received unique publish items versus total number of clients. Note that this shows the mean of Received Unique Publish Messages (RUPM) for only those clients that could make a connection to the server and receive data.

According to Figure 4.12, if the publish interval is higher or equal to the pull interval, the client will receive most of the messages. However as we have discussed in Section 4.6.3, this will generate an unnecessary number of messages. Looking at the figure again, we see that when the pull interval is lower than the publish interval, the clients will miss some updates, regardless of the number of users. So, with the pull approach, we need to know the *exact* publish interval. However, the publish interval tends to change, which makes it difficult for a pure pull implementation to adapt its pull interval.

With CometD and DWR, almost all messages are received by the users (having up to a total of 2000 users). Higher than 2000 users, we see that some data miss begins to occur for both push libraries.

With pull, a lower pull interval leads to more accuracy. With a pull interval of 1, the mean number of received items can be as high as 9. With a pull interval of 5 this number drops to 6.

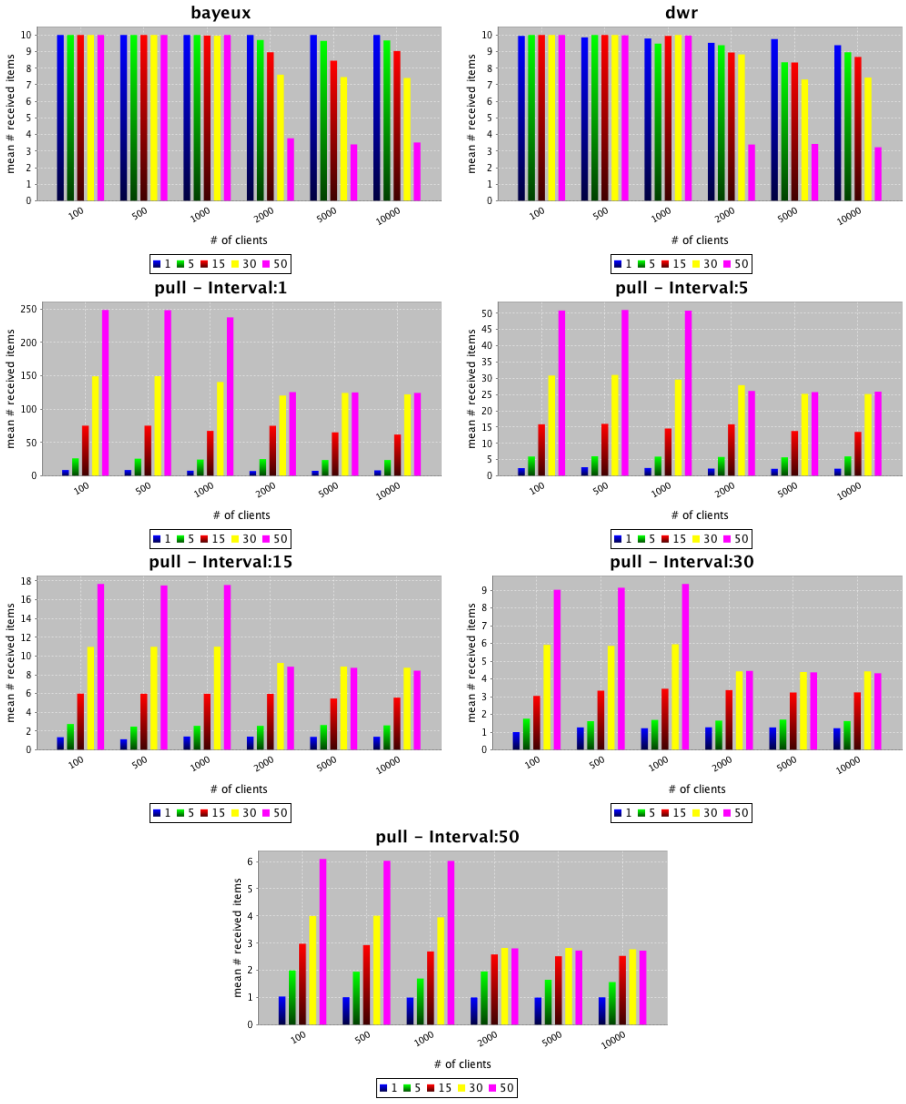

Figure 4.11 Mean Number of Received Publish Messages

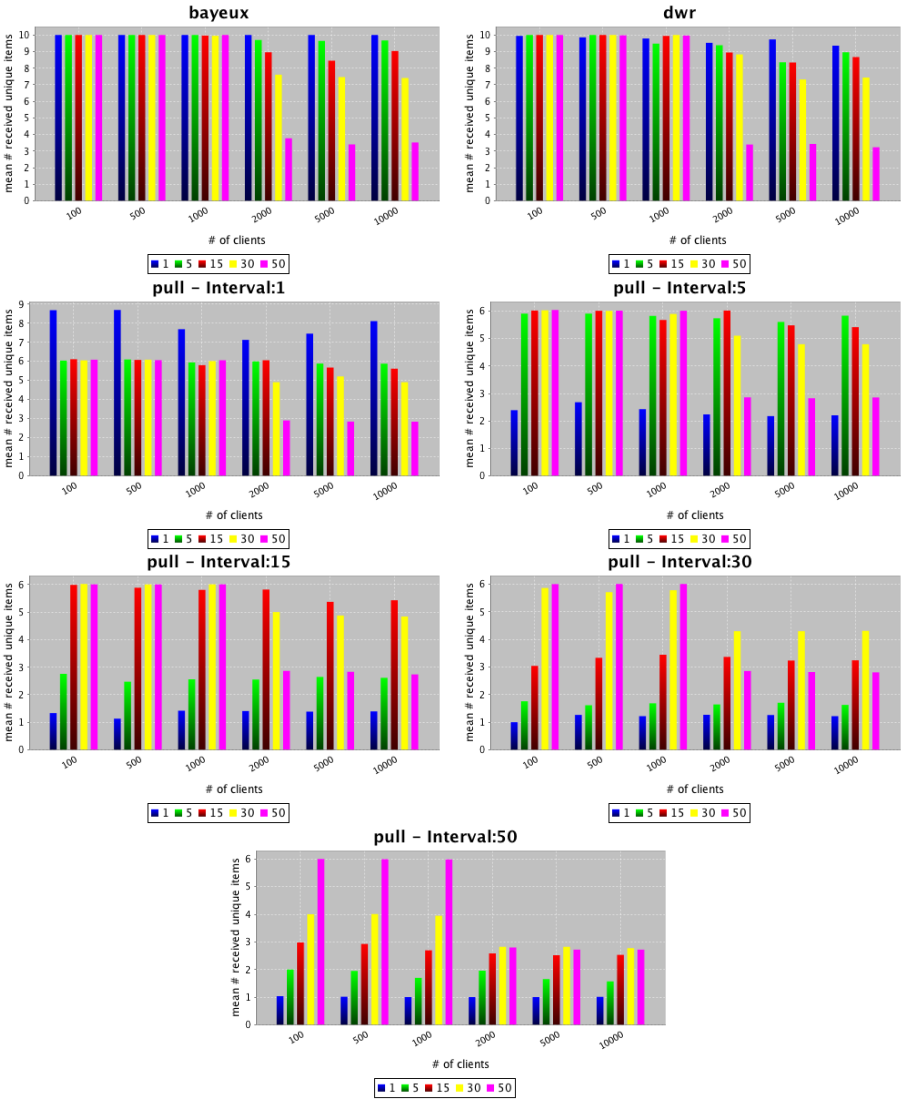

Figure 4.12 Mean Number of Received Unique Publish Messages.

4.6.5 Received Message Percentage

Figure 4.13 shows us the Received Message Percentage (RMP). We notice that RMP is significantly higher with Cometd and DWR, compared to pull. We can also see that RMP decreases significantly in all approaches after 2000 users. This might depend on many factors, such as, Jetty application server queuing all the connections, or perhaps, before a client can process the test run ends. It might also lie within the client simulator, or our statistics server that have to cope with generating a high number of clients and receive a huge number of messages. See Section 4.7.2 for a discussion of possible factors that might have an effect on the results.

However, if we compare RMP of the three approaches for 2000 users or more, we see that Cometd and DWR clients still receive more data than pull, and thus have a higher degree of reliability.

4.6.6 Network Traffic

Figure 4.14 shows the results. We notice that, in Cometd and DWR, the number of TCP packets traveled to/from the server increases as the number of users increases, rising up to 80,000 packets with Cometd and 250,000 with DWR. We see that up to 2000 users, the publish interval has almost no effect, showing that the administrative costs of establishing a long polling connection is negligible. After 2000 users, there is a difference between different publish intervals, this might be caused by many factors including connection timeouts. With pull, in worst case scenario (`pullInterval =1 , publishInterval = 50, and 1000 users`), Network Traffic (NT) rises up to 550,000 packets. This is almost 7 times more than Cometd and 2 times more than DWR. This number decreases with lower pull intervals, however as we have discussed in Section 4.6.1, high pull intervals will lead to high trip-time and low data coherence. Also note that, with pull, NT increases as the publish interval increases. This is because a longer publish interval will lead to a longer test run, in turn leading to more pull requests.

4.7 Discussion

In this section we discuss our findings and try to answer our research questions. We also present the threats to the validity of our experiment.

4.7.1 The Research Questions Revisited

RQ1 Data coherence: RQ1 inquired how fast the state changes (new messages) on the server were being propagated to the clients. We expected that the pull-based clients would have a higher trip-time value and thus a lower degree of data coherence. Our findings in Section 4.6.1 support our expectations. Even with a pull interval of 1 second, the trip-time

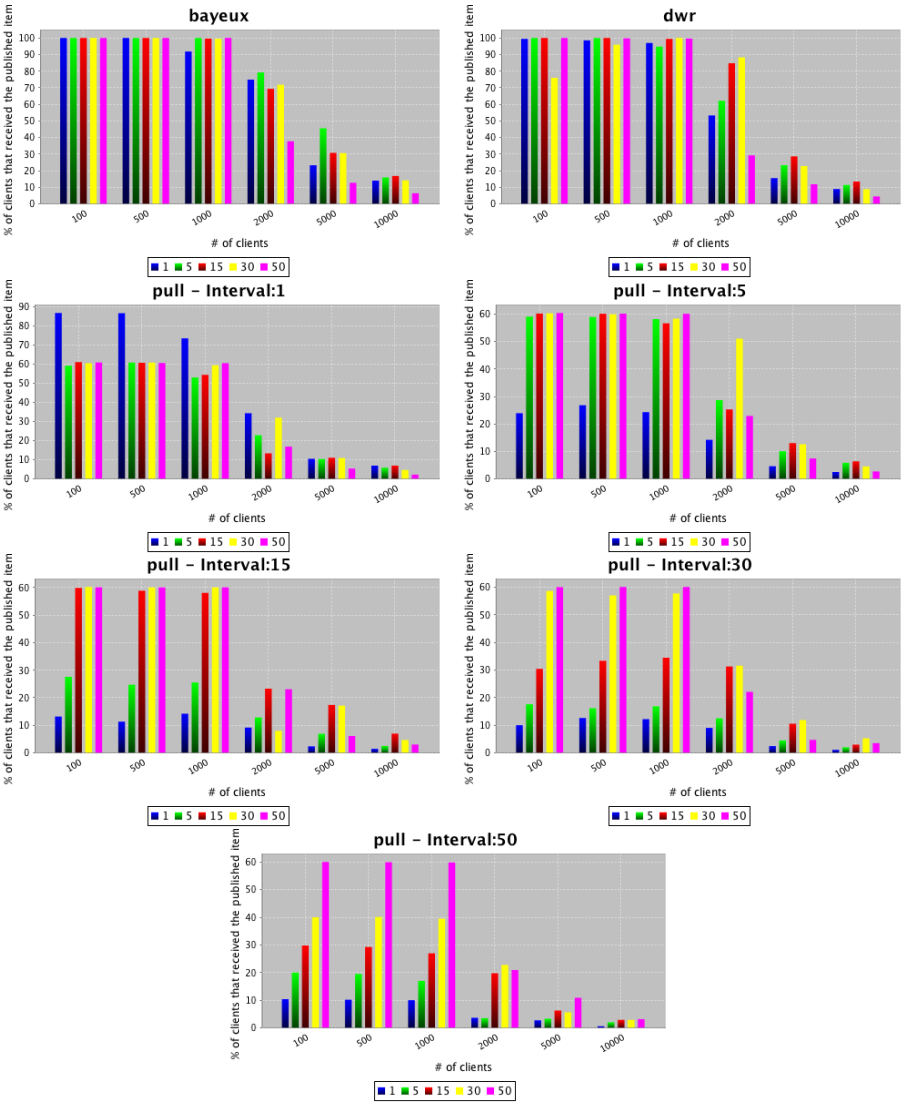

Figure 4.13 Percentage of data items that are delivered to all clients

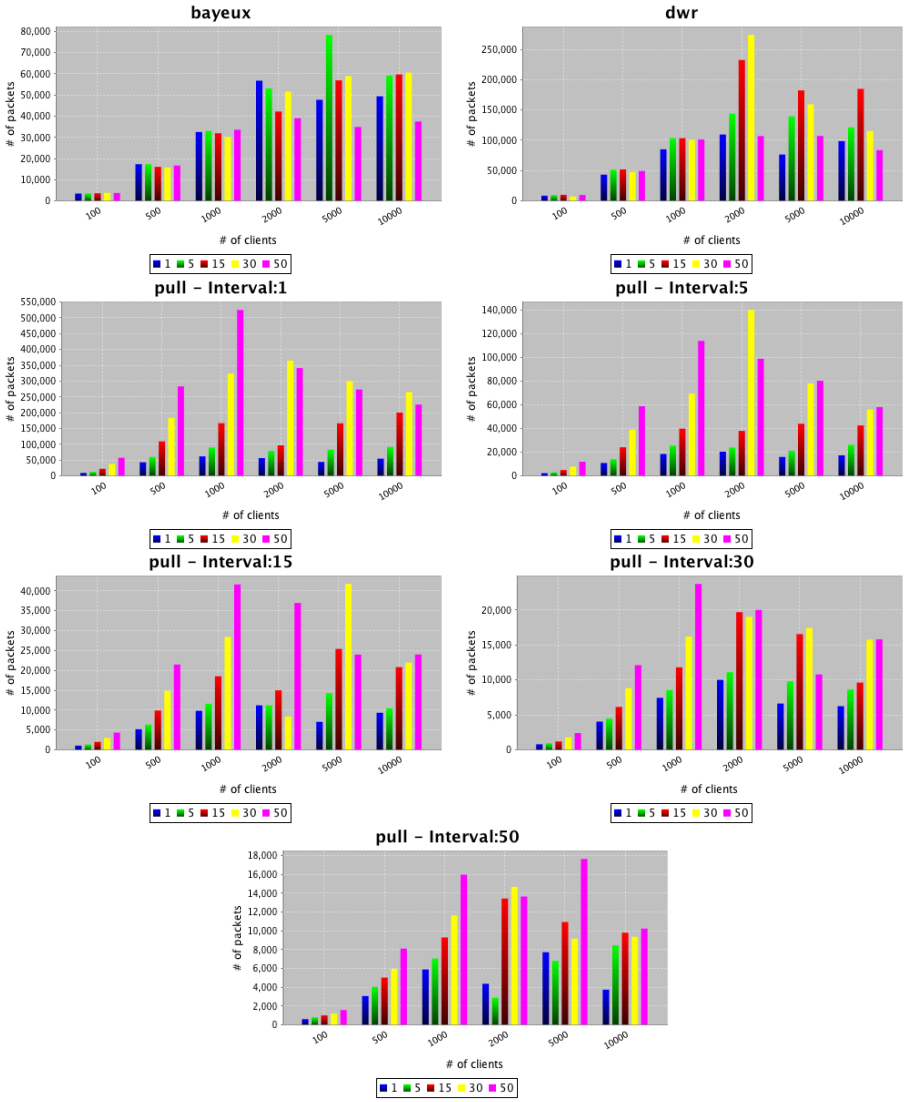

Figure 4.14 Total number of TCP packets coming from/to the server

with pull is higher than push. This finding indicates that the push-based style supports a higher degree of data coherence.

RQ2 Scalability and server performance: RQ2 addressed the issue with the scalability and the overall performance of the server. Considering our previous results (Bozdag et al., 2007), we initially expected push to cause significantly higher server load, compared to pull. However, our new findings in Section 4.6.2 show that, even though push causes more CPU usage, the difference is not significant. With push, the server CPU is not saturated, even with 10000 users. This indicates that both approaches scale well to thousands of clients.

RQ3 Network traffic: RQ3 tried to find to what extend the network traffic generated between the server and clients is influenced. We initially expected a smaller pull interval to create many unnecessary requests and a high network traffic. Our findings in Section 4.6.3 and Section 4.6.6 supported these predictions. Pull causes the clients to receive a higher number of redundant messages compared to push, leading to unnecessary network traffic. With pull, the number of traveling TCP packets to and from the server is 7 times more than CometD.

RQ4 Reliability: RQ4 questioned the reliability of each approach. We expected that pull-based clients would miss many more data items compared to push clients. Our findings in Section 4.6.5 and Section 4.6.4 show that pull clients with a low pull interval are more up-to-date compared to those with a high pull interval, but also receive many redundant data if the publish interval is high. However, they are still missing many data items compared to the push-based clients.

4.7.2 Threats to Validity

A fundamental question concerning results from an experiment is how valid the results are (Wohlin et al., 2000). In this section the validity of our findings is discussed with respect to *internal* and *external* validity. We structure our discussion according to Wohlin et al. (2000), which in turn is based on the work of Campbell and Stanley (1963), and T.D.Cook and D.T.Campbell (1979).

Internal Validity

In an experiment, different treatments may be applied to the same object at different times. Then there is a risk that the history affects the experimental results, since the circumstances are not the same on both occasions (Wohlin et al., 2000). We have the following threats to internal validity:

Non-deterministic nature of the distributed environments.
Our experimental setup is distributed on different machines and each test run for each technique is run on different times of the day. A network congestion

at the time of the test, might for example have an effect on the trip-time variable (See Section 4.6.1). We also use the supercomputer (DAS3) to simulate many clients and at the time of testing there might be other users running tasks, which might affect the network bandwidth. In order to limit these effects on the network latency, we placed all the machines in the same LAN, used the same test-script in all the simulated clients and allocated the same bandwidth.

Reliability of the tools.
We use several tools to obtain the result data. The shortcomings and the problems of the tools themselves can have an effect on the outcome. For example, from all the results we see that the performance suffers after 2000 users for all approaches. This degradation might be caused by the server, client simulator, or the statistics server. Debugging distributed applications, to find the cause of such behavior, proved to be very difficult.

Time and Data Coherence.
The time calculation can also be a threat to the internal validity. To measure the trip-time, the difference between the data creation date and data receipt date is calculated. However if the time on the publisher and the clients is different, the trip-time is calculated incorrectly. In order to prevent this, we made sure that the time on the server and client machines are synchronized by using the same time server.

We measure the data coherence by taking the trip-time. However, the data itself must be correct, i.e., the received data must be the same data that was sent by the server. We rely on HTTP in order to achieve this data correctness. However, additional experiments must include a self check to ensure this requirement.

External validity

The external validity deals with the ability to generalize results (Wohlin et al., 2000). There is a risk that the used *push* libraries are not good representatives of *push*, making it difficult to draw general conclusions about the whole *push* approach. In order to minimize the risk, we have used two libraries instead of one, and as we have shown in Section 4.6, we see the same pattern, which confirms our findings for push. However, we did only use a single web application server, because at the time of the testing Jetty was the only open source and stable Java server that supported NIO. In the future, different application servers should be included, e.g., Sun's Grizzly.[16]

We only used one type of sample application, namely the stock ticker. In this scenario, we had a single channel with many users, where a data item is sent to all the push clients. However, there are other use cases, such as a chat application, where there will be multiple channels with many users. In

[16] `https://grizzly.dev.java.net/`

this scenario, a data item will only be sent to the subscribers of that particular channel. This will have effects on the scalability and other dependent variables. Therefore, further tests with different use cases are necessary.

In order to limit the external factors that affect the trip-time, we placed all the test machines in the same network and all the users are granted with the same bandwidth. In a real-life scenario, users are located at different parts of the world, and have different bandwidth properties, leading to a bigger variance in the trip-time. This should be taken into account before deciding on actual parameters of a web application using *push*.

During the experiment execution, if a large volume of data exchange occurs, this might lead to concurrency on the access to the shared resources. To minimize this threat, we run each server (application server, client generator, statistics server) on different machines. Note that only the client generator is located in a cluster (DAS3). Other servers are located outside the cluster.

4.8 Related Work

There are a number of papers that discuss server-initiated events, known as *push*, however, most of them focus on client/server distributed systems and non HTTP multimedia streaming or multi-casting with a single publisher (Acharya et al., 1997; Juvva and Rajkumar, 1999; Franklin and Zdonik, 1998; Ammar et al., 1998; Trecordi and Verticale, 2000). The only work that focuses on Ajax is the white-paper of Khare (2005). Khare discusses the limits of the pull approach for certain Ajax applications and mentions several use cases where a push application is much more suited. However, the white-paper does not mention possible issues with this *push* approach such as scalability and performance. Khare and Taylor (2004) propose a push approach called ARRESTED. Their asynchronous extension of Rest, called A+REST, allows the server to broadcast notifications of its state changes. The authors note that this is a significant implementation challenge across the public Internet.

The research of Acharya et al. (1997) focuses on finding a balance between push and pull by investigating techniques that can enhance the performance and scalability of the system. According to the research, if the server is lightly loaded, pull seems to be the best strategy. In this case, all requests get queued and are serviced much faster than the average latency of publishing. The study is not focused on HTTP.

Bhide et al. (2002) also try to find a balance between push and pull, and present two dynamic adaptive algorithms: *Push and Pull* (PaP), and *Push or Pull* (PoP). According to their results, both algorithms perform better than pure pull or push approaches. Even though they use HTTP as messaging protocol, they use custom proxies, clients, and servers. They do not address the limitations of browsers nor do they perform load testing with high number of users.

Hauswirth and Jazayeri (1999) introduce a component and communication model for push systems. They identify components used in most *Publish/Sub-*

scribe implementations. The paper mentions possible problems with scalability, and emphasizes the necessity of a specialized, distributed, broadcasting infrastructure.

Eugster et al. (2003) compare many variants of *Publish/Subscribe* schemes. They identify three alternatives: *topic-based*, *content-based*, and *type-based*. The paper also mentions several implementation issues, such as events, transmission media and qualities of service, but again the main focus is not on web-based applications.

Martin-Flatin (1999) compares push and pull from the perspective of network management. The paper mentions the publish/subscribe paradigm and how it can be used to conserve network bandwidth as well as CPU time on the management station. Flatin suggests the 'dynamic document' solution of Netscape (1995), but also a 'position swapping' approach in which each party can both act as a client and a server. This solution, however, is not applicable to web browsers. Making a browser act like a server is not trivial and it induces security issues.

As far as we know, there has been no empirical study conducted to find out the actual tradeoffs of applying pull/push on browser-based or Ajax applications.

4.9 Concluding Remarks

In this chapter we have compared pull and push solutions for achieving web-based real time event notification and data delivery. The contributions of this chapter include:

- An experimental design permitting the analysis of pull and push approaches to web data delivery, and the identification of key metrics, such as the mean publish trip-time, received (unique) publish messages, and the received message percentage (Section 4.4).
- A reusable software infrastructure, consisting of our automated distributed Ajax performance testing framework Chiron, Grinder scripts imitating clients and a sample application written for Cometd, DWR, and *pull* (Section 4.5).
- Empirical results highlighting the tradeoffs between push and pull based approaches to web-based real time event notification, and the impact of such characteristics as the number of concurrent users, the publish interval, on, for instance, server performance (Section 4.6).

Our experiment shows that if we want high data coherence and high network performance, we should choose the push approach. Pull cannot achieve the same data coherence, even with low pull intervals. Push can also handle a high number of clients thanks to the continuations (Jetty, 2006) mechanism of Jetty, however, when the number of users increases, the reliability in receiving messages decreases.

With the pull approach, achieving total data coherence with high network performance is very difficult. If the pull interval is higher than the publish interval, some data miss will occur. If it is lower, then the network performance will suffer, in some cases pull causes as high as 7 times more network traffic compared to push. Pull performs close to push only if the pull interval equals to publish interval, but never better. Besides, in order to have pull and publish intervals equal, we need to know the exact publish interval beforehand. The publish interval on the other hand is rarely static and predictable. This makes pull useful only in situations where the data is published according to some pattern.

These results allow web engineers to make rational decisions concerning key parameters such as pull and push intervals, in relation to, e.g., the anticipated number of clients. Furthermore, the experimental design and the reusable software infrastructure allows them to repeat similar measurements for their own (existing or to be developed) applications. We have released Chiron (open source) through our website (See Section 4.5).

Our future work includes adopting and testing a hybrid approach that combines pull and push techniques for Ajax applications to gain the benefits of both approaches. In this approach for example, users can specify a maximum trip-time, and if the server is under high load, it can switch some push users to pull. We believe that such optimizations can have a positive effect on the overall performance. We also intend to extend our testing experiments with different web application containers such as Grizzly, or different push server implementations that are based on holding a permanent connection (e.g., Lightstreamer[17]) as opposed to the long polling approach discussed in this chapter.

[17] http://www.lightstreamer.com

Crawling AJAX by Inferring User Interface State Changes*

Chapter 5

AJAX is a very promising approach for improving rich interactivity and responsiveness of web applications. At the same time, AJAX techniques shatter the metaphor of a web 'page' upon which general search crawlers are based. This chapter describes a novel technique for crawling AJAX applications through dynamic analysis and reconstruction of user interface state changes. Our method dynamically infers a 'state-flow graph' modeling the various navigation paths and states within an AJAX application. This reconstructed model can be used to generate linked static pages. These pages could be used to expose AJAX sites to general search engines. Moreover, we believe that the crawling techniques that are part of our solution have other applications, such as within general search engines, accessibility improvements, or in automatically exercising all user interface elements and conducting state-based testing of AJAX applications. We present our open source tool called CRAWLJAX which implements the concepts discussed in this chapter. Additionally, we report a case study in which we apply our approach to a number of representative AJAX applications and elaborate on the obtained results.

5.1 Introduction

The web as we know it is undergoing a significant change. A technology that has gained a prominent position lately, under the umbrella of WEB 2.0, is AJAX (Asynchronous JavaScript and XML) (Garrett, 2005), in which a clever combination of JavaScript and Document Object Model (DOM) manipulation, along with asynchronous server communication is used to achieve a high level of user interactivity. Highly visible examples include Google Maps, Google Documents, and the recent version of Yahoo! Mail.

With this 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;

*This chapter was published in the Proceedings of the 8th International Conference on Web Engineering (ICWE 2008) (Mesbah et al., 2008).

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.

One way to address these challenges is through the use of a crawler that can automatically walk through different states of a highly dynamic Ajax site, create a model of the navigational paths and states, and generate a traditional linked page-based static version. The generated static pages can be used, for instance, to expose Ajax sites to general search engines or to examine the accessibility (Atterer and Schmidt, 2005) of different dynamic states. Such a crawler can also be used for conducting state-based testing of Ajax applications (Marchetto et al., 2008b) and automatically exercising all user interface elements of an Ajax site in order to find e.g., link-coverage, broken-links, and other errors.

To date, no crawler exists that can handle the complex client code that is present in Ajax applications. The reason for this is that crawling Ajax is fundamentally more difficult than crawling classical multi-page web applications. In traditional web applications, states are explicit, and correspond to pages that have a unique URL assigned to them. In Ajax applications, however, the state of the user interface is determined dynamically, through changes in the DOM that are only visible after executing the corresponding JavaScript code.

In this chapter, we propose an approach to analyze and reconstruct these user interface states automatically. Our approach is based on a crawler that can exercise client side code, and can identify clickable elements (which may change with every click) that change the state within the browser's dynamically built DOM. From these state changes, we infer a *state-flow graph*, which captures the states of the user interface, and the possible transitions between them. This graph can subsequently be used to generate a multi-page static version of the original Ajax application.

The underlying ideas have been implemented in a tool called Crawljax.[1]

We have performed an experiment of running our crawling framework over a number of representative Ajax sites to analyze the overall performance of our approach, evaluate the effectiveness in retrieving relevant clickables, assess the quality and correctness of the detected states and generated static pages, and examine the capability of our tool on real sites used in practice and the scalability in crawling sites with thousands of dynamic states and clickables. The cases span from internal to academic and external commercial Ajax web sites.

The chapter is structured as follows. We start out, in Section 5.2 by exploring the difficulties of crawling and indexing Ajax. In Sections 5.3 and 5.4, we present a detailed discussion of our new crawling techniques, the generation process, and the Crawljax tool. In Section 5.5 the results of applying our methods to a number of Ajax applications are shown, after which Section 5.6

[1]The tool is available for download from `http://spci.st.ewi.tudelft.nl/crawljax/`.

discusses the findings and open issues. Section 5.7 presents various applications of our crawling techniques. We conclude with a brief survey of related work, a summary of our key contributions, and suggestions for future work.

5.2 Challenges of Crawling Ajax

Ajax has a number of properties making it extremely difficult for, e.g., search engines to crawl such web applications.

5.2.1 Client-side Execution

The common ground for all Ajax applications is a JavaScript engine which operates between the browser and the web server, and which acts as an extension to the browser. This engine typically deals with server communication and user interface rendering. Any search engine willing to approach such an application must have support for the execution of the scripting language. Equipping a general search crawler with the necessary environment complicates its design and implementation considerably. The major search giants such as Google[2] currently have little or no support for executing JavaScript due to scalability and security issues.

5.2.2 State Changes & Navigation

Traditional web applications are based on the multi-page interface paradigm consisting of multiple (dynamically generated) unique pages each having a unique URL. In Ajax applications, not every state change necessarily has an associated Rest-based (Fielding and Taylor, 2002) URI (see Chapter 2). Ultimately, an Ajax application could consist of a single-page with a single URL. This characteristic makes it very difficult for a search engine to index and point to a specific state on an Ajax application. For crawlers, navigating through traditional multi-page web applications has been as easy as extracting and following the hypertext links (or the `src` attribute) on each page. In Ajax, hypertext links can be replaced by events which are handled by the client engine; it is not possible any longer to navigate the application by simply extracting and retrieving the internal hypertext links.

5.2.3 Dynamic Document Object Model (DOM)

Crawling and indexing traditional web applications consists of following links, retrieving and saving the HTML source code of each page. The state changes in Ajax applications are dynamically represented through the run-time changes on the DOM. This means that the source code in HTML does not represent the state anymore. Any search engine aimed at crawling and indexing such applications, will need to have access to this run-time dynamic document object model of the application.

[2] `http://googlewebmastercentral.blogspot.com/2007/11/spiders-view-of-web-20.html`

```
1  <a href="javascript:OpenNewsPage();">
2  <a href="#" onClick="OpenNewsPage();">
3  <div onClick="OpenNewsPage();">
4  <a href="news.html" class="news">
5  <input type="submit" class="news"/>
6  <div class="news">
7  <!-- jQuery function attaching events to elements
8       having attribute class="news" -->
9  $(".news").click(function() {
10   $("#content").load("news.html");
11 });
```

Figure 5.1 Different ways of attaching events to elements.

5.2.4 Delta-communication

Ajax applications rely on a delta-communication (see Chapter 2) style of interaction in which merely the state changes are exchanged asynchronously between the client and the server, as opposed to the full-page retrieval approach in traditional web applications. Retrieving and indexing the delta state changes, for instance, through a proxy between the client and the server, could have the side-effect of losing the context and actual meaning of the changes. Most of such delta updates become meaningful after they have been processed by the JavaScript engine on the client and injected into the DOM.

5.2.5 Elements Changing the Internal State

To illustrate the difficulties involved in crawling Ajax, consider Figure 5.1. It is a highly simplified example, showing different ways in which a news page can be opened.

The example code shows how in Ajax sites, it is not just the hypertext link element that forms the doorway to the next state. Note the way events (e.g., `onClick`, `onMouseOver`) can be attached to DOM elements at run-time. As can be seen, a `div` element (line 3) can have an `onclick` event attached to it so that it becomes a *clickable* element capable of changing the internal DOM state of the application when clicked. The necessary event handlers can also be programmatically registered in Ajax. The jQuery[3] code responsible (lines 9–11) for attaching the required functionality to the `onClick` event handlers using the `class` attribute of the elements can also be seen.

Finding these clickables at run-time is another non-trivial task for a crawler. Traditional crawlers as used by search engines will simply ignore all the elements (not having a proper `href` attribute) except the one in line 4, since they rely on JavaScript only.

[3] `http://jquery.com`

5.3 A Method for Crawling Ajax

The challenges discussed in the previous section will make it clear that crawling Ajax based on static analysis of, e.g., the HTML and JavaScript code is not feasible. Instead, we rely on a dynamic approach, in which we actually exercise clicks on all relevant elements in the DOM. From these clicks, we reconstruct a *state-flow graph*, which tells us in which states the user interface can be. Subsequently, we use these states to generate static, indexable, pages.

An overview of our approach is visualized in Figure 5.3. As can be seen, the architecture can be divided in two parts: (1) inferring the state machine, and (2) using the state machine to generate indexable pages.

In this section, we first summarize our state and state-flow graph definition, followed by a discussion of the most important steps in our approach.

5.3.1 User Interface States

In traditional multi-page web applications, each state is represented by a URL and the corresponding web page. In Ajax however, it is the internal structure change of the DOM tree on the (single-page) user interface that represents a state change. Therefore, to adopt a generic approach for all Ajax sites, we define a state change as a change on the DOM tree caused either by server-side state changes propagated to the client, or client-side events handled by the Ajax engine.

5.3.2 The State-flow Graph

The user interface state changes in Ajax can be modeled by recording the paths (events) to these DOM changes to be able to navigate the different states. For that purpose we define a *state-flow graph* as follows:

Definition 1 *A **state-flow graph** for an* Ajax *site* **A** *is a 3 tuple* $< \boldsymbol{r}, \boldsymbol{V}, \boldsymbol{E} >$ *where:*

1. $\boldsymbol{r}$ *is the root node (called Index) representing the initial state after* **A** *has been fully loaded into the browser.*
2. $\boldsymbol{V}$ *is a set of vertices representing the states. Each* $\boldsymbol{v} \in \boldsymbol{V}$ *represents a run-time state in* **A**.
3. $\boldsymbol{E}$ *is a set of edges between vertices. Each* $(\boldsymbol{v_1}, \boldsymbol{v_2}) \in \boldsymbol{E}$ *represents a clickable* $\boldsymbol{c}$ *connecting two states if and only if state* $\boldsymbol{v_2}$ *is reached by executing* $\boldsymbol{c}$ *in state* $\boldsymbol{v_1}$.

Our state-flow graph is similar to the *event-flow graph* (Memon et al., 2001), but different in that in the former vertices are *states*, where as in the latter vertices are *events*.

As an example of a state-flow graph, Figure 5.2 depicts the visualization of the state-flow graph of a simple Ajax site. It illustrates how from the start page three different states can be reached. The edges between states are labeled with an identification (either via its ID-attribute or via an XPath expression) of the element to be clicked in order to reach the given state. Thus, clicking on the `//DIV[1]/SPAN[4]` element in the Index state leads to the `S_1` state, from which two states are reachable namely `S_3` and `S_4`.

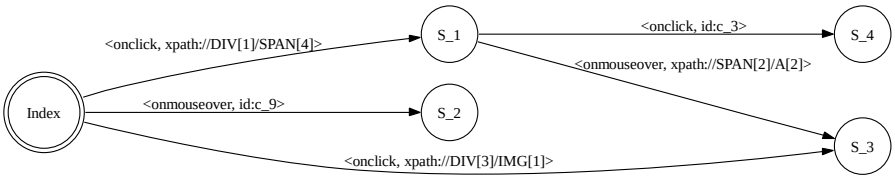

Figure 5.2 The state-flow graph visualization.

5.3.3 Inferring the State Machine

The state-flow graph is created incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed.

The following components, also shown in Figure 5.3 participate in the construction of the state flow graph:

- Embedded Browser: Our approach is based on an embedded browser interface (with different implementations: IE, Mozilla) capable of executing JavaScript and the supporting technologies required by AJAX (e.g., CSS, DOM, `XMLHttpRequest`).
- Robot: A robot is used to simulate user input (e.g., `click`, `mouseOver`, text input) on the embedded browser.
- Controller: The controller has access to the embedded browser's DOM and analyzes and detects state changes. It also controls the Robot's actions and is responsible for updating the State Machine when relevant changes occur on the DOM. After the crawling process is over, the controller also calls the Sitemap and Mirror site generator processes.
- Finite State Machine: The finite state machine is a data component maintaining the state-flow graph, as well as a pointer to the current state.

The algorithm used by these components to actually infer the state machine is shown in Algorithm 3. The start procedure (lines 1-8) takes care of initializing the various components and processes involved. The actual, recursive, crawling procedure starts at line 10: the main steps are explained below.

5.3.4 Detecting Clickables

There is no direct way of obtaining all clickable elements in a DOM-tree, due to the reasons explained in Section 5.2. Therefore, our algorithm makes use of a set of *candidate elements*, which are all exposed to an event type (e.g., click, mouseOver). We use the *click* event type to present our algorithm, note, however, that other event types can be used just as well to analyze the effects on the DOM in the same manner.

We distinguish three ways of obtaining the *candidate* elements:

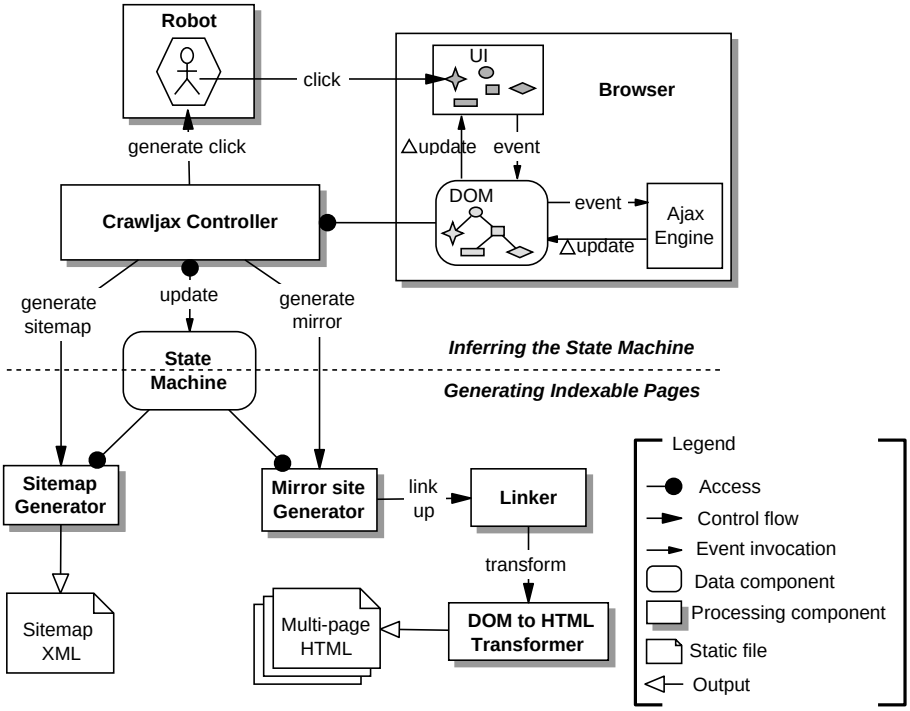

Figure 5.3 Processing view of the crawling architecture.

- In a *Full Auto Scan* mode, the candidate clickables are labeled as such based on their HTML tag element name. For example, all elements with a tag `div`, `a`, `span`, `input` are considered as candidate clickable. Tag element selection can also be constrained by the attributes (using wildcards). This is the mode that is displayed in Algorithm 3.

- In the *annotation* mode, we allow the HTML elements to have an attribute `crawljax="true"`. This gives users the opportunity to explicitly mark certain elements as to be crawled, or elements to be excluded from the process by setting the attribute to false. Note that this mode requires access to the source code of the application for applying the annotations.

- In the *configured* mode, we allow a user to specify by means of a domain-specific language which elements should be clicked (explained in more detail in Section 5.3.8). This allows the most precise control over the actual elements to be clicked.

Note that, if desirable, these modes can be combined. After the candidate elements have been found, the algorithm proceeds to determine whether these elements are indeed clickable. For each candidate element, the crawler instructs the robot to execute a click (line 15) on the element (or other event types, e.g., mouseOver), in the browser.

Algorithm 3 Full Auto Scan

Require: α is the maximum allowed depth level. τ is the similarity threshold used by the edit distance method.

```
1: procedure START (url, Set tags)
2: browser ← initEmbeddedBrowser(url)
3: robot ← initRobot()
4: sm ← initStateMachine()
5: crawl(null, 0)
6: linkupAndSaveAsHTML(sm)
7: generateSitemap(sm)
8: end procedure
9:
10: procedure CRAWL (State ps, depth)
11: if depth < α then
12:   cs ← sm.getCurrentState()
13:   Δupdate ← diff(ps, cs)
14:   Set C ← getCandidateClickables(Δupdate, tags)
15:   for c ∈ C do
16:     robot.fireEvent(c, 'click')
17:     dom ← browser.getDom()
18:     if distance(cs.getDom(), dom) > τ then
19:       xe ← getXpathExpr(c)
20:       ns ← State(c, xe, dom)
21:       sm.addState(ns)
22:       sm.addEdge(cs, ns, c, 'click')
23:       sm.changeState(ns)
24:       depth++
25:       crawl(cs, depth)
26:       depth−−
27:       sm.changeState(cs)
28:       if browser.history.canBack then
29:         browser.history.goBack()
30:       else
31:         browser.reload()
32:         List E ← sm.getShortestPathTo(cs)
33:         for e ∈ E do
34:           robot.fireEvent(e.getXpathExpr(), 'click')
35:         end for
36:       end if
37:     end if
38:   end for
39: end if
40: end procedure
```

5.3.5 Creating States

After firing an event on a candidate clickable, the algorithm compares the resulting DOM tree with the DOM tree as it was just before the event fired, in order to determine whether the event results in a state change.

For this purpose the *edit distance* between two DOM trees is calculated (line 17) using the Levenshtein (Levenshtein, 1996) method. A similarity threshold τ is used under which two DOM trees are considered clones. This threshold $(0.0 - 1.0)$ can be defined by the developer. A threshold of 0 means two DOM states are seen as clones if they are *exactly* the same in terms of structure and content. Any change is, therefore, seen as a state change.

If a change is detected according to our similarity metric, we create (line 19) a new state and add it to the state-flow graph of the state machine (line 20). In order to recognize an already met state, we compute a hashcode for each DOM state, which we use to compare every new state to the list of already visited states on the state-flow graph. Thus, in line 19 if we have a state containing the particular DOM tree already, that state is returned, otherwise a new state is created.

Furthermore, a new edge is created on the graph (line 21) between the state before the event and the current state. The element on which the event was fired is also added as part of the new edge. Moreover, the current state pointer of the state machine is also updated to this newly added state at that moment (line 22).

5.3.6 Processing Document Tree Deltas

After a clickable has been identified, and its corresponding state created, the `crawl` procedure is recursively called (line 23) to find new possible states in the changes made to the DOM tree.

Upon every new (recursive) entry into the `crawl` procedure, the first thing done (line 12) is computing the differences between the previous document tree and the current one, by means of an enhanced *Diff* algorithm (Chawathe et al., 1996; Mesbah and van Deursen, 2007b). Such "delta updates" may be due, for example, to a server request call that injects new elements into the DOM. The resulting delta updates are used to find new candidate clickables (line 13), which are then further processed in a depth-first manner.

It is worth mentioning that in order to avoid a loop, a list of visited elements is maintained to exclude already checked elements in the recursive algorithm. We use the tag name, the list of attribute names and values, and the XPath expression of each element to conduct the comparison. Additionally, a depth number can be defined to constrain the depth level of the recursive function.

5.3.7 Navigating the States

Upon completion of the recursive call, the browser should be put back into the state it was in before the call. Unfortunately, navigating (back and forth)

through an AJAX site is not as easy as navigating a classical web site. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function of the browser does not bring us to the previous state. This complicates traversing the application when crawling AJAX. We distinguish two situations:

Browser History Support

It is possible to programatically register each state change with the browser history through frameworks such as the jQuery history/remote plugin[4] or the Really Simple History library.[5] If an AJAX application has support for the browser history (line 25), then for changing the state in the browser, we can simply use the built-in history back functionality to move backwards (line 26).

Click Through From Initial State

In case the browser history is not supported, which is the case with many AJAX applications currently, the only way to get to a previous state is by saving information about the elements and the order in which their execution results in reaching to a particular state. Once we have such information, we can reload the application (line 28) and follow and execute the elements from the initial state to the desired state. As an optimization step, we use Dijkstra's shortest path algorithm (Dijkstra, 1959) to find the shortest element execution path on the graph to a certain state (line 29).

We initially considered using the ID attribute of a clickable element to find it back after a reload of the page. When we `reload` the application in the browser, all the internal objects are replaced by new ones and the ID attribute would be a way to follow the path to a certain state by clicking on those elements whose IDs have been saved in the state machine. Soon we realized that firstly, not all AJAX sites assign ID attributes to the elements and, secondly, if IDs are provided, they are not always persistent, i.e., they are dynamically set and can change with each reload.

To overcome these challenges, we adopt XPath to provide a better, more reliable, and persistent element identification mechanism. For each state changing element, we reverse engineer the XPath expression of that element which gives us its exact location on the DOM (line 18). We save this expression in the state machine (line 19) and use it to find the element after a reload, persistently (line 31).

Note that because of side effects of the element execution, there is no guarantee that we reach the exact same state when we traverse a path a second time. It is, however, as close as we can get.

```
crawl MyAjaxSite {
  url: http://spci.st.ewi.tudelft.nl/aowe/;
  navigate Nav1 {
    event: type=mouseover xpath=/HTML/BODY/SPAN[3];
    event: type=click id=headline;
    ...
  }
  navigate Nav2 {
    event: type=click
           xpath="//DIV[contains(.,"Interviews")]";
    event: type=input id=article "john doe";
    event: type=click id=search;
  } ...
}
```

Figure 5.4 An instance of CASL.

5.3.8 CASL: Crawling Ajax Specification Language

To give users control over which candidate clickables to select, we have developed a Domain Specific Language (DSL) (van Deursen et al., 2000) called Crawling AJAX Specification Language (CASL). Using CASL, the developer can define the elements (based on IDs and XPath expressions) to be clicked, along with the exact order in which the crawler should crawl the AJAX application. CASL accepts different types of `events`. The event types include `click`, `mouseover`, and `input` currently.

Figure 5.4 shows an instance of CASL. `Nav1` tells our crawler to crawl by first firing an event of type `mouseover` on the element with XPath `/HTML/BODY/SPAN[3]` and then clicking on the element with ID `headline` in that order. `Nav2` commands the crawler to crawl to the `Interviews` state, then insert the text '`john doe`' into the input element with ID `article` and afterward click on the `search` element. Using this DSL, the developer can take control of the way an AJAX site should be crawled.

5.3.9 Generating Indexable Pages

After the crawling AJAX process is finished, the created state-flow graph can be passed to the generation process, corresponding to the bottom part of Figure 5.3.

The first step is to establish links for the DOM states by following the *outgoing edges* of each state in the state-flow graph. For each clickable, the element type must be examined. If the element is a hypertext link (an `a`-element), the `href` attribute is updated. In case of other types of clickables (e.g., `div`, `span`) we replace the element by a hypertext link element. The `href` attribute in both situations represents the link to the name and location of the generated static page.

[4] `http://stilbuero.de/jquery/history/`
[5] `http://code.google.com/p/reallysimplehistory/`

Case	Ajax site	Clickable Elements
C1	spci.st.ewi.tudelft.nl/demo/aowe/	testing span 2 Second link Topics of Interest
C2	PetStore	Hairy Cat
C3	www.4launch.nl	<div onclick="setPrefCookies('Gaming', 'DESTROY', 'DESTROY'); loadHoofdCatsTree('Gaming', 1, '')">Gaming <td onclick="open_url('..producteninfo.php?productid=037631',..)">Harddisk Skin</td>
C4	www.blindtextgenerator.com	<input type="radio" value="7" name="radioTextname" class="js-textname iradio" id="idRadioTextname-EN-li-europan"/> <a id="idSelectAllText" title="Select all" href="#">
C5	site.snc.tudelft.nl	organisatie ...
C6	www.gucci.com[a]	booties <div class="darkening"/>

[a] http://www.gucci.com/nl/uk-english/nl/spring-summer-08/womens-shoes/

Table 5.1 Case objects and examples of their clickable elements.

After the linking process, each DOM object in the state-flow graph is transformed into the corresponding HTML string representation and saved on the file system in a dedicated directory (e.g., /generated/). Each generated static file represents the style, structure, and content of the Ajax application as seen in the browser, in exactly its specific state at the time of crawling.

Here, we can adhere to the Sitemap Protocol,[6] generating a valid instance of the protocol automatically after each crawling session consisting of the URLs of all generated static pages.

5.4 Tool Implementation: Crawljax

We have implemented the concepts presented in this chapter in a tool called Crawljax. Crawljax is released under the open source BSD license and is available for download. More information about the tool can be found on our website http://spci.st.ewi.tudelft.nl/crawljax/.

Crawljax is implemented in Java. We have engineered a variety of software libraries and web tools to build and run Crawljax. Here we briefly mention the main modules and libraries.

The embedded browser interface has two implementations: IE-based on Watij[7] and Mozilla-based on XULRunner.[8] Webclient[9] is used to access the run-time DOM and the browser history mechanism in the Mozilla browser. For the Mozilla version, the Robot component makes use of the java.awt.Robot class to generate native system input events on the embedded browser. The IE version uses an internal Robot to simulate events.

[6] http://www.sitemaps.org/protocol.php
[7] http://watij.com
[8] http://developer.mozilla.org/en/docs/XULRunner/
[9] http://www.mozilla.org/projects/blackwood/webclient/

The generator uses JTidy[10] to pretty-print DOM states and Xerces[11] to serialize the objects to HTML. In the Sitemap Generator, XMLBeans[12] generates Java objects from the Sitemap Schema,[13] which after being used by CRAWLJAX to create new URL entries, are serialized to the corresponding valid XML instance document.

The *state-flow graph* is based on the JGrapht[14] library. The grammar of CASL is implemented in ANTLR.[15] ANTLR is used to generate the necessary parsers for CASL. In addition, StringTemplate[16] is used for generating the source-code from CASL. Log4j is used to optionally log various steps in the crawling process, such as the identification of DOM changes and clickables. CRAWLJAX is entirely based on Maven[17] to generate, compile, test (JUnit), release, and run the application.

5.5 Case Studies

In order to evaluate the effectiveness, correctness, performance, and scalability of the proposed crawling method for AJAX, we have conducted a number of case studies, which are described in this section, following Yin's guidelines for conducting case studies (Yin, 2003).

5.5.1 Subject Systems

We have selected 6 AJAX sites for our experiment as shown in Table 5.1. The case ID, the actual site, and a number of real clickables to illustrate the type of the elements can be seen for each case object.

Our selection criteria include the following: sites that use AJAX to change the state of the application by using JavaScript, assigning events to HTML elements, asynchronously retrieving delta updates from the server and performing partial updates on the DOM.

The first site C1 in our case study is an AJAX test site developed internally by our group using the jQuery AJAX library. Although the site is small, it is representative by having different types of dynamically set clickables as shown in Figure 5.1 and Table 5.1.

Our second case object, C2, is Sun's Ajaxified PETSTORE 2.0[18] which is built on the Java ServerFaces, and the Dojo AJAX toolkit. This open-source web application is designed to illustrate how the Java EE Platform can be used to develop an AJAX-enabled Web 2.0 application and adopts many advanced rich AJAX components.

[10] http://jtidy.sourceforge.net
[11] http://xerces.apache.org/xerces-j/
[12] http://xmlbeans.apache.org
[13] http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd
[14] http://jgrapht.sourceforge.net
[15] http://www.antlr.org
[16] http://www.stringtemplate.org
[17] http://maven.apache.org
[18] http://java.sun.com/developer/releases/petstore/

The other four cases are all external Ajax sites and we have no access to their source-code. C4 is an Ajax site that can function as a tool for comparing the visual impression of different typefaces. C3 (online shop), C5 (sport center), and C6 (Gucci) are all single-page commercial sites with many clickables and states.

5.5.2 Experimental Design

Our goals in conducting the experiment include:

G1 Effectiveness: evaluating the effectiveness of obtaining high-quality results in retrieving relevant clickables including the ones dynamically injected into the DOM,

G2 Correctness: assessing the quality and correctness of the states and static pages automatically generated,

G3 Performance: analyzing the overall performance of our approach in terms of input size versus time,

G4 Scalability: examining the capability of Crawljax on real sites used in practice and the scalability in crawling sites with thousands of dynamic states and clickables.

Environment & Tool Configuration.
We use a laptop with Intel Pentium M 765 processor 1.73GHz, with 1GB RAM and Windows XP to run Crawljax.

Configuring Crawljax itself is done through a simple `crawljax.properties` file, which can be used to set the URL of the site to be analyzed, the tag elements Crawljax should look for, the depth level, and the similarity threshold. There are also a number of other configuration parameters that can be set, such as the directory in which the generated pages should be saved in.

Output.
We determine the average DOM string size, number of candidate elements, number of detected clickables, number of detected states, number of generated static pages, and performance measurements for crawling and generating pages separately for each experiment object. The actual generated linked static pages also form part of the output.

Method of Evaluation.
Since other comparable tools and methods are currently not available to conduct similar experiments as with Crawljax, it is difficult to define a baseline against which we can compare the results. Hence, we manually inspect the systems under examination and determine which expected behavior should form our reference baseline.

G1. For the experiment we have manually added extra clickables in different states of C1, especially in the delta updates, to explore whether clickables dynamically injected into the DOM can be found by CrawlJax. A reference model was created manually by clicking through the different states in a browser. In total 16 clickables were noted of which 10 were on the top level, i.e., index state. To constrain the reference model for C2, we chose two product categories, namely CATS and DOGS, from the five available categories. We annotated 36 elements (product items) by modifying a JavaScript method which turns the items retrieved from the server into clickables on the interface. For the four external sites (C3–C6) which have many states, it is very difficult to manually inspect and determine, for instance, the number of expected clickables and states. Therefore, for each site, we randomly selected 10 clickables in advance by noting their tag name, attributes, and XPath expression. After each crawling process, we checked the presence of the 10 elements among the list of detected clickables.

G2. After the generation process the generated HTML files and their content are manually examined to see whether the pages are the same as the corresponding DOM states in Ajax in terms of *structure*, *style*, and *content*. Also the internal linking of the static pages is manually checked. To test the clone detection ability we have intentionally introduced a clone state into C1.

G3. We measure the time in milliseconds taken to crawl each site. We expect the crawling performance to be directly proportional to the input size which is comprised of the average DOM string size, number of candidate elements, and number of detected clickables and states.

We also measure the generation performance which is the period taken to generate the static HTML pages from the inferred state-flow graph.

G4. To test the capability of our method in crawling real sites and coping with unknown environments, we run CrawlJax on four external cases C3–C6. We run CrawlJax with depth level 2 on C3 and C5 each having a huge state space to examine the scalability of our approach in analyzing tens of thousands of candidate clickables and finding clickables.

5.5.3 Results and Evaluation

Table 5.2 presents the results obtained by running CrawlJax on the subject systems. The measurements were all read from the log file produced by CrawlJax at the end of each process.

G1. As can be seen in Table 5.2, for C1 CrawlJax finds all the 16 expected clickables and states with a precision and recall of 100%.

For C2, 33 elements were detected from the annotated 36. One explanation behind this difference could be the way some items are shown to the user in PetStore. PetStore uses a `Catalog Browser` to show a set of the total number of the product items. The 3 missing product items could be the ones that were

Case	DOM string size (byte)	Candidate Elements	Detected Clickables	Detected States	Generated Static Pages	Crawl Performance (ms)	Generation Performance (ms)	Depth	Tags
C1	4590	540	16	16	16	14129	845	3	A, DIV, SPAN, IMG
C2	24636	1813	33	34	34	26379	1643	2	A, IMG
C3	262505	150	148	148	148	498867	17723	1	A
		19247	1101	1071	1071	5012726	784295	2	A, TD
C4	40282	3808	55	56	56	77083	2161	2	A, DIV, INPUT, IMG
C5	165411	267	267	145	145	806334	14395	1	A
		32365	1554	1234	1234	6436186	804139	2	A, DIV
C6	134404	6972	83	79	79	701416	28798	1	A, DIV

Table 5.2 Results of running CRAWLJAX on 6 AJAX applications.

never shown on the interface because of the navigational flow e.i., the order of clickables.

CRAWLJAX was able to find 95% of the expected 10 clickables (noted initially) for each of the four external sites C3–C6.

G2. The clone state introduced in C1 is correctly detected and that is why we see 16 states being reported instead of 17. Inspection of the static pages in all cases shows that the generated pages correspond correctly to the DOM state.

G3. When comparing the results for the two internal sites, we see that it takes CRAWLJAX 14 and 26 seconds to crawl C1 and C2 respectively. As can be seen, the DOM in C2 is 5 times and the number of candidate elements 3 times higher. In addition to the increase in DOM size and the number of candidate elements, CRAWLJAX cannot rely on the browser Back method when crawling C2. This means for every state change on the browser CRAWLJAX has to reload the application and click through to the previous state to go further. This reloading and clicking through has a negative effect on the performance. The generation time also doubles for C2 due to the increase in the input size. It is clear that the running time of CRAWLJAX increases linearly with the size of the input. We believe that the execution time of a few minutes to crawl and generate a mirror multi-page instance of an AJAX application automatically without any human intervention is very promising. Note that the performance is also dependent on the CPU and memory of the machine CRAWLJAX is running on, as well as the speed of the server and network properties of the case site. C6, for instance, is slow in reloading and retrieving updates from its server and that increases the performance measurement numbers in our experiment.

G4. CRAWLJAX was able to run smoothly on the external sites. Except a few minor adjustments (see Section 5.6) we did not witness any difficulties. C3 with depth level 2 was crawled successfully in 83 minutes resulting in

19247 examined candidate elements, 1101 detected clickables, and 1071 detected states. The generation process for the 1071 states took 13 minutes. For C5, CrawlJax was able to finish the crawl process in 107 minutes on 32365 candidate elements, resulting in 1554 detected clickables and 1234 states. The generation process took 13 minutes. As expected, in both cases, increasing the depth level from 1 to 2 expands the state space greatly.

5.6 Discussion

5.6.1 Back Implementation

CrawlJax assumes that if the Browser Back functionality is implemented, then it is implemented correctly. An interesting observation was the fact that even though Back is implemented for some states, it is not correctly implemented i.e., calling the Back method brings the browser in a different state than expected which naturally confuses CrawlJax. This implies that the Back method to go to a previous state is not reliable and using the reload and click-through method is much more safe in this case.

The click-through method is not without limitations either. When the behavior of an Ajax application, in terms of the client-side user interface changes, is non-deterministic, the click-through method is likely to fail in finding an exact click-path and state match. The order in which clickables are chosen could generate different states. Even executing the same clickable twice from an state could theoretically produce two different DOM states depending on, for instance, server-side factors. We are currently exploring ways to cope with this issue. One possible direction is seeking a way to re-detect elements on the click-path, even if their location is changed after a reload. Another possibility is resetting the server-side state after a reload, which requires access to the server-side logic.

5.6.2 Constantly Changing DOM

Another interesting observation in C2 in the beginning of the experiment was that every element was seen as a clickable. This phenomenon was caused by the `banner.js` which constantly changed the DOM with textual notifications. Hence, we had to either disable this banner to conduct our experiment or use a higher similarity threshold so that the textual changes were not seen as a relevant state change for detecting clickables.

5.6.3 Cookies

Cookies can also cause some problems in crawling Ajax applications. C3 uses Cookies to store the state of the application on the client. With Cookies enabled, when CrawlJax reloads the application to navigate to a previous state, the application does not start in the expected initial state. In this case, we had to disable Cookies to perform a correct crawling process.

5.6.4 State Space

The set of found states and generated HTML pages is by no means complete, i.e., CRAWLJAX generates a static instance of the AJAX application but not necessarily *the* instance. This is partly inherent in dynamic web applications. Any crawler can only crawl and index a snapshot instance of a dynamic web application in a point of time.

The number of possible states in the state space of almost any realistic web application is huge and can cause the well-know *state explosion problem* (Valmari, 1998). Just as a traditional web crawler, CRAWLJAX provides the user with a set of configurable options to constrain the state space such as the maximum search depth level, the similarity threshold, maximum number of states per domain, maximum crawling time, and the option of ignoring external links and links that match some pre-defined set of regular expressions, e.g., mail:*, *.ps, *.pdf.

The current implementation of CRAWLJAX keeps the DOM states in the memory which can lead to an state explosion and out of memory exceptions with approximately 3000 states on a machine with a 1GB RAM. As an optimization step we intend to abstract and serialize the DOM state into a database and only keep a reference in the memory. This saves much space in the memory and enables us to handle much more states. With a cache mechanism, the essential states for analysis can be kept in the memory while the other ones can be retrieved from the database when needed in a later stage.

5.7 Applications

As mentioned in the introduction, we believe that the crawling and generating capabilities of our approach have many applications for AJAX sites.

5.7.1 Search Engines

We believe that the crawling techniques that are part of our solution can serve as a starting point and be adopted by general search engines to be able to crawl AJAX sites. General web search engines, such as Google and Yahoo!, cover only a portion of the web called the *publicly indexable web* which consists of the set of web pages reachable purely by following hypertext links, ignoring forms (Barbosa and Freire, 2007) and client-side scripting. The pages not reached this way are referred to as the *hidden-web*, which is estimated to comprise several millions of pages (Barbosa and Freire, 2007). With the wide adoption of AJAX techniques that we are witnessing today this figure will only increase. Although there has been extensive research on crawling and exposing the data behind forms (Barbosa and Freire, 2007; de Carvalho and Silva, 2004; Lage et al., 2004; Ntoulas et al., 2005; Raghavan and Garcia-Molina, 2001), crawling the hidden-web induced as a result of client-side scripting in general and AJAX in particular has gained very little attention so far. Consequently, while AJAX techniques are very promising in terms of improving

rich interactivity and responsiveness, Ajax sites themselves may very well be ignored by the search engines.

5.7.2 Discoverability

There are some industrial proposed techniques that assist in making a modern Ajax website more accessible and discoverable by general search engines.

Graceful Degradation

In web engineering terms, the concept behind *Graceful Degradation* (Florins and Vanderdonckt, 2004) is to design and build for the latest and greatest user-agent and then add support for less capable devices, i.e., focus on the majority on the mainstream and add some support for outsiders. Graceful Degradation allows a web site to 'step down' in such a way as to provide a reduced level of service rather than failing completely. A well-known example is the menu bar generated by JavaScript which would normally be totally ignored by search engines. By using HTML list items with hypertext links inside a `noscript` tag, the site can degrade gracefully.

Progressive Enhancement

The term *Progressive Enhancement*[19] has been used as the opposite side to Graceful Degradation. This technique aims for the lowest common denominator, i.e., a basic markup HTML document, and begins with a simple version of the web site, then adds enhancements and extra rich functionality for the more advanced user-agents using CSS and JavaScript.

Server-side Generation

Another way to expose the hidden-web content behind Ajax applications is by making the content available to search engines at the server-side by providing it in an accessible style. The content could, for instance, be exposed through RSS feeds. In the spirit of Progressive Enhancement, an approach called *Hijax*[20] involves building a traditional multi-page website first. Then, using unobtrusive event handlers, links and form submissions are intercepted and routed through the `XMLHttpRequest` object. Generating and serving both the Ajax and the multi-page version depending on the visiting user-agent is yet another approach. Another option is the use of XML/XSLT to generate indexable pages for search crawlers (Backbase, 2005). In these approaches, however, the server-side architecture will need to be quite modular, capable of returning delta changes as required by Ajax, as well as entire pages.

[19] `http://hesketh.com/publications/progressive_enhancement_paving_way_for_future.html`
[20] `http://www.domscripting.com/blog/display/41`

Mirror Site Generation

The Graceful Degradation and Progressive Enhancement approaches mentioned constrain the use of AJAX and have limitations in the content exposing degree. It is very hard to imagine a single-page desktop-style AJAX application that degrades into a plain HTML website using the same markup and client-side code. The more complex the AJAX functionality, the higher the cost of weaving advanced and accessible functionality into the components.[21] The server-side generation approaches increase the complexity, development costs, and maintainability effort as well. We believe our proposed solution can assist the web developer in the automatic generation of the indexable version of their AJAX application (Mesbah and van Deursen, 2008b), thus significantly reducing the cost and effort of making AJAX sites more accessible to search engines. Such an automatically built mirror site can also improve the accessibility[22] of the application towards user-agents that do not support JavaScript.

5.7.3 Testing

When it comes to states that need textual input from the user (e.g., input forms) CASL can be very helpful to crawl and generate the corresponding state. The Full Auto Scan, however, does not have the knowledge to provide such input automatically. Therefore, we believe a combination of the three modes to take the best of each could provide us with a tool not only for crawling but also for automatic testing of AJAX applications.

The ability to automatically exercise all the executable elements of an AJAX site gives us a powerful test mechanism. The crawler can be utilized to find abnormalities in AJAX sites. As an example, while conducting the case study, we noticed a number of *404 Error*s and exceptions on C_3 and C_4 sites. Such errors can easily be detected and traced back to the elements and states causing the error state in the inferred state-flow graph. The asynchronous interaction in AJAX can cause race conditions (Chapter 2) between requests and responses, and the dynamic DOM updates can also introduce new elements which can be sources of faults. Detection of such conditions by analyzing the generated state machine and static pages can be assisted as well. In addition, testing AJAX sites for compatibility on different browsers (e.g., IE, Mozilla) can be automated using CRAWLJAX.

The crawling methods and the produced state machine can be applied in conducting state machine testing (Andrews et al., 2005) for automatic test case derivation, verification, and validation based on pre-defined conditions for AJAX applications. Chapter 6 presents our automatic testing approach that is based on CRAWLJAX.

[21] http://blogs.pathf.com/agileajax/2007/10/accessibility-a.html
[22] http://bexhuff.com/node/165

5.8 Related Work

The concept behind CrawlJax, is the opposite direction of our earlier work RetJax presented in Chapter 3, in which we try to reverse engineer a traditional multi-page website to Ajax.

The work of Memon et al. (Memon et al., 2003, 2001) on GUI Ripping for testing purposes is related to our work in terms of how they reverse engineer an event-flow graph of desktop GUI applications by applying dynamic analysis techniques.

There are some industrial proposed approaches for improving the accessibility and discoverability of Ajax as discussed in Section 5.7.

There has been extensive research on crawling the hidden-web behind forms (Barbosa and Freire, 2007; Dasgupta et al., 2007; de Carvalho and Silva, 2004; Lage et al., 2004; Ntoulas et al., 2005; Raghavan and Garcia-Molina, 2001). This is sharp contrast with the the hidden-web induced as a result of client-side scripting in general and Ajax in particular, which has gained very little attention so far. As far as we know, there are no academic research papers on crawling Ajax at the moment.

5.9 Concluding Remarks

Crawling Ajax is the process of turning a highly dynamic, interactive web-based system into a static mirror site, a process that is important to improve searchability, testability, and accessibility of Ajax applications. This chapter proposes a crawling method for Ajax. The main contributions of the chapter are:

- An analysis of the key problems involved in crawling Ajax applications;
- A systematic process and algorithm to infer a state machine from an Ajax application, which can be used to generate a static mirror site. Challenges addressed include the identification of clickable elements, the detection of DOM changes, and the construction of the state machine;
- The open source tool CrawlJax, which implements this process;
- Six case studies used to evaluate the effectiveness, correctness, performance, and scalability of the proposed approach.

Although we have been focusing on Ajax in this chapter, we believe that the approach could be applied to any DOM-based web application.

Future work consists of conducting more case studies to improve the ability of finding clickables in different Ajax settings. The fact that the tool is available for download for everyone, will help to identify exciting case studies. Furthermore, strengthening the tool by extending its functionality, improving the performance, and the state explosion optimization are other directions we

foresee. Exposing the hidden-web induced by AJAX using CRAWLJAX and conducting automatic state-based testing of AJAX application based on the reverse engineering techniques are other applications we will be working on.

Invariant-Based Automatic Testing of Ajax User Interfaces*

Chapter 6

Ajax-based Web 2.0 applications rely on stateful asynchronous client/server communication, and client-side run-time manipulation of the DOM tree. This not only makes them fundamentally different from traditional web applications, but also more error-prone and harder to test. We propose a method for testing Ajax applications automatically, based on a crawler to infer a flow graph for all (client-side) user interface states. We identify Ajax-specific faults that can occur in such states (related to DOM validity, error messages, discoverability, back-button compatibility, etc.) as well as DOM-tree invariants that can serve as oracle to detect such faults. We implemented our approach in Atusa, a tool offering generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite covering the paths obtained during crawling. We describe two case studies evaluating the fault revealing capabilities, scalability, required manual effort and level of automation of our approach.

6.1 Introduction

Many new web trends have recently appeared under the Web 2.0 umbrella, changing the web significantly, from read-only static pages to dynamic user-created content and rich interaction. Many Web 2.0 sites rely heavily on Ajax (Asynchronous JavaScript and XML) (Garrett, 2005), a prominent enabling technology in which a clever combination of JavaScript and Document Object Model (DOM) manipulation, along with asynchronous client/server delta-communication (Mesbah and van Deursen, 2008a) is used to achieve a high level of user interactivity on the web.

With this new change comes a whole set of new challenges, mainly due to the fact that Ajax shatters the metaphor of a web 'page' upon which many classic web technologies are based. One of these challenges is testing such applications (Bozdag et al., 2009; Marchetto et al., 2008b; Mesbah et al., 2008). With the ever-increasing demands on the quality of Web 2.0 applications, new techniques and models need to be developed to test this new class of software. How to automate such a testing technique is the question that we address in this paper.

In order to detect a fault, a testing method should meet the following conditions (Morell, 1988; Richardson and Thompson, 1988): *reach* the fault-execution, which causes the fault to be executed, *trigger* the error-creation,

*This chapter has been accepted for publication in the Proceedings of the 31st International Conference on Software Engineering (ICSE 2009) (Mesbah and van Deursen, 2009).

which causes the fault execution to generate an incorrect intermediate state, and *propagate* the error, which enables the incorrect intermediate state to propagate to the output and cause a detectable output error.

Meeting these reach/trigger/propagate conditions is more difficult for Ajax applications compared to classical web applications. During the past years, the general approach in testing web applications has been to request a response from the server (via a hypertext link) and to analyze the resulting HTML. This testing approach based on the page-sequence paradigm has serious limitations meeting even the first (reach) condition on Ajax sites. Recent tools such as Selenium[1] use a capture/replay style for testing Ajax applications. Although such tools are capable of executing the fault, they demand a substantial amount of manual effort on the part of the tester.

Static analysis techniques have limitations in revealing faults which are due to the complex run-time behavior of modern rich web applications. It is this dynamic run-time interaction that is believed (Huang et al., 2005) to make testing such applications a challenging task. On the other hand, when applying dynamic analysis on this new domain of web, the main difficulty lies in detecting the various doorways to different dynamic states and providing proper interface mechanisms for input values.

In this paper, we discuss challenges of testing Ajax (Section 6.3) and propose an automated testing technique for finding faults in Ajax user interfaces. We extend our Ajax crawler, Crawljax (Sections 6.4–6.5), to infer a state-flow graph for all (client-side) user interface states. We identify Ajax-specific faults that can occur in such states and generic and application-specific invariants that can serve as oracle to detect such faults (Section 6.6). From the inferred graph, we automatically generate test cases (Section 6.7) that cover the paths discovered during the crawling process. In addition, we use our open source tool called Atusa (Section 6.8), implementing the testing technique, to conduct a number of case studies (Section 6.9) to discuss (Section 6.10) and evaluate the effectiveness of our approach.

6.2 Related Work

Modern web interfaces incorporate client-side scripting and user interface manipulation which is increasingly separated from server-side application logic (Stepien et al., 2008). Although the field of rich web interface testing is mainly unexplored, much knowledge may be derived from two closely related fields: traditional web testing and GUI application testing.

Traditional Web Testing. Benedikt et al. (2002) present VeriWeb, a tool for automatically exploring paths of multi-page web sites through a crawler and detector for abnormalities such as navigation and page errors (which are configurable through plugins). VeriWeb uses SmartProfiles to extract candidate input values for form-based pages. Although VeriWeb's crawling algorithm

[1] `http://selenium.openqa.org`

has some support for client-side scripting execution, the paper provides insufficient detail to determine whether it would be able to cope with modern AJAX web applications. VeriWeb offers no support for generating test suites as we do in Section 6.7.

Tools such as WAVES (Huang et al., 2005) and SecuBat (Kals et al., 2006) have been proposed for automatically assessing web application security. The general approach is based on a crawler capable of detecting data entry points which can be seen as possible points of security attack. Malicious patterns, e.g., SQL and XSS vulnerabilities, are then injected into these entry points and the response from the server is analyzed to determine vulnerable parts of the web application.

A model-based testing approach for web applications was proposed by Ricca and Tonella (2001). They introduce ReWeb, a tool for creating a model of the web application in UML, which is used along with defined coverage criteria to generate test-cases. Another approach was presented by Andrews et al. (2005), who rely on a finite state machine together with constraints defined by the tester. All such model-based testing techniques focus on classical multi-page web applications. They mostly use a crawler to infer a navigational model of the web. Unfortunately, traditional web crawlers are not able to crawl AJAX applications (see Chapter 5).

Logging user session data on the server is also used for the purpose of automatic test generation (Elbaum et al., 2003; Sprenkle et al., 2005). This approach requires sufficient interaction of real web users with the system to generate the necessary logging data. Session-based testing techniques are merely focused on synchronous requests to the server and lack the complete state information required in AJAX testing. Delta-server messages (Mesbah and van Deursen, 2008a) from the server response are hard to analyze on their own. Most of such delta updates become meaningful after they have been processed by the client-side engine on the browser and injected into the DOM.

Exploiting static analysis of server-side implementation logic to abstract the application behavior is another testing approach. Artzi et al. (2008) propose a technique and a tool called Apollo for finding faults in PHP web applications that is based on combined concrete and symbolic execution. The tool is able to detect run-time errors and malformed HTML output. Halfond and Orso (2007) present their static analysis of server-side Java code to extract web application request parameters and their potential values. Such techniques have limitations in revealing faults that are due to the complex run-time behavior of modern rich web applications.

GUI Application Testing. Reverse engineering a model of the desktop (GUI), to generate test cases has been proposed by Memon (2007). AJAX applications can be seen as a hybrid of desktop and web applications, since the user interface is composed of components and the interaction is event-based. However, AJAX applications have specific features, such as the asynchronous client/-server communication and dynamic DOM-based user interface, which make

them different from traditional GUI applications (Marchetto et al., 2008b), and therefore require other testing tools and techniques.

Current Ajax Testing Approaches. The server-side of AJAX applications can be tested with any conventional testing technique. On the client, testing can be performed at different levels. Unit testing tools such as JsUnit[2] can be used to test JAVASCRIPT on a functional level. The most popular AJAX testing tools are currently capture/replay tools such as Seleninum, WebKing,[3] and Sahi,[4] which allow DOM-based testing by capturing events fired by user (tester) interaction. Such tools have access to the DOM, and can assert expected UI behavior defined by the tester and replay the events. Capture/replay tools demand, however, a substantial amount of manual effort on the part of the tester (Memon, 2007).

Marchetto et al. (2008b) have recently proposed an approach for state-based testing of AJAX applications. They use traces of the application to construct a finite state machine. Sequences of semantically interacting events in the model are used to generate test cases once the model is refined by the tester. In our approach, we crawl the AJAX application, simulating real user events on the user interface and infer the abstract model automatically.

6.3 Ajax Testing Challenges

In AJAX applications, the state of the user interface is determined dynamically, through event-driven changes in the browser's DOM that are only visible after executing the corresponding JAVASCRIPT code. The resulting challenges can be explained through the reach/trigger/propagate conditions as follows.

6.3.1 Reach

The event-driven nature of AJAX presents the first serious testing difficulty, as the event model of the browser must be manipulated instead of just constructing and sending appropriate URLs to the server. Thus, simulating user events on AJAX interfaces requires an environment equipped with all the necessary technologies, e.g., JAVASCRIPT, DOM, and the XMLHttpRequest object used for asynchronous communication.

One way to *reach* the fault-execution automatically for AJAX is by adopting a web crawler, capable of detecting and firing events on clickable elements on the web interface. Such a crawler should be able to exercise all user interface events of an AJAX site, crawl through different UI states and infer a model of the navigational paths and states.

[2] http://jsunit.net
[3] http://www.parasoft.com/jsp/products/home.jsp?product=WebKing
[4] http://sahi.co.in/w/

6.3.2 Trigger

Once we are able to derive different dynamic states of an Ajax application, possible faults can be triggered by generating UI events. In addition input values can cause faulty states. Thus, it is important to identify input data entry points, which are primarily comprised of DOM forms. In addition, executing different sequences of events can also trigger an incorrect state. Therefore, we should be able to generate and execute different event sequences.

6.3.3 Propagate

In Ajax, any response to a client-side event is injected into the single-page interface and therefore, faults propagate to and are manifested at the DOM level. Hence, access to the dynamic run-time DOM is a necessity to be able to analyze and detect the propagated errors.

Automating the process of assessing the correctness of test case output is a challenging task, known as the oracle problem (Weyuker, 1982). Ideally a tester acts as an oracle who knows the expected output, in terms of DOM tree, elements and their attributes, after each state change. When the state space is huge, it becomes practically impossible. In practice, a baseline version, also known as the Gold Standard (Binder, 1999), of the application is used to generate the expected behavior. Oracles used in the web testing literature are mainly in the form of HTML comparators (Sprenkle et al., 2007) and validators (Artzi et al., 2008).

6.4 Deriving Ajax States

Here, we briefly outline our Ajax crawling technique and tool called Crawljax (see Chapter 5) and the extensions for testing purposes. Crawljax can exercise client side code, and identify clickable elements that change the state within the browser's dynamically built DOM. From these state changes, we infer a *state-flow graph*, which captures the states of the user interface, and the possible event-based transitions between them.

We define an Ajax UI state change as a change on the DOM tree caused either by server-side state changes propagated to the client, or client-side events handled by the Ajax engine. We model such changes by recording the paths (events) to these DOM changes to be able to navigate between the different states.

As an example of a state-flow graph, Figure 6.1 displays the state-flow graph of a simple Ajax site. From the start page three different states can be reached. The edges between states are labeled with an identification (ID-attribute or an XPath expression) of the element and the event type to reach the next state.

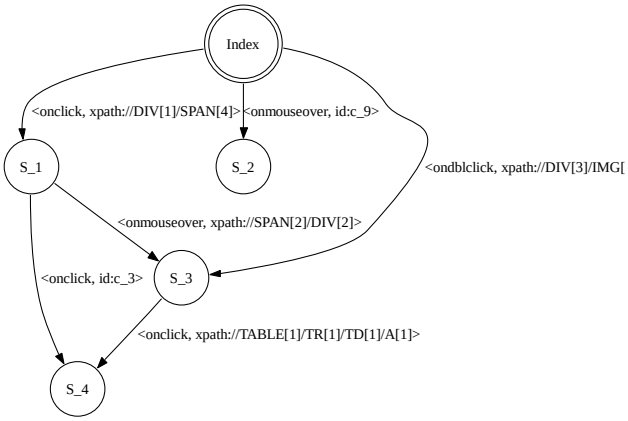

Figure 6.1 The state-flow graph visualization.

Inferring the State Machine. The state-flow graph is created incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed.

The following components participate in the construction of the graph: CRAWLJAX uses an *embedded browser* interface (with different implementations: IE, Mozilla) supporting technologies required by AJAX; A *robot* is used to simulate user input (e.g., `click`, `mouseOver`, text input) on the embedded browser; The *finite state machine* is a data component maintaining the state-flow graph, as well as a pointer to the current state; The *controller* has access to the browser's DOM and analyzes and detects state changes. It also controls the robot's actions and is responsible for updating the state machine when relevant changes occur on the DOM. The algorithm used by these components to actually infer the state machine is discussed below: the full algorithm along with its testing-specific extensions is shown in Algorithms 4 and 5 (Section 6.8).

Detecting Clickables. CRAWLJAX implements an algorithm which makes use of a set of *candidate elements*, which are all exposed to an event type (e.g., click, mouseOver). In automatic mode, the candidate clickables are labeled as such based on their HTML tag element name and attribute constraints. For instance, all elements with a tag `div`, `a`, and `span` having attribute `class="menuitem"` are considered as candidate clickable. For each candidate element, the crawler fires a click on the element (or other event types, e.g., mouseOver), in the embedded browser.

Creating States. After firing an event on a candidate clickable, the algorithm compares the resulting DOM tree with the way as it was just before the event fired, in order to determine whether the event results in a state change. If a change is detected according to the Levenshtein edit distance, a new state is created and added to the state-flow graph of the state machine. Furthermore,

a new edge is created on the graph between the state before the event and the current state.

Processing Document Tree Deltas. After a new state has been detected, the crawling procedure is recursively called to find new possible states in the partial changes made to the DOM tree. CRAWLJAX computes the differences between the previous document tree and the current one, by means of an enhanced *Diff* algorithm to detect AJAX partial updates which may be due to a server request call that injects new elements into the DOM.

Navigating the States. Upon completion of the recursive call, the browser should be put back into the previous state. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function of the browser is usually insufficient. To deal with this AJAX crawling problem, we save information about the elements and the order in which their execution results in reaching a given state. We then can reload the application and follow and execute the elements from the initial state to the desired state. CRAWLJAX adopts XPath to provide a reliable, and persistent element identification mechanism. For each state changing element, it reverse engineers the XPath expression of that element which returns its exact location on the DOM. This expression is saved in the state machine and used to find the element after a reload. Note that because of side effects of the element execution and server-side state, there is no guarantee that we reach the exact same state when we traverse a path a second time. It is, however, as close as we can get.

6.5 Data Entry Points

In order to provide input values on AJAX web applications, we have adopted a reverse engineering process, similar to (Benedikt et al., 2002; Huang et al., 2005), to extract all exposed data entry points. To this end, we have extended our crawler with the capability of detecting *DOM forms* on each newly detected state (this extension is also shown in Algorithm 3).

For each new state, we extract all form elements from the DOM tree. For each form, a `hashcode` is calculated on the attributes (if available) and the HTML structure of the input fields of the form. With this hashcode, custom values are associated and stored in a database, which are used for all forms with the same code.

If no custom data fields are available yet, all data, including input fields, their default values, and options are extracted from the DOM form. Since in AJAX forms are usually sent to the server through JAVASCRIPT functions, the `action` attribute of the form does not always correspond to the server-side entry URL. Also, any element (e.g., A, DIV) could be used to trigger the right JAVASCRIPT function to submit the form. In this case, the crawler tries to identify the element that is responsible for form submission. Note that the tester

can always verify the *submit* element and change it in the database, if necessary. Once all necessary data is gathered, the form is inserted automatically into the database. Every input form provides thus a data entry point and the tester can later alter the database with additional desired input values for each form.

If the crawler does find a match in the database, the input values are used to fill the DOM form and submit it. Upon submission, the resulting state is analyzed recursively by the crawler and if a valid state change occurs the state-flow graph is updated accordingly.

6.6 Testing Ajax States Through Invariants

With access to different dynamic DOM states we can check the user interface against different constraints. We propose to express those as invariants on the DOM tree, which we thus can check automatically in any state. We distinguish between invariants on the DOM-tree, between DOM-tree states, and application-specific invariants. Each invariant is based on a fault model (Binder, 1999), representing Ajax-specific faults that are likely to occur and which can be captured through the given invariant.

6.6.1 Generic DOM Invariants

Validated DOM. Malformed HTML code can be the cause of many vulnerability and browser portability problems. Although browsers are designed to tolerate HTML malformedness to some extent, such errors have led to browser crashes and security vulnerabilities (Artzi et al., 2008). All current HTML validators expect all the structure and content be present in the HTML source code. However, with Ajax, changes are manifested on the single-page user interface by partially updating the dynamic DOM through JavaScript. Since these validators cannot execute client-side JavaScript, they simply cannot perform any kind of validation.

To prevent faults, we must make sure that the application has a valid DOM on every possible execution path and modification step. We use the DOM tree obtained after each state change while crawling and transform it to the corresponding HTML instance. A W3C HTML validator serves as oracle to determine whether errors or warnings occur. Since most Ajax sites rely on a single-page interface, we use a *diff* algorithm to prevent duplicate occurrences of failures that may be the result of a previous state.

No Error Messages in DOM. Our state should never contain a string pattern that suggests an error message (Benedikt et al., 2002) in the DOM. Error messages that are injected into the DOM as a result of client-side (e.g., 404 Not Found, 400 Bad Request) or server-side errors (e.g., Session Timeout, 500 Internal Server Error, MySQL error) can be detected automatically. The prescribed list of potential fault patterns should be configurable by the tester.

Other Invariants. In line with the above, further generic DOM-invariants can be devised, for example to deal with accessibility, link discoverability, or security constraints on the DOM at any time throughout the crawling process. We omit discussion of these invariants due to space limitations.

6.6.2 State Machine Invariants

Besides constraints on the DOM-tree in individual states, we can identify requirements on the state machine and its transitions.

No Dead Clickables. One common fault in classical web applications is the occurrence of *dead links* which point to a URL that is permanently unavailable. In Ajax, clickables that are supposed to change the state by retrieving data from the server, through JavaScript in the background, can also be broken. Such error messages from the server are mostly swallowed by the Ajax engine, and no sign of a dead link is propagated to the user interface. By listening to the client/server request/response traffic after each event (e.g., through a proxy), dead clickables can be detected.

Consistent Back-Button. A fault that often occurs in Ajax applications is the broken Back-button of the browser. As explained in Section 5.3, a dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function makes the browser completely leave the application's web page. It is possible to programatically register each state change with the browser history and frameworks are appearing which handle this issue. However, when the state space increases, errors can be made and some states may be ignored by the developer to be registered properly. Through crawling, upon each new state, one can compare the expected state in the graph with the state after the execution of the Back-button and find inconsistencies automatically.

6.6.3 Application-specific Invariants

We can define invariants that should always hold and could be checked on the DOM, specific to our Ajax application in development. In our case study, Section 6.9.2, we describe a number of application-specific invariants. Constraints over the DOM-tree can be easily expressed as invariants in Java, for example through an XPath expression. Typically, this can be coded into one or two simple Java methods. The resulting invariants can be used to dynamically search for invariant violations.

6.7 Testing Ajax Paths

While running the crawler to derive the state machine can be considered as a first full test pass, the state machine itself can be further used for testing purposes. For example, it can be used to execute different paths to cover the

```
@Test
public void testcase1() {
browser.goToUrl(url);

/*Element-info: SPAN class=expandable-hitarea */
browser.fireEvent(new Eventable(new Identification(
  "xpath", "//DIV[1]/SPAN[4]"), "onclick"));

Comp.AssertEquals(oracle.getState("S_1").getDom(), browser.getDom());

/*Element-info: DIV class=hitarea id=menuitem2 */
browser.fireEvent(new Eventable(new Identification(
  "xpath", "//SPAN[2]/DIV[2]"), "onmouseover"));

Comp.AssertEquals(oracle.getState("S_3").getDom(), browser.getDom());

/*Element-info: Form, A href=#submit */
handleForm(2473584);

Comp.AssertEquals(oracle.getState("S_4").getDom(), browser.getDom());
}

private void handleForm(long formId) {
 Form form = oracle.getForm(formId);
 if (form != null) {
  FormHandler.fillFormInDom(browser, form);
  browser.fireEvent(form.getSubmit());
 }
}
```

Figure 6.2 A generated JUnit test case.

state machine in different ways. In this section, we explain how to derive a test suite (implemented in JUnit) automatically from the state machine, and how this suite can be used for testing purposes.

To generate the test suite, we use the *K shortest paths* (Yen, 1971) algorithm which is a generalization of the shortest path problem in which several paths in increasing order of length are sought. We collect all sinks in our graph, and compute the shortest path from the index page to each of them. Loops are included once. This way, we can easily achieve all transitions coverage.

Next, we transform each path found into a JUnit test case, as shown in Figure 6.2. Each test case captures the sequence of events from the initial state to the target state. The JUnit test case can fire events, since each edge on the state-flow graph contains information about the event-type and the element the event is fired on to arrive at the target state. We also provide all the information about the clickable element such as tag name and attributes, as code comments in the generated test method. The test class provides API's to access the DOM (`browser.getDom()`) and elements (`browser.getElementBy(how, value)`) of the resulting state after each event, as well as its contents.

If an event is a form submission (annotated on the edge), we generate all the required information for the test case to retrieve the corresponding input values from the database and insert them into the DOM, before triggering the event.

6.7.1 Oracle Comparators

After each event invocation the resulting state in the browser is compared with the expected state in the database which serves as oracle. The comparison can take place at different levels of abstraction ranging from textual (Sprenkle et al., 2007) to schema-based similarity as proposed in Chapter 3.

6.7.2 Test-case Execution

Usually extra coding is necessary for simulating the environment where the tests will be run, which contributes to the high cost of testing (Bertolino, 2007). We provide a framework to run all the generated tests automatically using a real web browser and generate success/failure reports. At the beginning of each test case the embedded browser is initialized with the URL of the Ajax site under test. For each test case, the browser is first put in its initial index state. From there, events are fired on the clickable elements (and forms filled if present). After each event invocation, assertions are checked to see if the expected results are seen on the web application's new UI state.

The generated JUnit test suite can be used in several ways. First, it can be run as is on the current version of the Ajax application, but for instance with a different browser to detect browser incompatibilities. Furthermore, the test suite can be applied to altered versions of the Ajax application to support regression testing: For the unaltered user interface, the test cases should pass, and only for altered user interface code failures might occur (also helping the tester to understand what has truly changed). The typical use of the derived test suite will be to take apart specific generated test cases, and augment them with application-specific assertions. In this way, a small test suite arises capturing specific fault-sensitive click trails.

6.8 Tool Implementation: Atusa

We have implemented our testing approach in an open source tool called Atusa [5] (Automatically Testing UI States of Ajax), available through our website.[6] It is based on the crawling capabilities of Crawljax and provides plugin *hooks* for testing Ajax applications at different levels. Its architecture can be divided into three phases:

preCrawling occurs after the application has fully been loaded into the browser. Examples include authentication plugins to log onto the system and checks on the HTML source code.

[5] Atusa is a Persian name, meaning beautiful body. In Ancient Persia, during the Achaemenid dynasty, Atusa was the daughter of Cyrus the Great, and a half-sister of Cambyses II. -She married Darius the Great and gave birth to Xerxes.

[6] `http://spci.st.ewi.tudelft.nl`

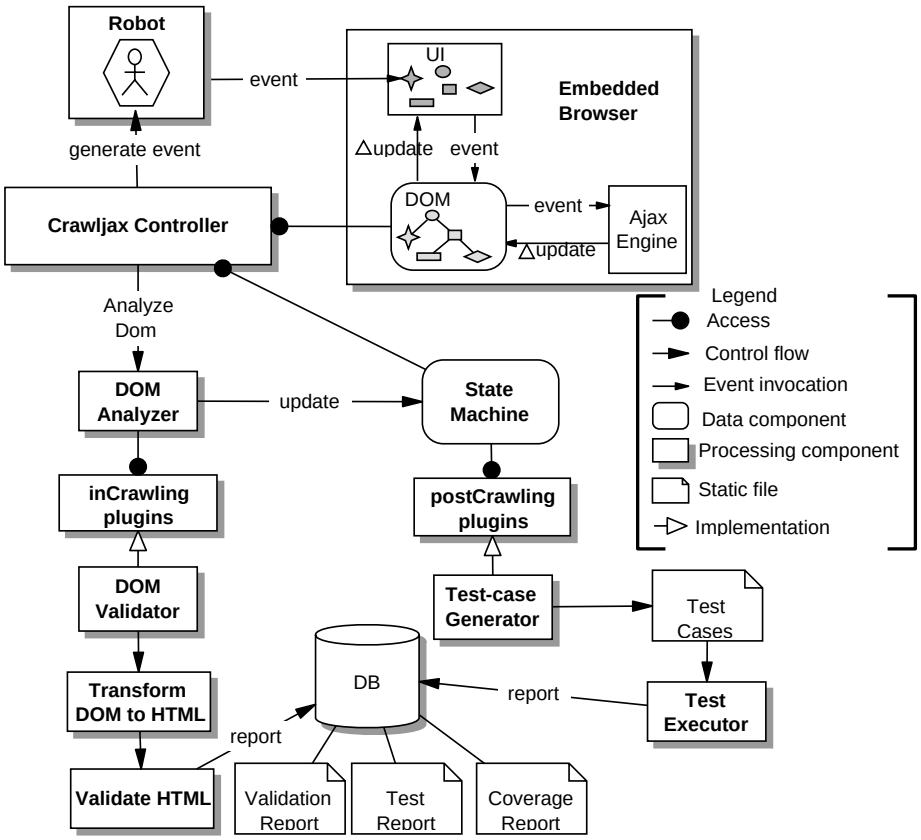

Figure 6.3 Processing view of ATUSA.

inCrawling occurs after each detected state change, different types of invariants can be checked through plugins such as Validated DOM, Consistent Back-button, and No Error Messages in DOM.

postCrawling occurs after the crawling process is done and the state-flow graph is inferred fully. The graph can be used, for instance, in a plugin to generate test cases from.

Algorithms 4 and 5 show the hooks along the crawling process. For each phase, ATUSA provides the tester with specific APIs to implement plugins for validation and fault detection. ATUSA offers generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite from the inferred state-flow graph. Figure 6.3 depicts the processing view of ATUSA, showing only the DOM Validator and Test Case Generator as examples of possible plugin implementations.

ATUSA supports looking for many different types of faults in AJAX-based applications, from errors in the DOM instance, to errors that involve the nav-

Algorithm 4 Pre/postCrawling hooks

```
1: procedure START (url, Set tags)
2: browser ← initEmbeddedBrowser(url)
3: robot ← initRobot()
4: sm ← initStateMachine()
5: preCrawlingPlugins(browser)
6: crawl(null)
7: postCrawlingPlugins(sm)
8: end procedure
9: procedure CRAWL (State ps)
10: cs ← sm.getCurrentState()
11: Δupdate ← diff(ps, cs)
12: analyseForms(Δupdate)
13: Set C ← getCandidateClickables(Δupdate, tags)
14: for c ∈ C do
15:    generateEvent(cs, c)
16: end for
17: end procedure
18: procedure ANALYSEFORMS (State cs)
19: for form ∈ cs.getForms() do
20:    id ← getHashCode(form)
21:    dbForm ← database.getForm(id)
22:    if dbForm == null then
23:       extractInsertForm(form, id)
24:    else
25:       fillFormInDom(browser, dbForm)
26:       generateEvent(cs, dbForm.getSubmit())
27:    end if
28: end for
29: end procedure
```

igational path, e.g., constraints on the length of the deepest paths (Benedikt et al., 2002), or number of clicks to a certain state. Whenever a fault is detected, the error report along the causing execution path is saved in the database so that it can be reproduced later easily.

Implementation. ATUSA is implemented in Java 1.6. The *state-flow graph* is based on the JGrapht library. The implementation details of the crawler can be found in Chapter 5. The plugin architecture is implemented through the Java Plugin Framework (JPF)[7] and we use Hibernate, a persistence and query service, to store the data in the database. Apache Velocity[8] templates assist us in the code generation process of JUnit test cases.

[7] http://jpf.sourceforge.net
[8] http://velocity.apache.org

Algorithm 5 InCrawling hook while deriving AJAX states

```
1: procedure GENERATEEVENT (State cs, Clickable c)
2: robot.fireEvent(c)
3: dom ← browser.getDom()
4: if distance(cs.getDom(), dom) > τ then
5:    xe ← getXpathExpr(c)
6:    ns ← State(dom)
7:    sm.addState(ns)
8:    sm.addEdge(cs, ns, Event(c, xe))
9:    sm.changeState(ns)
10:   inCrawlingPlugins(ns)
11:   crawl(cs)
12:   sm.changeState(cs)
13:   if browser.history.canBack then
14:      browser.history.goBack()
15:   else
16:      browser.reload()
17:      List E ← sm.getPathTo(cs)
18:      for e ∈ E do
19:         robot.fireEvent(e)
20:      end for
21:   end if
22: end if
23: end procedure
```

6.9 Empirical Evaluation

In order to assess the usefulness of our approach in supporting modern web application testing, we have conducted a number of case studies, set up following Yin's guidelines (Yin, 2003).

Goal and Research Questions. Our goal in this experiment is to evaluate the fault revealing capabilities, scalability, required manual effort and level of automation of our approach. Our research questions can be summarized as:

RQ1 What is the fault revealing capability of ATUSA?

RQ2 How well does ATUSA perform? Is it scalable?

RQ3 What is the automation level when using ATUSA and how much manual effort is involved in the testing process?

6.9.1 Study 1: TUDU

Our first experimental subject is the AJAX-based open source TUDU [9] web application for managing personal todo lists, which has also been used by other

[9] http://tudu.sourceforge.net

LOC Server-side		LOC Client-side		DOM string size		Candidate Clickables		Detected Clickables		Detected States		Detected Entry Points		DOM Violations		Back-button		Generated Test Cases		Coverage Server-side		Coverage Client-side		Detected Faults		Manual Effort		Performance	
3k		11k (ext) 580 (int)		24908 (byte)		332		42		34		4 forms 21 inputs		182		false		32		73%		35% (ext) 75% (int)		80%		26.5 (minutes)		5.6 (minutes)	

Table 6.1 TUDU case study.

researchers (Marchetto et al., 2008b). The server-side is based on J2EE and consists of around 12K lines of Java/JSP code, of which around 3K forms the presentation layer we are interested in. The client-side extends on a number of AJAX libraries such as DWR and Scriptaculous,[10] and consists of around 11k LOC of external JAVASCRIPT libraries and 580 internal LOC.

To address RQ3 we report the time spent on parts that required manual work. For RQ1-2, we configured ATUSA through its `properties` file (1 minute), setting the URL of the deployed site, the tag elements that should be included (`A, DIV`) and excluded (`A:title=Log out`) during the crawling process, the depth level (2), the similarity threshold (0.89), and a maximum crawling time of 60 minutes. Since TUDU requires authentication, we wrote (10 minutes) a `preCrawling` plugin to log into the web application automatically.

As shown in Table 6.1, we measure average DOM string size, number of candidate elements analyzed, detected clickables and states, detected data entry points, detected faults, number of generated test cases, and performance measurements, all of which are printed in a log file by ATUSA after each run.

In the initial run, after the login process, ATUSA crawled the TUDU application, finding the doorways to new states and detecting all possible data entry points recursively. We analyzed the data entry points in the database and provided each with custom input values (15 minutes to evaluate the input values and provide useful values). For the second run, we activated (50 seconds) the DOM Validator, Back-Button, Error Detector, and Test Case Generator plugins and started the process. ATUSA started crawling and when forms were encountered, the custom values from the database were automatically inserted into the browser and submitted. Upon each detected state change, the invariants were checked through the plugins and reports were inserted into the database if faults were found. At the end of the crawling process, a test suite was generated from the inferred state-flow graph.

To the best of our knowledge, there are currently no tools that can automatically test AJAX dynamic states. Therefore, it is not possible to form a base-line for comparison using, for instance, external crawlers. To assess the effectiveness of the generated test suite, we measure code coverage on the client as well as the presentation-tier of the server. Although the effectiveness is not directly implied by code coverage, it is an objective and commonly used indicator of the quality of a test suite (Halfond and Orso, 2007). To that end, we instrumented the presentation part of the server code (`tudu-dwr`) with Clover and the client-side JAVASCRIPT libraries with JSCoverage,[11] and deployed the web application. For each test run, we bring the TUDU database to the original state using a SQL script. We run all the test cases against the instrumented application, through ATUSA's embedded browser, and compute the amount of coverage achieved for server- and client-side code. In addition, we manually seeded 10 faults, capable of causing inconsistent states (e.g., DOM malformdness, adding values longer than allowed by the database, adding duplicate

[10] `http://script.aculo.us`
[11] `http://siliconforks.com/jscoverage/`

todo items, removing all items instead of one) and measured the percentage of faults detected. The results are presented in Table 6.1.

Findings. Based on these observations we conclude that: The use of Atusa can help to reveal generic faults, such as DOM violations, automatically; The generated test suite can give us useful code coverage (73% server-side and 75% client-side; Note that only partial parts of the external libraries are actually used by TUDU resulting in a low coverage percentage) and can reveal most DOM-based faults, 8 of the 10 seeded faults were detected, two faults were undetected because during the test execution, they were silently swallowed by the JavaScript engine and did not affect the DOM. It is worth mentioning that increasing the depth level to 3 significantly increased the measured crawling time passed the maximum 60 minutes, but dit not influence the fault detection results. The code coverage, however, improved by approximately 10%; The manual effort involved in setting up Atusa (less than half an hour in this case) is minimal; The performance and scalability of the crawling and testing process is very acceptable (it takes Atusa less than 6 minutes to crawl and test TUDU, analyzing 332 clickables and detecting 34 states).

6.9.2 Study 2: Finding Real-Life Bugs

Our second case study involves the development of an Ajax user interface in a small commercial project. We use this case study to evaluate the manual effort required to use Atusa (RQ3), and to assess the capability of Atusa to find faults that actually occurred during development (RQ1).

Subject System. The case at hand is Coachjezelf (cjz, "Coach Yourself"),[12] a commercial application allowing high school teachers to assess and improve their teaching skills. cjz is currently in use by 5000-6000 Dutch teachers, a number that is growing with approximately 1000 paying users every year.

The relevant part for our case is the interactive table of contents (TOC), which is to be synchronized with an actual content widget. In older versions of cjz this was implemented through a Java applet; in the new version this is to be done through Ajax, in order to eliminate a Java virtual machine dependency.

The two developers working on the case study spent around one week (two person-weeks) building the Ajax solution, including requirements elicitation, design, understanding and evaluating the libraries to be used, manual testing, and acceptance by the customer.

The Ajax-based solution made use of the `jQuery`[13] library, as well as the `treeview`,[14] `history-remote`,[15] and `listen` plugins for `jQuery`. The libraries comprise around 10,000 lines of JavaScript, and the custom code is around 150 lines of JavaScript, as well as some HTML and CSS code.

[12] See `www.coachjezelf.nl` for more information (in Dutch).
[13] `jquery.com`
[14] `http://bassistance.de/jquery-plugins/jquery-plugin-treeview/`
[15] `http://stilbuero.de/jquery/history/`

```
1 //case one: warn about collapsible divs within expandable items
2 String xpathCase1 =  "//LI[contains(@class,'expandable')]/DIV[contains(@class,'collapsable')]";
3
4 //case two: warn about collapsible items within expandable items
5 String xpathCase2 = "//LI[contains(@class,'expandable')]/UL/LI[contains(@class,'collapsable')]";
```

Figure 6.4 Example invariants expressed using XPath in Java.

Case study setup. The developers were asked (1) to try to document their design and technical requirements using invariants, and (2) to write the invariants in Atusa plugins to detect errors made during development. After the delivery of the first release, we evaluated (1) how easy it was to express these invariants in Atusa; and (2) whether the (generic or application-specific) plugins were capable of detecting faults.

Application-Specific Invariants. Two sets of invariants were proposed by the developers. The first essentially documented the (external) `treeview` component, capable of (un)folding tree structures (such as a table of contents).

The `treeview` component operates by setting HTML class attributes (such as `collapsible`, `hit-area`, and `lastExpandable-hitarea`) on nested list structures. The corresponding style sheet takes care of properly displaying the (un)folded (sub)trees, and the JavaScript intercepts clicks and re-arranges the class attributes as needed.

Invariants were devised to document constraints on the class attributes. As an example, the `div`-element immediately below a `li`-element that has the class `expandable` should have class `expandable-hitarea`. Another invariant is that expandable list items (which are hidden) should have their CSS display type set to "none".

The second set of invariants specifically dealt with the code written by the developers themselves. This code took care of synchronizing the interactive display of the table of contents with the actual page shown. Clicking links within the page affects the display of the table of contents, and vice versa.

This resulted in essentially two invariants: one to ensure that within the table of contents at most one path (to the current page) would be open, and the other that at any time the current page as marked in the table of contents would actually be displayed in the content pane.

Expressing such invariants on the DOM-tree was quite easy, requiring a few lines of Java code using XPath. An example is shown in Figure 6.4.

Failures Detected. At the end of the development week, Atusa was used to test the new Ajax interface. For each type of application-specific invariant, an `inCrawling` plugin was added to Atusa. Six types of failures were automatically detected: three through the generic plugins, and three through the application-specific plugins just described. An overview of the type of failures found and the invariant violations that helped to detect them is provided in Table 6.2.

Failure	Cause	Violated Invariant	Invariant type
Images not displayed	Base URL in dynamic load	Dead Clickables	Generic
Broken synchronization in IE	Invalid HTML id	DOM-validator	Generic
Inconsistent history	Issue in `listen` library	Back-Button	Generic
Broken synchronization in IE	Backslash versus slash	Consistent current page	Specific
Corrupted table	Coding error	`treeview` invariants, Consistent current page	Specific
Missing TOC Entries	Incomplete input data	Consistent current page	Specific

Table 6.2 Faults found in CJZ-AJAX.

The application-specific failures were all found through two invariant types: the *Consistent current page*, which expresses that in any state the table and the actual content should be in sync, and the *treeview invariants*. Note that for certain types of faults, for instance the treeview corrupted table, a very specific click trail had to be followed to expose the failure. ATUSA gives no guarantee of covering the complete state of the application, however, since it tries a huge combination of clickables recursively, it was able to detect such faults, which were not seen by developers when the application was tested manually.

Findings. Based on these observations we conclude that: The use of ATUSA can help to reveal bugs that are likely to occur during AJAX development and are difficult to detect manually; Application-specific invariants can help to document and test the essence of an AJAX application, such as the synchronization between two widgets; The manual effort in coding such invariants in Java and using them through plugins in ATUSA is minimal.

6.10 Discussion

6.10.1 Automation Scope

User interface testing is a broad term, dealing with testing how the application and the user interact. This typically is manual in nature, as it includes inspecting the correct display of menus, dialog boxes, and the invocation of the correct functionality when clicking them. The type of user interface testing that we propose does not replace this manual testing, but augments it: Our focus is on finding programming faults, manifested through failures in the DOM tree. As we have seen, the highly dynamic nature and complexity of AJAX make it error-prone, and our approach is capable of finding such faults automatically.

6.10.2 Invariants

Our solution to the oracle problem is to include invariants (as also advocated by, e.g., Meyer (Meyer, 2008)). AJAX applications offer a unique opportunity for specifying invariants, thanks to the central DOM data structure. Thus, we are able to define generic invariants that should hold for all AJAX applications, and we allow the tester to use the DOM to specify dedicated invariants. Furthermore, the state machine derived through crawling can be used to express invariants, such as correct Back-button behavior. Again, this state machine can be accessed by the tester to specify his or her own invariants. These invariants make our approach much more sophisticated than *smoke tests* for user interfaces (as proposed by e.g., Memon (Memon, 2007)) — which we can achieve thanks to the presence of the DOM and state machine data structures. Note that just running CRAWLJAX would correspond to conducting a smoke test: the difficulty with web applications (as opposed to, e.g., Java Swing ap-

plications) is that it is very hard to determine when a failure occurs – which is solved in Atusa through the use of invariants.

6.10.3 Generated versus hand-coded JavaScript

The case studies we conducted involve two different popular JavaScript libraries in combination with hand-written JavaScript code. Alternative frameworks exist, such as Google's Web Toolkit (GWT)[16] in which most of the client-side code is generated. Atusa is entirely independent of the way the Ajax application is written, so it can be applied to such systems as well. This will be particularly relevant for testing the custom JavaScript code that remains to be hand-written, and which can still be tricky and error-prone. Furthermore, Atusa can be used by the developers of such frameworks, to ensure that the generated DOM states are correct.

6.10.4 Manual Effort

The manual steps required to run Atusa consist of configuration, plugin development, and providing custom input values, which for the cases conducted took less than an hour. The hardest part is deciding which application-specific invariants to adopt. This is a step that is directly connected with the *design* of the application itself. Making the structural invariants explicit not only allows for automated testing, it is also a powerful design documentation technique. Admittedly, not all web developers will be able to think in terms of invariants, which might limit the applicability of our approach in practice. Those capable of documenting invariants can take advantage of the framework Atusa provides to actually implement the invariants.

6.10.5 Performance and Scalability

Since the state space of any realistic web application is huge and can cause the well-know *state explosion problem*, we provide the tester with a set of configurable options to constrain the state space such as the maximum search depth level, the similarity threshold, maximum number of states per domain, maximum crawling time, and the option of ignoring external links and links that match some pre-defined set of regular expressions.The main component that can influence the performance and scalability is the crawling part. The performance of Atusa in crawling an Ajax site depends on many factors such as the speed at which the server can handle requests, how fast the client-side JavaScript can update the interface, and the size of the DOM tree. Atusa can scale to sites comprised of thousands of states easily.

6.10.6 Application Size

The two case studies both involve around 10,000 lines of JavaScript library code, and several hundred lines of application code. One might wonder

[16] `http://code.google.com/webtoolkit/`

whether this is too small to be representative. However, our results are based on *dynamic* analysis rather than static code analysis, hence the amount of code is not the determining factor. Instead, the size of the derived state machine is the factor limiting the scalability of our approach, which is only moderately (if at all) related to the size of the JavaScript code.

6.10.7 Threats to Validity

Some of the issues concerning the *external* validity of our empirical evaluation have been covered in the above discussion on scope, generated code, application size, and scalability. Apart from the two case studies described here, we conducted two more (on TaskFreak[17] and the Java PetStore 2.0[18]), which gave comparable results. With respect to *internal* validity, we minimized the chance of Atusa errors by including a rigorous JUnit test suite. Atusa, however, also makes use of many (complex) third party components, and we did encounter several problems in some of them. While these bugs do limit the current applicability of our approach, they do not affect the validity of our results. As far as the choice of faults in the first case study is concerned, we selected them form the TUDU bug tracking system, based on our fault models which we believe are representative of the types of faults that occur during Ajax development. The choice is, therefore, not biased towards the tool but the fault models we have. With respect to *reliability*, our tools and the TUDU case are open source, making the case fully reproducible.

6.10.8 Ajax Testing Strategies

Atusa is a first, but essential step in testing Ajax applications, offering a solution for the reach/trigger/propagate problem. Thanks to the plugin-based architecture of Atusa, it now becomes possible to extend, refine, and evaluate existing software testing strategies (such as evolutionary, state-based, category-partition, and selective regression testing) for the domain of Ajax applications.

6.11 Concluding Remarks

In this chapter, we have proposed a method for testing Ajax applications automatically. Our starting point for supporting Ajax-testing is CrawlJax, a crawler for Ajax applications that we proposed in our earlier work (Mesbah et al., 2008), which can dynamically make a full pass over an Ajax application. Our current work resolves the subsequent problems of extending the crawler with data entry point handling to *reach* faulty Ajax states, *triggering* faults in those states, and *propagating* them so that failure can be determined. To that end, this chapter makes the following contributions:

[17] `http://www.taskfreak.com`
[18] `https://blueprints.dev.java.net/petstore/`

1. A series of fault models that can be automatically checked on any user interface state, capturing different categories of errors that are likely to occur in AJAX applications (e.g., DOM violations, error message occurrences), through (DOM-based) generic and application-specific invariants which server as oracle.
2. An algorithm for deriving a test suite achieving all transitions coverage of the state-flow graph obtained during crawling. The resulting test suite can be refined manually to add test cases for specific paths or states, and can be used to conduct regression testing of AJAX applications.
3. An open source tool called ATUSA implementing the approach, offering generic invariant checking components as well as a plugin-mechanism to add application-specific state validators and test suite generation.
4. An empirical validation, by means of two case studies, of the fault revealing capabilities and the scalability of the approach, as well as the level of automation that can be achieved and manual effort required to use the approach.

Given the growing popularity of AJAX applications, we see many opportunities for using ATUSA in practice. Furthermore, the open source and plugin-based nature of ATUSA makes it a suitable vehicle for other researchers interested in experimenting with other new techniques for testing AJAX applications.

Our future work will include conducting further case studies, as well as the development of ATUSA plugins, capable of spotting security vulnerabilities in AJAX applications.

Chapter 7 Conclusion

With the advent of Ajax technologies, a new paradigm for developing interactive web applications has emerged. This dissertation has focused on better understanding this shifting web paradigm, and the consequences of moving from the classical multi-page model to an Ajax-based single-page style. Specifically to that end, this work has examined this new class of software from three main software engineering perspectives:

Software Architecture to gain an abstract understanding of the key architectural properties of Ajax applications;

Software Reengineering to understand the implications of a migration from classical multi-page web systems to single-page Ajax variants;

Software Analysis and Testing to explore strategies for analyzing and testing this new breed of web application.

7.1 Contributions

The main contributions of the thesis can be summarized as follows:

- A new component- and push-based architectural style, called Spiar, for single-page Ajax web applications. The style results from a study of different major Ajax frameworks, investigating their salient architectural properties, key elements, and constraints on those elements required to achieve the desired properties. We provide a detailed comparison of what the classical web architecture, called Rest, provides, and what the modern Ajax systems require. Our style captures the essence of Ajax frameworks and can be seen as an abstract model of different architectural implementations. As such, Spiar allows to evaluate the tradeoffs between different architectural properties of Ajax systems.
- A process for migrating multi-page web systems to single-page Ajax-based interfaces through reverse and model-driven engineering. The reverse engineering steps of the process have been implemented in a tool called Retjax. Retjax is capable of inferring a navigational model of the web interface by conducting a meta-model clustering technique for web page classification, which we call schema-based clustering. We have shown that schema-based clustering is more accurate and reliable in grouping web pages with structural similarity than approaches which are based on the HTML syntax trees. Additionally, Retjax is capable of producing a list of candidate UI components for migration, through a simplification process of clusters along the navigational paths.

- An automated, distributed software testing infrastructure, for conducting performance analyses of Ajax web applications. The proposed infrastructure is implemented in an open source framework called Chiron. Chiron can simulate thousands of concurrent web users and collect data on network usage, server performance, and data coherence. We have used Chiron to conduct an empirical study for a comparison of the performance tradeoffs of using push- (Comet) and pull-based web data delivery techniques on Ajax applications. The results of our empirical study help engineers to anticipate the effects of key parameters such as pull and push intervals, and the number of web clients on, for instance, data coherence and server performance.

- A crawling method for Ajax, based on dynamic analysis of single-page web interfaces. The method infers a state-flow graph of the navigational paths by running the web application in an embedded browser, detecting and executing clickable elements, and analyzing the user interface state changes. The technique is implemented in an open source tool called Crawljax, which can automatically make a full pass over Ajax web interfaces.

- An automated technique for testing Ajax user interfaces. The technique is based on an extension of the crawling technique to dynamically find the doorways to different states and data entry points on Ajax web interfaces. We propose to use invariants, as oracle, on the DOM tree and the inferred state machine to detect faults. The technique is implemented in an open source tool called Atusa, offering a plugin-mechanism to add invariant checking components and application-specific state validators. Atusa provides a number of generic invariant plugins, such as DOM validation and test suite generation.

7.2 Research Questions Revisited

In the beginning of this thesis, we formulated a set of research questions. We believe that the contributions indicate that we have successfully met the objectives. We will now discuss the results for each chapter individually and with respect to other chapters.

Research Question 1

> *What are the fundamental architectural differences and tradeoffs between designing a classical and an* Ajax*-based web application? Can current architectural styles describe* Ajax*? If not, can we propose an architectural style taylored for* Ajax*?*

In order to answer the first research question, Chapter 2 based its foundation on the software architecture literature, to understand the key properties of a new, complex, and dynamic software family. Software architecture turned

out to be an appropriate framework for the study of web evolution, since it enabled us to describe a family of web systems by abstracting their similarities and differences.

We studied a number of Ajax systems and evaluated different variants of Ajax client/server interactions. It turned out that despite their differences in implementation approaches, a common pattern could be seen in their architecture in terms of how the different components were interacting to increase the level of responsiveness and interactivity in web settings. It became obvious to us that the components and their interactions were far more complex and fine-grained than what the classical web architecture was prescribing.

Chapter 2 determines the focus of this thesis by describing, through an architectural style called Spiar, the target system, namely Ajax-based web applications with a single-page user interface where the state changes are synchronized between the client and the server through a component-based model with push capabilities. This chapter described the main architectural differences with respect to classical web systems, e.g., client/server delta communication, asynchronous interaction, interface delta updates, component-based. These differences in turn serve as the basis for the problem definition, motivation and possible solutions for all subsequent chapters.

Spiar, can be used to describe not only Ajax-based architectures but also any type of Rich Internet Application by replacing some of the architectural elements and fine-tuning the architectural properties. For instance, the DOM-based representational model can be replaced with a Flash-based model. This replacement changes the *standards-based* property to *proprietary*. The delta communication remains intact since in all RIAs, state changes occur partially and incrementally.

Research Question 2

> *Is it possible to support the migration process (Ajaxification) of multi-page web applications to single-page* Ajax *interfaces? Can reverse engineering techniques help in automating this process?*

While Chapter 2 helped us gain a thorough understanding of the target system, in Chapter 3 we proposed a systematic approach to migration of multi-page web applications (source) to Ajax-based single-page interfaces composed of UI components (target).

Chapter 3 focused on reverse engineering a navigational model of the source system, by retrieving all possible pages automatically, clustering similar pages based on a page meta-model (schema-based) notion along the navigational path, simplifying the clusters and the navigational model, and performing a step-wise comparison of the changes when going up the navigational path, to detect candidate UI components.

It turned out in Chapter 3 that this schema-based similarity metric can result in a much higher precision and recall, compared with approaches that are based directly on the HTML code, when detecting clone pages with similar structures. However, to automatically group pages in clusters, Chapter 3

applied the transitive property of clone pairs. For this reason, it is possible that the algorithm finds larger clusters than expected.

Although Chapter 3 proposed a complete migration approach, it specifically focused on the first part, namely reverse engineering an abstract model of the source multi-page interface for the purpose of migration towards a component-based single-page interface.

Automating a migration process is a daunting task, still very ad hoc, and due to the complexity of web systems, full automation is probably not achievable. Nevertheless, reverse engineering techniques, implemented in a tool called RETJAX, proved to be a great vehicle in automating parts of the process of program comprehension and inferring abstract models from the source system.

The applicability of the proposed approach is currently limited to web applications with simple user interfaces. A limitation of the approach is currently the static analysis technique adopted to retrieve client pages from the source system. In fact, RETJAX navigates anchors/links that are found in the HTML code of navigated pages. However, a URL link, could generate different client pages at multiple requests from the server.

The retrieval phase of RETJAX is, to certain extent, similar to what CRAWLJAX does, i.e., inferring an abstract model of the web application. The edit distance method used is also the same in both approaches. The difference lies in the fact that RETJAX conducts a static analysis of the HTML code to detect hyperlinks, while CRAWLJAX performs a dynamic analysis of all possible clickable elements to find state changes. It is thus possible to augment RETJAX with the capabilities of CRAWLJAX to retrieve a more complete navigational model from the target system.

The second part of the migration process, namely single-page model definition and target UI model transformation, is missing in this thesis. We have, however, initiated research (Gharavi et al., 2008) to adopt a model-driven engineering approach to AJAX, which could be integrated in the migration process.

RETJAX is geared towards classic multi-page web applications and although the intention is to use the resulting candidate components for migration towards AJAX, the tool itself is not AJAX specific. In fact, RETJAX can be used as a clustering tool on any standards-based web application. The schema-based clustering approach has various applications in cases where web page comparison on an structural level plays an important role.

Research Question 3

> *What are the challenges for analyzing and testing* AJAX *applications in an automatic approach?*

AJAX applications have a number of characteristics that make them challenging for automatic analysis and testing. First of all, like classical web systems, AJAX applications have a distributed client/server nature, and testing distributed systems is known to be very demanding. AJAX settings do, however, pose a greater challenge than conventional web settings, since the client

plays a more prominent role in the client/server distributed spectrum, as more logic is being ported from the server-side to the client-side.

In addition, unlike traditional object-oriented systems, the heterogeneous nature of web applications increases the difficulty of useful analysis based on inspections of the source code. Static analysis techniques, which are commonly used with success on object-oriented applications, have serious limitations in analyzing modern Ajax applications where understanding the run-time behavior is crucial. As we have seen in Chapter 5, even simply finding and following links is not possible any longer on Ajax interfaces by static analysis of the source code. Using dynamic analysis to gather data from a running web program seems then the right direction to take, which in turn has its own difficulties and limitations, such as incompleteness and scalability (Cornelissen et al., 2009).

Research Question 3.1

> *What are the tradeoffs of applying pull- and push-based data delivery techniques on the web? Can we set up an automated distributed test environment to obtain empirical data for comparison?*

Conducting an experiment to compare the actual tradeoffs of pull and push based web data delivery techniques turned out to be a difficult task, as explained in Chapter 4. Our first attempt in conducting the experiment consisted of a great deal of manual work, from starting the servers, the client simulation processes, to gathering and saving the run-time information, parsing the data, and generating graphs. The difficulty in coordination of the different modules in such a distributed environment encouraged us to opt for an automated approach, implemented in Chiron. Besides decreasing the manual effort, such an automated testing environment greatly increases the accuracy of the experiment, since similar test runs can be sequentially performed with exactly the same parameters.

The proposed experiments are very complex, since the execution environment contains various distributed interacting concurrent software processes and distributed hardware devices. For these reasons, the execution of our experiments is subject to validation threats, as discussed thoroughly in Chapter 4.

Although the experiment in Chapter 4 has been conducted on one sample application, and the Comet push servers are still experimental, the results show a promising trend in web data delivery techniques which clearly out perform pull-based approaches in terms of data coherence and network performance. Currently, scalability of the server, when a high number of clients has to be served, is the main concern when push is used.

Chiron is not bound to any specific web technology and almost every aspect of the tool is configurable, and since it has been made open source, similar experiments by others can be carried out to dynamically analyze the runtime performance properties of web systems.

Research Question 3.2

Can Ajax*-based web applications be crawled automatically?*

Simply, statically analyzing and retrieving anchors/links in the HTML source code, as conducted by classical web crawlers, is not sufficient any longer. The run-time manipulation of the DOM object within the browser, the fact that resources are not bound to a specific URL, and the extensive use of JavaScript for binding events to any type of HTML elements (`DIV`, `SPAN`), all contribute to the complexity of crawling Ajax-based web interfaces.

Despite all these challenges, we were successful in proposing a method for automatically detecting clickables, and navigating through dynamic doorways of Ajax applications, as explained in Chapter 5. In fact, to the best of our knowledge, Crawljax is the first Ajax crawler available, capable of making a full pass of a running Ajax application.

The method proposed is based on dynamic analysis of the web interface, as seen from the web user's perspective. We have opted for using a real browser (embedded in our crawling tool), which we believe is the best way to analyze highly dynamic web applications, since the browser forms the natural engine for all the required technologies, e.g., DOM, JavaScript. The fundamental elements in our approach are access to the run-time DOM object and the ability to generate events on the browser interface, to systematically run and analyze the interface changes and reverse engineer a state-flow graph of the application's navigational structure.

While Retjax, as presented in Chapter 3, reverse engineers a multi-page web application to a single-page one, Crawljax does the opposite, i.e., it extracts an abstract model (multi-page) of the different states from a single-page web application.

We strongly believe that such a crawler has many applications, when it comes to analyzing and testing, not only Ajax-based, but also any standards-based web application. Even non standard-based interfaces (RIAs) could reuse a similar approach for crawling if access to run-time interface elements can be provided.

Research Question 3.3

Can Ajax*-based web user interfaces be tested automatically?*

In Chapter 6, we proposed to dynamically change the state and analyze the changes through Crawljax and find faults automatically by defining generic and application-specific invariants that should hold in the entire state space. The crawler was first extended to detect and provide interface mechanisms for handling forms, which are the data entry points in web applications. Through these data entry points, various data inputs can be inserted on the interface and sent to the server systematically.

One of the main challenges in automated testing is the process of assessing the correctness of test output. In Chapter 6, we have opted for using invariants on DOM states as well as the inferred state machine to specify the expected

correct behavior. Violations in the invariants can be seen as candidate faults. The invariants can be generic, such as DOM validation and correctness, or application-specific, constraining DOM elements' properties, relations, and occurrences.

The proposed technique in Chapter 6 provides evidence that automatic testing of dynamic interfaces is possible through invariants. Invariants not only help during the test process to automate the oracle, but at the same time they can form an appropriate vehicle for documenting the system behavior, especially in web-based applications where documenting the dynamic behavior is far from trivial.

As a comparison, CHIRON, presented in Chapter 4, analyzes the behavior of the web application by simulating a high number of clients and examining system properties such as server performance, network usage, data coherence, etc. On the other hand, ATUSA focuses on testing the run-time behavior of one single web application, by simulating user events on the client interface and analyzing the response.

ATUSA does not replace manual user interface testing, but augments it. One limitation of our approach is the state space explosion, which is an inherent problem in all state-based testing approaches and our tool is not an exception, although we do provide a number of mechanisms to constrain the state space, as discussed in Chapter 6.

7.3 Evaluation

In this section we discus our research challenges and the threats to validity of our results.

The work described in this thesis started under the Single Page Computer Interaction (SPCI) project, a project initiated by:

- Backbase, a computer software company specialized in creating AJAX-based web applications, founded in 2003 in Amsterdam, see 2.2.3 for more details about their AJAX framework.
- CWI, the Dutch National Research Institute for Mathematics and Computer Science, and
- the Delft University of Technology.

In fact, the term AJAX was not even coined yet when the project's proposal was written. The goal of the project was to better understand the implications of adopting modern standards-based single-page web applications.

One challenge we faced at the start of the project was that the literature on modern web applications in general and AJAX in particular, appeared to be a scarcely populated area. Many traditional software and web engineering techniques appeared to be unfit for AJAX and therefore had to be adapted to the new requirements.

Empirical evaluation (Wohlin et al., 2000) is an effective way to asses the validity of any new concept or method. As far as the proposed Spiar architectural style (Chapter 2) is concerned, empirical evaluation is very challenging, if not impossible. Therefore, we have taken an analytical approach in evaluating Spiar by discussing the rationale behind the style and its scope regarding different web architectures. We have conducted a controlled experiment (Wohlin et al., 2000) in Chapter 4 and have empirically evaluated Retjax (Chapter 3), Crawljax (Chapter 5), and Atusa (Chapter 6) by conducting case studies (Yin, 2003; Kitchenham et al., 1995) and reporting the results. One challenge in conducting the case studies was the lack of existing comparable methods and tools to form a comparison baseline (Kitchenham et al., 2002) for the results.

Another issue we faced revolved around finding industrial real-world cases for the evaluation part, which can be seen as an external threat to validity. Due to the relatively new nature of the technology, we were not able to find as many industrial Ajax-based applications to conduct experiments on, as we had wished for. Therefore, most of our case study subjects are open source applications. Although the size of the chosen case subjects is relatively small, they are representative of the type of web applications our methods are focused on, i.e., standards-based single-page web applications. With respect to reliability and replication, the open source nature of both the case subjects and the tools makes the experiment and case studies fully reproducible.

As far as the internal validity is concerned, we minimized the chance of development errors by including rigorous unit testing for all the implemented tools. We have used various external libraries and third party components, and while we did encounter problems in some of them, they do not affect the validity of our results. Sections 4.7 and 6.10 provide a more detailed discussion of the threats to validity.

Our emphasis in this thesis has been on ideal ultimate standards-based 'single-page' web interfaces. As we have seen in this work, single-page web interfaces are great if the surrounding web technologies, such as crawlers and browsers, also support them. We believe that for the time being, a *hybrid* approach which unifies the benefits of both dynamic fine-grained single-page and static large-grain multi-page approaches, is the best option for developing web applications without having to face the negative side effects of the two ends, i.e., issues with amnesic pages without any history, linking, bookmarking, indexing by general search engines on the one hand, and low latency, responsiveness and interactivity on the other hand.

It is worth mentioning that for closed (authenticated) web applications, single-page interfaces are ideal since many of the issues mentioned above, such as discoverability by search engines, do not play a role.

Our Spiar architectural style is deduced from single-page Ajax applications. Whether such a style can support hybrid architectures needs further exploration. Our crawling and testing methods should have no difficulties in dealing with such hybrid settings, although here again, more experiments need to be carried out.

A great deal of our work has focused on Ajax from a client's perspective, since we believe the client-side is where the main changes have taken place. Further work is needed in exploring the implications on the server-side by inspecting the server-side code. In Chapter 3, we opted for analyzing the generated web pages to build a navigational model. For a migration process, it is also valuable to analyze the server side code (e.g., JSP, PHP) to infer an augmented abstract model of the application.

Our analysis of the client-side behavior has been founded mainly on dynamic analysis. Statically analyzing the client-side JavaScript and server-side code for program comprehension, analysis, and testing could also result in interesting results. Each approach considers only a subset of possible executions in its own way, and hence, static and dynamic analysis techniques could be combined (Ernst, 2003) to augment one another.

7.4 Future Work and Recommendations

Ajax development field is young, dynamic, and changing rapidly. Certainly, the work presented in this dissertation needs to be incrementally enriched and revised, taking into account experiences, results, and innovations as they emerge from the industry as well as the research community. Conducting case studies in larger web projects, in particular projects of industrial nature in the industry-as-laboratory style (Potts, 1993), forms an important part of the future work. Improving the quality and performance of the proposed methods and tools is a natural extension of the work presented in this research.

An interesting direction to investigate comprises abstracting from the implementation details through a Model-driven Engineering (Schmidt, 2006) approach, in which the Ajax application (i.e., the UI components and their topology) is defined in a modeling language and the corresponding web code is automatically generated. This approach enables us to define the application once and generate the same application to different frameworks.

The user interface components and the event-based interaction between them form the founding elements in Ajax-based web systems, whereas in classic web applications the notions of web pages and hypertext links are central. Therefore, modeling Ajax applications requires a different approach than what the current web modeling methods (Ceri et al., 2000; Conallen, 2003; Koch and Kraus, 2002) provide.

We have started (Gharavi et al., 2008) adopting a model-driven approach for Ajax web application development. How an Ajax web application can be modeled best, while having the ultimate goal of code generation from the models in mind, is an interesting direction for future work. In particular, future work encompasses modeling event-based interactions between user interface components in an abstract way in single-page web applications.

Our crawler currently compares the DOM tree after each event generation to find out whether a state change has occurred. One possible enhancement of CrawlJax in future work is adopting an embedded browser with support

for DOM *mutation events* (W3C, a). The mutation event module was introduced in DOM Level 2 and is designed to allow notification of document structural changes, including attribute and text modifications. This way, by subscribing to the mutation events, the crawler is automatically informed of DOM changes and thus redundant DOM comparisons can be avoided, which in turn increases the crawling performance. Another issue is coping with DOM changes that are not directly caused by an event fired by the crawler, such as pushed content from the server.

The dynamic user interface components and the huge combinations of click trails in changing the state, makes modern web interfaces very challenging to analyze and test. Atusa, currently detects and executes clickable elements in a top-down, depth-first manner. This implies that first, the inferred state machine is one possible instance of the state space, and second, the order of the events could have an influence on the faults executed and detected. How these parameters influence the effectiveness of the testing approach, and whether click trails that mimic a web user's actions would help the tool in finding more relevant faults, are all questions that form possible extensions of the work presented in this thesis. In addition, further work is needed to explore possibilities of implementing useful and robust oracle comparators and element identification mechanisms in the generated test cases (from the state machine by Atusa) for conducting regression testing.

Testing modern web applications for security vulnerabilities is far from trivial. Currently, we are exploring (Bezemer et al., 2009) ways Atusa can be used to spot security violations in single-page web applications comprised of various web widgets created by different developers. Generally, each web widget should operate in its own environment. As any program code, widgets can be used for malicious purposes. Security becomes an important aspect when third-parties are allowed to build and include new widgets in public catalogs. Example scenarios include when a malicious widget changes the content of another widget to trick the user into releasing sensitive information, or even worse, listens to the account details a user enters in another widget (e.g., PayPal or Email widgets) and sends the data to a malicious site.

Automatically detecting security vulnerabilities such as Cross-Site Scripting (XSS) (Wassermann and Su, 2008) and taking preventive measures by, for instance, instrumenting the JavaScript code (Yu et al., 2007), in Ajax applications are other interesting research areas that require more attention.

7.5 Concluding Remarks

The work presented in this dissertation aims at advancing the state-of-the-art in comprehending, analyzing, and testing standards-based single-page web applications, by means of a new architectural style, a significant set of techniques and tools, and case study reports. These contributions are aimed at helping software and web engineers better comprehend and deal with the complexity of highly dynamic and interactive web systems.

A Single-page AJAX Example Application

Appendix A

This appendix contains a very simple example to present the underlying concepts behind AJAX, which are described in Chapters 1 and 2.

A.1 The HTML Single-page

Figure A.1 shows the main (and only) HTML page of the example site. This is the page that is retrieved by the browser on the initial request to the server. The page contains a number of elements. The HEADER includes links to a JAVASCRIPT file and a style sheet for presentation. The BODY consists of a heading (*H3*) and a container (DIV). As can be seen, the DIV container encompasses some plain text as well as a clickable element (A).

```
1  <html>
2  <head>
3   <script type="text/javascript" src="ajax.js"></script>
4   <link rel="stylesheet" href="style.css" type="text/css">
5   <title>Single-page Ajax Site</title>
6  </head>
7  <body>
8   <span>
9   <h3>Welcome to the Ajax demo site!</h3>
10  <div id="container">
11   This is where the remote content is injected.
12   <a href="#" onclick="getData('server.php', 'container');">Click</a>
13  </div>
14  </span>
15 </body>
16 </html>
```

Figure A.1 The Index HTML page.

A.2 The Run-time DOM

After the HTML content is received from the server, the browser parses the page and creates a Document Object Model (DOM) instance, which can be seen as a tree representing the run-time elements and their properties, as shown in Figure A.2. This DOM-tree is then used to build the browser user interface. Figure A.2 also shows the rendered page after the initial load by the browser.

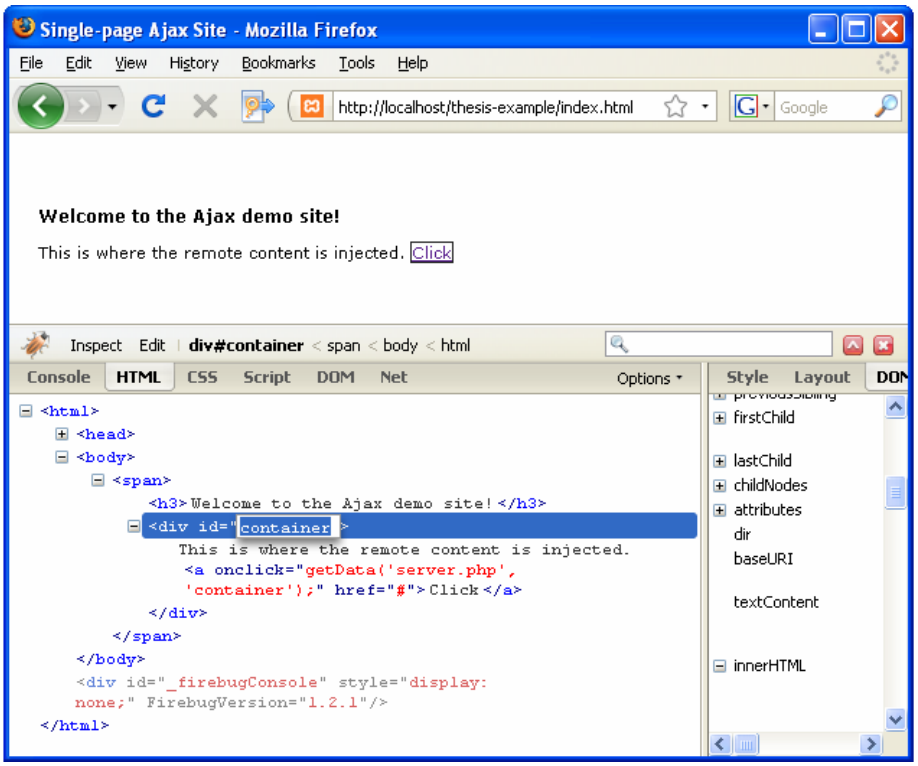

Figure A.2 The run-time Document Object Model after the page is fully loaded into the browser (bottom) and the rendered user interface (top).

A.3 Requesting Data

What we would like to achieve is to get web content, not as whole pages but as fragments from the server, and update the page dynamically without a page refresh. To that end, we use JavaScript to access the XMLHttpRequest object and the run-time DOM-tree to achieve our goal.

Figure A.3 presents the JavaScript code that is included in the HTML page (see line 3 in Figure A.1). Our JavaScript code begins by instantiating a correct XMLHttpRequest object. Note that different browsers (e.g., IE, Firefox) have different implementations for this object. The `loadFragment` function shows how the XMLHttpRequest object can be used to transfer data between the browser/server. The method takes a number of parameters:

method can have a value of GET or POST. A variety of other HTTP methods (W3C, b) are also possible.

url may be either a relative or an absolute URL which points to a resource.

```
if(navigator.appName == "Microsoft Internet Explorer") {
  var xmlHttpRequestObject = new ActiveXObject("Microsoft.XMLHTTP");
} else {
  var xmlHttpRequestObject = new XMLHttpRequest();
}

var COMPLETED = 4;
var SUCCESS = 200;

function loadFragment(method, url, params, target_element_id, callbackFunction) {
   xmlHttpRequestObject.open(method, url, true);
   xmlHttpRequestObject.onreadystatechange = function() {
     if (xmlHttpRequestObject.readyState == COMPLETED
         && xmlHttpRequestObject.status == SUCCESS) {
      callbackFunction(target_element_id, xmlHttpRequestObject.responseText);
     }
   }

   if (method == "POST") {
     xmlHttpRequestObject.setRequestHeader("Content-type",
      "application/x-www-form-urlencoded");
   } else {
     xmlHttpRequestObject.setRequestHeader("Content-type",
      "text/plain");
   }

   xmlHttpRequestObject.send(params);
}

function insertIntoDOMElement(target_element_id, data) {
  var element = document.getElementById(target_element_id);
  element.innerHTML = data;
}

function getData(url, target_element_id) {
 loadFragment('GET', url, null, target_element_id, insertIntoDOMElement);
}

function submitData(url, target_element_id) {
 var params = "val1=" + document.getElementById('val1').value +
                       "&val2=" + document.getElementById('val2').value +
                       "&calc=" + document.getElementById('calc').value;

 loadFragment('POST', url, params, target_element_id, insertIntoDOMElement);
}
```

Figure A.3 The JAVASCRIPT code. XMLHttpRequest is used to asynchronously transfer data between the web server and the browser.

params a string representing the request content. Used with the POST method to send form parameters to the server.

target_element_id the ID of the DOM element where the response content should be injected into.

callbackFunction handles the response content.

Using these parameters, the `loadFragment` function sends a request to the server and calls the callback function when a valid response is obtained from the server.

The `getData` function is a wrapper function which calls the `loadFragment` function with the correct parameters.

The `getData` function is attached to the `onclick` attribute of the clickable element, as can be seen in line 12 of Figure A.1. This function is called with two parameters, namely `server.php` as the URL, and `container` as the target element ID. This gives us the desired behavior that when the element is clicked by the user, the `getData` method is called, which in turn calls the `loadFragment` function to get data from the `server.php` URL on the server.

A.4 The Server-side Code

Figure A.4 show the contents of our `server.php` which simply returns the code for a HTML form.

```
1  <?php
2  echo ("<form>Calculation: <br/>
3  <input type='text' id='val1'> (value1) <br/>
4  <input type='text' id='calc'> (method: add, sub, mul, div) <br/>
5  <input type='text' id='val2'> (value2) = <br/>
6  <div id='result'></div>
7  <input type='button' value='Calculate'
8   onclick='submitData(\"calculate.php\", \"result\")'>
9  </form>");
10 ?>
```

Figure A.4 server.php source code.

A.5 DOM Injection

Once the response of the server is successfully arrived at the browser, the callback function is called. The `insertIntoDOMElement` function takes the ID of the target element and the response data, as parameters. It uses the ID to find a reference to the corresponding run-time element in the DOM instance (line 28 in Figure A.3). Once a reference has been found, the response data is injected into the inner part of that element. As soon as the content is injected

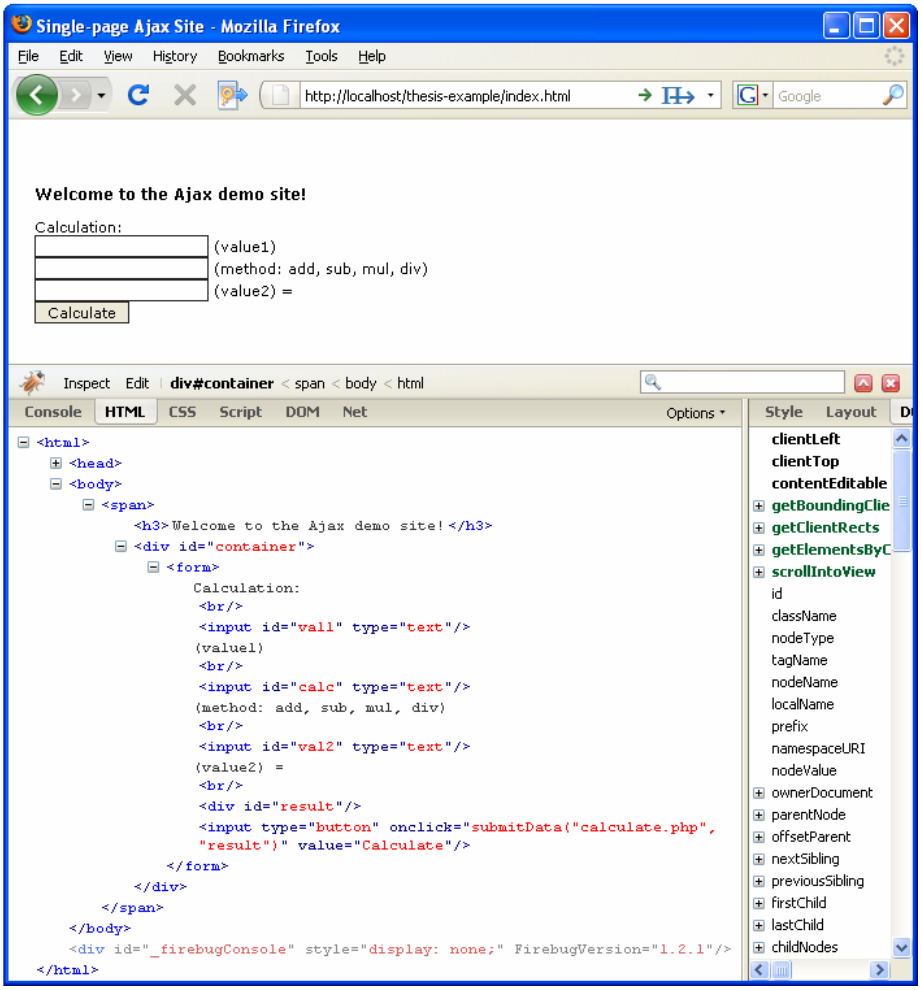

Figure A.5 The run-time DOM instance after the new content is retrieved and injected into the DOM-tree (bottom) and the updated user interface (top).

into the target DOM element (in this case the DIV element with ID *content*), the browser updates the user interface.

This way a complete client/server round-trip can be made and the user interface of the browser can be updated in the background, without having to refresh the page. Figure A.5 depicts the updated DOM and browser user interface.

A.6 Submitting Data

The new state contains a form which we use as a simple calculator. The user fills in the first and second digits along with one of the four supported operations for addition (add), subtraction (sub), multiplication (mul), and division (div). Upon clicking on the button labeled *Calculate*, an event is fired which calls the `submitData` JavaScript function (see lines 36–41 in Figure A.3), with `calculate.php` as URL and `result` as the target element ID. This function simply retrieves the input values filled in by the user on the form and calls the `loadFragment` function with relevant parameter, i.e., 'POST' as *method*, and the input values as *params*.

Figure A.6 shows the server-side PHP code that is responsible for the calculation. Based on the value of the three request parameters (`val1`, `val2`, `calc`), a response in returned to the browser.

```
1  <?php
2  function calculate($val1, $val2, $calc) {
3   if(is_numeric($val1) && is_numeric($val2) && $calc != null)
4   {
5     switch($calc) {
6               case "add" : $result= $val1 + $val2; break;
7               case "sub" : $result= $val1 - $val2; break;
8               case "mul" : $result= $val1 * $val2; break;
9               case "div" : $result= $val1 / $val2; break;
10    }
11    return "Result: <b>$result</b>";
12  }
13  else{
14    return "<span class='warn'>Invalid input, please try again!</span>";
15  }
16 }
17
18 $num1 = trim($_REQUEST['val1']);
19 $num2 = trim($_REQUEST['val2']);
20
21 echo calculate($num1, $num2, $_REQUEST['calc']);
22 ?>
```

Figure A.6 Server-side PHP calculator.

The response is then injected into the `DIV` element with ID *result*. Figure A.7 shows the request/response, focusing on the response returned by the server as a delta fragment. It also shows the updated browser user interface, in which the result of the calculation can be seen. Note that all the user interface updates have taken place on a single page. This is evident by the fact that the address-bar of the browser is still pointing to the same index HTML page.

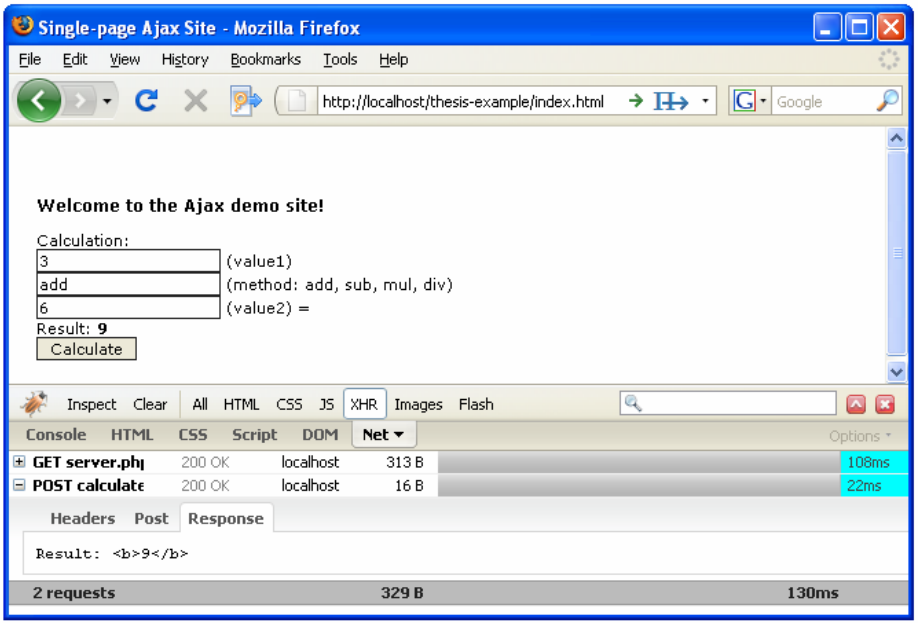

Figure A.7 The request/response traffic (bottom) and the updated user interface (top).

Samenvatting*

Analyse en Testing van Ajax-gebaseerde Single-page Web-applicaties
Ali Mesbah

Inleiding en Probleemdefinitie

Desktop-applicaties worden meer en meer vervangen door web-applicaties. Dit brengt grote voordelen met zich mee: geen gedoe met installatie, toegankelijkheid vanaf elke computer die een internetconnectie heeft, en altijd de beschikking over de nieuwste versie van de applicatie.

Tot voor kort was de browser niet meer dan een programma om een serie hypertekstpaginas, opgemaakt in HTML, te bekijken. Dit klassieke multi-pagemodel heeft echter zijn langste tijd gehad: veel aandacht gaat tegenwoordig uit naar een nieuwe type web-applicaties, genaamd Ajax (Garrett, 2005). Het is een antwoord op de beperkte interactiviteit van de bestaande toestandsloze webinteracties. In deze nieuwe benadering is het interactiemodel gebaseerd op één enkele pagina, samengesteld uit diverse gebruikersinterfacecomponenten, met als doel web-applicaties veel interactiever te maken. In plaats van de hele pagina te verversen vinden de wijzigingen plaats op componentniveau. Tegenwoordig bevatten web-applicaties allerlei rijke gebruikersinterface-widgets, zoals sliders, tabs, in- en uitklappende fragmenten, en zooming, welke geheel in de browser draaien. De techniek die dit alles mogelijk maakt is bekend onder de naam Ajax: een afkorting voor Asynchronous JavaScript and XML.

JavaScript is de geïnterpreteerde scripting-taal die door alle browsers gesproken wordt: hiermee kunnen de fraaie gebruikersinterface-widgets worden gemaakt, en acties aan bepaalde gebruikersinterface-events (zoals specifieke clicks) worden gehangen. Het asynchrone deel van Ajax zorgt ervoor dat de web-applicatie niet meer hoeft te wachten op resultaten die het van bv. een webserver wil halen: de applicatie, en dus de gebruikersinteractie, kan gewoon doorgaan terwijl op de achtergrond de browser informatie van de webserver betrekt. Mede hierdoor krijgt de web-applicatie het responsieve karakter dat we van desktopapplicaties gewend zijn. De rol van XML in Ajax, tot slot, zit hem in de communicatie tussen de webbrowser en server. Bijzonder aan Ajax is dat deze communicatie gebruikmaakt van "deltas", waarmee alleen wijzigingen in de toestand van de gebruikersinterface worden doorgegeven. Zo wordt voorkomen dat de volledige toestand van de pagina steeds opnieuw van de server gehaald moet worden. In plaats daarvan wordt deze toestand bijgehouden binnen de browser, middels het Document Object Model (DOM).

*Deze samenvatting is gedeeltelijk verschenen in twee Nederlandse publicaties: (Mesbah and van Deursen, 2006) en (van Deursen and Mesbah, 2008).

Echter, de introductie van deze technologie verloopt niet probleemloos. Voor de software-ontwikkelaar die overweegt Ajax toe te passen blijft namelijk nog een reeks vragen open. Heeft het gebruik van Ajax invloed op de architectuur? Hoe kunnen traditionele web-applicaties worden gemigreerd naar moderne single-page Ajax varianten? Hoe staat het met de betrouwbaarheid en veiligheid wanneer veel JavaScript code wordt geladen? Hoe bereikbaar zijn de dynamische toestanden voor zoekmachines? Hoe zit het met de performance, aan de kant van zowel de server als de browser? Is al dat geprogrammeer in JavaScript nog wel te doorgronden voor de gemiddelde software-ontwikkelaar? En: als de code niet gemakkelijk kan worden doorgrond, hoe kan dan aannemelijk worden gemaakt dat de applicatie doet wat zij moet doen?

Het vinden van een antwoord op dergelijke vragen is van groot belang voor de praktijk, en is een uitdaging die dit proefschrift aangaat.

Resultaten

De belangrijkste resultaten van dit proefschrift kunnen als volgt worden samengevat.

Architectuur. Er zijn talloze software-raamwerken en -bibliotheken geschreven die gebaseerd zijn op Ajax, en er komen nog dagelijks nieuwe bij. Om de essentie van Ajax te begrijpen, hebben we getracht een wat abstracter perspectief op Ajax te ontwikkelen. In dit kader hebben we diverse Ajax-raamwerken bestudeerd om hun architectuurkenmerken in kaart te brengen, zoals voorgesteld in hoofdstuk 2. Onze analyse van de bestaande literatuur over software-architectuur en -raamwerken heeft geleid tot onze nieuwe architectuurstijl genaamd Spiar. Deze stijl combineert een serie gewenste eigenschappen (t.w. interactiviteit en ontwikkelgemak) met vereisten waaraan een Ajax-architectuur moet voldoen (zoals componentgebaseerde gebruikersinterfaces en deltacommunicatie tussen client en server) om deze eigenschappen te realiseren. We hebben aan de hand van onze Spiar-architectuurstijl laten zien hoe concepten vanuit de architectuurwereld, zoals architectuureigenschappen, -restricties, en -elementen, ons kunnen steunen om een complexe en dynamische technologie als Ajax te begrijpen. Ons werk is gestoeld op de grondslagen van de klassieke webstijl (Rest) en biedt een analyse van deze stijl met betrekking tot het bouwen van moderne web-applicaties. Spiar beschrijft de essentie van Ajax en software engineering-principes die door ontwikkelaars gebruikt kunnen worden tijdens het bouwen en analyseren van Ajax-applicaties.

Migratie. De kernvraag die in hoofdstuk 3 wordt behandeld is hoe we automatisch bestaande traditionele multi-page web-applicaties kunnen migreren naar single-page Ajax. Hiertoe hebben we onderzocht hoe single-page gebruikersinterfacecomponent-kandidaten gedetecteerd kunnen worden in traditionele web-applicaties. Het begrijpen van het navigatiemodel van het te migreren systeem is cruciaal in een migratieproces. Ons onderzoek naar dit probleem richt zich erop om met behulp van reverse engineering-algoritmen

een dergelijk model te creëren, door alle mogelijke pagina's automatisch te benaderen, soortgelijke pagina's te clusteren met een meta-model (schemagebaseerde) vergelijkingsmethode, de clusters van pagina's te vereenvoudigen, en de clusters op het navigatiepad te vergelijken om mogelijke componentkandidaten te detecteren. Deze techniek is geïmplementeerd in het gereedschap "RETJAX".

Performance. Normaal gesproken vragen webclients zelf actief aan de server of er nieuwe data beschikbaar is (via de zogeheten "pull" techniek). Met een nieuwe technologie voor AJAX-applicaties, COMET, krijgt de server de mogelijkheid om zelf nieuwe data te pompen naar de webclient ("push" techniek). Om de voor- en nadelen van elke aanpak te bestuderen hebben we in hoofdstuk 4 aan geautomatiseerde en gedistribueerde testing infrastructuur "CHIRON" gewerkt en de broncode openbaar gemaakt. CHIRON kan duizenden webgebruikers tegelijkertijd simuleren en informatie verzamelen over netwerkverkeer, serverprestaties, en data-coherentie. De resultaten van ons empirisch onderzoek laten zien wat de effecten van de kernparameters kunnen zijn, zoals push en pull intervallen, en het aantal webgebruikers, bv. op datacoherentie en serverprestaties.

Bereikbaarheid. Het gebruik van AJAX kan problemen veroorzaken t.a.v. de bereikbaarheid door zoekmachines, zoals die van Google. Dergelijke zoekmachines zijn gebaseerd op een robot die automatisch alle pagina's van een web-applicatie afloopt. Vanuit het oogpunt van schaalbaarheid en veiligheid voeren ze daarbij momenteel geen client-side code uit. Het is echter juist dit soort code waar AJAX-applicaties van afhankelijk zijn, en een aanzienlijk deel van de in een website aanwezige informatie is onbereikbaar indien JAVASCRIPT niet wordt uitgevoerd. Met andere woorden, op dit moment bevinden AJAX-applicaties zich in het zogenaamde "hidden web", d.i. het gedeelte van het wereldwijde web dat niet door zoekmachines bereikt wordt. Traditionele crawling-technieken die gebaseerd zijn op statische analyse om automatisch links in HTML-broncode te detecteren en de betreffende webpagina's van de server te halen, ondersteunen niet het dynamische aspect van AJAX. Ons onderzoek naar dit probleem richt zich erop om methoden te vinden waarmee de zeer dynamische toestanden van AJAX-applicaties automatisch genavigeerd kunnen worden. Hiertoe hebben we een algoritme ontwikkeld en deze geïmplementeerd in het openbaar beschikbare gereedschap "CRAWLJAX". Zoals beschreven in hoofdstuk 5 is CRAWLJAX in staat automatisch clickables te vinden en door te klikken, en intussen een model van gebruikersinterfacetoestanden en de transities daartussen af te leiden uit een AJAX-applicatie. Voor zover wij weten is CRAWLJAX de eerste beschikbare AJAX crawler.

Betrouwbaarheid. Met AJAX worden de grenzen opgezocht van wat mogelijk is met HTML, JAVASCRIPT, en HTTP. Bovendien werkt AJAX met asynchrone communicatie, en wordt aan de server- en client-kant een toestand synchroon gehouden door delta's heen en weer te sturen. De resulterende programmatuur kan behoorlijk complex zijn: het programmeren van AJAX is bijzon-

der foutgevoelig. Geautomatiseerd testen van een AJAX-applicatie kan helpen sommige van de typische AJAX-fouten te vinden. Net als bij het onderzoek naar bereikbaarheid, speelt hier het automatisch crawlen (doorklikken) van een AJAX-applicatie een sleutelrol. De uitdaging daarbij bestaat niet alleen uit het invullen van invoervelden, maar ook uit het bepalen wanneer de geteste applicatie zich incorrect gedraagt (het "oracle-probleem"). De oplossing die in dit proefschrift in hoofdstuk 6 is onderzocht, spitst zich toe op de mate waarin (generieke en applicatiespecifieke) structurele invarianten op de DOM-boom hier een rol in kunnen spelen. Bovendien hebben we een algoritme ontwikkeld waarmee uit het gedistilleerde model tijdens het crawlen automatisch test cases gegenereerd kunnen worden. Onze testing techniek is geïmplementeerd in het open-source gereedschap "ATUSA". ATUSA biedt de testers een serie test-plugins, en een infrastructuur waarin ontwikkelaars zelf met gemak specifieke plugins kunnen bouwen en toevoegen.

Conclusie

Met dit proefschrift hebben we beoogd de stand der techniek te versterken op het gebied van het begrijpen, analyseren, en testen van standaard-gebaseerde single-page web-applicaties. Dit heeft geleid tot een nieuwe architectuurstijl, een uitgebreide verzameling van technieken, bijbehorende software-gereedschappen, en rapporten van uitgevoerde empirische evaluaties. Deze contributies zijn bedoeld om software engineers beter te laten omgaan met de complexiteit en het zeer dynamische karakter van interactieve websystemen.

Bibliography

Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M., and Shuster, J. E. (1999). UIML: An appliance-independent XML user interface language. In *WWW '99: 8th International Conference on World Wide Web*, pages 1695–1708. (Cited on page 59.)

Acharya, S., Franklin, M., and Zdonik, S. (1997). Balancing push and pull for data broadcast. In *SIGMOD '97: ACM SIGMOD International Conference on Management of Data*, pages 183–194. ACM Press. (Cited on page 103.)

Alager, S. and Venkatsean, S. (1993). Hierarchy in testing distributed programs. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG'93)*, pages 101–116. Springer-Verlag. (Cited on pages 17, 86, and 87.)

Allaire, J. (2002). Macromedia Flash MX-A next-generation rich client. Macromedia white paper. `http://www.adobe.com/devnet/flash/whitepapers/richclient.pdf`. (Cited on page 5.)

Ammar, M., Almeroth, K., Clark, R., and Fei, Z. (1998). Multicast delivery of web pages or how to make web servers pushy. Proceedings of the Workshop on Internet Server Performance. (Cited on page 103.)

Andrews, A., Offutt, J., and Alexander, R. (July 2005). Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345. (Cited on pages 16, 18, 126, and 131.)

Antoniol, G., Di Penta, M., and Zazzara, M. (2004). Understanding web applications through dynamic analysis. In *IWPC '04: 12th IEEE International Workshop on Program Comprehension*, page 120. IEEE Computer Society. (Cited on page 72.)

Arnold, R. S. (1993). *Software Reengineering*. IEEE Computer Society Press. (Cited on page 16.)

Artzi, S., Kieżun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. D. (2008). Finding bugs in dynamic web applications. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA'08)*, pages 261–272. ACM. (Cited on pages 131, 133, and 136.)

Asleson, R. and Schutta, N. T. (2005). *Foundations of Ajax*. Apress. (Cited on page 51.)

Atterer, R. and Schmidt, A. (2005). Adding usability to web engineering models and tools. In *Proceedings of the 5th International Conferencee on Web Engineering (ICWE'05)*, pages 36–41. Springer. (Cited on page 108.)

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33. (Cited on pages 34 and 44.)

Backbase (2005). Designing rich internet applications for search engine accessibility. backbase.com Whitepaper. (Cited on page 125.)

Barbosa, L. and Freire, J. (2007). An adaptive crawler for locating hidden-web entry points. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 441–450. ACM Press. (Cited on pages 17, 124, and 127.)

Barone, P., Bonizzoni, P., Vedova, G. D., and Mauri, G. (2001). An approximation algorithm for the shortest common supersequence problem: an experimental analysis. In *SAC '01: ACM symposium on Applied computing*, pages 56–60. ACM Press. (Cited on page 63.)

Barrett, D. J., Clarke, L. A., Tarr, P. L., and Wise, A. E. (1996). A framework for event-based software integration. *ACM Trans. Softw. Eng. Methodol.*, 5(4):378–421. (Cited on page 45.)

Bass, L., Clements, P., and Kazman, R. (2003). *Software architecture in practice, 2nd ed.* Addison-Wesley. (Cited on pages 30 and 33.)

Beizer, B. (1990). *Software Testing Techniques (2nd ed.)*. Van Nostrand Reinhold Co. (Cited on page 16.)

Benedikt, M., Freire, J., , and Godefroid, P. (2002). VeriWeb: Automatically testing dynamic web sites. In *Proc. 11th Int. Conf. on World Wide Web (WWW'02)*. (Cited on pages 130, 135, 136, and 141.)

Berners-Lee, T. (1996). WWW: Past, present, and future. *IEEE Computer*, 29(10):69–77. (Cited on page 1.)

Berners-Lee, T., Masinter, L., and McCahill, M. (1994). *RFC 1738: Uniform Resource Locators (URL)*. W3C. (Cited on pages 2 and 31.)

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *ICSE Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE Computer Society. (Cited on pages 16 and 139.)

Bezemer, C.-P., Mesbah, A., and van Deursen, A. (2009). Automated security testing of web widget interactions. Technical Report TUD-SERG-2009-011, Delft University of Technology. (Cited on page 162.)

Bhide, M., Deolasee, P., Katkar, A., Panchbudhe, A., Ramamritham, K., and Shenoy, P. (2002). Adaptive push-pull: Disseminating dynamic web data. *IEEE Trans. Comput.*, 51(6):652–668. (Cited on pages 35, 48, 79, and 103.)

Binder, R. V. (1999). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley. (Cited on pages 133 and 136.)

Bouras, C. and Konidaris, A. (2005). Estimating and eliminating redundant data transfers over the Web: a fragment based approach: Research articles. *Int. J. Commun. Syst.*, 18(2):119–142. (Cited on pages 43, 48, and 52.)

Bozdag, E. (2007). Integration of HTTP push with a JSF Ajax framework. Master's thesis, Delft University of Technology. (Cited on pages 28 and 41.)

Bozdag, E., Mesbah, A., and van Deursen, A. (2007). A comparison of push and pull techniques for Ajax. In *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE'07)*, pages 15–22. IEEE Computer Society. (Cited on pages 21, 35, 76, and 101.)

Bozdag, E., Mesbah, A., and van Deursen, A. (2009). Performance testing of data delivery techniques for Ajax applications. *Journal of Web Engineering*, 0(0). To appear. (Cited on pages 21, 48, 75, and 129.)

Briand, L. C., Morasca, S., and Basili, V. R. (2002). An operational process for goal-driven definition of measures. *IEEE Trans. Softw. Eng.*, 28(12):1106–1125. (Cited on page 83.)

Brodie, D., Gupta, A., and Shi, W. (2005). Accelerating dynamic web content delivery using keyword-based fragment detection. *J. Web Eng.*, 4(1):079–099. (Cited on page 52.)

Brodie, M. L. and Stonebraker, M. (1995). *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann Publishers Inc. (Cited on page 15.)

Campbell, D. and Stanley, J. (1963). *Experimental and Quasi-Experimental Designs for Research*. Rand-McNally. (Cited on page 101.)

Carzaniga, A., Picco, G. P., and Vigna, G. (1997). Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 22–32. ACM Press. (Cited on pages 31 and 47.)

Ceri, S., Fraternali, P., and Bongio, A. (2000). Web modeling language (WebML): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157. (Cited on page 161.)

Challenger, J., Dantzig, P., Iyengar, A., and Witting, K. (2005). A Fragment-based approach for efficiently creating dynamic Web content. *ACM Trans. Inter. Tech.*, 5(2):359–389. (Cited on page 52.)

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504. ACM Press. (Cited on page 115.)

Chen, J., Hierons, R. M., and Ural, H. (2006). Overcoming observability problems in distributed test architectures. *Inf. Process. Lett.*, 98(5):177–182. (Cited on pages 17, 86, and 87.)

Chikofsky, E. J. and Cross II, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17. (Cited on page 16.)

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., and Little, R. (2002). *Documenting Software Architectures: Views and Beyond*. Pearson Education. (Cited on pages 30 and 39.)

Coda, F., Ghezzi, C., Vigna, G., and Garzotto, F. (1998). Towards a software engineering approach to web site development. In *Proceedings of the 9th international Workshop on Software specification and design (IWSSD'98)*, page 8. IEEE Computer Society. (Cited on page 13.)

Conallen, J. (2003). *Building Web Applications with UML (2nd Edition)*. Addison-Wesley. (Cited on pages 68, 71, and 161.)

Cordy, J. R., Dean, T. R., and Synytskyy, N. (2004). Practical language-independent detection of near-miss clones. In *CASCON '04: Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–12. IBM Press. (Cited on page 72.)

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering (TSE)*. To appear. (Cited on page 157.)

Crane, D., Pascarello, E., and James, D. (2005). *Ajax in Action*. Manning Publications Co. (Cited on pages 24 and 51.)

Dasgupta, A., Ghosh, A., Kumar, R., Olston, C., Pandey, S., and Tomkins, A. (2007). The discoverability of the web. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 421–430. ACM Press. (Cited on page 127.)

de Carvalho, A. F. and Silva, F. S. (2004). Smartcrawl: a new strategy for the exploration of the hidden web. In *WIDM '04: Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 9–15. ACM Press. (Cited on pages 17, 124, and 127.)

De Lucia, A., Francese, R., Scanniello, G., and Tortora, G. (2004a). Reengineering web applications based on cloned pattern analysis. In *IWPC '04: 12th IEEE International Workshop on Program Comprehension*, page 132. IEEE Computer Society. (Cited on pages 61 and 72.)

De Lucia, A., Francese, R., Scanniello, G., and Tortora, G. (2005). Understanding cloned patterns in web applications. In *IWPC '05: 13th International Workshop on Program Comprehension*, pages 333–336. IEEE Computer Society. (Cited on pages 62 and 71.)

De Lucia, A., Scanniello, G., and Tortora, G. (2004b). Identifying clones in dynamic web sites using similarity thresholds. In *International Conference on Enterprise Information Systems*, pages 391–396. (Cited on pages 68 and 71.)

Deshpande, Y. and Hansen, S. (2001). Web engineering: Creating a discipline among disciplines. *IEEE MultiMedia*, 8(2):82–87. (Cited on page 13.)

van Deursen, A., Klint, P., and Verhoef, C. (1999). Research issues in software renovation. In *Fundamental Approaches to Software Engineering (FASE '99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag. (Cited on page 15.)

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36. (Cited on page 117.)

van Deursen, A. and Mesbah, A. (2008). Ajax probleemloos? *Automatisering Gids*, 41(50). (Cited on page 171.)

Di Lucca, G., Fasolino, A., and Faralli, F. (2002a). Testing web applications. In *ICSM '02: International Conference on Software Maintenance*, pages 310–319. IEEE Computer Society. (Cited on page 18.)

Di Lucca, G. A., Di Penta, M., and Fasolino, A. R. (2002b). An approach to identify duplicated web pages. In *COMPSAC '02: 26th International Computer Software and Applications Conference*, pages 481–486. IEEE Computer Society. (Cited on page 61.)

Di Lucca, G. A. and Fasolino, A. R. (2006). Testing web-based applications: The state of the art and future trends. *Inf. Softw. Technol.*, 48(12):1172–1186. (Cited on page 16.)

Di Lucca, G. A., Fasolino, A. R., Pace, F., Tramontana, P., and De Carlini, U. (2002c). Comprehending web applications by a clustering based approach. In *IWPC '02: 10th International Workshop on Program Comprehension*, page 261. IEEE Computer Society. (Cited on page 71.)

Di Lucca, G. A., Fasolino, A. R., Pace, F., Tramontana, P., and de Carlini, U. (2002d). WARE: A tool for the reverse engineering of web applications. In *CSMR '02: 6th European Conference on Software Maintenance and Reengineering*, pages 241–250. IEEE Computer Society. (Cited on page 71.)

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271. (Cited on page 116.)

Direct Web Remoting (2007). Reverse Ajax documentation. `http://getahead.org/dwr/reverse-ajax`. (Cited on pages 24 and 82.)

Draheim, D., Lutteroth, C., and Weber, G. (2005). A source code independent reverse engineering tool for dynamic web sites. In *CSMR '05: 9th European Conference on Software Maintenance and Reengineering*, pages 168–177. IEEE Computer Society. (Cited on pages 60 and 71.)

Elbaum, S., Karre, S., and Rothermel, G. (2003). Improving web application testing with user session data. In *Proc. 25th Int Conf. on Software Engineering (ICSE'03)*, pages 49–59. IEEE Computer Society. (Cited on pages 18 and 131.)

Emmerich, W., Ellmer, E., and Fieglein, H. (2001). TIGRA an architectural style for enterprise application integration. In *ICSE '01: 23rd International Conference on Software Engineering*, pages 567–576. IEEE Computer Society. (Cited on page 52.)

Erenkrantz, J. R., Gorlick, M., Suryanarayana, G., and Taylor, R. N. (2007). From representations to computations: the evolution of web architectures. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE'07)*, pages 255–264. ACM. (Cited on page 53.)

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *Proceedings of the Workshop on Dynamic Analysis (WODA'03)*, pages 24–27. (Cited on page 161.)

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131. (Cited on pages 31, 46, 81, and 104.)

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, UC, Irvine, Information and Computer Science. (Cited on pages 4, 15, 25, 30, 31, 34, 35, 39, and 52.)

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. (Cited on page 2.)

Fielding, R. and Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Trans. Inter. Tech. (TOIT)*, 2(2):115–150. (Cited on pages 25, 31, 45, 52, 75, 76, and 109.)

Florins, M. and Vanderdonckt, J. (2004). Graceful degradation of user interfaces as a design method for multiplatform systems. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 140–147. ACM Press. (Cited on page 125.)

Folmer, E. (2005). *Software Architecture analysis of Usability*. PhD thesis, Univ. of Groningen, Mathematics and Computer Science. (Cited on page 33.)

Franklin, M. and Zdonik, S. (1998). data in your face: push technology in perspective. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 516–519. ACM Press. (Cited on pages 52 and 103.)

Fuggetta, A., Picco, G. P., and Vigna, G. (1998). Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361. (Cited on pages 31 and 35.)

Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., and Shim, K. (2000). XTRACT: a system for extracting document type descriptors from XML documents. In *SIGMOD '00: ACM SIGMOD international conference on Management of data*, pages 165–176. ACM Press. (Cited on page 65.)

Garrett, J. (February 2005). Ajax: A new approach to web applications. Adaptive path. `http://www.adaptivepath.com/publications/essays/archives/000385.php`. (Cited on pages 6, 7, 8, 23, 24, 55, 75, 107, 129, and 171.)

Gharavi, V., Mesbah, A., and van Deursen, A. (2008). Modelling and generating Ajax applications: A model-driven approach. In *Proceedings of the 7th ICWE International Workshop on Web-Oriented Software Technologies (IWWOST'08)*, pages 38–43. (Cited on pages 19, 21, 60, 156, and 161.)

Ginige, A. and Murugesan, S. (2001). Web engineering: An introduction. *IEEE MultiMedia*, 8(1):14–18. (Cited on page 13.)

Halfond, W. and Orso, A. (2007). Improving test case generation for web applications using automated interface discovery. In *Proceedings of the ESEC/FSE conference*, pages 145–154. ACM. (Cited on pages 131 and 144.)

Hassan, A. E. and Holt, R. C. (2002). Architecture recovery of web applications. In *ICSE '02: 24th International Conference on Software Engineering*, pages 349–359. ACM Press. (Cited on page 71.)

Hauswirth, M. and Jazayeri, M. (1999). A component and communication model for push systems. In *7th European Software Engineering Conference (ESEC/FSE-7)*, pages 20–38. Springer-Verlag. (Cited on pages 31, 45, 52, 79, and 103.)

Huang, Y.-W., Tsai, C.-H., Lin, T.-P., Huang, S.-K., Lee, D. T., and Kuo, S.-Y. (2005). A testing framework for web application security assessment. *J. of Computer Networks*, 48(5):739–761. (Cited on pages 130, 131, and 135.)

Jazayeri, M. (2005). Species evolve, individuals age. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 3–12. IEEE Computer Society. (Cited on page 13.)

Jetty (2006). Jetty webserver documentation - continuations. Mortbay Consulting, `http://docs.codehaus.org/display/JETTY/Continuations`. (Cited on pages 48, 90, and 104.)

Juvva, K. and Rajkumar, R. (1999). A real-time push-pull communications model for distributed real-time and multimedia systems. Technical Report CMU-CS-99-107, School of Computer Science, Carnegie Mellon University. (Cited on page 103.)

Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N. (2006). Secubat: a web vulnerability scanner. In *Proc. 15th int. conf. on World Wide Web (WWW'06)*, pages 247–256. ACM. (Cited on page 131.)

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J. (1998). The architecture tradeoff analysis method. In *4th IEEE International Conference on Engineering of Complex Computer Systems*, pages 68–78. IEEE Computer Society. (Cited on page 31.)

Khare, R. (2005). Beyond Ajax: Accelerating web applications with Real-Time event notification. `http://www.knownow.com/products/docs/whitepapers/KN-Beyond-AJAX.pdf`. (Cited on pages 24, 52, and 103.)

Khare, R., Rifkin, A., Sitaker, K., and Sittler, B. (2002). mod_pubsub: an open-source event router for Apache. (Cited on page 52.)

Khare, R. and Taylor, R. N. (2004). Extending the Representational State Transfer (REST) architectural style for decentralized systems. In *26th International Conference on Software Engineering (ICSE'04)*, pages 428–437. IEEE Computer Society. (Cited on pages 31, 32, 45, 52, 53, 78, and 103.)

Kitchenham, B., Pickard, L., and Pfleeger, S. L. (1995). Case studies for method and tool evaluation. *IEEE Softw.*, 12(4):52–62. (Cited on pages 18 and 160.)

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., Emam, K. E., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734. (Cited on page 160.)

Koch, N. and Kraus, A. (2002). The expressive power of UML-based web engineering. In *IWWOST '02: 2nd International Workshop on Web-oriented Software Technology*, pages 105–119. CYTED. (Cited on page 161.)

Koch, P.-P. (March 2005). Ajax, promise or hype? `http://www.quirksmode.org/blog/archives/2005/03/ajax_promise_or.html`. (Cited on page 9.)

Koschke, R. and Eisenbarth, T. (2000). A framework for experimental evaluation of clustering techniques. In *IWPC 'oo: 8th International Workshop on Program Comprehension*, page 201. IEEE Computer Society. (Cited on page 69.)

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Program*, 1(3):26–49. (Cited on page 31.)

Lage, J. P., da Silva, A. S., Golgher, P. B., and Laender, A. H. F. (2004). Automatic generation of agents for collecting hidden web pages for data extraction. *Data Knowl. Eng.*, 49(2):177–196. (Cited on pages 17, 124, and 127.)

Lanubile, F. and Mallardo, T. (2003). Finding function clones in web applications. In *CSMR '03: 7th European Conference on Software Maintenance and Reengineering*, page 379. IEEE Computer Society. (Cited on page 72.)

Lehman, M. M. and Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc. (Cited on page 13.)

Levenshtein, V. L. (1996). Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10:707–710. (Cited on pages 62 and 115.)

Madhavan, J., Ko, D., Kot, L., Ganapathy, V., Rasmussen, A., and Halevy, A. (2008). Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252. (Cited on page 17.)

Marchetto, A., Ricca, F., and Tonella, P. (2008a). A case study-based comparison of web testing techniques applied to ajax web applications. *Int. Journal on Software Tools for Technology Transfer*, 10(6):477–492. (Cited on page 18.)

Marchetto, A., Tonella, P., and Ricca, F. (2008b). State-based testing of Ajax web applications. In *Proc. 1st IEEE Int. Conference on Sw. Testing Verification and Validation (ICST'08)*, pages 121–130. IEEE Computer Society. (Cited on pages 108, 129, 132, and 144.)

Martin-Flatin, J.-P. (1999). Push vs. pull in web-based network management. `http://arxiv.org/pdf/cs/9811027`. (Cited on page 104.)

Memon, A. (2007). An event-flow model of GUI-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157. (Cited on pages 131, 132, and 148.)

Memon, A., Banerjee, I., and Nagarajan, A. (2003). GUI ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering (WCRE'03)*, pages 260–269. IEEE Computer Society. (Cited on pages 72 and 127.)

Memon, A., Soffa, M. L., and Pollack, M. E. (2001). Coverage criteria for GUI testing. In *Proceedings ESEC/FSE'01*, pages 256–267. ACM Press. (Cited on pages 111 and 127.)

Merrill, C. L. (2006). Using Ajax to improve the bandwidth performance of web applications. `http://www.webperformanceinc.com/library/reports/AjaxBandwidth/`. (Cited on page 10.)

Mesbah, A. (2007). Ajaxifying classic web applications. In *Proceedings of the 29th International Conference on Software Engineering, Doctoral Symposium (ICSE'07)*, pages 81–82. IEEE Computer Society. (Cited on page 21.)

Mesbah, A., Bozdag, E., and van Deursen, A. (2008). Crawling Ajax by inferring user interface state changes. In *Proc. 8th Int. Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society. (Cited on pages 21, 107, 129, and 150.)

Mesbah, A. and van Deursen, A. (2005). Crosscutting concerns in J2EE applications. In *Proceedings of the 7th International Symposium on Web Site Evolution (WSE'05)*, pages 14–21. IEEE Computer Society. (Cited on page 21.)

Mesbah, A. and van Deursen, A. (2007a). An architectural style for Ajax. In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 44–53. IEEE Computer Society. (Cited on page 21.)

Mesbah, A. and van Deursen, A. (2007b). Migrating multi-page web applications to single-page Ajax interfaces. In *Proc. 11th Eur. Conf. on Sw. Maintenance and Reengineering (CSMR'07)*, pages 181–190. IEEE Computer Society. (Cited on pages 21, 24, 55, and 115.)

Mesbah, A. and van Deursen, A. (2006). De architectuur van Ajax ontrafeld. *Informatie*, 12(10):50–56. (Cited on page 171.)

Mesbah, A. and van Deursen, A. (2008a). A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software*, 81(12):2194–2209. (Cited on pages 21, 23, 129, and 131.)

Mesbah, A. and van Deursen, A. (2008b). Exposing the hidden-web induced by Ajax. Technical Report TUD-SERG-2008-001, Delft University of Technology. (Cited on pages 20 and 126.)

Mesbah, A. and van Deursen, A. (2009). Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers*, page 11 pp. IEEE Computer Society. (Cited on pages 21 and 129.)

Meyer, B. (2008). Seven principles of software testing. *IEEE Computer*, 41(8):99–101. (Cited on page 148.)

Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B. (1997). Potential benefits of delta encoding and data compression for HTTP. In *ACM SIGCOMM Conf. on Applications, technologies, architectures, and protocols for computer communication*, pages 181–194. ACM. (Cited on pages 43 and 52.)

Monroe, R. T. and Garlan, D. (1996). Style-based reuse for software architectures. In *ICSR '96: 4th International Conference on Software Reuse*, pages 84–93. IEEE Computer Society. (Cited on page 30.)

Morell, L. (1988). Theoretical insights into fault-based testing. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 45–62. (Cited on page 129.)

Murugesan, S., Deshpande, Y., Hansen, S., and Ginige, A. (2001). Web engineering: a new discipline for development of web-based systems. *Web Engineering*, 2016(0):3–13. (Cited on page 13.)

Naaman, M., Garcia-Molina, H., and Paepcke, A. (2004). Evaluation of ESI and class-based delta encoding. In *8th International Workshop Web content caching and distribution*, pages 323–343. Kluwer Academic Publishers. (Cited on page 43.)

Netscape (1995). An exploration of dynamic documents. `http://hoolan.net/spec/WWW/pushpull/`. (Cited on pages 11, 79, and 104.)

Newman, W. M. and Sproull, R. F. (1979). *Principles of Interactive Computer Graphics*. McGraw-Hill. 2nd Edition. (Cited on pages 46 and 52.)

Ntoulas, A., Zerfos, P., and Cho, J. (2005). Downloading textual hidden web content through keyword queries. In *JCDL '05: Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, pages 100–109. ACM Press. (Cited on pages 17, 124, and 127.)

Offutt, J. (2002). Quality attributes of web software applications. *IEEE Softw.*, 19(2):25–32. (Cited on pages 33 and 35.)

O'Reilly, T. (2005). What is Web 2.0: Design patterns and business models for the next generation of software. Oreillynet. `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`. (Cited on page 5.)

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering (ICSE'94)*, pages 279–287. IEEE Computer Society Press. (Cited on page 13.)

Parsons, D. (2007). Evolving architectural patterns for web applications. In *Proceedings of the 11th Pacific Asia Conference on Information Systems (PACIS)*, pages 120–126. (Cited on page 52.)

Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52. (Cited on pages 15, 30, 35, and 52.)

Potts, C. (1993). Software-engineering research revisited. *IEEE Softw.*, 10(5):19–28. (Cited on page 161.)

Puerta, A. and Eisenstein, J. (2002). XIML: a common representation for interaction data. In *IUI '02: 7th international conference on Intelligent user interfaces*, pages 214–215. ACM Press. (Cited on page 59.)

Raghavan, S. and Garcia-Molina, H. (2001). Crawling the hidden web. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 129–138. Morgan Kaufmann Publishers Inc. (Cited on pages 17, 124, and 127.)

Rajapakse, D. C. and Jarzabek, S. (2005). An investigation of cloning in web applications. In *ICWE '05: 5th International Conference on Web Engineering*, pages 252 – 262. Springer. (Cited on page 72.)

Ricca, F. and Tonella, P. (2001). Analysis and testing of web applications. In *ICSE'01: 23rd Int. Conf. on Sw. Eng.*, pages 25–34. IEEE Computer Society. (Cited on pages 18, 60, 71, and 131.)

Ricca, F. and Tonella, P. (2003). Using clustering to support the migration from static to dynamic web pages. In *IWPC '03: 11th IEEE International Workshop on Program Comprehension*, page 207. IEEE Computer Society. (Cited on pages 61 and 71.)

Richardson, D. and Thompson, M. (1988). The RELAY model of error detection and its application. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 223–230. (Cited on page 129.)

Rosenblum, D. S. and Wolf, A. L. (1997). A design framework for internet-scale event observation and notification. In *ESEC/FSE '97: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 344–360. Springer-Verlag New York, Inc. (Cited on page 31.)

Russell, A. (2006). Comet: Low latency data for the browser. `http://alex.dojotoolkit.org/?p=545`. (Cited on pages 11, 12, 24, and 79.)

Russell, A., Wilkins, G., and Davis, D. (2007). Bayeux - a JSON protocol for publish/subscribe event delivery protocol, 0.1draft3. `http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html`. (Cited on page 81.)

Schiemann, D. (2007). The forever-frame technique. `http://cometdaily.com/2007/11/05/the-forever-frame-technique`. (Cited on page 81.)

Schmidt, D. C. (2006). Model-driven engineering. *Computer*, 39(2):25–31. (Cited on pages 19 and 161.)

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. (Cited on page 30.)

Sinha, A. (1992). Client-server computing. *Communications of the ACM*, 35(7):77–98. (Cited on pages 31 and 52.)

Smullen III, C. W. and Smullen, S. A. (March 2008). An experimental study of ajax application performance. *Journal of Software*, 3(3):30–37. (Cited on page 10.)

Sommerville, I. (2007). *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 8 edition. (Cited on page 16.)

Sousa, J. P. and Garlan, D. (2002). Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3: IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43. Kluwer, B.V. (Cited on page 52.)

Sprenkle, S., Gibson, E., Sampath, S., and Pollock, L. (2005). Automated replay and failure detection for web applications. In *ASE'05: Proc. 20th IEEE/ACM Int. Conf. on Automated Sw. Eng.*, pages 253–262. ACM. (Cited on page 131.)

Sprenkle, S., Pollock, L., Esquivel, H., Hazelwood, B., and Ecott, S. (2007). Automated oracle comparators for testing web applications. In *Proc. 18th IEEE Int. Symp. on Sw. Reliability (ISSRE'07)*, pages 117–126. IEEE Computer Society. (Cited on pages 133 and 139.)

Srinivasan, R., Liang, C., and Ramamritham, K. (1998). Maintaining temporal coherency of virtual data warehouses. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'98)*, page 60. IEEE Computer Society. (Cited on page 79.)

Stepien, B., Peyton, L., and Xiong, P. (2008). Framework testing of web applications using TTCN-3. *Int. Journal on Software Tools for Technology Transfer*, 10(4):371–381. (Cited on page 130.)

Stroulia, E., El-Ramly, M., Iglinski, P., and Sorenson, P. (2003). User interface reverse engineering in support of interface migration to the web. *Automated Software Eng.*, 10(3):271–301. (Cited on pages 16 and 72.)

Suryanarayana, G., Erenkrantz, J. R., Hendrickson, S. A., and Taylor, R. N. (2004). PACE: An architectural style for trust management in decentralized applications. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 221. IEEE Computer Society. (Cited on page 52.)

Taylor, R. N., Medvidovic, N., Anderson, K. M., E. J. Whitehead, J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. (1996). A component- and message-based architectural style for GUI software. *IEEE Trans. Softw. Eng.*, 22(6):390–406. (Cited on pages 31 and 52.)

T.D.Cook and D.T.Campbell (1979). *QuasiExperimentation Design and Analysis Issues for Field Settings*. Houghton Mifflin Company. (Cited on page 101.)

Teo, H.-H., Oh, L.-B., Liu, C., and Wei, K.-K. (2003). An empirical study of the effects of interactivity on web user attitude. *Int. J. Hum.-Comput. Stud.*, 58(3):281–305. (Cited on page 33.)

Tonella, P. and Ricca, F. (2004). Statistical testing of web applications. *J. Softw. Maint. Evol.*, 16(1-2):103–127. (Cited on pages 60 and 71.)

Trecordi, V. and Verticale, G. (2000). An architecture for effective push/pull web surfing. In *2000 IEEE International Conference on Communications*, volume 2, pages 1159–1163. (Cited on page 103.)

Tzerpos, V. and Holt, R. C. (1999). MoJo: A distance metric for software clusterings. In *WCRE '99: 6th Working Conference on Reverse Engineering*, pages 187–193. IEEE Computer Society. (Cited on page 69.)

Umar, A. (1997). *Object-oriented client/server Internet environments*. Prentice Hall Press. (Cited on page 31.)

Valmari, A. (1998). The state explosion problem. In *LNCS: Lectures on Petri Nets I, Basic Models, Advances in Petri Nets*, pages 429–528. Springer-Verlag. (Cited on page 124.)

Vanderdonckt, J., Bouillon, L., and Souchon, N. (2001). Flexible reverse engineering of web pages with VAQUISTA. In *WCRE '01: 8th Working Conference on Reverse Engineering*, page 241. IEEE Computer Society. (Cited on page 72.)

W3C. Document Object Model (DOM). `http://www.w3.org/DOM/`. (Cited on pages 4, 6, and 162.)

W3C. XMLHttpRequest. W3C Working Draft `http://www.w3.org/TR/XMLHttpRequest/`. (Cited on pages 10 and 164.)

W3C (1995). Common gateway interface (cgi)/1.1 specification. `http://www.w3.org/CGI/`. (Cited on page 3.)

W3C (1999). Chunked transfer coding. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1`. (Cited on page 81.)

W3C (Mar. 21 2004). URIs, Addressability, and the use of HTTP GET and POST. W3C Tag Finding. (Cited on page 49.)

W3C Technical Architecture Group (Dec. 15, 2004). Architecture of the World Wide Web, Volume One. W3C Recommendation. (Cited on pages 48 and 52.)

Wang, Y., Rutherford, M. J., Carzaniga, A., and Wolf, A. L. (2005). Automating experimentation on distributed testbeds. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE'05)*, pages 164–173. ACM. (Cited on page 17.)

Wassermann, G. and Su, Z. (2008). Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, pages 171–180. ACM. (Cited on page 162.)

Welsh, M., Culler, D., and Brewer, E. (2001). Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243. (Cited on page 90.)

Welsh, M. and Culler, D. E. (2003). Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*. (Cited on page 90.)

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, 25(4):465–470. (Cited on page 133.)

White, A. (2006). Measuring the benefits of ajax. `http://www.developer.com/java/other/article.php/3554271`. (Cited on page 10.)

Willemsen, J. (November 2006). Improving user workflows with single-page user interfaces. `http://www.uxmatters.com/MT/archives/000149.php`. (Cited on page 10.)

Wohlin, C., Host, M., and Henningsson, K. (2005). Empirical research methods in software and web engineering. In *Web Engineering*, pages 409–429. Springer Verlag. (Cited on page 18.)

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers. (Cited on pages 18, 101, 102, and 160.)

Yen, J. Y. (1971). Finding the k shortest loopless paths in a network. *Manag. Sci.*, 17(11):712–716. (Cited on page 138.)

Yin, R. K. (2003). *Case Study Research: Design and Methods*. SAGE Publications Inc, 3d edition. (Cited on pages 18, 119, 142, and 160.)

Yu, D., Chander, A., Islam, N., and Serikov, I. (2007). JavaScript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'07)*, pages 237–249. ACM. (Cited on page 162.)

Zakon, R. H. (2006). Hobbes' internet timeline. Online. `http://zakon.org/robert/internet/timeline/`. (Cited on page 2.)

Curriculum Vitae

Personal Data

Full name
Ali Mesbah

Date of birth
May 23, 1978

Place of birth
Karaj, Tehran, Iran

Education

April 2006 – May 2009
PhD Student (AiO) at Delft University of Technology, Delft, under the supervision of prof. dr. Arie van Deursen.

May 2005 – March 2006
PhD Student (AiO) at the Centrum Wiskunde & Informatica (CWI), Amsterdam, under the supervision of prof. dr. Arie van Deursen.

September 1997 – June 2003
BSc and MSc degrees in Computer Science from Delft University of Technology, specialized in Software Engineering.

September 1994 – May 1997
International General Certificate of Secondary Education (IGCSE) and International Baccalaureate (IB) diplomas from Arnhem International School.

Employment

November 2001 – Present
Software engineer at West Consulting BV, Delft, The Netherlands.

October 2000 – October 2001
Web developer at Active Interface Solutions, Rotterdam, The Netherlands.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty

of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08