INSTITUTE FOR SOFTWARE SYSTEMS
UNIVERSITY OF TECHNOLOGY (TUHH)

PROJECT WORK

User Event Tracking for Test Case Generation for Web Applications

Author:
Anna MEMPEL

Supervisor:
Prof. Dr. Sibylle SCHUPP

February 16, 2012

Abstract

Testing of applications is an essential factor for successful software development. When it comes to business web applications testing ensures the functioning of software, which in turn ensures satisfied customers. Therefore, testing is a key success factor for software development business. Yet testing web applications is not trivial. Depending on the respective application different aspects have to be considered.
This thesis presents an approach to testing web applications based on the events, alias interactions, that users generate in a web application. User interactions like clicks or form filling are recorded by a proxy. Therefore a proxy modifies HTML and JavaScript code and adds extra tracking functions. The collected information is represented as a click sequence, which describes a test case. The test cases can be executed automatically and for example be used to regression-test the web application. The goal is to reduce manual testing efforts and redundancy in test suites and to maximize the number of interactions that are tested.

Affidavit

I hereby declare that the following project work thesis ”User Event Tracking for Test Case Generation for Web Applications” has been written only by the undersigned and without any assistance from third parties.
Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Hamburg, February 16, 2012

Contents

Glossary

Event

As event in the scope of this thesis we define

- Clicks on links or other elements in order to navigate through a web application
- Provided information by filling and submitting a form

HTTP GET

For a definition of the HTTP GET method we refer to RFC 2616 [1] of the World Wide Web Consortium: "The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.[. . .]“

HTTP POST

For a definition of the HTTP POST method we refer to RFC 2616 [1] of the World Wide Web Consortium:"The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources;
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation.

The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI.[. . .]"

Proxy

A proxy can interrupt HTTP communication and modify requests and responses.

Regular Expression

Regular expressions (RegEx) are an instrument for matching strings. It is possible to express anchors (e.g., start or end of string), character classes (e.g., control characters, digits, words), etc. in a formal language that can be interpreted.

Same Origin Policy

For a definition of Same Origin Policy we refer to the World Wide Web Consortium [2]: "An origin is defined by the scheme, host, and port of a URL. Generally speaking, documents retrieved from distinct origins are isolated from each other. For example, if a document retrieved from `http://example.com/doc.html` tries to access the DOM of a document retrieved from `https://example.com/target.html`, the user agent will disallow access because the origin of the first document, (http, example.com, 80), does not match the origin of the second document (https, example.com, 443)."

Test Case
As test case in the scope of this thesis we define a sequence of events or a sequence of interactions.

Acronyms

AJAX	Asynchronous JavaScript and XML
CFG	Control Flow Graph
DOM	Document Object Model
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
MTC	Module Test Case
PTC	Performance Test Case
RegEx	Regular Expression
SOP	Same Origin Policy
STC	Structural Test Case
WTC	Workflow Test Case
XSS	Cross-Site Scripting

1 Introduction

Web applications have become a very important part of the business of many companies these days. A single failure in an application may deadlock a whole company. And it might cost the company that once developed the faulty application - and did not test it well - more than just money. Companies whose everyday business is developing and selling business applications have a special interest in their software working properly. Business web applications in the context of this thesis deal with a varying amount of users and provide much of functionality. They are extended continuously and updates are done at irregular intervals (e.g., every month). Such web applications need to be tested continuously so that potential bugs are detected before users spot them.

Testing web applications in general is not a trivial task. There are many questions to be clarified beforehand. Web applications often contain code that is executed on the server (e.g., servlets) and code that is executed in the clients browser (e.g., JavaScript code). One question is how to test server-side and client-side code in general. For server-side code one can for example send HTTP requests to the server to see whether an exception occurs or not. In order to do this, the web application needs to be up and running. One might also have a look at the logfiles of the server but then one needs access to it. Client-side code is also hard to test, because it is often based on JavaScript and Asynchronous JavaScript and XML (AJAX), which makes it highly dynamic. In addition, there are many different browsers, operating systems, screen resolutions, etc. available that the users of the web application might use and that one has to consider [3]. Looking at a more detailed level, there are further possibilities to consider regarding the testing methods that can be used: If the source code of an application is available, there are several ways to find the presence of bugs. Structural testing for example can tell about the node and edge coverage, and data-flow testing helps to analyze dependencies in the source code. Wang et al. [4] present an approach for testing interactions between pages of a web application. They generate test sequences to cover all pairwise interactions, e.g., a link from one page to another as seen in figure 1. Therefore, a graph model of the application is generated “where each node represents a web page (or a portion of it), and each edge represents a direct link from one node to another” [4]. With such a model it is possible to test all links that exist in a web application. In a web application where every page contains a link to every other page, the number of links (or possibilities of interaction) is

$$\frac{n(n-1)}{2} \tag{1}$$

A web application with four pages would have six links. An application with 20 pages would already have 190 possibilities of interaction. Complex business web applications have usually more than 20 pages and usually the pages contain other elements that one can interact with (e.g., frames) and therefore need to be considered. Testing all interactions becomes a very complex task.

To circumvent this complex task, testing can be limited to a subset of interactions. On this base, two assumptions are made for this thesis. Firstly, it is assumed that an application under test has a fixed configuration (e.g., a fixed constellation of parameters)

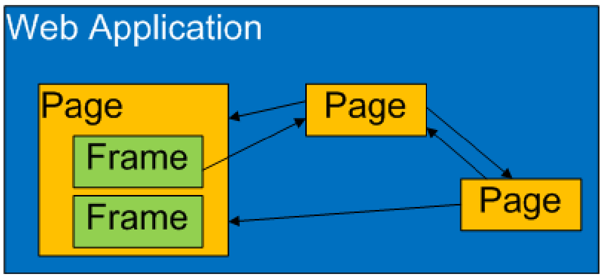


Figure 1: Exemplary link structure of a web application; arrows indicate links from one page or frame to another

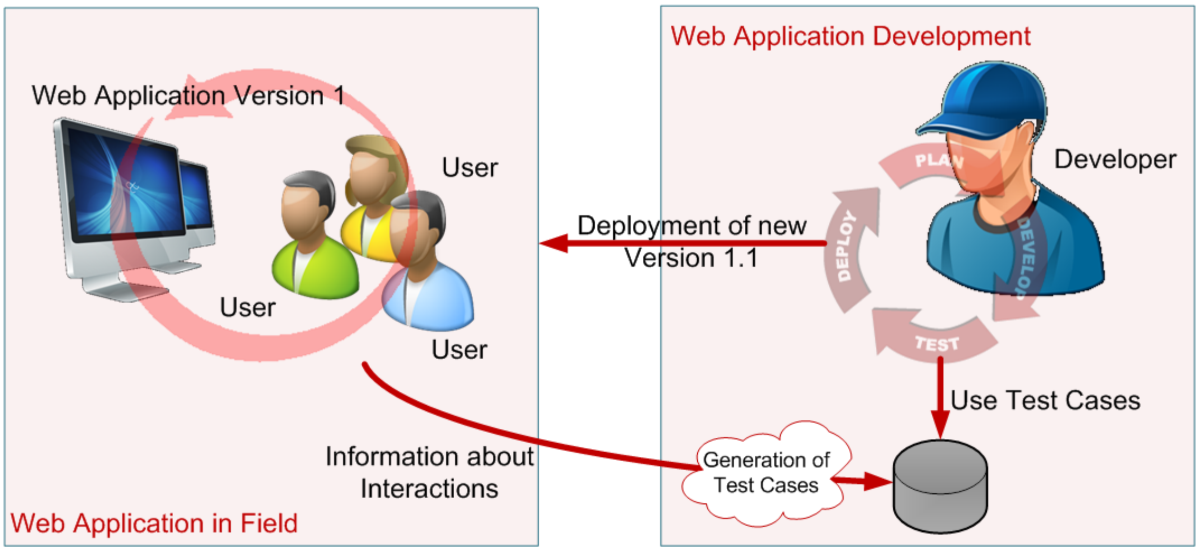

Web Application Version 1
User
User
User
Information about Interactions
Web Application in Field
Web Application Development
Developer
PLAN
DEVELOP
TEST
DEPLOY
Deployment of new Version 1.1
Use Test Cases
Generation of Test Cases

Figure 2: Example field of application; users use a released version of a web application while developers create new features using collected information about the user interactions for regression tests

that leads to a fixed number of possible interactions. Secondly, it is assumed, that users of a web application do not use 100% of the provided functionality and therefore do not use all possible interactions. For example, in a web mail application a specific user might always carry out a sequence of three actions: opening a new mail document, writing mail text, and sending the mail. All other functionality, like the calendar or address book, is not important for the user. Links to pages of this part of the application will never be used.

The second assumptions opens up the perspective to let the users generate test cases with interactions they do with the web application. In business cases the users of a web page and especially their satisfaction are very important for success. The developer of the web application has a special interest in a well-functioning application. It is desirable that potential bugs are not detected by the users.

An exemplary field where user-generated test cases can be used is regression testing. Let us assume, like shown in figure 2, a released version of a business web application used in several companies (user side on the left). In all companies all user interactions (clicks on navigation elements like links, form filling) are recorded continuously. Since every user uses the same application somehow differently, a wide range of interactions is recorded this way. Let us assume that in the meantime the web application is developed further (developer side on the right) and gets some new features. During this process test cases can be generated from the gathered information about interactions with the released version. These test cases can be used to test the existing (old) functionality in the new version of the software.

This thesis presents an approach to record user interactions with a web application and to use the recorded information to automatically generate test cases for testing the web application. The goal is to minimize the runtime and eliminating the redundancy of tests by replacing manually generated test cases by automatically, uniformly generated test cases. The approach is not based on browser plugins, since plugins are dependent on a specific web browser. The approach is rather based on a proxy that neither acts on the server-side nor on the client-side. Another advantage is that users are not burdened with setup. They do not have to install third party programs or to set up the proxy in the browser settings, since the proxy is URL based [5].

Section 2 discusses the definition of user events and how they are tracked with the proxy approach. Besides the WebQuilt proxy an alternative approach is discussed. Section 3 gives a short overview of software testing, and shows how user events recorded by the WebQuilt proxy are represented as test cases. An example field of application of the generated test cases is also presented.

Section 4 is related to the application of the proxy approach to a real-world application. The standardization and reproduction of existing test cases are discussed to estimate the quality of the presented approach, and the contribution of the proxy to the testing process is shown.

2 Tracking User Events via Proxy

Logging user activities on a web application is a common activity. Administrators log on to the back-end to solve upcoming internal problems (e.g., database problems as result of a complex query many users trigger at the same time) effectively. Programmers log on to the front-end to get useful information for enhancement of their web application (e.g., to test different layouts of an online shop to find out which layout best leads to successful orders).

For our approach we are interested in logging the events a user carries out in order to interact with, and navigate through the application. As event we define

- Clicks on links or other elements in order to navigate through the web application
- Provided information by filling and submitting a form.

Logging such events, in the following also called interactions, can be approached in different ways. On the one hand, one can prepare the application to log events and then consult the logs on the server-side. The advantage is that the users do not need to care about the logging procedure. They do not need to configure anything. But a problem might be that the code and the logs are only available to administrators of the website and that it is not possible to change the code. On the other hand one can log users actions on the client-side. In this case, the users need to download and install some kind of software (e.g., a browser plugin) that is responsible for event recognition. But it might be a problem to handle compatibility with a wide range of web browsers and operation systems. An alternative to server or client-side logging is a proxy that acts in the middle of client and server (refer to figure 3 in section 2.1.1). Hong et al. [5] therefore propose a proxy for logging web usage on *any* website. The proxy that is used in this approach is part of the WebQuilt tool and can track clicks on links and form filling. Atterer and colleagues [6] propose a similar approach that is not URL-based and that also recognizes mouse positions and scrolling. In our approach we make use of the WebQuilt proxy to track user a a web application.

The following section 2.1 treats the WebQuilt proxy, the principle of operation, and the state of the release version. Extensions of the proxy are explained in detail. Section 2.2 discusses an alternative approach.

2.1 WebQuilt Proxy

The WebQuilt project, that was once designed for web site usability analysis, provides tools to log and illustrate web activity. WebQuilt comes along with a proxy that records user interactions with web applications. It is based on Java Servlet and JSP technology [7]. For our approach we made use of the released WebQuilt Proxy v1.0RC1.

WebQuilt contains four additional components: the action inferencer, the graph merger, the graph layout component, and the visualization component, but these are not available for download or development.

2.1.1 General Principle of Operation

In General the WebQuilt proxy works like a normal proxy (refer to figure 3): It interceptsHyper Text Transfer Protocol (HTTP) communication.

The proxy receives HTTP requests e.g., for an Hyper Text Markup Language (HTML) document from a client, modifies them, and forwards the modified request to the server. Then the proxy receives a document as response, modifies it, and forwards it to the client. Unlike common proxies, the WebQuilt proxy is URL based. That means that all links in a HTML page are modified to redirect through the proxy again. The user does not have to specify the proxy in the web browser, instead the proxy is used automatically when clicking a modified link. The modification of HTML documents happens in several steps. When a request is received, WebQuilt-specific parameters are registered and saved. These parameters are appended by the proxy to all links and used to identify

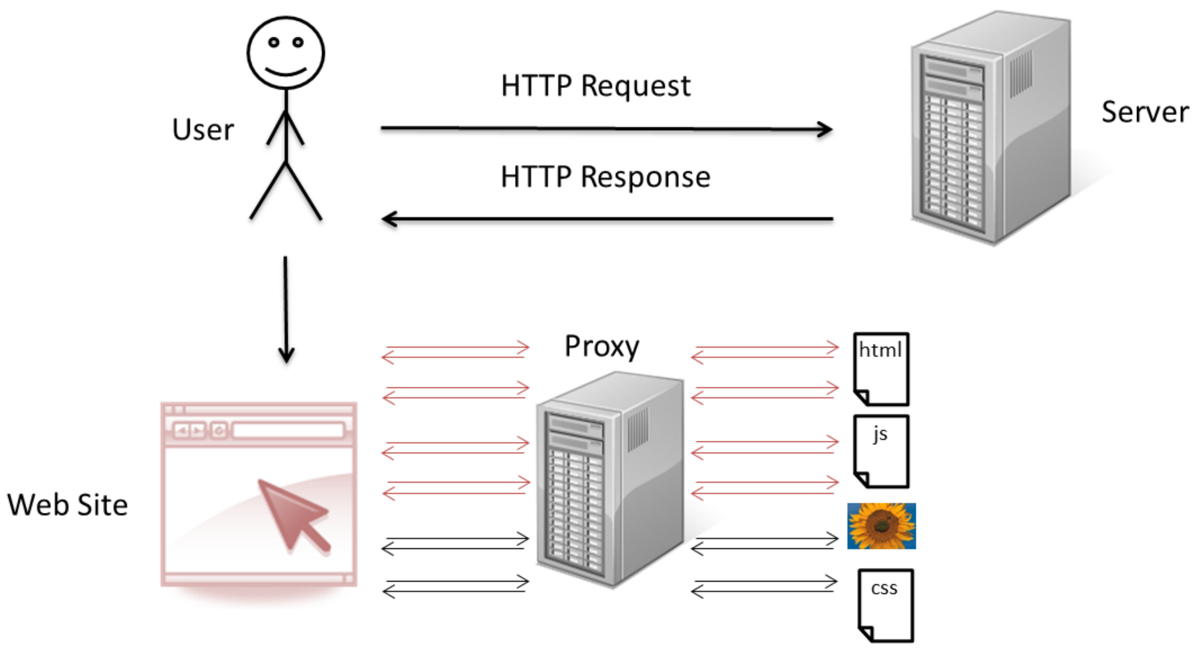


Figure 3: Interrupting HTTP communication with a proxy

the user interactions explained in the following section. An URL modified by WebQuilt looks as follows:

```
http://www.proxy.de?wq_replace=www.example.de&wq_linkid=4&param=test
```

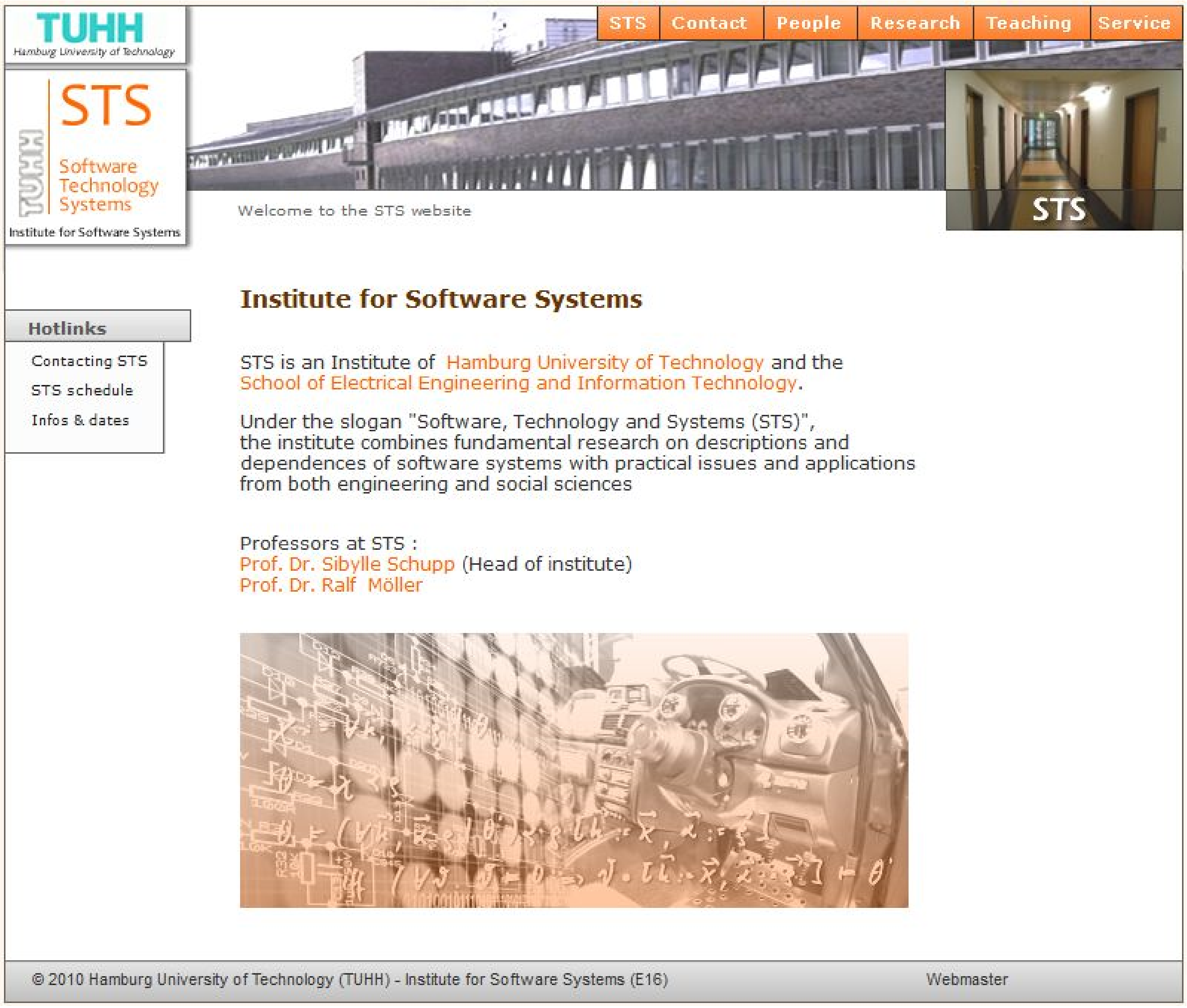
The WebQuilt-specific parameters always start with "wq_".

In the next step the originally requested document is requested. The HTML code of the response is then analyzed and all links are modified like mentioned. In the last step the modified response is cached and then sent back to the client.

2.1.2 Principle of Event Logging

The WebQuilt-specific parameters are used to log the user actions on a page and to redirect links through the proxy. For example one parameter represents the originally requested URL inclusive original parameters and one the ID of the corresponding page. One parameter corresponds to the ID of the link the user clicked on. For example, the ID of the third link in the Document Object Model (DOM) tree, represented by $< a >$ tags, is 3. Another parameter represents the ID of the current page's HTML frame parent, and one the frame number of the current page and so on. The whole list of parameters is documented in the WebQuilt "ReadMe" file [7]. Based on the information gained from the WebQuilt parameters and further calculations, the WebQuilt proxy builds a logfile as shown in tables 2 to 5.

Table 2 shows an example logfile of a user that visited the website of the Institute for Software Systems of the Hamburg University of Technology (figure 4). The first row of table 2 shows the start of the logged sequence. The user's session is active for almost three



Welcome to the STS website

Institute for Software Systems

STS is an Institute of Hamburg University of Technology and the School of Electrical Engineering and Information Technology.

Under the slogan "Software, Technology and Systems (STS)", the institute combines fundamental research on descriptions and dependences of software systems with practical issues and applications from both engineering and social sciences

Professors at STS :
Prof. Dr. Sibylle Schupp (Head of institute)
Prof. Dr. Ralf Möller

© 2010 Hamburg University of Technology (TUHH) - Institute for Software Systems (E16)

Figure 4: Website of the Institute for Software Systems of the Hamburg University of Technology [8]

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
175467	0	12	-1	200	-1	-1	GET	`http://www.sts.tu-harburg.de/`	-1
178986	12	13	-1	200	-1	3	GET	`http://www.sts.tu-harburg.de/contact/index.html`	`contact/index.html`
183230	14	15	-1	200	-1	8	GET	`http://www.sts.tu-harburg.de/service/index.html`	`../service/index.html`
190631	16	17	-1	403	-1	10	GET	`http://www.sts.tu-harburg.de/service/intern/faq.html`	`intern/faq.html`

Table 2: Example logfile of the website of the Institute for Software Systems of the Hamburg University of Technology where the user clicked different links

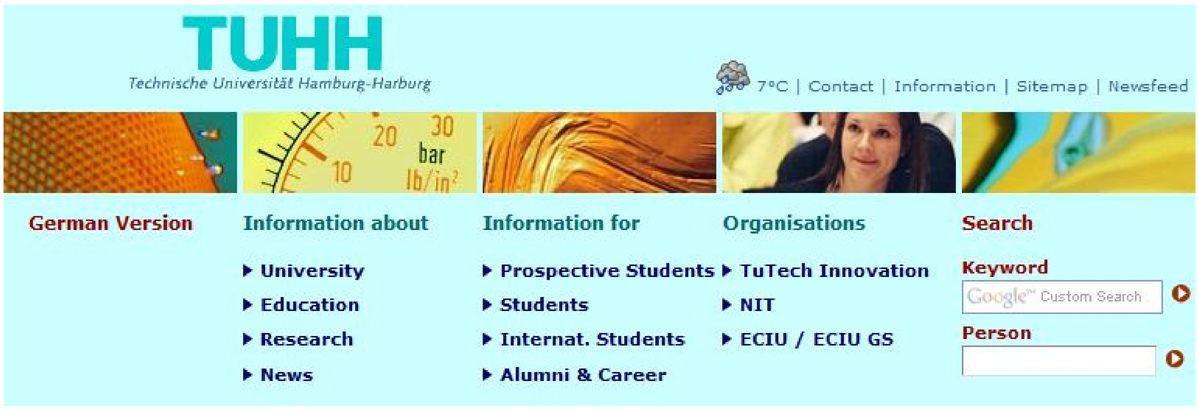


Figure 5: Website of the Hamburg University of Technology [9]

minutes (column one). Columns two to four show different IDs: Column two represents the page the user came from. Column three is the current page. These numbers are based on the user's HTTP session. The page on which the user started the session has ID 1, the page visited next has ID 2 and so on. Column four shows the frame parent of the current page, or -1 if there is none. Column five contains the server status code. Common server status codes are 200 "OK", 301 "Moved Permanently", 403 "Forbidden" or 404 "Not Found". The ID in column six tells about the current frame (e.g., in a frameset) or -1 in case it is not a frame. Column seven is important, it gives the ID of the link the user clicked. Column eight denotes the HTTP method that was used. Column nine and ten are also very important. Column nine shows the current URL and a potential parameter string (query) that comes along with it. Column ten contains the *href* element of the link the user clicked on[1]. The link cell (column ten) of the second row shows that the user clicked on a link `contact/index.html` with ID 3 to get to `http://www.sts.tu-harburg.de/contact/index.html`. The third row shows that the user navigated to `http://www.sts.tu-harburg.de/service/index.html` afterwards via clicking link 8. Then it was tried to open `http://www.sts.tu-harburg.de/service/intern/faq.html` but the server returned the error code 403, since the user did not have sufficient rights to request the page.

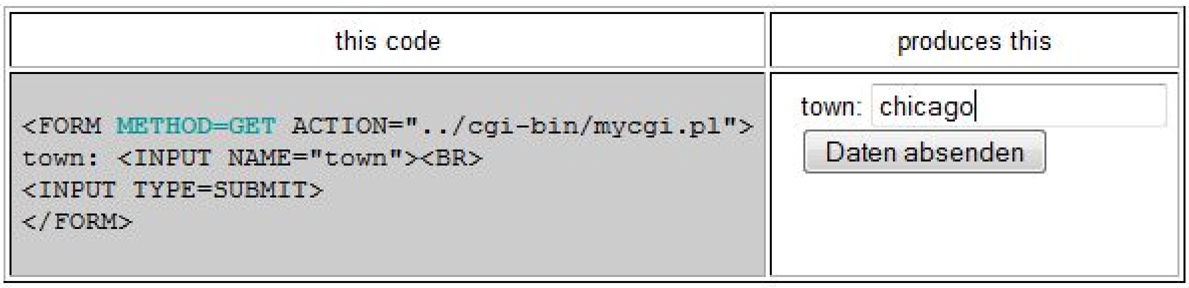
Table 3 shows how WebQuilt recognizes the usage of the back button of the browser. Row one shows that the user visited the website of the Hamburg University of Technology (figure 5). It was clicked on `/tuhh/studium/studierende.html` (row two) and afterwards `/tuhh/studium/studieninteressierte.html` (row three). The IDs in column two, which depicts the page the user came from, are the same for row two and three. That means that the user came from the same page in both cases. From that fact it can be derived that the back button of the browser must have been used in between.

Table 4 shows an example logfile of a page with a GET form (figure 6) where the name of a city needs to be filled in. The user typed "chicago" and submitted the form. In a

[1]Column ten was inserted within this project and is not part of the release version of the WebQuilt proxy.

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
21981	0	2	-1	200	-1	-1	GET	`http://www.tu-harburg.de/`	-1
29292	2	4	-1	200	-1	11	GET	`http://www.tu-harburg.de/tuhh/studium/studierende/rund-um-das-studium.html`	`/tuhh/studium/studierende.html`
40060	2	9	-1	200	-1	10	GET	`http://www.tu-harburg.de/tuhh/studium/studieninteressierte/bachelor-studiengaenge.html`	`/tuhh/studium/studieninteressierte.html`

Table 3: Example logfile of the website of the Hamburg University of Technology where the user used the back button of the browser



```
<FORM METHOD=GET ACTION="../cgi-bin/mycgi.pl">
town: <INPUT NAME="town"><BR>
<INPUT TYPE=SUBMIT>
</FORM>
```

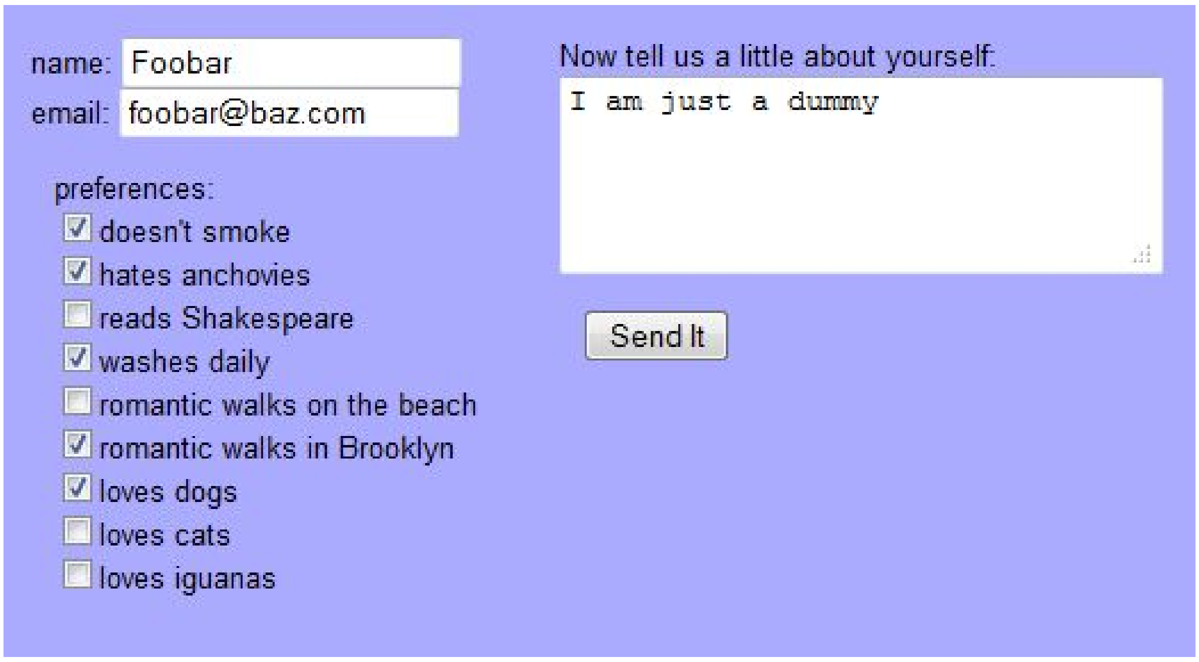
Figure 6: Example GET form [10]

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
41776	0	2	-1	200	-1	-1	GET	`http://www.htmlcodetutorial.com/forms/_FORM_METHOD_GET.html`	-1
43016	2	3	-1	200	-1	-1	GET	`http://www.htmlcodetutorial.com/cgi-bin/mycgi.pl?town=chicago` town=chicago	-1

Table 4: Example logfile of a website with a GET form where the user filled in information

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
13602	0	6	-1	200	-1	-1	GET	`http://www.htmlcodetutorial.com/forms/_FORM_METHOD_POST.html`	-1
14313	6	7	-1	200	-1	-1	POST	`http://www.htmlcodetutorial.com/cgi-bin/mycgi.pl` realname=Foobar& email=foobar%40baz.com& nosmoke=on& hatesanchovies=on& washesdaily=on& brooklyn=on& dogs=on& myself=I+am+just+a+dummy	-1

Table 5: Example logfile of a website with a POST form where the user filled in information


name: Foobar
email: foobar@baz.com

preferences:
[✓ doesn't smoke]
[✓ hates anchovies]
reads Shakespeare
[✓ washes daily]
romantic walks on the beach
[✓ romantic walks in Brooklyn]
[✓ loves dogs]
loves cats
loves iguanas

Now tell us a little about yourself:
I am just a dummy

Figure 7: Example POST form [11]

GET request all parameters are appended to the URL. The WebQuilt proxy recognizes them and lists the parameters in column nine (town=chicago).

Table 5 shows an example logfile of a page with a POST form (figure 7) where some information needs to be filled in. The user filled in the name, the email, and the *myself* field and checked several checkboxes. Then the form was submitted. In a POST request nothing is appended to the URL. The WebQuilt proxy recognizes the parameters anyway and lists them in column nine.

The proxy does not capture IP addresses. Nevertheless it logs parameters that can contain personal data like usernames and passwords. The proxy provides a function to replace passwords with wildcards before writing them to the logfile. Enabling this function makes generated test cases useless, since they can not be executed automatically if the password is unknown (refer to tables 8 and 9 in section 4.2). Therefor all parameters, even passwords, are saved in the clear. According to §15 of the German Teleservices Act [12] users need to be advised in case personal characteristics are logged.

Tables 2 to 5 always show just a small piece of a logfile. In general, logfiles can become very large, since all actions of a user are tracked during a session. Thus in a normal use case all the different scenarios shown can occur many times in a single logfile.

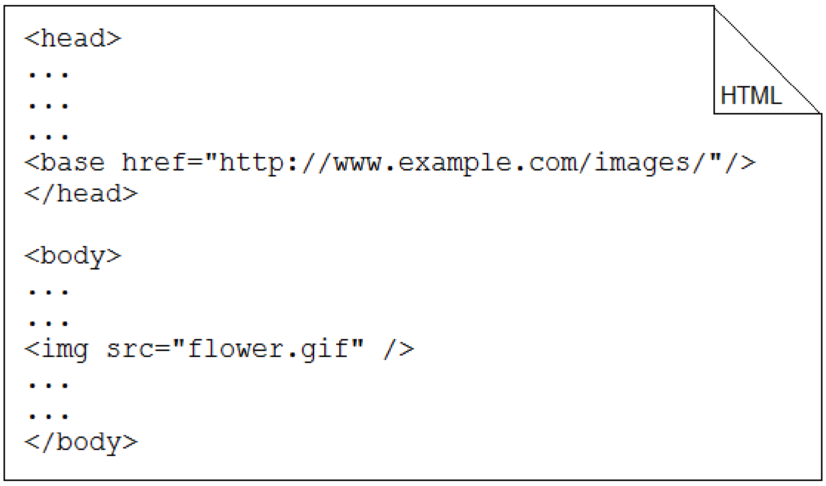
2.1.3 Capabilities, Problems and Limitations of the Release Version

The WebQuilt proxy is delivered as web application within a pre-configured Apache Tomcat server. The server simply needs to be started with a script file in order to get the proxy up and running. This fact enables the proxy to be set up on the fly on any machine.

The proxy modifies links in HTML pages in order to redirect through the proxy again and to identify user actions. As described in section 2.1.1 the interactions the proxy can capture in the release version are

- clicking a link,
- clicking the back button of the web browser,
- submitting a GET form, and
- submitting a POST form.

In the release version of the proxy the $< base >$ tag of an HTML page is modified. This tag is used to set the base URL for links in the website. But the proxy places the tag at the end of the *head* part of the modified HTML page as shown in figure 8. Code that is written above the $< base >$ tag (e.g., an include of a JavaScript file) does not know about this $< base >$, so related sources are not available. This leads to errors and malfunctioning web applications when using the proxy. We solved this problem with



```
<head>
...
...
...
<base href="http://www.example.com/images/"/>
</head>

<body>
...
...
<img src="flower.gif" />
...
...
</body>
```

Figure 8: Example HTML file with base tag at the end of the HEAD part

several changes in the HTML parsing and modification algorithm of WebQuilt.

In general the set of user interactions that can be captured by the proxy is limited. The proxy records the links, the user clicks on as well as a click on the back button of the browser. Also, when the user fills a form, all values written to the fields are recorded. But there is for example no information captured about the position of the cursor on the screen. With such information the graphical arrangement of the elements (e.g., buttons) of a page could be tested [5]. The proxy approach of Atterer et al. [6] provides this functionality.

One big limitation of the release version of the proxy are dynamic web applications that use JavaScript. Nowadays, many web applications use AJAX requests to load content dynamically. The proxy is not able to modify JavaScript code. This leads to several problems. First of all, there are many situations where URLs are used in JavaScript function calls. One example is the $window.open(URL, windowName)$ method that is used to open a pop-up window. If the URL is not modified before the call is executed, the browser security restrictions (Same Origin Policy (SOP)) will prevent the pop-up from executing JavaScript code in the parent window (Cross-Site Scripting (XSS)) [5]. The reason is that the parent window has a modified WebQuilt URL but the pop-up not, so the origin is different. How we tried to overcome this JavaScript limitation is described in chapter 2.1.5.

The second problem are JavaScript events. With events like *onclick* it is possible to interact with a web application. For example a menu structure in the Graphical User Interface (GUI) of a web application can technically be a HTML table where clicks on table rows would switch between menus or submenus (refer to listing 4 in section 2.1.6). The proxy can not capture such navigation. How we tried to deal with this limitation is shown in chapter 2.1.6.

The proxy also has some limitations regarding the scheme of URIs. URIs consist of four parts, where the first part describes the scheme:

$$< schemename >< hierarchicalpart > [? < query >][\# < fragment >].$$

The proxy is not able to detect clicks on links with *mailto* URI scheme that open a new email document in the user's email client. Therefore such events are by-passed and not listed in logfiles. The *https* scheme leads to exceptions. Referring to [13] and [5] WebQuilt supports Hyper Text Transfer Protocol Secure (HTTPS) but we were not able to use it with this version. In our tests the proxy only worked with the *http* scheme.

2.1.4 Extending the Logfile Format

In order to overcome WebQuilts limitations we did several small and bigger modifications. The first modification extends the logfile format of the proxy. When modifying HTML pages, different parameters are appended to the links in the code. In this way every link gets an ID or the ID of a parent frame appended as parameter. The parameters are filtered by the proxy when a request for this URL passes it. In this way actions are identified and logged (refer to section 2.1.1). In order to provide easier identification of the links the user clicked on, the logging algorithm was modified. The *href* element value of a link is extracted and appended as additional parameter *wq_link*. The *href* value is the exact text that is in the final HTML page, for example the text *people.html* in listing 1:

Listing 1: Example code for a HTML link

```
<a href="people.html">
```

It is important to keep existing encoding in the *href* parameter value in order to be able to find the exact value later while testing. Therefore the *wq_link* parameter is appended to the link URL without further encoding.

2.1.5 Extending the Parsing Algorithms

A second extension was done to enable the proxy to parse JavaScript additionally to HTML code. Script code can contain URLs that need to be modified in order to redirect through the proxy again. But since the code is executed on the client-side, URLs might not be detected when the request passes the proxy. The reason is that URLs are often computed dynamically with variables. Consider the code from listing 2:

Listing 2: Example JavaScript code containing an URL

```
...
var bar = "login?browserCheck=1&eChar=%E2%82%AC";
...
window.open(foo() + bar,"Browsercheck");
...
```

There are several difficulties regarding dynamic URLs as shown in listing 2. First of all, it might not be obvious that the variable *bar* contains some part of a URL. Secondly, even if it is known that a call of *window.open()* always contains a URL as first parameter one needs to handle the concatenation of strings to deal with the whole URL.

When it comes to proxying JavaScript code two different situations need to be considered: JavaScript code can be embedded in an HTML file with simple $< script >$ tags on the one hand. In this case the code is placed between a pair of $< script >$ and $< /script >$ tag. On the other, hand one can include one or more JavaScript files via including a *src* element in a $< script >$ tag like shown in listing 3:

Listing 3: Example code for including a JavaScript file

```
<script type="text/javascript"
        src="http://www.example.com/js/example.js">
```

Parsing the code in the first case can be done at the same time when the HTML file is proxied. For the second case it is necessary to redirect the whole JavaScript file through the proxy.

For handling the second case the proxy was modified to

- modify the *src* parameter of $< script >$ tags via appending the original URL as parameter. Thereby a request for the file will go through the proxy, e.g., `http://proxy?wq_replace=http://www.example.com/js/example.js`.

- spot incoming requests with content type *application/x-javascript* etc., which marks the content as JavaScript code.
- handle these requests separately to parse and modify the JavaScript code.

In order to parse and modify JavaScript code it was first tried to use the Rhino API [14] to include a full JavaScript parser. Parsing the code would enable syntax check. But since dependent JavaScript code can be interspersed throughout different documents and script blocks, it is difficult to keep track of the right order while parsing. Another problem comes from the fact that JavaScript normally runs in a web browser. Since the proxy does not provide a browser environment, no browser variables (e.g., document) are available when parsing the code in the proxy. This leads to parsing errors and needs to be treated. In order to keep the JavaScript code analysis and modification simple, we abandoned the implementation of a parser. Instead the JavaScript code is scanned with regular expressions to find URLs at common places in source code. One of this places is a call to *open(method, url, async)* that is used to open an AJAX request. The code can be scanned for such calls with a regular expression like this:

```
/.*\.+(open)+\(+.*,.*,.*\)+;+.*/
```

If a method call was found with such a regular expression, the second parameter, the URL, needs to be modified. This is done on the client-side by a JavaScript function that the proxy inserts into all HTML pages it proxies. The function adds the original URL as parameter to the URL of the proxy. Other places where URLs are used and that need to be handled are

- calls of *window.open(URL,name,specs,replace)*
- replacements of the document URL with *location.href=*
- or with calls of *location.replace(URL)*
- changes of the *src* property of an element with *src=*

For all these cases a Regular Expression (RegEx) was created that finds the respective pattern in JavaScript code. But there is a challenge that makes modification of JavaScript code difficult, even if a complete list of patterns to modify would exist: there is no coding standard. The result of that is that every web application may have differently looking code with different formatting etc. It is not possible to guarantee that the regular expression will find every dynamically generated URL in every website.

This problem aside, the XSS limitation was offset: URLs of popups generated with *window.open()* get modified. Therefore popups have the same origin as the parent windows. This enables pop-ups to execute JavaScript in the parent window and AJAX requests to successfully execute.

2.1.6 Extending the List of Observable Events

Based on the explained JavaScript parsing extension another modification was done in order to extend the list of events the proxy can capture (refer to section 2.1.3). With JavaScript it is possible to change the content or structure of a web application without sending new HTTP requests. In such cases the code for all elements is already present but not all elements are visible at the same time. A click on an element will toggle the visibility of elements so that the user can access new functions and other functions are hidden. Such cases can not be recognized by the released version of the proxy, since no HTTP request will pass through it. Since the web application used to measure the quality of the proxy (refer to section 4) uses many of such navigation elements, the proxy was extended to capture such events. Therefore every HTML element that has an *onclick* event gets an additional function appended to the existing *onclick* functions. When the event fires, an AJAX call to the proxy will log the ID of the clicked element. Elements that had no *onclick* event before proxying do not get one, since a click on the element was not relevant before and this should not change. The difficulties of this extension are the different ways to build an HTML page. One way is to write static HTML code with elements specified by distinct tags like shown in listing 4. The opening tag of an HTML element can contain an *onclick* parameter.

Listing 4: Example code for statically building of HTML code

```
<body>
 <table id="menue">
  <tr onclick="selectPage('overview');">overview</tr>
  <tr onclick="selectPage('settings');">settings</tr>
 </table>
</body>
```

HTML code like the one in listing 4 will be parsed by the proxy and the *onclick* element will be recognized, extracted, modified, and written back to the modified version of the page. This case is easy.

Another way to built an HTML page is dynamically with the help of JavaScript like shown in listing 5.

Listing 5: Example code for dynamically building of HTML code

```
<head>
 <script type="text/javascript">

 function init() {
         var table = $("#menue");
         for (var i in getMenueItems())
                 table.append("<tr onclick='selectPage("
                         + i
                         + ");'>"
                         + i
11                              + "</tr>");
12  }
13
14  window.onload = init;
15
16  </script>
17 </head>
18
19 <body>
20  <table id="menue"></table>
21 </body>
```

In this example the HTML code is built dynamically when the page is loaded and the *init* function is called. HTML code like this can only by modified by the proxy at the time the JavaScript code is modified. In order to include such modification the same technique as described in the previous section was used: By applying several regular expressions the code is scanned for *onclick* parameters in HTML code. If such a parameter is spotted a short piece of code is inserted before the first instruction inside the *onclick* parameter. With this a new *onclick* event is added to the element. Line 7 to 11 in listing 5 would be modified like shown in listing 6:

Listing 6: Example code for adding a new onclick event

```
7  table.append("<tr onclick='"
8          + "var request = new XMLHttpRequest();"
9          + "request.open('GET',"
10         + proxyURL
11         + "'?wq_clicked_id='"
12         + this.id
13         + ", true);"
14         + "request(send);"
15         + "selectPage("
16         + i
17         + ");'>"
18         + i
19         + "</tr>");
```

When a user clicks this modified table row, an AJAX request is asynchronously sent to the proxy. The proxy can then log the ID of the element that was clicked on. Afterwards all *onclick* events are executed that existed for this element before.

This modification comes along with the same problem as the one described in the last section: it can not be guaranteed that the regular expression will find every *onclick* parameter in the code.

This modification is also not appropriate for a live usage of the proxy. If a user triggers this asynchronous logging and shortly afterwards a synchronous event occurs, the logging order might get screwed up. Additionally this modification infringes the proxy concept of intercepting HTTP communication: The client triggers the logging by directly requesting the proxy. This modification therefore is a temporary workaround in order to be able to use the proxy with heavy JavaScript and AJAX related applications (refer to section 4).

2.2 Alternative Approach

With the shown modifications (refer to sections 2.1.4 to 2.1.6) the WebQuilt proxy can now modify JavaScript code. Also the logfile format was extended and with a workaround it was tried to extend the list of events the proxy can capture. Figure 9 shows how the modified and extended version of the WebQuilt proxy works. [h]

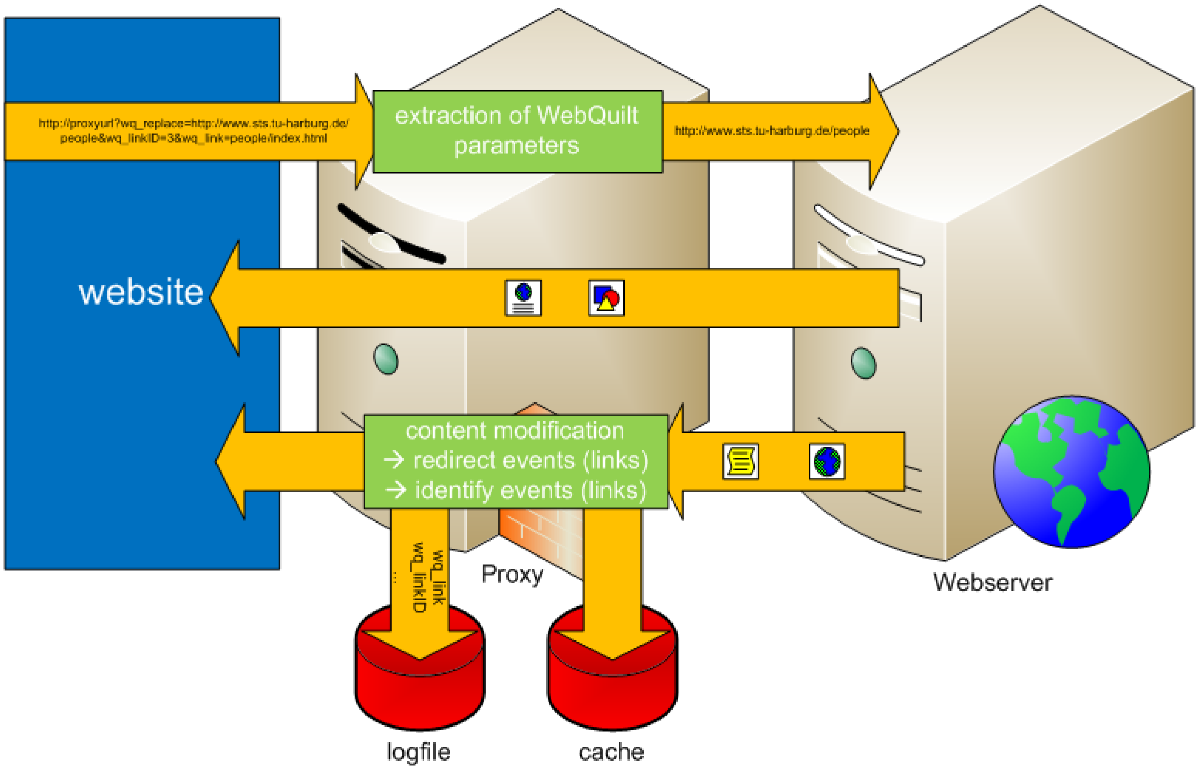

http://proxyurl?wq_replace=http://www.sts.tu-harburg.de/people&wq_linkID=3&wq_link=people/index.html
extraction of WebQuilt parameters
http://www.sts.tu-harburg.de/people
website
content modification
→ redirect events (links)
→ identify events (links)
wq_link
wq_linkID
...
Proxy
Webserver
logfile
cache

Figure 9: WebQuilt Proxy (extended)

The released version as well as the extended version of the WebQuilt proxy is error-prone. Since the proxy modifies the content the user will see, it can not be ensured that errors found with the generated test cases are not caused by the proxy itself (e.g., due to implementation faults).

An alternative is a proxy that does not change the code of the response page. Such an approach is shown in figure 10. Compared with WebQuilt this proxy is not URL-based. The web browser of the users needs to be configured in order to redirect all traffic through the proxy. When a request reaches the proxy the request is forwarded unchanged. When the server returns a response, the unchanged response is stored in a lookup-database and

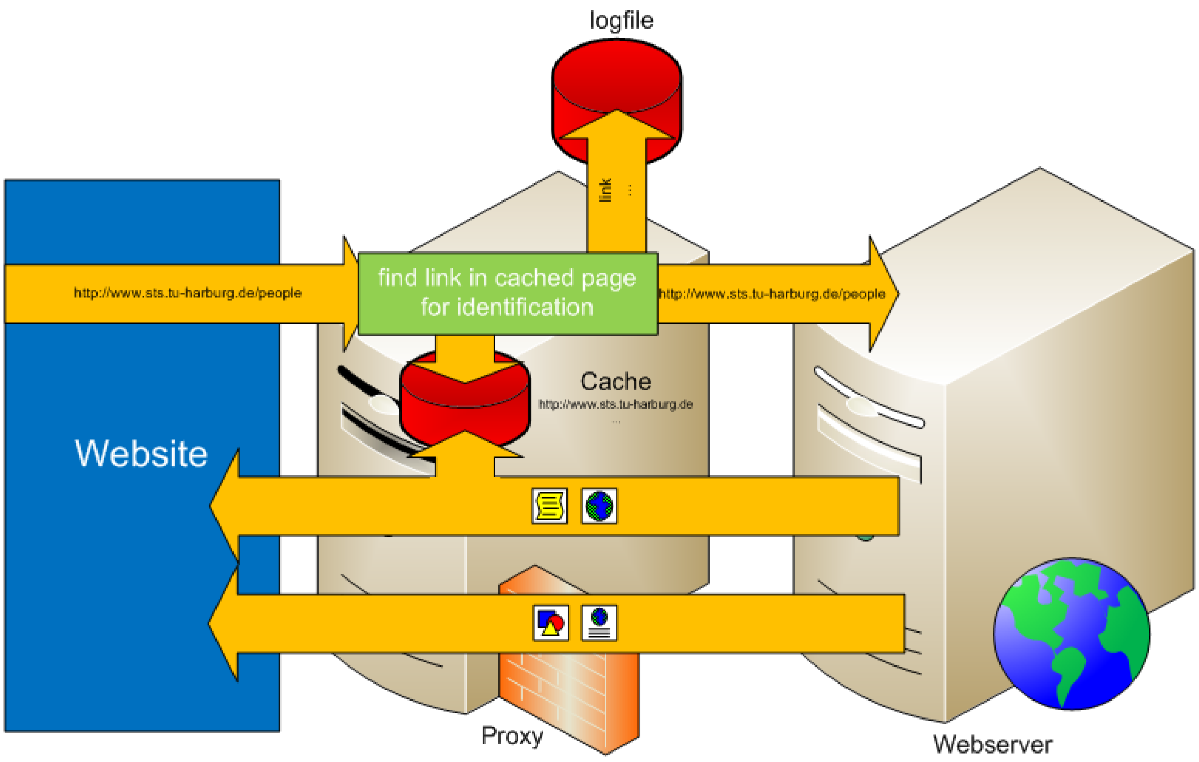

logfile
link
...
http://www.sts.tu-harburg.de/people
find link in cached page for identification
http://www.sts.tu-harburg.de/people
Cache
http://www.sts.tu-harburg.de
...
Website
Proxy
Webserver

Figure 10: Alternative approach

forwarded to the user. Let us assume the user clicks a link in the page that was returned. The request will go through the proxy. The proxy will now search the lookup-database for the document the request came from (source). The source can for example be identified by the HTML referrer header. The stored document contains the link the user clicked on. Different algorithms could compute unique characteristics (e.g., an ID specifying its position in the DOM) for the link. For this the HTML and JavaScript code would be analyzed similar to the WebQuilt approach. Afterwards the link and its characteristics are stored in a logfile, just like in the WebQuilt approach.

The advantage of this approach is that the page content is not modified, so the web application will be displayed correctly in any case. Errors found by test cases are definitely not generated by the proxy. Another advantage is the same origin that is preserved for all documents that are transferred during a concrete client-server-communication. This comes from the fact that the proxy is not URL-based. But at the same time this is also a disadvantage, because the user needs to configure the web browser in order to redirect the requests through the proxy.

The main problem that exists with this approach comes from dynamic links and is basically the same as with the WebQuilt proxy. When a link is calculated or changed dynamically on the client-side with JavaScript, it might not be possible to find it in the cached page, since the cached page does not contain the calculated or changed version of the link. To solve this problem it is necessary to analyze the cached page for patterns, that could modify links and derive possible changes from that. In this way it might be possible to find dynamically changed links and recognize events caused by those links. In the worst case it would not be possible to track all users actions on pages that use JavaScript and AJAX for dynamic code changes.

Another problem is the recognition of events that do not cause an HTTP request. As described in 2.1.4 it is possible to toggle the visibility of content on the client-side with JavaScript events and therefore provide or hide parts of the functionality. Since this happens on the client-side, this alternative proxy will not recognize such events. To overcome this problem in a nice way it might be helpful to make use of the approach of Atterer et al. [6]. In this approach the proxy injects a JavaScript file into the response, which is responsible for logging events. The script appends *onclick* events to the elements of an HTML page. This solution would change the concept of this alternative approach as follows: The proxy will also change the code on the client-side like WebQuilt does. But this code modification is different compared to WebQuilt. The first modification happens in the proxy by injecting a script to the response page. The second modification happens on the client-side. The injected script will append an *onclick* event to the elements. Therefore this modification does not parse HTML or JavaScript code.

3 Test Case Generation

The user event tracking approaches presented make it possible to capture interactions of the users with a web application. In order to make use of this information for testing purposes, software testing will be shortly explained in section 3.1. Section 3.2 shows how user events recorded by the WebQuilt proxy are represented as test cases . The test cases are classified and an algorithm for similarity checks is presented. Finally, an example of an application for the generated test cases is shown in 3.3 including automated execution of test cases.

3.1 Software Testing

Referring to IEEE 610 [15], in (software) testing the whole software or a part of it is executed under specified terms. Results are monitored, logged, and analyzed. Many different methods and techniques exist to test software in an efficient way. Pezze and Young [16] describe five levels of testing: module testing, integration testing, system testing, acceptance testing and regression testing. Module testing tests the behavior of modules against expectations and specifications. With integration tests the compatibility of the modules of an application is checked. System testing, including systemwide tests, actually represents the last step of integration testing. Acceptance tests check the match of the users expectations with the final application. With regression testing possible faults are uncovered that were introduced during further development.

When it comes to methods on how to test Pezz and Young present for example functional testing to test the program specifications, structural testing to test branches, statements etc. and data flow testing to test variable definitions and uses. Such methods can be applied to the testing levels mentioned but not all methods work for all levels. Regardless of the level and selected method of testing, almost always test cases are defined.

3.2 From User Events to Test Cases

A test case in the scope of a web application is an event sequence, or technically rather a sequence of URLs specifying the pages and potential parameters as input (e.g., for forms) [17]. With the WebQuilt approach as well as with the alternative approach presented we log event sequences generated by the user of the web application. One logfile or an event sequence respectively represents one test case.

3.2.1 Classification of Test Cases

For a web application there are many aspects that should be tested. One aspect is the functionality of the back-end, another is the functionality of the front-end, and yet another is the proper layout of the GUI. In this thesis we focus on testing methods where testing can be done from the user perspective. The different testing methods can be represented by different classes of test cases.

Module Test Cases This type of test case tests connectivity from one module of the web application to another. Two modules of a web application can be connected for example by a link.

Functional Test Cases This type of test case tests some isolated functionality of the web application and the desired output e.g., that a pop-up opens or that field validation works.

Content Test Cases This type of test case tests the correctness of the displayed content, e.g., text that is based on calculations.

Workflow Test Cases This type of test case tests a workflow of the application from start to end, e.g., logging in, opening a mail document, writing text, sending the mail.

Non-Functional Test Cases This type of test case tests usability related aspects like arrangement of the elements of a website.

Performance Test Cases This type of test case tests the performance of a web application e.g., if the application can deal with an extreme high amount of requests (stress test).

Security Test Cases This type of test case tests security issues e.g., finds and uses vulnerabilities to do a cross site scripting attack.

Structural Test Cases This type of test case tests the structure of the software e.g., a path, branches or statements in the Control Flow Graph (CFG).

The presented WebQuilt proxy approach captures click sequences that can be used as test cases of type Module Test Case (MTC), Workflow Test Case (WTC), Performance Test Case (PTC), and Structural Test Case (STC).

3.2.2 Compilation of Test Cases to Test Suites via Similarity Check

Test cases are usually grouped in test suites in a way that is useful for the respective testing level. For example for regression testing several methods for selecting an appropriate set of test cases exist in order to reduce the required testing time [18].

For a web application that is continuously developed further the set of test cases also needs to be extended continuously. This can be done with the presented proxy approaches. In order to avoid redundancy in such a suite of tests a similarity-check algorithm needs to be established. This algorithm checks whether a newly generated test case would increase the interaction coverage of a test suite. This coverage can be expressed as

$$c_I = \frac{\#\text{ exercised interactions } i(p_j, p_k),\ i \subset I, p_j, p_k \subset P}{\#\text{ interactions } I} \tag{2}$$

where I represents the set of possible interactions that exist for any two pages p (or parts of them) from the set of pages P that exist for a web application. The interaction coverage increases, if the test case to be added contains a sequence of events that is not a subsequence of the event sequence of another test case. The event sequence of a test case t_1 is a subsequence of the events of another test case t_2 if all elements of all events of t_1 are equal to all elements of a sequence of events of t_2. Technically an event is identified by several elements that are logged by the WebQuilt proxy (refer to 2.1.1): the ID of the parent element in the page, the HTTP response code, the ID of the frame, the ID of the link the user clicked on, the method type, an URL and parameter and the *href* parameter of the link. All these elements need to be equal for two events in order to declare these events as equal. For example the event sequence shown in table 6 is a subsequence of the event sequence shown in table 7 because all parameters that identify an event are equal for the respective events in both sequences.

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
22528	0	2	-1	200	-1	-1	GET	`http://www.sts.tu-harburg.de/`	`-1`
35568	2	4	-1	200	-1	3	GET	`http://www.sts.tu-harburg.de/contact/index.html`	`contact/index.html`
37131	4	5	-1	200	-1	8	GET	`http://www.sts.tu-harburg.de/service/index.html`	`../service/index.html`
46034	5	6	-1	200	-1	15	GET	`http://www.sts.tu-harburg.de/service/benutzerantrag.html`	`benutzerantrag.html`

Table 6: Example logfile of a test case t_1, the event sequence is a subset of the event sequence shown in table 7

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
60741	0	7	-1	200	-1	-1	GET	`http://www.sts.tu-harburg.de/`	-1
63556	7	8	-1	200	-1	3	GET	`http://www.sts.tu-harburg.de/contact/index.html`	`contact/index.html`
65256	8	9	-1	200	-1	8	GET	`http://www.sts.tu-harburg.de/service/index.html`	`../service/index.html`
68703	9	10	-1	200	-1	15	GET	`http://www.sts.tu-harburg.de/service/benutzerantrag.html`	`benutzerantrag.html`
70505	10	11	-1	200	-1	11	GET	`http://www.sts.tu-harburg.de/service/benutzbestim.html`	`benutzbestim.html`

Table 7: Example logfile of a test case t_2, the event sequence shown in table 6 is a subset of this event sequence

The *time*, *from* and *to* fields that are always logged by the WebQuilt proxy and therefore part of the logfile are not important for the comparison of two events.

For the similarity check algorithm four scenarios are important: Firstly, let us assume a test suite T_1 that contains the test case t_2 from table 7. With the WebQuilt proxy, test case t_1 from table 6 is generated. Since the event sequence of t_1 is a subsequence of the event sequence of t_2, like shown in figure 11 A, t_1 would not increase the interaction coverage and therefore not be added to T_1.

Secondly, when we assume a test suite T_2 that contains t_1, adding t_2 would increase the interaction coverage but also add redundancy because some of the events are already tested by t_1. Therefore t_2 would be added and t_1 would be removed. The same procedure (inverted) would be applied if the event sequence of t_2 would be a subsequence of the event sequence of t_1 like shown in figure 11 B.

Thirdly, let us assume a test suite T_3 containing a test case t_1 with a sequence of events. Now a test case t_2 is generated where the first $x, x \in \mathbb{N}$ events (head) are equal to the last x events (tail) of t_1 like shown in figure 11 C. To keep the redundancy minimal but increase the interaction coverage, t_1 and t_2 would be merged to a new test case t_3 that contains events e_1 to e_n of t_1 and events $e_x + 1$ to e_m of t_2. t_3 would be added to the test suite, replacing t_1.

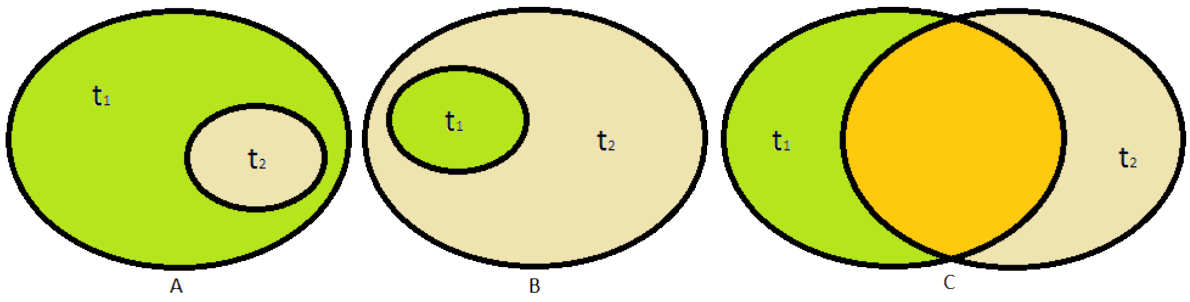


Figure 11: A test case t_2 where the event sequence is a complete subsequence of the event sequence of t_1 (A); a test case t_1 where the event sequence is a complete subsequence of the event sequence of t_2 (B); and two test cases t_1 and t_2 where some part of the event sequence of t_1 is equal to some part of the event sequence of t_2 (C)

In a fourth case where test suite T_4 contains t_1 and a part in the middle of the event sequence of t_1 is similar to a part in the middle of the event sequence of a new test cases t_2, t_2 is added to the test suite. This will increase redundancy because some events of t_2 are already tested with t_1. It is not possible to split up or shorten t_2, since this will break up possible dependencies between the events and this can cause t_2 to be useless.

A pseudecode version of the similarity-check algorithm that considers all cases is shown in listing 7.

Listing 7: Similarity check algorithm for test suite extension

```
public boolean addTestCaseToSuite(TestCase tNew) {
        for (all TestCases t in the TestSuite) {
                if (tNew is a subset of t
                        or tNew is equal to t) {
                                return false;
                        }
                if (t is a subset of tNew) {
                        remove t from TestSuite;
                        add tNew to TestSuite;
                        return true;
                }
                if (t.tail is equal to tNew.head
                        or tNew.tail is equal to t.head) {
                                merge t and tNew;
                                return true;
                        }
        }
        add tNew to TestSuite;
        return true;
}
```

3.3 Test Case Application - Regression Testing

The test cases that can be generated with the presented proxy approach can for example be applied in regression testing. Let us assume a web application P and an established test suite T. When the web application is developed further (e.g., new features are added or changed) it may happen that in this modified version P' existing functionality was changed unintentionally. In such a scenario regression testing attempts to use T to test P' [19].

3.3.1 Test Case Execution

The test cases that are generated by the proxy represent click sequences that the users did to interact with a web application. The click sequences are stored in logfiles. Every entry represents one event. Several tools exist that can execute click sequences automatically in web browsers and act as a remote control. These tools usually require the click sequence represented in some special input format. One of these tools is Selenium, which "is possibly the most widely-used open source solution" [20] for web application test automation. Selenium comes along with a web driver and "a collection of language specific bindings to drive a browser" [21]. It is for example possible to write a Java JUnit test to test a test case or a whole test suite generated by the proxy. Selenium can handle click sequences in different ways so that it is possible to use the proxy output directly. The logfiles contain the *href* parameter of the links the users clicked on. Selenium is able to use this parameter to find and click elements in a web page. The logfiles also contain name and value of form elements that the user filled. Selenium can find and fill form elements, too. The logfiles also contain information about the usage of the back button of the browser. Selenium provides methods to use this button.

Listing 8 shows a pseudocode example of a JUnit test that uses the Selenium WebDriver to execute test cases generated with the WebQuilt proxy. The driver will *open* a specified web browser at the given *startURL*. It iterates over all *entries in a logfile* (the test case). When the *user clicked a link*, the driver will search the *link* and *click* it. Before it must be checked if the user *used the back button*. If so, the driver will also use the back button. When the user *filled a form* the driver will iterate over *all fields of the form* and write the *values* into them. Afterwards it will *submit* the form.

Listing 8: Pseudocode of a Selenium test suite execution program

```
@Test
public void test() throws Exception {
        selenium.open(startURL);
        for (all entries in a logfile) {
                if (user clicked a link) {
                if (user used the back button
                        of the browser before) {
                        selenium.useBackButton();
                }
                selenium.click(link);
                assert(selenium.HTTPCode == HTTPCode);
        } else if (user filled a form) {
                for (all fields of the form) {
                        selenium.fillTheFieldWith(value);
                        }
                        selenium.click(submit);
                        assert(selenium.HTTPCode == HTTPCode);
        }
        }
}
```

Checking the result of the test case is done with assert statements. The HTTP response code that Selenium retrieves is always compared to the one expected (found in the log file). If an assertion fails, the whole test fails. The test also fails if selenium can not find an expected element like the link to click on. A fail indicates that the test case (the

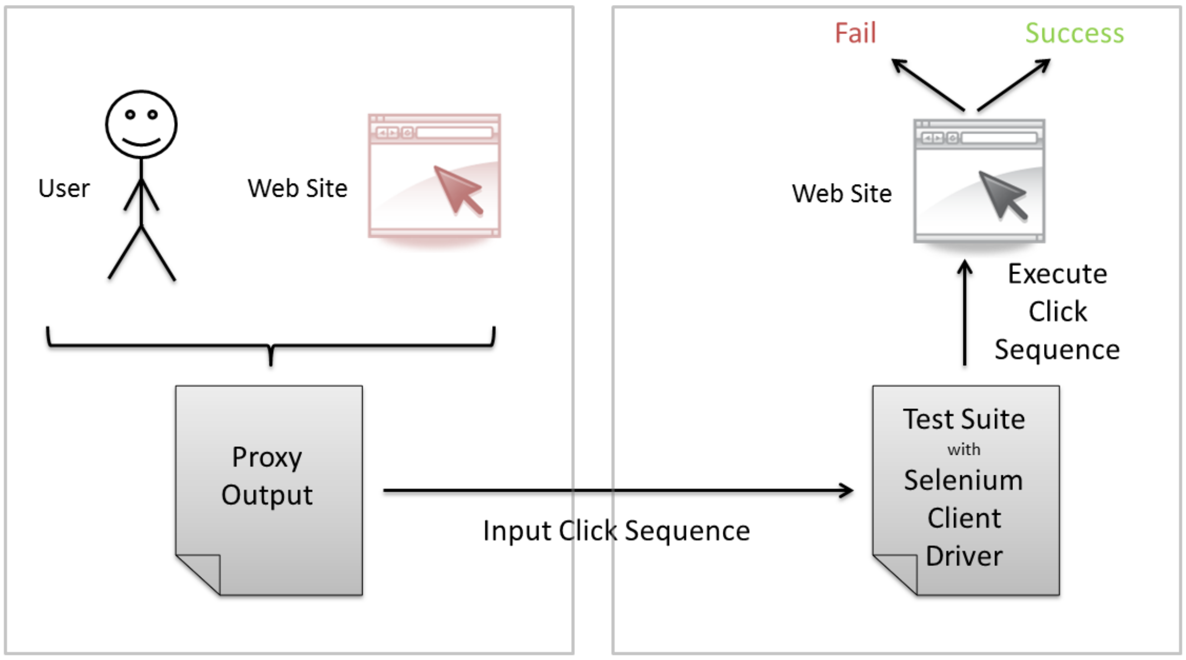


Figure 12: Regression testing; users use a released version of a web application and produce event sequences that are logged. Developers use the logged information to regression test the further developed web application

click sequence) could not be executed the same way as before. This can indicate a fault.

The algorithm in listing 8 can be used as shown in figure 12. Users use a released version of the web application and produce event sequences with their interactions that are logged. The developers of the web application in the meantime develop the application further and invent new features. They use the generated logfiles and run these test cases against the new version. If tests fail it might indicate unintended changes to existing code. But it might also indicate that test cases are not usable anymore in case functionality was removed. Therefore it is necessary that the developers check the meaning of the test results manually. New test cases for the features that are invented will be initially generated by the users after the new version is released. To test the new features with the proxy beforehand the developers can make use of the proxy to record their usage of these features.

4 Application to Real World Software

In order to measure the quality of the WebQuilt proxy we used an existing web application and a set of existing test cases. The application is a web management software for professional customer communication via eMail, fax, letter, SMS, social media etc. The web application makes use of many AJAX calls and other dynamic JavaScript code. For this thesis the focus is on a part of the application that uses less dynamic code and therefore can be tested with help of the proxy (refer to 2.1.4). Section 4.1 discusses the standardization and section 4.2 the reproduction of existing test cases. Section 4.3 shows how our approach contributed to testing the application.

4.1 Standardization of Test Cases

For this part of the web application a test suite S with 143 test cases t exist. This number does not include duplicate test cases and test cases that are not valid anymore due to software changes. These duplicates and invalid test cases were eliminated manually. The test cases t were generated manually by many different persons. In a standardized version every test case t consists of a list of events $e_1,\ldots,e_n$ and an expected result r but they need not have a uniform level of detail. For example one test case might have this event list

- e_1: click the “Reports” tab
- r: everything is displayed correctly

which is not very detailed and the expected result is open to interpretation: Only the person who generated the test case once might have known the meaning of “correctly” at this time.

For 113 of the existing test cases the list of events can be represented as pure click sequence or a form filling or a combination. For example a test case for a successful login to the application is

- e_1: (on the login page) enter existing username
- e_2: enter correct password
- e_3: click login
- r: login successful

and the test case for a failed login is

- e_1: (on the login page) enter existing username
- e_2: enter wrong password
- e_3: click login
- r: login denied

Such test cases only consist of events $e_1,\ldots,e_n$ that the users of the application do on the page. Therefore such test cases can be generated by the proxy and are the base for the proxy quality calculation.

There are other test cases that require some interaction with the back-end of the application, like

- e_1:(in a config file on the server) set workflow parameter “configure.clients = true”
- e_2: click the “Clients” tab
- r: clients are editable

Such test cases are not within the scope of the presented WebQuilt approach.

4.2 Reproduction of Test Cases

By trying to reproduce all of the 113 test cases, the quality q of the proxy is measured. The quality can be in general expressed with formula 3:

$$\text{Quality } q = \frac{\#\text{ reproduced test cases } t_r,\ t \subset t}{\#\text{ test cases } t,\ t \subset S} \tag{3}$$

In an experimental setup of the software, the click sequences of the 113 existing test cases were clicked manually. In this way 97 test cases could be generated by the proxy. The equivalent test case for a successful login is shown in table 8, the test case for a failed login is shown in table 9.

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
7058	0	1	-1	200	-1	-1	GET	`http://example.com/app/login`	-1
18197	1	2	-1	200	-1	-1	POST	`http://example.com/app/index` username=amempel password=asd4fg	-1

Table 8: Example logfile of the reproduction of a successful login

Time	From	To	Parent ID	Code	Frame ID	Link ID	Method	URL+Parameter	Link
7058	0	1	-1	200	-1	-1	GET	`http://example.com/app/login`	-1
18197	1	2	-1	200	-1	-1	POST	`http://example.com/app/login?error=authentication&username=amempel` username=amempel password=asasdf	-1

Table 9: Example logfile of the reproduction of a failed login

With 113 test cases given and 97 test cases regenerated the quality of the current version of the proxy is

$$q = \frac{97}{113} = 0,86 = 86\% \tag{4}$$

The 16 test cases that could not be regenerated are those click sequences where fail or success is indicated by an AJAX response, for example

- e_1: click the “Template” tab
- e_2: click the “File” tab
- e_3: click the “error.txt”

- e_4: click “delete”
- r: template can not be deleted

The proxy is in the current version not able to log the AJAX request and response, so the result of the test case can not be captured.

4.3 Contribution to Testing Process

Regenerating 86% of the test cases for this part of the web application is just a small contribution to the testing process that exists for the whole application. But it gives a foretaste of how testing will look like when the proxy is capable of dealing with highly dynamic pages. Automatic execution of automatically generated test cases will speed up the testing process extremely. A manual test showed that a developer or tester that knows the application very well and is responsible for testing needs about 30 seconds to execute the test cases of tables 8 and 9 in a sequence and verify the result. An automatic execution of both test cases with Selenium takes only 6 seconds on the same machine.

During the reproduction process all duplicate or invalid test cases were eliminated manually before executing the click sequences. Therefore the redundancy was minimized. An implementation of the similarity check algorithm (refer to section 3.2.2) would not have reduced redundancy more effective. Eliminating duplicates and invalid test cases in the first place resulted in 113 test cases that were clicked manually to end up with 97 reproduced rest cases. Using a similarity check algorithm means clicking all test cases, including duplicates, manually to end up with the same 97 reproduced test cases.

5 Future Work

In order to be able to track user events even on pages that use AJAX and JavaScript, the WebQuilt proxy is unsuitable. Since the proxy modifies the content the users see, it can never be guaranteed that the faults found are not generated by the proxy itself. The presented alternative addresses this problem. We would like to implement such a proxy that does not change the code but has similar HTML analysis and link-identification algorithms for logging. When implementing such a proxy a solution for the dynamic link difficulty needs to be found. Dynamic links are composed on the client-side, for example when the user clicks an element. A disadvantage of the presented alternative is that the proxy is not URL-based. The consequence is that the users need to specify the proxy in the web browser.

We would like to add some functionality to the proxy that enables the user to indicate the start and end of a sequence. This would enable easy and precise test case generation. In the current version of the proxy users needs to end the session in order to indicate the end of a sequence or start on the proxy main page to start a new session. A simple solution would be a link or a button, injected into the user interface by the proxy, that sends a request to the proxy to invalidate the session.

6 Conclusions

This thesis has presented an approach to track users interactions with a web site to automatically generate test cases for testing a web application.

It was shown that it is possible to track interactions with an URL-based proxy that intercepts HTTP communication. The proxy analyzes and manipulates HTML and JavaScript code on the way from server to client. Every link the proxy finds this way gets several parameters appended. The parameters are used to identify interactions and to redirect through the proxy again when a user clicks the link. On the way from client to server the proxy filters out the parameters and generates and saves log information. This way form filling as well as clicks on links or other elements in a web application to navigate through the application can be recorded. The users do not have to manipulate the browser or install any third party program.

By trying to modify and extend the WebQuilt proxy, new problems and limitations of this approach were covered and it was shown that this proxy is error-prone (refer to sections 2.1.4 to 2.1.6). An alternative is a proxy that modifies the content of the web page to a lesser extent.

It was also shown that one logfile that contains an event sequence represents a test case. Test cases are usually grouped in test suites. For the expansion of test suites we have given an algorithm for checking the similarity of test cases (refer to section 3.2.2). The algorithm only adds a new test case to a suite, if the interaction coverage would be increased. The test cases generated by the proxy can be applied in various testing methods and levels. One example application is regression testing. Developers can use test cases that were generated by users of the current version of an application to regression test a further developed version. In this way unintended changes could be spot. With the Selenium tool we illustrated how the generated test cases could be executed automatically (refer to section 3.3.1).

The presented proxy approach was also tested in real world applications. On web applications that do not contain dynamic elements and do not make use of AJAX the proxy works very well. It can record click sequences with all link combinations that are possible in order to navigate through the web application. The proxy was tested with a complex web application that uses JavaScript extensively. The test was restricted to a part of the application that can largely be tested with the proxy, with respect to the limitations. The whole application was tested manually in the past. From this procedure several test cases exist for the part that was tested with the proxy. We were possible to reproduce 86% of the existing test cases with the proxy (refer to section 4.2). Running the produced test cases automatically is likely to increase the effectivity compared to manually testing these cases many times over (refer to section 4.3).

References

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616, Hypertext Transfer Protocol – HTTP/1.1," 1999. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`.

[2] "Definition of Same Origin Policy," December 2011. `http://www.w3.org/Security/wiki/Same_Origin_Policy`.

[3] "Aspects of Web Application Testing," December 2011. `http://robdmoore.id.au/blog/2011/03/12/web-application-testing/`.

[4] W. Wang, S. Sampath, Y. Lei, and R. Kacker, "An Interaction-Based Test Sequence Generation Approach for Testing Web Applications," in *Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*, (Washington, DC, USA), pp. 209–218, IEEE Computer Society, 2008.

[5] J. I. Hong and J. A. Landay, "WebQuilt: a framework for capturing and visualizing the web experience.," in *Proceedings of the 10th International Conference on World Wide Web*, pp. 717–724, 2001.

[6] R. Atterer, M. Wnuk, and A. Schmidt, "Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction," in *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, (New York, NY, USA), pp. 203–212, ACM, 2006.

[7] "WebQuilt Readme File," November 2011. `http://dub.washington.edu:2007/projects/webquilt/download/RC1/WebQuiltProxy-v1.0RC1/README.html`.

[8] "Homepage of the Institute for Software Systems of the Hamburg University of Technology," February 2012. `http://www.sts.tu-harburg.de/`.

[9] "Homepage of the Hamburg University of Technology," February 2012. `http://www.tu-harburg.de/index_e.html`.

[10] "Example Website with GET Form," February 2012. `http://www.htmlcodetutorial.com/forms/_FORM_METHOD_GET.html`.

[11] "Example Website with POST Form," February 2012. `http://www.htmlcodetutorial.com/forms/_FORM_METHOD_POST.html`.

[12] "Telemediengesetz," 2010. `http://www.gesetze-im-internet.de/tmg/`.

[13] J. I. Hong, J. Heer, S. Waterson, and J. A. Landay, "WebQuilt: A proxy-based approach to remote web usability testing," *Information Systems*, vol. 19, no. 3, pp. 263–285, 2001.

[14] "Rhino API," November 2011. `http://grepcode.com/snapshot/repo1.maven.org/maven2/com.jolira/rhino/1.7.3.1/`.

[15] IEEE, “IEEE 610.12 Standard Glossary of Software Engineering Terminology,” 1990.

[16] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques.* John Wiley & Sons, 2005.

[17] F. Ricca and P. Tonella, “Analysis and testing of Web applications,” in *Proceedings of the 23rd International Conference on Software Engineering*, ICSE ’01, (Washington, DC, USA), pp. 25–34, IEEE Computer Society, 2001.

[18] G. Rothermel and M. J. Harrold, “Analyzing Regression Test Selection Techniques,” *IEEE Transactions on Software Engineering*, vol. 22, pp. 529–551, August 1996.

[19] G. Rothermel and M. J. Harrold, “Selecting tests and identifying test coverage requirements for modified software,” in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA ’94, (New York, NY, USA), pp. 169–184, ACM, 1994.

[20] “Introduction to Selenium,” January 2012. `http://seleniumhq.org/docs/01_introducing_selenium.html`.

[21] “Selenium Project Homepage,” January 2012. `http://seleniumhq.org`.

List of Figures

List of Tables

List of Listings