

Automated analysis of dynamic web services

Jonas Boustedt

`jbt@hig.se`

Masters Thesis in Computer Science
Uppsala University, Sweden

5th March 2002

Abstract

Web applications appear as mazes to a user. Using a web browser, the user explores each web page without seeing the structure of the entire service. For a software tester, it would be convenient to have a map, in form of a graph, describing the functional topology of the service. In that way, it would be possible to analyse the possible paths which can be navigated to discover redundancies and circularities for example. A web spider tool can automate the construction of such a graph. The spider can request a document from the application, find all references to other documents in it, and explore them recursively until all the references have been analysed. However, web services often produce dynamic responses which means that the content cannot be distinctly represented by its reference, i.e., the responses must be classified in a way that matches the users perception. The main problem is to find suitable criteria for this classification. This study describes how to make such a tool and it surveys ideas for how to create a classifying identifier for dynamic responses. The implemented spider was used to make experiments on selected web services, using different models for web node identification. The result is a proposal of suitable criteria for classification of dynamic responses, coming from web applications. These criteria are implemented in algorithms which use the parse structure and the set of internal references as the dominant terms of identification.

Keywords: Web Analysis, Graph Editor, Web Spider

Contents

1	Introduction	1
1.1	Problem definition	2
1.2	Expected result	3
1.3	Questions at issue	3
1.3.1	Web node equivalence	3
1.3.2	Accessing the web	3
1.3.3	User interaction	3
1.3.4	Graph definition	4
1.3.5	Visualization of graphs	4
1.4	Method	4
1.5	Delimitation	4
2	Related work	5
3	Background	7
3.1	World Wide Web	7
3.1.1	Hypertext Transfer Protocol	8
3.1.2	Cookies	8
3.1.3	Adding security using HTTPS	8
3.1.4	Hypertext Mark-up Language	8
3.1.5	Web browsers	9
3.1.6	Frames	9
3.1.7	User input by forms	10
3.1.8	Client side scripts	10
3.2	Web applications	11
3.2.1	Public services	11
3.2.2	Intranets	11
3.3	Web servers	12
3.4	Dynamic server side responses	12
3.4.1	Server-parsed scripts	12
3.4.2	Interfacing against external programs	12
3.5	Web Spiders	13
3.5.1	Search engines	13
3.5.2	Intelligent agents	13
3.5.3	Web Robot exclusion	13

4	Realisation	15
4.1	Web node equivalence	15
4.1.1	Edges	15
4.1.2	Titles	16
4.1.3	Name/value-pairs in forms	16
4.1.4	Partial parse trees	16
4.1.5	Repeated patterns in the parse prefix	17
4.1.6	The equivalence algorithm, all put together	18
4.2	Accessing the web	19
4.2.1	The web spider algorithm	19
4.2.2	Implementation	22
4.3	User interaction	23
4.3.1	Forms	23
4.3.2	Methods for passing information	24
4.3.3	Secure HTTP and authentication	24
4.4	Graph definition	25
4.4.1	GML	25
4.5	Visualizing graphs	25
4.5.1	VGJ	26
5	The implemented tool	28
5.1	The Web Graph Tool	28
5.1.1	The file menu	29
5.1.2	The sites menu	29
5.1.3	The options menu	29
5.2	Presentation of graphs	30
5.3	Results from running the tool	31
5.3.1	Analysing black box sites	32
5.3.2	Analysing white box sites	34
5.4	Delimitation	39
5.4.1	Support for HTML version 3.2	39
5.4.2	Client side scripts - Dynamic HTML	39
5.4.3	User input into forms	39
5.4.4	Cookies unsupported	40
5.4.5	Multithreading	40
6	Discussion	42
6.1	The expected versus the achieved result	42
6.1.1	Web node equivalence	43
6.2	Previous research and results	43
6.3	Usefulness	43
6.4	Acknowledgements	44
7	Conclusions	45
7.1	Answers to the questions at issue	45
7.1.1	Web node equivalence	45
7.1.2	Accessing the web	46
7.1.3	User interaction	46
7.1.4	Graph definition	46
7.1.5	Visualization of graphs	46

7.2 Further research	46
References	48
Appendix	51
A	51

List of Figures

3.1	An example of HTML code.	9
3.2	The rendered HTML document.	9
3.3	The HTML code for an input form.	10
3.4	How a form is visualized in a web browser.	10
4.1	A partial parse tree.	16
4.2	A truncated partial parse tree.	17
4.3	The patternTruncator algorithm.	18
4.4	The complete equivalence algorithm.	19
4.5	The spider algorithm.	20
4.6	Subprograms used by the spider algorithm.	20
4.7	A graphical view of the spider algorithm in action.	21
4.8	An example of GML.	26
5.1	The main GUI for the tool.	28
5.2	Several options can be selected.	30
5.3	A generated graph with a poor layout.	31
5.4	Graphs can be edited manually.	31
5.5	A graph showing a news application.	33
5.6	The graph of a commercial site.	33
5.7	The graph editor can show a graph under construction.	34
5.8	Forms are rendered by the tool.	35
5.9	An incomplete graph for the <i>Othello</i> application.	36
5.10	The graph for an incomplete <i>Human vs. Computer</i> game.	36
5.11	The graph for an incomplete <i>Computer vs. Computer</i> game.	36
5.12	A complete graph for the <i>Human vs. Computer</i> game.	37
5.13	A complete graph for the <i>Othello</i> application.	37
5.14	The spider tool showing statistics.	38
5.15	A graph showing three web services.	39
5.16	A self-referencing, growing document.	40
A.1	The class diagram for the application.	51

Chapter 1

Introduction

A large number of the modern information systems today, are accessible through the World Wide Web. These systems are designed as distributed client/server applications which means that several users, i.e., clients, can connect to a service through a web server system. The clients use a piece of software, a browser, to exchange information between the user and the application.

Traditionally, the way to design web sites was to construct them as large hierarchic menus from where the user could passively select the desired information. Now they are often highly dynamic systems where the actions of the users have impact on the system's behaviour and contents. As an example, we can access a bank service, log in, and then perform actions, e.g., pay the bills, transfer currency between accounts or even buy stock shares. Clearly, these applications are very advanced and delicate. The services must not be error prone and must be secure, hence there is a need for software testing that is well-structured and well-documented. Manual software testing is complicated, sometimes tedious and always expensive, which has led to a need for tools that support automated testing.

Today, there are tools for automated testing of web services. They mainly focus on testing functionality, performance and scalability. It is usual to design them as macro recorders with the ability to replay the macros, simulating a huge number of simultaneous clients. Two examples of such benchmarking tools are *LoadRunner* and *WinRunner* from *Mercury* [1].

If the tool supports a scripting language, the test engineers can write script programs, fine-tune and parameterise the test. This requires a good understanding of the structure and functionality of the actual web service. Such knowledge is gained either from a well-defined specification or from empirical experience. Much of this knowledge could be described by a graph. The graph would describe the functional topology of the service, thus showing how to navigate through it, revealing every sub service (link) and its attributes, such as input values and types. It could be used as a specification, from which tests cases could be designed and it could be useful when comparing updates of services. This approach could be useful for test engineers if the graphs could be automatically generated by a software tool.

Web spiders are tools used to explore the web in order to collect information. The principle is to find links in the retrieved documents and follow them recursively. Along the way, information is collected and stored. Traditionally

the contents in the web were static files. However, modern web services generate both static and dynamic responses. This means that the content, received as a response to a specific URI¹ request, often is generated from a database and thus it is variable. The response can be affected by the users behaviour, i.e., explicit user input (parameters), implicit input (navigation history) or by the state of the service, e.g., stocks and bank accounts. The server can also have a random behaviour or the responses can depend on the actions of other clients. Obviously, when designing a spider tool that manages this type of behaviour, a number of new problems arise.

1.1 Problem definition

Given a web application, which criteria can be used to classify its output² in order to automatically produce a graph that represents the application's functional topology?

The nodes of the graph should represent the application's "states" from a user's perspective. A common definition of a state is that a system has reached a certain state when the combination of all involved flow control parameters has a certain unique value. As a consequence of this definition, there will be an immediate state transition when any of the parameters is changed. From one state there is a well-defined, finite set of possible state transitions to other states. However, there is no way to access a web application's internal parameters from the client side; the web application acts as a "black box". Instead, we have to define the state of the system by either the input to the system, its output or both. The input to the application is the requested URI and additional parameters. This could be an adequate definition for static applications, where each URI is mapped to a specific static resource, typically an HTML document. However, when the application has a dynamic behaviour, there is no guarantee that a specific URI is mapped to a certain output. Furthermore, the input is often concealed to the user. From the user's perspective, the dynamic output from the application represents its states. The output from the application normally consist of structured HTML code which has a logical meaning to the user. In fact, the response is normally the only entity that keeps the state since the application itself is stateless. The application is only active for a short moment when it is using the user input and possibly some auxiliary data³ to determine its state and produce the output. In addition, the responses define the actions that can be performed in the current state through input parameters and a set of hyper-links (URIs). Each link represents a potential state transition and the links are represented by edges in the graph. Hence, it is natural to define the application's states by classifying the output coming from the application into equivalence classes that represent the nodes in the graph.

Naturally, using the proposed definition of states, a program that explores a web application must examine all possible responses coming from the web server. This is only possible if the responses can be identified, i.e., we must define some

¹URI is the generic set of all references to resources. Both URL (Universe Resource Locator) and URN (Universe Resource Name) are URIs

²A web application's output is the web server response to an HTTP request, a requested URI.

³A database can also keep state information.

criteria that can be used to determine if two responses are equivalent or not, i.e., if they should be represented by the same node in the graph. The fact that the responses can have dynamic contents complicates the problem, since the criteria for defining equivalence must allow “equal” responses to have variable content to some extent. A user would probably allow some variations in the output and still regard them as belonging to the same logical state. As an example, we can consider a page displaying the balance of a bank account. This leads to the main question, the problem of *web node equivalence*.

1.2 Expected result

This study will suggest suitable criteria and algorithms for web node equivalence. In order to validate and enhance the criteria a tool must be developed. It is expected that the tool can produce graphs that can be used as an aid for web service testers. However, it will merely be a prototype for experiments and demonstrations and it should be useful for the further work on how the generated graphs could be used in software testing.

1.3 Questions at issue

The problem can be subdivided into five groups of questions.

1.3.1 Web node equivalence

Dealing with dynamic contents, how should the spider classify the server responses into nodes, in order to build the graph? Solving the problem of *web node equivalence* is the key to success, i.e., if we know how to compare responses from web servers, we can make the classification. Which criteria are possible candidates for web node equivalence and identification and which are the most suitable ones?

1.3.2 Accessing the web

The tool must be able to communicate with remote web servers through the Internet. The search space, the set of resources on the Internet, is huge and must therefore be defined for the web application at issue since it could point to external resources. Without a well-defined search space the analysis could end up with a graph of the entire Internet. Also, the stop criteria must be well-defined. Otherwise the analysis would never end. Which are the underlying principles of a web spider and how should this web spider be designed in order to fulfil the specific requirements for this type of application?

1.3.3 User interaction

The data input from clients affect the response. How should the data be dealt with, i.e., how should the web spider know which data to insert, in order to automate the analysis? Some interaction from the operator is probably required. Could the same data be repeated? Could it be reused and manipulated? How

should the input forms be presented to the user, and how should the entered values be captured?

1.3.4 Graph definition

How should the graph be defined and abstracted? A well-structured, possibly extensible, language is needed to store information of the graph structure and other data attributes collected in the traverse. Are there some standards?

1.3.5 Visualization of graphs

One of the main ideas is to be able to visualize the functional topology of a web application. The graph should be editable and storable. It should also be possible to send it to a printer. The problem is not trivial since the renderer must be able to organise the nodes and edges into a well-structured view. Thus a rendering tool must be developed, or are there such tools available?

1.4 Method

A spider engine must be developed in order to investigate methods for response classification and user interaction. That requires knowledge about the concepts of client/server applications on the Internet and some practical skills in programming and usage of APIs (not rocket science but yet time consuming).

Theoretical ideas, for how classification of responses can be done, will be generated and evaluated using the tool. Experiments will verify or reject these ideas. Some ideas are collected from other researchers and some are new. The method is in other words innovative, experimental and iterative.

The structure of the response is more important than its literal content. There are several aspects to consider such as: parse trees, links, titles and contents. However it is hard to make a conclusion a priori of which criteria will be successful, thus an iterative process combining new ideas with experiments and test will be used.

1.5 Delimitation

This study focuses on the analysis of web services, creation and visualization of functional graphs. The aim is not to produce complete functional specifications that could be used for test case generation. However, the results, if successful, could probably be used in further research to achieve that goal.

Chapter 2

Related work

Jonsson, L of and Magnenat discuss the possibility to construct test schedules for a web service if a state graph of the service is provided [2, 3]. Such a state (transition) diagram would be comprehensive and useful for a test suite developer. This fact urge to create such a graph automatically. However, how should it be constructed? Using the principles of a web spider would make it possible to explore a web service and collect information along the road by issuing requests and receiving responses. Links, titles, input parameter names and values, script detection, etc. are examples of such information.

Filippo Ricca and Paulo Tonella have developed similar tools for analysis and testing of web applications and their results were presented in April at *TACAS 2001* [4]. The tool *ReWeb* downloads and analyses the web pages and then creates an UML model of the web service. *TestWeb* generates and executes a set of test cases based on the model from *ReWeb*. Both tools are semiautomatic. However, the problem of web node equivalence is not discussed in their paper. It is stated that all versions of the same page are merged. That is exactly the issue here: how do we know that a page is a potential variant of itself?

WWWPal [5] is an example of another type of tool. It analyses the contents of a web server and produces graphs showing all the references to documents. It can also create synthesised web pages containing a skeletal representation of the generated graph. However, dynamic behaviour of the web server is not considered. This is a tool suitable for analysing information server systems where links are bound to specific documents, i.e., the link identifies the document.

Yet another example of an automated analysis of web sites using a spider combined with having the result represented in a graph is *WebCutter* [6]. It is a tool for tailored information searching. It combines the *search paradigm* with the *browse paradigm* by presenting the dynamic search results from a site in a graph. The generated graph is interactive, thus letting a user browse the graph by clicking the nodes.

One of the non-trivial problems in this study is the web node identification, i.e., how the contents (server responses) can be mapped into nodes and how they can be compared to each other (equivalence). Web node equivalence is discussed by Luca de Alfaro in a paper on *Model Checking the World Wide Web* [7]. Alfaro talks about the following aspects of web node comparison and equivalence:

- **Textual comparison.**

The actual text contents in two pages (s, t) can be compared and equivalence would be based on identical text in the pages.

$$s = t \Leftrightarrow \text{text}(s) = \text{text}(t)$$

Comment¹: this approach does not consider equivalence on a higher level of abstraction. Any single character can cause a mismatch. An HTTP request is required.

- **Link comparison.**

If two pages contain the same links they can be considered as potentially equal.

$$s = t \Leftrightarrow \text{links}(s) = \text{links}(t)$$

Comment: having this definition could lead to a loss of data but the structure of the graph will be preserved in a minimal form. An HTTP request is required.

- **Original URL comparison.**

Pages are identified by their referred URLs (a_0, b_0).

$$s = t \Leftrightarrow a_0 = b_0$$

Comment: this is very efficient since it is sufficient to visit each node only once. It will work in a static context, but is useless for dynamic services. An HTTP request is not required.

- **Final URL comparison.**

Pages are identified by their final identifiers - after redirections and completions into fully qualified URLs (a_n, b_k).

$$s = t \Leftrightarrow a_n = b_k$$

Comment: if two responses originates from different locations, they are not likely to be defined as equivalent even if the contents are identical. An HTTP request is required.

- **Redirection sequence comparison.**

The sequence of redirection URLs obtained until a page response occurs can be used to identify the page. If two such URL sets have a common member, the pages are equivalent. This criterion is more robust than the other URL comparisons.

$$s = t \Leftrightarrow \{a_0, \dots, a_n\} \cap \{b_0, \dots, b_k\} \neq \emptyset$$

Comment: an HTTP request is required.

Finally it is stated that a mix of the above definitions is likely to be the optimal equivalence definition. The mix could be tailored for the web domain in question. However, redirection sequence comparison is used in de Alfaro's implementation, "for the sake of simplicity".

¹The comments here are the authors comments on de Alfaro's proposed criteria.

Chapter 3

Background

The following sections explain some concepts that the reader should be familiar with in order to comprehend the problems discussed in the following chapters. The initiated reader could skip these sections.

3.1 World Wide Web

The World Wide Web (WWW) [8] is a community of servers that offer multimedia information and interactivity through the Internet. These servers are normally accessed via client-side software tools called browsers. The communication between the client and the server is performed using the Hypertext Transfer Protocol (HTTP), and the information is normally structured in documents by the Hypertext Mark-up Language (HTML); however any type of data can be transferred, e.g., pictures. The HTML language can describe structure and presentation of information and can define hyperlinks, i.e., references to other sources of information, typically other HTML documents. A hyperlink is structured as a Universe Resource Identifier (URI) [9].

The original use of the web was to let the web browser passively display the contents of a given URI. The only interaction with the server side was to send new URI requests (by typing in a new URI or by clicking on a hyperlink), receive the response and display it. In this type of static applications, the web server is acting alone on the server side. In order to update information on the server side, the static files (documents) had to be manually edited.

However, according to James E. Goldman the Internet is transforming from a government funded entity designed for research and education to a privately funded entity that offers commercial services [10]. Along with this increased commercial use new needs come for improved user interaction. A web shop must be able to show products and prices that are up to date. Moreover, a customer should be able to buy the products in a simple manner. This requires some user-input mechanisms for selection and text fields for the credit card number. The user interactions can also affect the state of the service, i.e., bank transactions should be able to be traced and must be stored in a database. Hence, HTML and HTTP have been enhanced during the years to meet these new demands. Most of all modern web services or web applications today have this dynamic behaviour.

3.1.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol [11] is a high level, platform independent, client/server protocol, residing in the application layer on top of TCP. It is a stateless protocol, hence it is quite different comparing with other TCP/IP applications. A client connects to a server, and makes a request, gets a response and then closes the connection. After this point, there is no state change anywhere from the protocols perspective. The next connection will have no other relation with the former. State awareness must be solved at a higher level, e.g., by using hidden parameters in the response or by using *cookies*.

3.1.2 Cookies

The use of cookies is a way for web applications to store persistent information on the client side. This information can be fetched (polled) by the web service in order to retrieve the state of the particular user. Using cookies is optional in web browsers since the browser actually writes the cookie on the client disk. Web services must handle users that do not accept cookies. According to Hal Berghel, the use of cookies have drawbacks since they affect both the clients storage and integrity [12].

3.1.3 Adding security using HTTPS

In order to provide security (communications privacy) for HTTP connections, a security layer can be applied. Secure Sockets Layer (SSL) by Netscape is a protocol that establishes a secure connection between computers [13]. An URI indicates the use of a SSL server by using the “https:” protocol specifier. The SSL layer resides on top of HTTP.

3.1.4 Hypertext Mark-up Language

HTML [14] is a fairly well-formed language that structures multimedia contents. It also suggests to the browser how the data should be presented to the end user. This is controlled by “tags” embedded in the textual contents in a hierarchic manner (tree structure). Some tags are more or less mandatory, e.g., the “root” tag <HTML>, and others (<HEAD>, <TITLE> and <BODY>). The tags that express structure (have contents) are all balanced, i.e., they consist of a start tag and an end tag (<TABLE>...</TABLE>). Tags without contents are not balanced, e.g., the line break tag
. Tags can contain embedded attributes. The anchor tag <A . . .> for example, defines a hyperlink (URI) using the attribute HREF. The contents between the start and end tags represent the hyperlink when the page is visualized by the browser. See Figure 3.1 for an example of HTML-code and Figure 3.2 to see how it is rendered by an HTML-browser.

Links can be expected in several other tags. To find them, the page must be parsed following the syntax of HTML. Unfortunately, browser developers have augmented the language standard in different directions. In addition the parsers in the browsers are very forgiving, i.e., the HTML documents will often be parsed and rendered even if the documents breaks the syntactical rules. This

```

<HTML>
  <HEAD>
    <TITLE>A table in HTML</TITLE>
  </HEAD>
  <BODY>
    <HR><H2>A table in HTML</H2><HR>
    <TABLE>
      <TR><TD>A11</TD><TD>A12</TD><TD>A13</TD></TR>
      <TR><TD>A21</TD><TD>A22</TD><TD>A23</TD></TR>
      <TR><TD>A31</TD><TD>A32</TD><TD>A33</TD></TR>
    </TABLE>
    This is a <A HREF="http://www.x.y/z.html">hyperlink</A>.<HR>
  </BODY>
</HTML>

```

Figure 3.1: An example of HTML code.

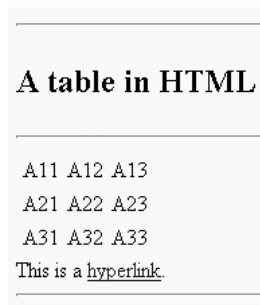


Figure 3.2: The rendered HTML document.

has caused a sloppiness¹ when writing HTML code. These circumstances lead to major problems when parsing a document, since the parser has to ignore many special cases and syntactic errors.

3.1.5 Web browsers

A web browser is an application that can fetch multimedia data from the Internet via HTTP or from a local file system. It can interpret and visualize HTML-documents, but can also render pictures of various file formats, play music, etc. Some of the most well-known browsers are Mosaic, Netscape and Explorer.

3.1.6 Frames

The users view of web pages are often structured by frames. Typically, the browser presents a number of important links at the left side or in the top or bottom. If the user selects one of those links, the browser will show the response to that action in another section. The browser is actually divided into a number of rectangular windows. Thus, we have a split vision of the service. This is controlled by the <FRAMESET> tag in HTML.

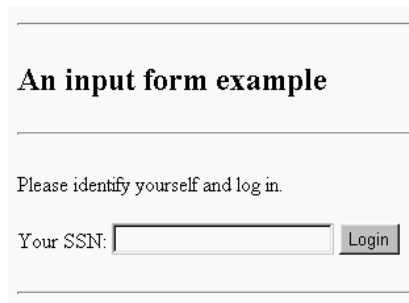
¹In the absence of HTML compilers, *trial and error* is the prevailing method for writing HTML code.

3.1.7 User input by forms

The technique for passing information to a server side application is very simple. The `<FORM>` tag implies that this section of the HTML document contains name/value pairs. A name/value pair is a name (a key) and a string value which can be set by the user. In order to send the information to the application, a submit-button must be activated. The forms specifies an action that should be performed on submit and it is typically an URI. In addition, a parameter passing method is specified. It could be either POST or GET. The web browser has to visualize this form in some way in order to prompt the user. The parameters are associated with several types of GUI-components (widgets). These can be text fields, check boxes, selection boxes, text areas, buttons or radio buttons. However, there are also “hidden” parameters, which are not visualized in the browser. These parameters can carry state information since they are sent back to the server application in the next HTTP request. See Figure 3.3 for an example of HTML-form code and Figure 3.4 to see how it is visualized.

```
<HTML>
<HEAD><TITLE>An input form example</TITLE></HEAD>
<BODY>
  <HR><H2>An input form example</H2><HR>
  <P>Please identify yourself and log in.
  <FORM METHOD=POST ACTION="http://www.secret.to/login.cgi">
    <INPUT TYPE=HIDDEN NAME="retries" VALUE="4">
    <P>Your SSN: <INPUT TYPE=TEXT NAME="ssn">
    <INPUT TYPE=SUBMIT VALUE="Login">
  </FORM>
  <HR>
</BODY>
</HTML>
```

Figure 3.3: The HTML code for an input form.



An input form example

Please identify yourself and log in.

Your SSN:

Figure 3.4: How a form is visualized in a web browser.

3.1.8 Client side scripts

On the client side, browsers can interpret scripts that are embedded in the downloaded pages. JavaScript, JScript and VBScript are examples of such

script languages. Some of the most common tasks for the scripts are checking user input, controlling “banners”, performing reactions on user actions, making nicer graphical appearance, and so on. HTML-documents containing scripts are often referred to as “Dynamic HTML”. In addition, Java applets can be executed in a JVM² on the client. The applet’s GUI³ appears embedded in the browser, but the execution is restricted to the JVM and they have no access to the browser’s API. Client side scripts can also handle URIs and issue associated actions, i.e., request the URI or it could even start new browser instances. URIs are treated as strings in the script languages, and can be built by function calls and substrings at run time. This fact has a dramatic impact on the spider application in this work. It is not sufficient to parse the contents; the scripts must actually be executed to ensure that all URIs are found.

3.2 Web applications

Web applications, or services, are composed of a number of sub-services that put together can interact with a user through a web browser. WWW itself is a service layer on the Internet, which offers the possibility to access other services. The typical server side architecture of a modern web service is a web server connected to a set of software applications, residing in an application server. The web server can activate these applications when it is resolving a request from a client. The request typically consists of a URI and input data entered by the user. The web server communicates with the application using some protocol, e.g., CGI⁴, and the response from the application is sent back to the client either directly or through the web server. The response is typically an HTML document and the contents is likely to vary depending on user input and the state of the application; the state of some database for example.

3.2.1 Public services

Anyone equipped with web browser software and an Internet connection can access any public web service. Some of them require a login procedure, with a prompt for username and password, to identify the users. This could be bank services, web shops, libraries, public service applications and naturally the classical “homepages” of organisations. Even some private persons offer interactive services but most people don’t have a permanent TCP/IP address nor do they have a server. The ISP⁵ provides a dynamic address for them that is fixed during the current session only.

3.2.2 Intranets

An Intranet is a private network based on TCP/IP protocols hidden from the Internet behind a firewall. Thus, only the members of the organisation can access the network. Others can access it from outside only by authorisation. An intranet’s web site acts as a normal web site within the private network. This

²Java Virtual Machine

³Graphical User Interface

⁴Common Gateway Interface

⁵Internet Service Provider

allows the company to present classified information and to offer internal web services to its employees. Goldman gives some examples of intranet applications: discussion forums, on-line polls, organisational directories, company policies and procedures and company forms [10]. Other examples are company news, time management and address books.

3.3 Web servers

The web server software distributes requests and responses between the client and the server. The information on the server is auxiliary to the server software and is normally a set of files (static or executable). Some well-known server software are: Apache (GNU), AERN, Microsoft IIS, NCSA and Netscape [15].

3.4 Dynamic server side responses

In order to generate dynamic server side responses, some code must be executed on the server side (host). It can be done by either letting the web server interpret code written in some script language, or by delegating the task to the operating system.

3.4.1 Server-parsed scripts

Web servers can run several script languages. The SSI⁶ directives were the first tool to make the pages act dynamically. They are put in HTML comment lines. However, merely directives do not make an expressive scripting language and since the demand for dynamic behaviour in the services on the web increased, so did the demand for better tools. ASP⁷ was the first server side scripting language, followed by many others, e.g., PHP⁸ and JSP⁹. The idea behind all of the embedded server-parsed languages is based upon embedding the script code within the HTML document. The web server parses the document and interprets or executes the script code (in an execution thread), replacing the code with its results, and then delivers the resulting web page to the browser. Often, the file name extensions of the URI reveals the type of scripting language: *.asp*, *.jsp*, *.php3*.

3.4.2 Interfacing against external programs

The “Common Gateway Interface” (CGI) is a specification that allows communication between a client and an executable program or script on a remote operating system through the WWW, using HTTP. It is a standard for information servers, such as web servers, to interface with external gateway programs. Gateways are programs, which receive requests for information and return the requested information, which can be generated dynamically in real time. The server and the gateway run as separate processes on the host operating system in contrast to server-parsed scripts. When a CGI program is executed, either

⁶Server Side Includes

⁷Active Server Pages (Microsoft - VB-script)

⁸Personal Home Pages (open-source by Apache Software Foundation)

⁹Java Server Pages (JavaScript executed on the server)

it can get its input data from the standard input stream or it can be provided as command line parameters. CGI also specifies environment variables that can be accessed by both the server and the program during execution. The most common input data are name/value pairs originating from a FORM request in an HTML page. The output from the program is transferred back to the client via the server and is typically a web page. The gateway can get information from various sources, e.g., a database, and then transform the data into a format that can be read by the client, i.e., if the client is a web browser, the data should be HTML encoded. The major drawback with CGI programs is the sometimes poor execution time. The operating system has to start up and kill a process for each request. One of the advantages is the possibility to interface against previously developed applications. There are many gateway programs that collect information from external systems. They can easily be modified to support CGI. Normally the URI ends with a “.cgi”. The CGI programs can be written in any programming language that produce executables, e.g., C, C++, Perl, Python, TCL, UNIX shell scripting languages (Bourne, C, Korn, ...) [15].

3.5 Web Spiders

A web spider is any application that traverses the web automatically. These applications are also known as web wanderers, web robots or web crawlers. Their intention is to find information in the web. It could be information in some specific topic or it could be a higher level search as indexing the web in to databases [16].

3.5.1 Search engines

Web search engines are web applications, which use huge databases to keep an index of information that can be found in the web. The index is generated by web robots that continuously search the web for information. The gathered information is indexed and is then put into a database. A client can connect to the search engine and type in some keywords into a search form. The database is searched for these keywords and the result is typically a list of URIs. Each URI refers to a document containing the keywords. However, since the indexing process is very time consuming, it is impossible to keep the index updated at all times. Thus, it is quite common that the links are broken or the content is changed etc. The major search engines, e.g., AltaVista, Yahoo, Evreka, etc., are only used to find static web pages in the web.

3.5.2 Intelligent agents

Some web spider applications, *intelligent agents* [17] can detect changes in selected web pages (static or dynamic). It could be weather reports, stock prices, etc. As a user, you subscribe to some specific URIs and the spider will automatically detect updates and inform you, e.g., by e-mail.

3.5.3 Web Robot exclusion

The Internet is crowded by all kind of robots, looking for information. The performance of a web server under “attack” from a robot can be considerably

reduced. Both the web masters and the web robot launchers have a mutual interest of keeping the Internet as fast as possible. Therefore, some steps have been taken to solve the problem. The “Web Robots Exclusion Protocol” is a very simple way of defining which parts of a site that should not be visited by robots. A plain text file `http://.../robots.txt` contains such information. Another way of doing it is by using a *HTML META tag (Robots Meta Tag)* in the actual page, inhibiting indexing and linking. Note that these methods require co-operation from the robot creators since this is just information to the robot and it could be ignored.

Chapter 4

Realisation

This chapter describes how the problems were solved, which criteria were chosen for web node equivalence and how they were implemented into algorithms.

4.1 Web node equivalence

The unsolved problem in the described spider algorithm is how to identify nodes and how to compare them. Using a highly pedantic method will cause a never-ending loop in the algorithm since “new” nodes will keep showing up. A sloppier method may lead to missed nodes. Dealing with dynamic services implies that some contents in a specific node will be dynamic and the rest remain constant. This is what has to be considered when formulating the equivalence criteria. Luca de Alfaro suggests five different methods that can be combined to identify nodes [7]. Some of those ideas are used but some new thoughts are implemented and evaluated.

4.1.1 Edges

Two nodes, that have the same set of node references (edges) are potentially equal. Two nodes that have different node sets cannot possibly be equivalent due to graph properties. Using this model, a web site is considered to be a structure with no other contents (attributes) than the edges themselves. If a new response does not have the same ordered set of edges as other nodes, it could not be identified with any of the existing nodes. Obviously, this is not a sufficient demand, but it is required, in order to maintain the graph properties. Actually, keeping the references in a set will make the definition of equivalence weak. The order of the references and repeated occurrences could be regarded. Thus, a stronger differentiation is achieved if the set is ordered and it allows duplicates. A simple implementation would be to put the references in a linear string. However, it is possible that some applications produce internal links¹ dynamically; the order of links could be changed or they could be repeated. This would lead to new nodes in the graph where an unification were expected. In this case an unordered set of links is better to use. An ordered set is chosen for the prototype.

¹Links within the application itself are edges in the graph.

4.1.2 Titles

According to W3C, one of the mandatory tags in the HTML language is the title tag [18]. If the title tag is missing, the web browser may not be able to render and display the document. The content in the title block will be displayed in the browsers title bar. Search engines also use it to describe a documents content. Thus, the title tag is expected in a well-formed HTML document. Using an objective perspective it would also be predictable that the title is unique to the actual page. Reversing this argument gives us a strong reason to believe that the title for a specific node is constant in time. The title criterion is optionally added to the set of edges criterion. The effect will be verified by experiments.

4.1.3 Name/value-pairs in forms

If there is one or more forms in a document, each input component is associated with a parameter name. When the form is submitted, the names are combined with the entered values into name/value pairs. The set of parameter names from these pairs is a potential identifier for an HTML document. Therefore, the identification algorithm will use the ordered set of parameter names.

4.1.4 Partial parse trees

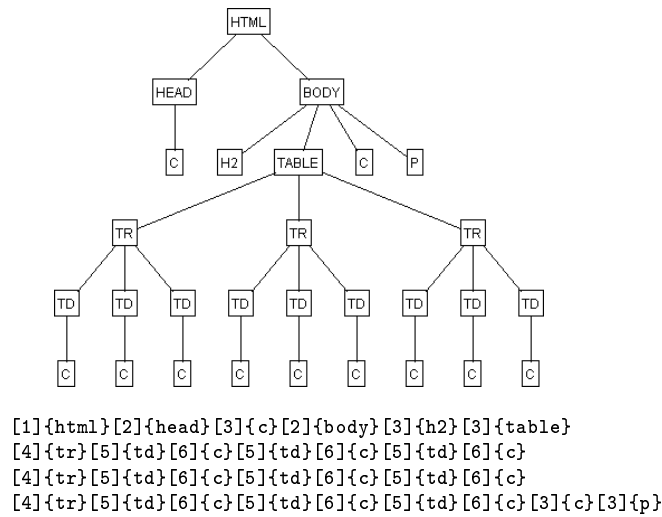


Figure 4.1: A partial parse tree.

The HTML document in Figure 3.1 is represented here as a partial parse tree. A prefix string is produced by a *depth first* traversal of the tree. Each node is represented by its depth and type, but only the non-leaf nodes are represented. The generated prefix string is displayed beneath the parse tree.

An HTML document can be described as a parse tree. Is it possible to find a method to delimit the parse tree in such a way that dynamic contents are excluded? Equivalence would then occur if it is possible to abstract two parse trees into a common, non-trivial, tree. As a first step in this direction, the parse

tree could be cut off at some fixed level. The root element and its children are kept and the textual content is ignored. However all HTML documents are isomorphic until level 2, so this approach would be far too naive. The implemented and verified method is to build a signature, or prefix, that describes the parse tree, i.e., every node with contents is tagged with its depth and type. Thus the parse tree is flattened by a traversing the tree by *depth first*, and the signature is stored as a linear string. When determining equivalence, the signatures are matched by string comparison. The leaves, representing the textual contents, are represented as an abstract *content* in the signature, allowing textual variations in the documents, hence it describes a partial parse tree. However, if the structure of the document is changed, the equivalence algorithm will not match.

As shown in Figure 4.1, the partial parse tree for the HTML document in Figure 3.1 can be described as a string. Brackets surround the level of a node in the parse tree and curly braces mark the node type. The element {c} symbolises textual contents.

4.1.5 Repeated patterns in the parse prefix

Some applications show dynamic listings of items, e.g., a list of bank accounts or a list of news titles. When this kind of applications are analysed, the number of items in those lists could vary during the analysis. The examination itself may affect the state of the service and thus affect the responses. In a news publishing system, every inset of a new article would affect the index of articles. The new version of the index page would now be regarded as a new node, using the partial parse tree algorithm described in section 4.1.4. Nevertheless, the amount of repeated items should not stop the unification of nodes. Either the parse prefix option must be turned off or a more sophisticated algorithm must be used to avoid a graph with a group of nodes that should be the same.

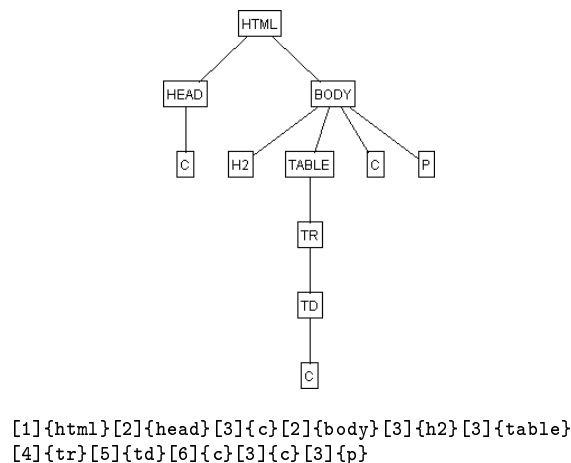


Figure 4.2: A truncated partial parse tree.

This figure shows the results from the *patternTruncator* algorithm, when applied to the prefix string from Figure 4.1. The resulting prefix string and its corresponding parse tree are truncated. The HTML table has been reduced to having only one row and one column.

A suggestion is to find adjacent repeated patterns in the parse tree prefix and replace such patterns with one instance. This implies that all repeated tables, rows, columns, lists and paragraphs for example, would be abstracted into a basic structure, containing all unique substructures, and the varying responses would be unified. Naturally, this is only the case when the other equivalence criteria are fulfilled. Figure 4.2 shows an example of this idea. Starting with the HTML document in Figure 3.1 and then creating the partial parse tree prefix shown in Figure 4.1, the prefix is truncated into the prefix shown in the figure.

The algorithm, `patternTruncator`, takes the parse tree prefix string and finds its sub-trees using the algorithm `divideTree`. The sub-trees are examined and if two adjacent sub-trees are identical, the second is removed, etc. The remaining sub-trees are then recursively processed by `patternTruncator`. During the end of the recursion, the sub-trees are concatenated into one truncated parse tree prefix.

```

patternTruncator(prefixString, level) : string
  resultString := head(prefixString, level)           to first occurrence of level
  prefixString := tail(prefixString, level)           from first occurrence of level
  subtreeList := divideTree(prefixString, level)      find all sub-trees of level
  truncatedList := findUniquePrefixes(subtreeList)   remove adjacent repetitions
  if truncatedList is empty then
    resultString := concat(resultString, prefixString) base case
  else
    foreach subtreeString in truncatedList do         recursive case
      resultString := concat(resultString, patternTruncator(subtreeString, level+1))
    end foreach
  endif
  return resultString

divideTree(prefixString, level) : stringList
  subtreeList :=  $\emptyset$ 
  while( head(prefixString) = level) do
    subtreeString := firstSubtree(prefixString, level)
    add(subtreeList, subtreeString)
    prefixString := remove(prefixString, subtreeString)
  endwhile
  return subtreeList

```

Figure 4.3: The `patternTruncator` algorithm.

4.1.6 The equivalence algorithm, all put together

The equivalence algorithm is a composition of the suggested variants of equivalence definitions. Some are optional since their importance for different kinds of services may vary. The fundamental criterion for the comparison is the set of edges found in the responses and therefore it is not optional. Then follows the optional comparison of the titles found in the document. The third optional criterion is the parse tree prefixes, which may have been truncated by the *patternTruncator algorithm*. As a last optional means of equivalence, the parameter names found in the possible forms are used.

```

equivalenceAlgorithm(a, b) : boolean
  boolean e := ( edges(a) = edges(b) )           Set of edges, not optional
  boolean t := ( title(a) = title(b) ) or not optionT Titles, optional
  boolean p := ( prefix(a) = prefix(b) ) or not optionP Parse trees, optional
  boolean f := ( fparam(a) = fparam(b) ) or not optionF Form parameters, optional
  return ( e and t and p and f )

```

Figure 4.4: The complete equivalence algorithm.

The parse tree prefix is optionally truncated by the *patternTruncator algorithm* when it is constructed from the parse tree.

4.2 Accessing the web

Having solved the problem of how nodes should be compared, it is now possible to use the definition of equivalence in an algorithm that generates a graph.

4.2.1 The web spider algorithm

The algorithms used to explore web services follows the same principle as proposed independently by Ricca and Tonella [4], with some differences.

The suggested algorithm uses two sets:

- R , a set of node objects, representing web server responses. Each node has a set of edges (URIs).
- U , a set of edge objects, representing URIs. Each edge has a set of attributes, e.g., a node reference, the URI, form parameters, etc.

The main principle is to iterate as long as U is not empty. Therefore the start edge (URI) is put into U and then the iteration can begin.

In each iterative step, an edge element (URI), u , is picked out from U . The URI from u is issued to a subprogram that fetches a web server response (an HTML document) and puts it in a new node object². The node object, r , is analysed in several ways. First the HTML document is parsed and converted into a parse tree. Then a parse tree prefix is produced and optionally truncated to prepare for the equivalence algorithm. If the HTML document contains an HTML form, the form is optionally completed with data. Then the parse tree is traversed in order to find all URIs, referring to other web resources. Each found URI is put in an edge object which is put in the node object's edge set. After the analysis of the document, the node object keeps only the most important attributes found in the document, such as: the parse tree prefix, the set of edges, the title and the form parameter names.

Now, it must be determined if the node object is a member of R or if it is a unique node. This is done by the set operation *member* that iterates through R and performs calls to the *equivalenceAlgorithm*. If, r , is already a member, there must exist an equivalent node, x , in R . Now, the node object in R that refers to r , must have its reference to r updated so that it refers to x instead. However if r is unique, it is put in R and all of its internal edges (URIs) are put in U . Thus the iteration will continue until the last edge (URI) in U has a

²When a node object is created, it actually contains the responded HTML document.

response node that, either is a member of R or has no URIs in it. The algorithm could be described in pseudo code (see Figure 4.5 and Figure 4.6).

```

 $U, T \subseteq \{\text{all edges (URIs)}\}$        $start, edge \in U$ 
 $R \subseteq \{\text{all nodes (HTTP Responses)}\}$    $node \in R$ 

Spider.run(start) : "graph"
   $R := \emptyset$ 
   $T := \emptyset$ 
   $U := \{start\}$ 

  while not empty(U) do
    edge := removeElement(U)
    node := getResponse(edge)
    bindEdgeToNode(edge, node)
    if member(node,R) is true then
      updateEdge(node,R)
    else
      T := findEdges(node)
      bindNodeToEdges(node, T)
      foreach edge in T do
        insertElement(edge,U)
      endforeach
      insertElement(node,R)
    endif
  endwhile
  return start

```

Figure 4.5: The spider algorithm.

```

member(node, R) : boolean
  foreach n in R do
    if equivalenceAlgorithm(node, n) is true then
      return true
    endif
  endforeach
  return false

updateEdge(node, R)
  e1 := parent(node)
  e2 := findEquivalent(node)
  if not e1 is null and not e2 is null then
    removeEdge(e1, node)
    insertEdge(e1, e2)
    updateReferences(valueOption, e2, node)
  endif

```

Figure 4.6: Subprograms used by the spider algorithm.

The subprogram *equivalenceAlgorithm*(*node*, *node*) in the algorithm returns true if the node elements (server responses) are “equivalent”. As described in section 4.4, the function composes several criteria, some of which are optional, to determine equivalence. The optional criteria can be used in order to fine-tune the behaviour of the algorithm, and thus adapt the analysis to the specific web application.

However, when reading the pseudo code in Figure 4.5 we must keep in mind

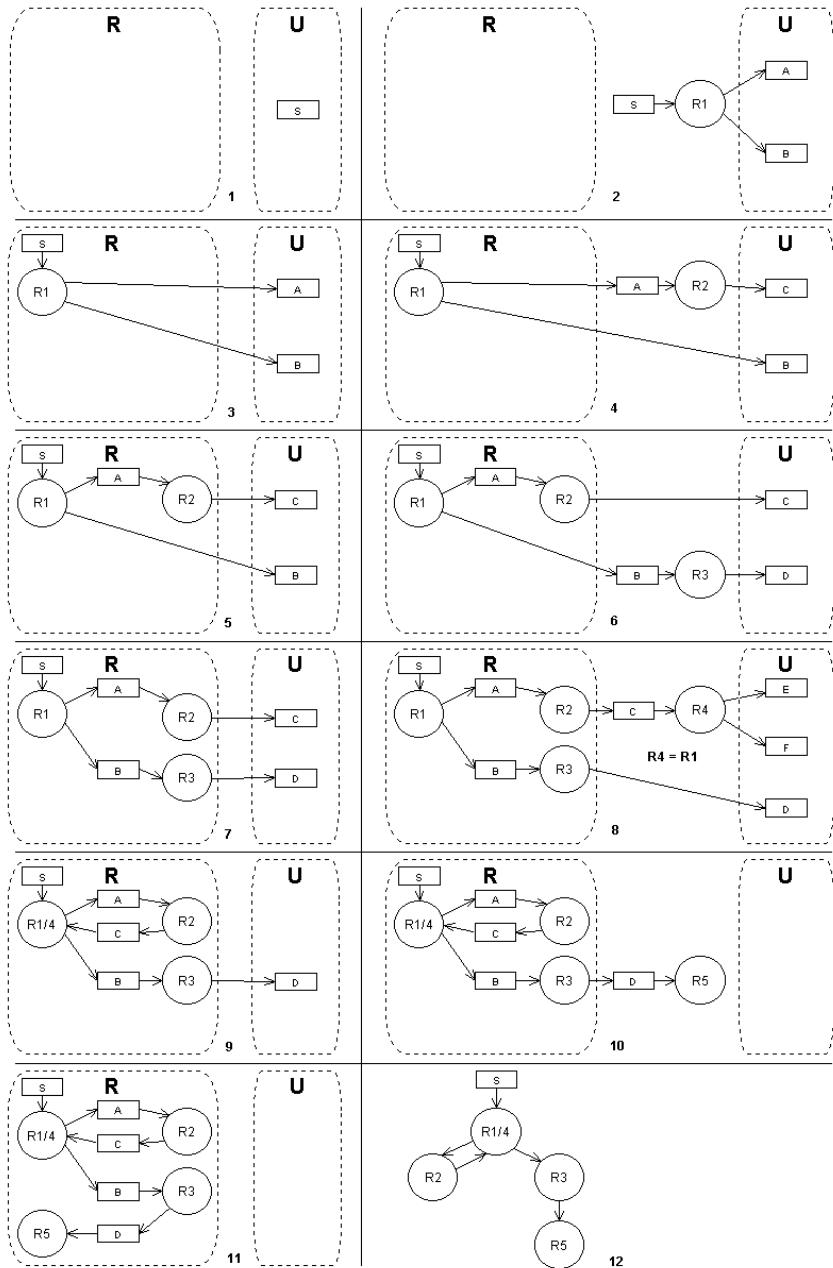


Figure 4.7: A graphical view of the spider algorithm in action.

that both the nodes and the edges are objects, having attributes and operations. An edge keeps a reference to a node and a node has a set of edge references. The *start* edge is returned from the algorithm and it is representing the entire graph, since the node and edge objects are traversable through operations.

In Figure 4.7, an example of an analysis is illustrated. The figures show how

the URIs and responses are related to each other, and how the graph gradually is constructed.

4.2.2 Implementation

In order to investigate methods for page identification, a spider engine was implemented using the Java³ JDK 2.0 platform. This choice was based on the high-level support for graphical user interfaces and networking facilities. The `javax.swing.text.html` package supports HTML 3.2 parsing and rendering. The parser structures the components of an HTML document into an abstract document model.

The HTML document model has some drawbacks when it is used in the web spider, since it is primarily intended to be used for constructing, editing and rendering HTML documents, but not for parsing them after their construction. In summary, the parser converts the HTML code into an abstract tree model containing various objects. When the tree is parsed during the analysis, some objects have to be accessed by method calls in order to get all of their attributes. Some of the objects has unique method names. In these cases the object reference must be type casted to the objects explicit type. This causes a problem, as the explicit type of a chosen object is unknown due to the abstract structure of the model.

In detail, the document is a container of elements and an element is something which has a name, a list of attributes, and a list of child elements. The element attributes are available by function calls. However, the type “Element” is a java interface, a pure declaration of a set of functions without definition (implementation). Any class⁴ can implement the interface and thereby share its type. This mechanism allows the document to consist of any objects that implements the interface “Element”. This is one aspect of polymorphism⁵. This is very nice as the document structure holds independently of the implementing classes. However, when the HTML parser builds a document, it uses rules that are not very well-documented. Sometimes it uses the default implementation of the Element interface, the “AbstractElement”. In other cases it creates very specific objects, i.e., the graphical components used in HTML forms: buttons, text fields, text areas and radio buttons. In these cases, the specific object type must be identified in order to properly access its attributes, such as entered user input. In addition, it is not obvious which HTML tags are associated with the elements in the document. As for an example, the `<TITLE>` tag cannot be found as an element in the document. On the contrary, it is an accessible “property” of the document. Therefore, it is hard to find out which are the underlying objects and this makes the analysis of a document a bit cumbersome. Several experiments were carried out to gain knowledge of the behaviour of the parser and the various objects it produces. This knowledge had to be implemented in the code that uses these Java classes in order to make appropriate type conversions and feasible exception handling, since an exception could be “thrown” if a

³Java is a trademark of Sun Microsystems Inc.

⁴A class is an encapsulation of data and code that defines an abstract datatype. The class can be instansiated into objects of this type.

⁵One definition of polymorphism is when references (function calls) to a type are bound to the implementing object in runtime. Neither the compiler nor the programmer knows the explicit type of the target object.

type cast was not successful.

The ability to render the document was the crucial argument for this choice of technology since our program must be able to present forms to an operator in order to fetch input parameters that are sent to the web service. The idea is to show the form in a window on the screen, letting the user enter information and then extract the parameters from the widget components in the document model.

In summary, the chosen Java classes makes it very easy to visualize HTML documents which makes it possible to fetch user input. They also provides a built-in HTML parser, working in the background, which converts HTML code into a document. The code that analyses HTML documents heavily depends on the (undocumented) behaviour of the parser. Future changes in the parsers implemented behaviour could affect the spider. On the other hand, it would have been a big effort to build a parser from scratch.

The Java class *URL* specifies the desired target URI and has a method *openConnection* which establishes the connection to the server. The class *URLConnection* abstracts the connection and it lets us issue operations like *getInputStream()* and *getOutputStream()*. Through these streams, data can be exchanged between the application and the server. An input stream reader can be assigned to an object of the class *HTMLDocument* which then gets the response from the server and parses the contents. The tree structure of the document object can then be traversed in order to analyse its contents on a higher level. One of the results of this analysis is the set of URIs pointing to other resources. The responses and URIs are abstracted by the user defined Java classes *Response* and *Link*.

4.3 User interaction

The ideal web service analysis tool is totally automatic. Provided with a start URI, options and constraints, the tool should start the process, do the analysis and then present the results as a graph. However, reality is too complex and thus the practical solution is to make the analysis semiautomatic.

4.3.1 Forms

When a form is detected in the analysed HTML document, its input parameters should be assigned some values before they are sent to the server side by issuing the action URI (see section 3.1.7). It is most likely that the values affect the state and behaviour of the server. Is it possible to automate this process? From where should these values be fetched? For input controls with an enumerated set of values, e.g., the check box and the radio button, these values could be combined into a finite number of states. Each combination could be sent to the server and its various responses would be analysed as usual. However, the form can have input controls that allow the user to type in arbitrary text strings, e.g., text fields and text areas. In these cases, the number of combinations would be almost infinite, and the process would never stop. This leaves us with two possibilities. Either the form values are fetched from some kind of database or they are entered by a human operator, ad hoc. However, if the values are stored in a database, how can they be known a priori? It would require

thorough preparations before any web application could be analysed. Thus, in this implementation, the input values are entered by the user in “runtime” or they are optionally left blank.

One of the main reasons to use the Java Swing API, when the spider was developed, was its ability to render HTML documents; including forms. When a document containing frames are received, a window running in a separate execution thread visualizes it. All input controls are modelled into standard Java Swing controls. The user can enter values into the controls and then press a submit button. At that point, the document tree is scanned and the entered values are collected and stored in a set of name/value pairs. When the action URI is finally issued, the name/value pairs are passed to the server. The user can optionally press the *enter data again* button and can then enter a new set of values into the same form. This can be done until all interesting combinations of data have been entered and each combination has been sent to the server.

4.3.2 Methods for passing information

The name/value pairs in a form are sent to the server in two ways, either using the GET or the POST method. The differences between these methods have impact on the implementation. Using the GET method, the parameter pairs constitutes the trailing part of the action URI, separated from it by a “?”. If the GET method would be used in Figure 3.4 and 3.3, the URI would be `http://www.secret.to/login.cgi?retries=4&ssn=991231-0126`. This implies that the values must follow the general rules for how URIs can be formed, since some characters are forbidden in URIs. Hence, the data values must be URI-encoded before they can appear in the URI. In addition, the total length of the URI must not exceed the stipulated maximum length of some hundreds of bytes. Thus, the program must augment the action URI with the parameters if the form method is GET.

The POST method is completely different. When the program is requesting the action URI and after the HTTP connection is established, the parameter values are streamed to the server. Using this method, there are no limitations of the length or the contents of the data. Nevertheless, the name/value pairs are still formatted as a list of pairs, each pair glued together by a “=”, and the pairs separated by an “&”. The data should not be URI-encoded by the program.

4.3.3 Secure HTTP and authentication

Some sites are password protected by the web server (not by the service). As a part of the HTTP protocol, the server can require authentication. This is implemented by the Java API and the only requirement is to implement a Java interface. A suitable implementation is by displaying an authentication window on the screen, letting an operator enter the user name and password. The drawback is that it is tedious and it requires the operator to know all possible logins. As a solution, the pairs of user names and passwords are saved together with the URI as a search key. Whenever the server requests authentication, the set of logins are searched and if found, the login is used automatically in the background. If the login would fail, a manual window is displayed.

The HTTPS protocol, described in section 3.1.3, is used by many web services. The Java API did not support this protocol when the tool was implemented. However, Sun Microsystems provided a reference implementation of a HTTPS package that was easily downloaded and plugged in. HTTPS will be included in the Java 2 Platform API, version 1.4.

4.4 Graph definition

The graphs should be defined and stored by a structured, flexible and well-defined representation. XML⁶ and XSL⁷ were first considered to be a good choice. XGMML⁸ [19] is an XML application for describing graphs, based on GML⁹ [20], a standard language specialised for graph definition in file format. However, it turned out that GML is used by several graph-drawing systems, including the one chosen for the spider tool. This fact motivated the choice of GML.

4.4.1 GML

GML, the *Graph Modeling Language*, is a portable file format for graphs. GML has been used in several graph drawing and analysing systems, including Graphlet [21], LEDA [22], GraVis [23] and VGJ [20]. A graph is defined as a block consisting of node definitions followed by edge definitions. Each node has the mandatory attribute *id* and the edges has *source* and *target*. In addition, any user-defined attributes can be added. These attributes may be plain name/value pairs or they can be composite objects, represented by a block of attributes, etc.

The possibility to create user-defined attributes is used by the web spider application. The user-defined attribute *data* is a composite attribute. It contains user input, the name/value-pairs from forms. Furthermore the original, absolute and received URIs are stored together with the response time (see Figure 4.8). This information should be available in the visualization of the graph, by clicking on the edge.

4.5 Visualizing graphs

There are several existing graph drawing tools that could be used together by the spider tool, e.g., AGD [24], daVinci [25], GEM [26], Graphlet [21], GraVis [23], Graphviz [27], Interactive Graph Drawing [28], LEDA [22], LINK [29], VGJ [30] and VCG [31]. Most of these tools are implemented using the programming language C++ because of its performance benefits. Some of the tools offer an API to interface against from external applications. This is precisely what is needed in the spider tool.

It would be very convenient if the graph drawing tool was developed using Java, since it would save lots of effort and time if the code could be accessed directly from the spider tool. VGJ is a visual graph editor, written in Java. Hence, combining the spider tool with VGJ leads to a tool entirely built on the

⁶eXtensible Markup Language

⁷eXtensible Stylesheet Language

⁸eXtensible Graph Markup and Modeling Language

⁹Graph Modeling Language

```

graph
[
  node
  [
    id 0
    label "Welcome"
    data
    [
      Form_1_input_1 "hidden : status"
      Form_1_input_2 "select : service"
    ]
  ]
  node
  [
    id 1
    label "Bye"
    data
    [
      Form_1_input_1 "hidden : status"
      Form_1_input_2 "hidden : previous"
    ]
  ]
  edge
  [
    source 0
    target 1
    data
    [
      OriginalURL "service.cgi"
      AbsoluteURL "http://192.168.1.4/cgi-bin/service.cgi"
      ReceivedURL "http://192.168.1.4/cgi-bin/service.cgi"
      Method "Post"
      Params "status=reset&service=quit"
      Accesstime "60"
    ]
  ]
]
]

```

Figure 4.8: An example of GML.

Java platform. One of the benefits of having all code written in Java is that the tool would be platform independent, i.e., executable from any operating system, having an JVM.

4.5.1 VGJ

VGJ, Visualizing Graphs with Java, is a tool for graph drawing and graph layout [30]. Graphs can be input into it in two ways: with a textual description, a GML file (see section 4.4.1), or by directly using the graph editor. The graphs can be output to a printer, or a file in postscript format.

The software package is used as an important component in the spider tool. Fortunately, the source code for it is available and free to use and is distributed under *the GNU General Public License* [32]. This made it possible to make some necessary modifications of its API in order to fit in properly. The GUI has been stripped from irrelevant features and it is now possible to create graphs from an external application by method calls. This makes it possible to draw the graph

for the web service during the analysis (in runtime). This graph “animation” shows how far the spider algorithm has progressed at a given point of time.

The user chooses the layout style of the graphs from a menu. The various layout algorithms are adapted to specific types of graphs. However, only one of the original layout algorithms remains in the modified version: *the tree algorithm*.

Tree algorithm

The tree algorithm implementation is referred to as *the Walker algorithm* or *tree layout algorithm* [33]. Trees are drawn so that:

- Nodes at same level lie on a straight line.
- Parents are centred over their children.
- There is vertical symmetry.
- Isomorphic sub trees are drawn identically.

Given the above properties and a minimum horizontal spacing, the tree has the minimum possible width. The algorithm does adjust for different sized nodes. If the graph is not a tree, a depth-first search will be used to identify a spanning tree, which will be used for layout. Before the algorithm is run, a node that will act as the root must be selected, if the graph contains cycles [30].

Chapter 5

The implemented tool

The main result of this study is the algorithm used for node identification and its implementation into a web spider prototype. Secondary results are generated from the experiments using the spider. This chapter describes the spider and the results of the experiments.

5.1 The Web Graph Tool

When the spider is started the main GUI appears on the screen (see Figure 5.1). In order to analyse a service or a site a start URI must be entered. By default, the constraining domain will be based on the path to the resource. Services may be distributed on several hosts and these can be specified in the “show sites” window.

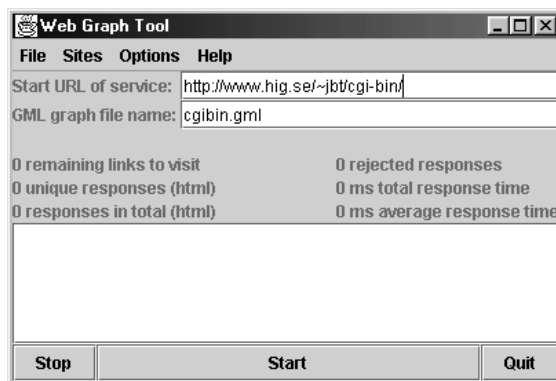


Figure 5.1: The main GUI for the tool.

It is possible to choose among some variations of the comparison algorithm and there are some other options regarding the user input. The produced GML-graph is stored in a plain text file. Using a Java-package called VGJ [30] the graphs can be rendered and presented when the analysis is finished. This package has been slightly modified in order to be able to show the growing graph during the analysis. The graph is drawn using a hierarchic tree-algorithm, which

puts the start node on top and the children are placed recursively downwards. The graph can be edited by the user in terms of moving nodes etc. All nodes and edges have attributes, which can be viewed by double clicking on the object. Postscript output to a printer or to a file is possible. VGJ (standard version) is a stand-alone package and can be used separately from the spider to view and edit the graphs at any time.

5.1.1 The file menu

From the file menu the current settings and results can be saved (save) or previous results can be reloaded and re-run without having to type in URIs, domains and options (open). Finally GML-files (graphs) can be visualized (show a graph).

5.1.2 The sites menu

There are two possible site windows; one for the allowed sites and one for prohibited sites or links. The idea is to specify the paths that the spider may or may not visit. The default accepting path is the longest possible path that can be extracted from the start URI, e.g., the start URI `http://www.x.y/z/q.html` would give the default accepting path `http://www.x.y/z/`. Any number of paths can be added. This is important since a web service is often distributed over a number of servers (hosts).

Detected links to hosts outside the specified sites should optionally be presented in runtime and it should be possible to insert them into the set accepted sites. This option is planned for, but still remains to be implemented.

5.1.3 The options menu

There are a set of options used to fine-tune the way the spider should operate. In some cases, a web site has special features that have to be considered. The most important options make it possible to decide variations of the equivalence algorithm. There is also a possibility to choose whether user input in forms should be prompted by rendering the HTML-documents containing forms, or if it should be done automatically by using empty input data. See Figure 5.2.

The options are as follows:

- **Depth first.** Should the spider follow new links first in a depth first manner?
- **Form input.** Should documents containing forms be rendered, allowing user input?
- **Scan client scripts for URIs.** The URI scanning algorithm for client scripts is not perfect. However, it finds most of the static URIs in the script code.
- **Use form parameter names in comparison.** Should the equivalence algorithm use the set of input parameter names?
- **Node labels.** Should the nodes in the graph be labelled with the document titles?

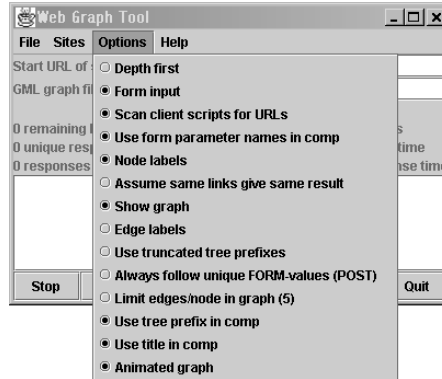


Figure 5.2: Several options can be selected.

- **Assume same links give same result.** Assumed static links will be visited once only. This also complies with dynamic links which have the same set of associated parameters and values as a previous request to the same URI.
- **Show graph.** The graph will be displayed automatically after the analysis.
- **Edge labels.** Edges will be labelled with the URI.
- **Use truncated tree prefixes.** Truncate the parse tree prefixes before they are used by the equivalence algorithm.
- **Always follow unique FORM-values.** New values in the name/value-pairs will force new requests.
- **Limit edges/node in graph.** No more than five edges will be drawn towards any node in the graph. This can be useful when a large graph with many back references is displayed.
- **Use tree prefix in comparison.** The prefix that identifies a response will be used to identify nodes by the unifying algorithm.
- **Use title in comparison.** The document title will be part of the unifying algorithm.
- **Animated graph.** A graph window will show the graph building up during analysis.

5.2 Presentation of graphs

The graphs are visualized by a call to the modified VGJ package (see section 4.5.1). This can be done either by an explicit selection of a GML-file, or automatically at the end of an analysis. Optionally, a graph window can display the construction of a graph during the analysis. In this mode the operator can watch the impact of various data input in the HTML-forms.

Normally, a web service resembles a tree. The tree layout algorithm in VGJ is well-suited for these cases, since it tries to draw the graph as a tree. However, this is not always the case. In fact, some sites have a structure more reminding of a web. The tree layout algorithm will not be suitable in those cases and the generated graphs have to be edited after the analysis to make them more comprehensive. The graph in Figure 5.3 represents a worst case scenario since it consists of eight nodes, each referring to every other node in the graph. Unfortunately, the edges are put on top of each other, thus concealing their sources and targets. The graph was manually edited in the VGJ window, into the graph shown in Figure 5.4.

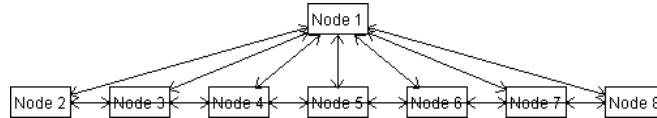


Figure 5.3: A generated graph with a poor layout.

This graph is captured from the tool, showing the results from analysing a web application consisting of eight nodes, all referring to each other. The tree algorithm is not optional in this case since it will generate only two levels in the tree.

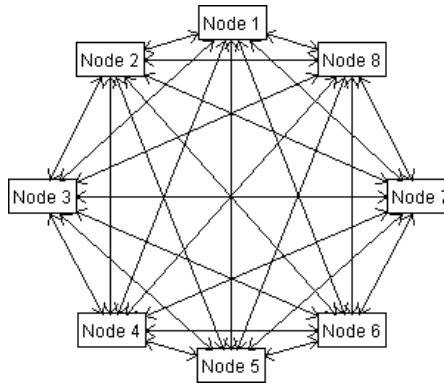


Figure 5.4: Graphs can be edited manually.

If the *tree layout algorithm* is unsuccessful, the graph can be edited manually, simply by dragging the nodes with the mouse.

5.3 Results from running the tool

During the development of the spider, several analyses of web services have been performed. The web services can be divided into two types. First we have the black box sites; these are analysed “as is”, with no knowledge of the implementation details. Web sites like university home pages, commercial sites and company intranets have been selected for analysis and graph production. However, since we did not have access to the full specification of these services,

we do not know to what extent these graphs actually reflect the complete structure of the analysed web services. The completeness can only be verified by a manual exploration of the site.

The other type can be referred as “white box” sites, i.e., applications with a well-known behaviour and structure. Some purely dynamic example applications (using CGI) have been developed and they were used in order to verify some special conditions where it was important to know exactly how the server application worked inside.

5.3.1 Analysing black box sites

Intranet service for news

One of the services was an intranet application for internal company news. There were hundreds of different documents in the service. Many of the documents contained cross-referencing links to other documents and almost all documents contained explicit back links to higher level documents. Since the spider algorithm does not identify responses by its referring URI, each occurring URI must be visited. This makes the algorithm inefficient, having an exponential growth of HTTP requests. Finally 383 nodes were identified and the graph was slowly displayed. However, a screenshot of the rendered graph shows that it is impossible to take in and understand the structure of such a vast service, see Figure 5.5. The analysis of the news service took over one hour to accomplish and over 4000 HTTP requests were made. At the time of this experiment, the truncated parse tree prefix algorithm was not yet developed. Having used that option, it is possible that the analysis would have ended up with a reduced number of nodes in the graph.

After this and other time consuming analyses, a special option (“assume same links give same result”) was added to the tool. With this option enabled, the spider will make repeated requests to a specific URI only if it has a unique set of associated input parameters and values. It means that if an input form is entered with exactly the same data twice, it will only cause one request to the server. This also automatically leads to the fact that there will only be one request to a specific static URI, regardless of the amount of references to it from within the service. This is obvious since there are no parameters associated to static URIs (static files on the server side).

Commercial sites

Most commercial sites use embedded client side scripts (dynamic HTML) to create more appealing and functional user interfaces. Typical tasks for the scripts are: control of animations, invocation of new browser instances, verification of user input before sending it to the server and other operations that involves URIs. Unfortunately, the spider cannot detect all URIs in these scripts and hence, when the spider is applied to this kind of web applications, some parts of the functionality will be missing in the graph representation. This malfunction is verified through a manual exploration of the web service, comparing the functions in it with the graph. A typical effect of client scripts is that they often start a new browser window to display an auxiliary service or site. The result from an analysis of a commercial site is presented in Figure 5.6.

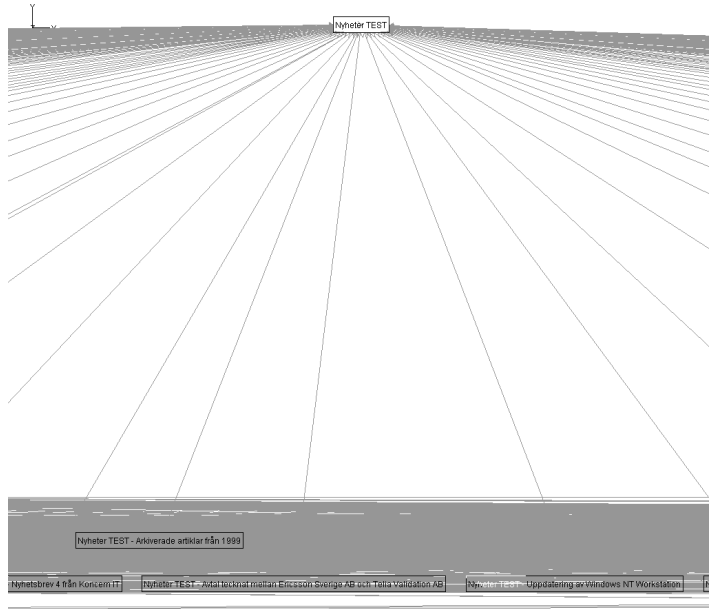


Figure 5.5: A graph showing a news application.

Sometimes the service is too vast to visualize. The shaded parts of the picture are the edges. The program works, but the graph editor is overloaded. The graph contains 383 nodes with thousands of edges.

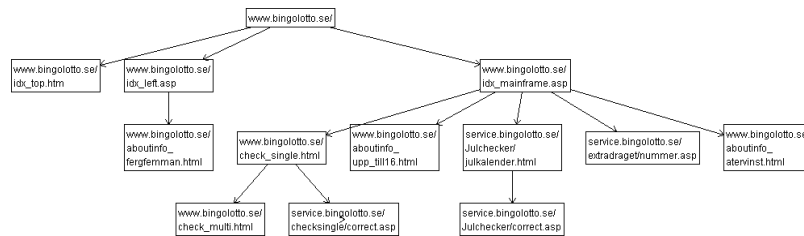


Figure 5.6: The graph of a commercial site.

This graph is the result from an analysis of the site *Bingolotto*. The major functions are detected and correctly represented in the graph. However, some of the functions are missing since client side scripts initiate them; typically the instantiation of new browser windows.

University homepages

Some homepages of staff members at the university were analysed. It was noticed that client side scripts were unusual in this context, so the analysis worked out fine. However, some server side applications were occasionally found and form input had to be entered. This is a mix of static and dynamic responses and typically, the static documents are ordered in a hierarchic manner. This can be viewed in Figure 5.7 showing the graph editor during an analysis.

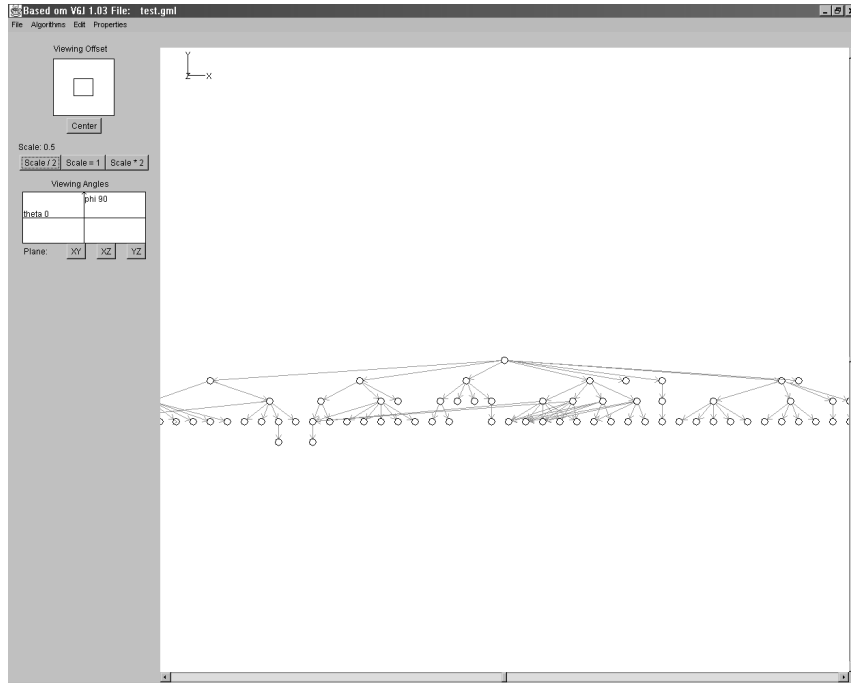


Figure 5.7: The graph editor can show a graph under construction.

The graph editor, based on VGJ, is a graph visualizing tool with the capability to create and edit graphs.

5.3.2 Analysing white box sites

In order to really examine the correctness of the tool, it must be used on a site with a well-known behaviour. For this purpose, a web service developed by the author was used. It is a web based *Othello* game implemented in C++ using CGI. It is purely dynamic, i.e., there is no static HTML documents involved at all. The C++ program generates all responses. One drawback is that the application is completely non-persistent. The state of the game is entirely decided by the parameters that are passed between the client and the server application. In the HTML responses, hidden parameters keep the state. Thus, no surprises can appear, i.e., the response on a specific set of input data is well-defined.

The game of *Othello*

The first state of the game is the “choose game” page. The user can select from three modes:

- Human vs. Computer.
- Computer vs. Human.
- Computer vs. Computer.

Once a choice is made, the user can begin to play. When it is the human’s move, he can select one of the possible moves from a select box. The computers move

is just confirmed by a click on the submit button. Regardless of the selected game mode, it will take about 60 moves to end the game. Normally the tool ends the search when there are no more links to visit.

However, a weakness in that approach was discovered. The game would not end, i.e., the final state “Game Over” was never reached. It is not hard to figure out why it behaves like this. After a couple of moves, the identification algorithm unifies the nodes. The new links from the “unified” node will not be added to the “not visited” set, because it is identified as “the same” response as the previously analysed one. The links are always included in the unifying algorithm and so are the names of the name/value-pairs of possible forms. In this case, a compromise must be done. If the values of the name/value pairs are different, the nodes are still unified. In order to keep the game going the links must be put in the set of not visited links again if they contain unique parameter values. Thus, the game will continue until it ends. *Othello* is certainly not a representative web service, however the principle is important. If input values are changed, the process should continue. This discovery was considered important and a new option was added to the tool.

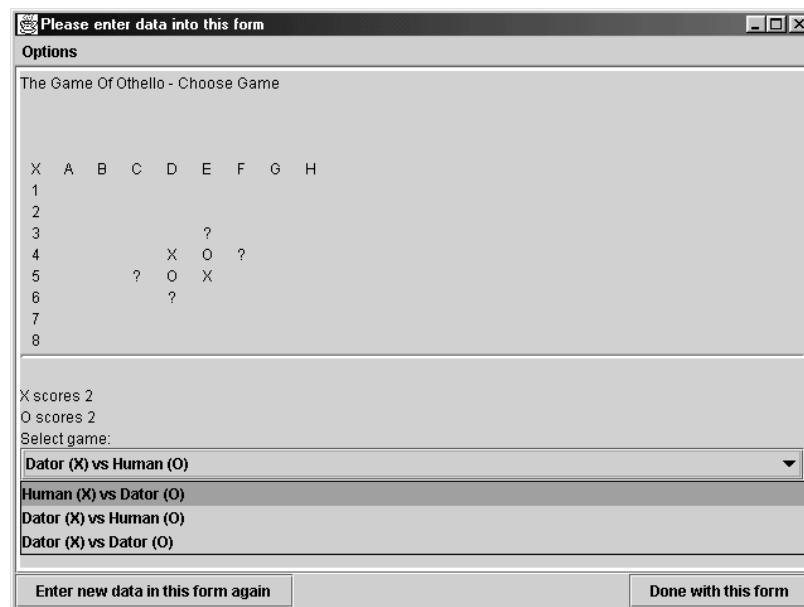


Figure 5.8: Forms are rendered by the tool.

When a form is encountered, the user can enter data if the “form input” option is set. It is possible to keep on entering new combinations of data by selecting *Enter new data in this form*. When there are no more interesting inputs to explore, the form is closed by selecting *Done with this form*.

The interesting subject here is that depending on which algorithm option chosen for the tool, the results were totally different. To perform the analysis, the option form input was enabled (see figure 5.8), the tool started, and when the start form was prompted all three game modes were selected. Then the form input was deactivated and the tool started to “play”. Figure 5.9 to Figure 5.14

shows the results from analysing the *Othello* application under different modes.

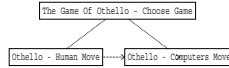


Figure 5.9: An incomplete graph for the *Othello* application.

All states of the game has not been generated, hence the graph is incomplete. Entering three different inputs in the first form starts three game modes. But the process ends before the game is completed, due to the way the *equivalence algorithm* works. It seems as if it is impossible to reach the *Human Move* node from the *Computers Move* node. Note that the little arrow head inside the *Computers Move* node is a self-reference.

Figure 5.9 shows the result of running the tool and entering all three game modes. The problem is that the game over state is never reached and there is no transition from *Computers Move* to *Human Move*. In Figure 5.10 only *Human vs. Computer* was selected and the missing state is *Game Over*.

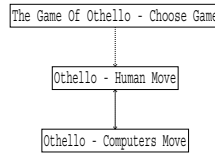


Figure 5.10: The graph for an incomplete *Human vs. Computer* game.

The graph is incomplete, even after a analysis of reduced state space, by selecting only the *Human vs. Computer* game. However the algorithm detects transitions between *Human Move* and *Computes Move* in both directions, but the *Game Over* state is not reached.

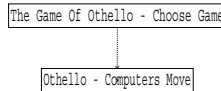


Figure 5.11: The graph for an incomplete *Computer vs. Computer* game.

Selecting only the *Computer vs. Computer* game will also fail since the *Game Over* state is not reached. Note the self-reference.

We will have the same poor results regardless of what game mode is selected. The equivalence algorithm classifies the responses and the process ends because the algorithm is not able to detect the differences in the responses. The reason for this is that the state information is kept in hidden parameters in the forms. However, only the parameter values change, not the parameter names. A change of the amount of parameters, their names or their order would classify the response as unique, but this is not the case when it comes to the values.

The proposed solution of this problem is to introduce a new option in the tool. It does not affect the *equivalenceAlgorithm*, but it affects the way links are added to the set of not visited URIs. Optionally, if a unique set of parameter values is detected in a response, the *action URI* found in the form is added to the set of not visited URIs, together with the new set of name/value pairs. This is the case even if the response is unified with previously received response. This option is put in the *add links* algorithm of the spider. This explains the optional call to *updateReferences* in Figure 4.6.

Using the new option, the game actually is played until all states of the application are surveyed. Hence, the graph is a complete functional view of the application. Unfortunately, a successful analysis is dependent on correct user input. If the user input excludes information that would bring the application into a new state, the corresponding sub graph will not be analysed.

Now it is possible to start the program and give a complete set of inputs to the first input frame of the application. Figure 5.12 shows the successful result after selecting the *Human vs. Computer* game.

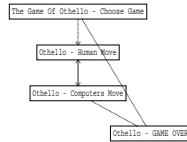


Figure 5.12: A complete graph for the *Human vs. Computer* game.

The process runs until the *Game Over* state is encountered. Thus all the states of the game are explored and the analysis is successful. *Note: the graph was edited by moving the Game Over node to the right. It was originally drawn centred at the bottom, concealing some of the other edges.*

If the operator selects all of the three game modes, they will be played simultaneously until game over. At last the generated graph in Figure 5.13 is a complete representation of the web application. All crucial user input was considered and entered by the operator.

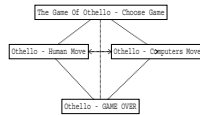


Figure 5.13: A complete graph for the *Othello* application.

This is the complete graph showing all of the states and state transitions of the *Othello* application. Note the self-reference in the *Computers Move* node.

Some statistics of the analysis is presented in the spider GUI during the process. They are:

- The current number of URIs in the *unvisited set*.
- The current number of unique responses in the *visited set* (according to the equivalence algorithm).
- The total number of responses.
- The number of failed requests.
- The accumulated response time.
- The mean access time.

Figure 5.14 shows a the spider GUI after the analysis of the *Othello* application, exploring all game modes. It is interesting to see that 251 responses were analysed and classified into only four unique nodes.

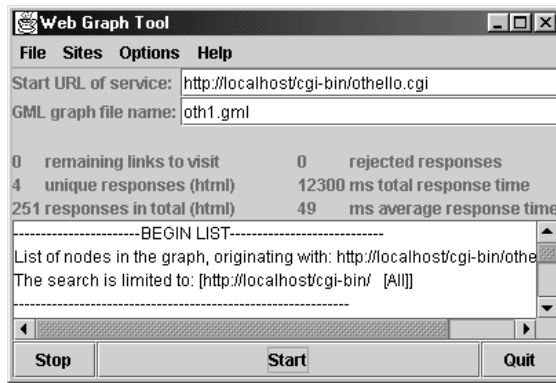


Figure 5.14: The spider tool showing statistics.

Analysing three sub services

There is no limit of the number of services that can be explored and analysed during one single pass of the spider. This is the reason for having constraints on the sites and hosts that are allowed. Without the sites list, potentially the entire Internet would be explored as long as there are new links to follow. However, inside the delimitation, all links must be explored and these links may point to different services. As an example of the state independence of the HTTP protocol, three applications are examined in parallel. The analysis is done in breadth first mode, meaning that the input forms from the different services are mixed with each other. It is very confusing for an operator to follow the program flow of the services. Using depth first makes it much easier for the operator to understand what is going on, since the forms are displayed separately for each sub service. However, the same graph, representing the three different services, is produced by the spider (see Figure 5.15).

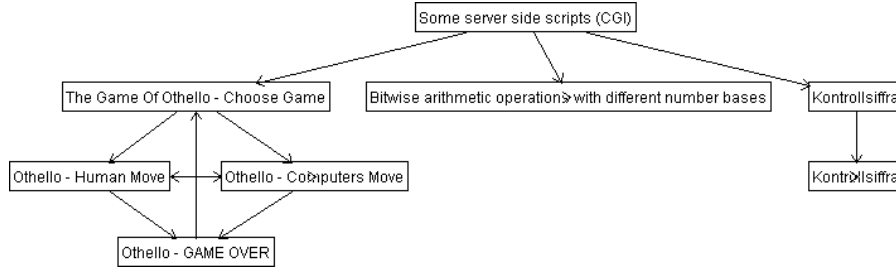


Figure 5.15: A graph showing three web services.

Experiments on dynamic tables

A server application that counts the number of accesses to it was developed in order to evaluate the truncated parse tree prefix algorithm. The application creates an HTML table containing a new row for each access. Using the spider without this algorithm causes a never-ending loop since each response is unique to the previously generated ones. Therefore, it has to be aborted. However, it turned out that the truncation works out fine. The result is a graph containing only one node. In Figure 5.16, a snapshot of specific response is shown to the left. In the middle is a part of the generated graph (after abortion). Finally, the complete graph containing only one node and a self-reference is shown to the right.

5.4 Delimitation

Some delimitation of the problem were made in order to find feasible solutions. The most important are highlighted in this section.

5.4.1 Support for HTML version 3.2

The implementation uses Java 1.3 (Swing) which has a parser that only supports HTML 3.2. Today XHTML 1.0¹ or at least HTML 4.01 are standards on the Internet [18].

5.4.2 Client side scripts - Dynamic HTML

HTML documents can contain script statements within `<SCRIPT>` tags. If links are hidden within the scripts, the ability to parse them is required. However implementing such a parser lies beyond the scope of this work since a semantic analysis would be required. Thus the problem is solved by a straight forward scanning for URIs in the comments within the `<SCRIPT>` tags.

5.4.3 User input into forms

The suggested tool will not try to generate or simulate user input. No other automated generation of form data than the default selections will be performed.

¹XHTML 1.0 is a reformulation of HTML 4.01 in XML

Access logger

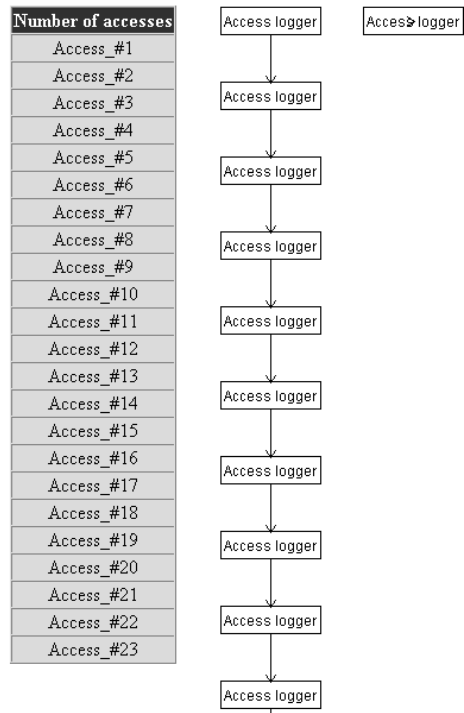


Figure 5.16: A self-referencing, growing document.

To the left is a snapshot of an HTML response from the web application, visualized in a web browser. Every new access causes the table to grow by one row. Apparently, the application has been accessed 23 times. If the spider starts an analysis of this application, it will keep on finding new nodes since an HTML table is a structure that indeed affects the structure of the parse tree. Thus the parse tree prefix will be affected, and the nodes will be considered as unique to each other. This will result in a never-ending graph, seen in the middle. However if the *patternTruncator* algorithm is used, the table structure will not affect the equivalence algorithm. As long as there is a table in the same relative position in the document, the responses will be unified. The graph will consist of only one node, referring to itself, shown to the right.

However, the operator (user) can manually insert data into the forms.

5.4.4 Cookies unsupported

Support for cookies is not yet implemented. Cookies are optional “footprints” of the state of a service, stored on the client. The web browser stores the data in the cookie file, in negotiation with the web server using the HTTP-protocol. The contents of the cookie can be sent back to the server on demand. In this study, the protocol for this negotiation has not been investigated.

5.4.5 Multithreading

A multithreaded implementation would dramatically speed up the tool. The bottleneck lies in the delayed server responses. If requests would be sent out

in parallel, lots of time would be gained. One could believe that this is the case only if two requests have different target hosts. However, the servers are designed as multithreaded platforms themselves and thus each request will cause a new thread on the server that will compete with other threads for CPU-time. To achieve this the main containers used by the algorithm must be synchronised and thread safe. It could easily be done in breadth-first search mode by querying all URIs originating from a common parent. Luca de Alfaro [7] also realised this and implemented it, but discusses the danger of this speed up. The servers may be overloaded. In a test environment such a behaviour is perfectly all right and is even desirable, but for commercial sites, this would rather be experienced as a hostile attack.

Chapter 6

Discussion

In this chapter some issues are raised for discussion. The results are discussed in section 6.1, the importance of other works within this area in section 6.2, usefulness in section 6.3 and finally, a number of acknowledgements are presented in section 6.4.

6.1 The expected versus the achieved result

The expected result was to develop a tool that could build a graph, describing a web service. This graph would be useful for software testers when they are designing tests for the service. The goal is achieved and there is a prototype implementation, which can make the analysis, present the graph and even print it. However, there are some shortcomings of the completeness of the analysis.

Many modern web services use client side scripts, embedded in the HTML code. The scripts code can conceal links from the tool by building URIs dynamically. The tool does not consider the syntax and semantics of the various scripting languages. However, the scripts are scanned for strings that look like an URI. This makes the analysis potentially incomplete for these kind of sites. Further work is needed to make the tool work properly concerning client side scripts.

It is also possible to imagine an application where a large number of different parameters can be input from HTML forms. It is often obvious which data to input in the form. However, some rare combinations of input will lead to certain states of the application which are not in the “mainstream”. Given a piece of finite time, long enough to try all combinations, we will eventually succeed in finding all states of the application. The problem is that a piece of finite time could be very long; it could take days, months or thousands of years to find all possible combinations¹. This could lead to an incomplete analysis of application. If the operator does not know all of the special cases and “backdoors” in the system, they will probably never be found.

In summary, as this study demonstrates, it was possible to solve the major problems and make a more or less functional tool.

¹It must be kept in mind that the data input is not inserted automatically; it is done by an human operator.

6.1.1 Web node equivalence

The web node equivalence problem is challenging and hard to define: what are the factors that make a node different from another? In the authors comprehension, some differences should not affect equivalence, such as the number of rows in a table or list or the number of paragraphs in a text block. This problem is solved by the *patternTruncator* algorithm (see section 4.1.5), an augment of the *partial parse tree* algorithm that neglects “repeated patterns” in the parse tree prefix.

The fundamentals of the unifying algorithm are based upon the set of internal links and it is not optional. The other criteria are optional: the set of names in the form name/value-pairs, the document titles and the parse tree prefixes (optionally truncated). The experiments have shown that the algorithm works.

6.2 Previous research and results

Many of the issues in the paper by Ricca and Tonella [4] are quite similar to the problems at hand in this work, and so are the solutions. The similarities of the spider algorithm in both works support my solution. They seem to have the same philosophy concerning user interaction and their tool is semiautomatic as well. They focused on white box testing, hence using knowledge of the server application.

6.3 Usefulness

In my opinion, the tool is very useful for analysing static web sites and dynamic web applications. The produced graph will show a “map” of the site (or application) which could be very useful for understanding how it is structured. Of course, using the tool for an analysis is a test itself. In addition, the graph can be used as a foundation when constructing tests for the application. It is also a good way of documenting the application. The graph is just not a abstraction of the application’s structure. It is also a collection of all the captured attributes from the analysis. All the URIs, form parameter names and input values are accessible from the graph by a simple “double click” on a node or edge.

The tool could be slightly modified to produce a “flat index” of a web site. This would be a structured HTML page containing all the URIs found in the entire site. The index page would be a collection of short cuts that could be useful for a designer, tester or user.

One drawback is the order in which the input forms are displayed. An input form will appear on the screen as soon as a server response contains an HTML form. There is no guarantee that the order will follow the logical order in the application. An addition, some URIs are issued many times. If they contain forms, the same form will be displayed several times, forcing the user to enter new input. This could be annoying and time consuming for the user.

What do the users think about the tool; could it be used as an aid for testing? In the discussion after the presentation of the tool, the invited software testers came up with some comments. They are pleased with the tools at hand and they did not think they could gain very much by using this tool. Perhaps they could use it as a complementary tool. The research problems in the scope of

this work are quite interesting, but perhaps the practical use is not interesting enough for the presumptive users. However, the tool is not completed yet. The ability to build a graph was focused in this work, and hopefully the tool can be further developed into a useful application for software testers.

6.4 Acknowledgements

As a gesture of deepest gratitude, the importance of the contributions from the following persons must be emphasized:

- Professor Bengt Jonsson at University of Uppsala, initiator, supervisor and examiner.
- Stefan Magnenat, supervisor at Validation AB, Stockholm.

Chapter 7

Conclusions

This study shows that it is possible to make a tool that performs an analysis of a web service and presents a model of the service in a graph that is visualized in the tool. It is possible to identify and classify the Web server responses by using its structure rather than using its reference (URI). However, this has to be further developed before it can be used as a professional tool. There is no guarantee that the produced graphs reflect a 100% true picture of the explored web service. One of the reasons for this is that the tool is unable to analyse the semantics of embedded client side scripts. The scripts must be interpreted in order understand its behaviour, e.g., how it produces URIs.

7.1 Answers to the questions at issue

The issued questions were answered during the development of the tool and the algorithms. The results are briefly summarised in this section.

7.1.1 Web node equivalence

Repeated requests to a web application through the same URI can give different responses. This is why the URI itself cannot be used as a criteria for equivalence. It is a better approach to compare the contents of the responses; the HTML documents.

Nodes are identified using a combination of criteria, some of which are optional. They are:

- The set of internal links (necessary).
- The set of parameter names (necessary).
- Partial parse trees (narrowing), with or without truncating (optional).
- Titles (optional).

It is hard to find one algorithm that suites all kinds of services and therefore the spider has many optional parameters to fine tune the analysis.

7.1.2 Accessing the web

An algorithm that searches through a web service was developed and verified. The Java API offers classes for a convenient, high level approach to the problem. The entire HTML page is modelled into an abstract document that contains the elements. The algorithm starts with a set (U) of not visited nodes containing only the start URI. An URI is selected from U , it is visited, the response is analysed and any occurrence of an URI is added to U . But not if the URI was visited before (node identification). The process continues until U is empty.

7.1.3 User interaction

When forms appear in an HTML document, the page is rendered in a GUI and the user (the operator) can enter data into it. In order to investigate all combinations of interest, the operator can post any number copies of the form carrying different input values. Each of these forms can potentially give raise to a new path in the graph.

7.1.4 Graph definition

Graphs are defined in text files using the GML language. Nodes and edges are defined separately and they can contain any set of attributes. These attributes are used to store data, e.g., URIs and the name/value pairs in forms. The information is stored in a well-structured format, which is suitable to parse by the graph editor. Arbitrary attributes can be added to the graph ad hoc.

7.1.5 Visualization of graphs

A package called VGJ (visualizing graphs with Java) is used to render the graphs. It supports the GML language and can read and visualize GML files. VGJ is a graph editor with many functions. It can draw new nodes and edges, edit them and even print the graphs. Some modifications of the code were necessary in order to build graphs “ad hoc” from the spider. Some of the edit modes were also removed from the tool. Double clicking on a node or an edge accesses the attributes.

7.2 Further research

The Java platform is convenient, but the swing XML parser and the HTML-package offers an API that makes it a bit cumbersome to analyse the structure of web documents. In addition, there is no support for parsing client side scripts and therefore this implementation uses a simple string-matching algorithm to find links within the script code. It would perhaps be a better idea to use a standardised specification such as DOM¹ in order to access the elements of a document. Modern web browsers provide a DOM API for script programmers and these browsers can parse and interpret client side scripts. Is it possible to write applications that interfaces to a web browser through DOM, thus using the browser’s parser, i.e., do they provide an open API?

¹Document Object Model

One of the major reasons to develop this tool was to get a visual representation of a web service. Using this representation (specification), would it be possible to select a path from within the graph and automatically generate scripting code for the various existing test tools?

Another development would be to let the tool detect changes in web services, provided with a previously generated graph of the same services. This would require an extended level of persistency of the nodes and edges. The parse tree prefixes would have to be stored in the graph definition file; the GML file. How should the changes be represented in the new graph? Would colour coding be feasible?

How should HTML frames be represented in the graph? Using frames in an web application allows the user to interact in a number of parallel browser windows. Normally, one of the windows is used as a “main menu” from which the user can make choices which have an impact on an other window’s content. The generated graph will show all the nodes (URIs) but it will not give any information of which nodes are reachable from a certain frame. This information would be very important since it reflects the users apprehension of the application. Could the problem be solved by colour coding the sub trees in the graph that are visualized in a certain frame?

Bibliography

- [1] Mercury Interactive Corporation. *WinRunner and LoadRunner*. URL: <http://www-svca.mercuryinteractive.com/products>, December 2001.
- [2] Bengt Jonsson, Daniel Löf, and Stefan Magnenat. Graphically structured specifications of www application interfaces and their use for automated test case generation. ASTEC competence center for Software Technology, 2000.
- [3] Daniel Löf. Automated testing of web application functionality. Masters thesis, University of Uppsala, Uppsala, Sweden, January 2000.
- [4] Filippo Ricca and Paolo Tonella. Building a tool for the analysis and testing of web applications : Problems and solutions. In T. Margaria and W. Yi, editors, *In proceedings of TACAS 2001*, LNCS 2031, pages 373–388, Berlin Heidelberg, 2001. Centro per la Ricerca Scientifica e Tecnologia, Italy, Springer-Verlag.
- [5] J. Punin and M. Krishnamoorthy. Wwupal system - a system for analysis and synthesis of web pages. In *Proceedings of the Web-Net 98 Conference, Orlando*, URL: http://www.cs.rpi.edu/~puninj/WEB-NET/wwupal_paper.html, November 1998. Dept. of Computer Science, Rensselaer Polytechnic Institute, NY, USA.
- [6] Y. Maarek and I. Shaul. Webcutter: A system for dynamic and tailorable site mapping. In *Proceedings of WWW 6 Conference, Santa Clara, USA*, URL: <http://www.scope.gmd.de/info/www6/technical/paper040/paper40.html>, 1997. IBM Haifa Research Laboratory.
- [7] Luca de Alfaro. Model checking the world wide web. In *Proceedings of the 13th Conference on Computer Aided Verification*, Berlin Heidelberg, 2001. University of California at Berkeley, USA, Springer-Verlag.
- [8] T. Berners-Lee, R. Cailliau, A. Loutonen, H. F. Nielsen, and A. Secret. The world wide web. In *Communications of the ACM*, Volume 37, Number 8, pages 76–82, New York, August 1994. ACM, ACM press.
- [9] Tim Berners-Lee et al. *RFC 2396 Universe Resource Identifiers (URI): Generic syntax*. World Wide Web Consortium, MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139, U.S.A., August 1998.

- [10] James E. Goldman, Phillip T. Rawles, and Julie R. Mariga. *Client/server information systems : a business-oriented approach*. John Wiley & Sons, Inc, New York, 1999.
- [11] Tim Berners-Lee et al. *RFC 1945 HTTP/1.0*. World Wide Web Consortium, MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139, U.S.A., May 1996.
- [12] Hal Berghel. Caustic cookies. In *Communications of the ACM*, Volume 44, Number 5, pages 19–22, New York, May 2001. ACM, ACM press.
- [13] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol Version 3.0*. Transport Layer Security Working Group, November 1996. Internet draft, expired in June 1997.
- [14] Dave Ragget. *HTML 3.2 Reference Specification : W3C Recommendation 14-Jan-1997*. World Wide Web Consortium, January 1997.
- [15] Roger Fournier. *A Methodology for Client/Server and Web Application Development*. Prentice-Hall, Inc, Upper Saddle River, NJ 07458, 1999.
- [16] Martijn Koster. Ww robots, wanderers and spiders. URL: <http://www.robotstxt.org/wc/robots.html>, December 2001.
- [17] Doug Reichen. Intelligent agents. In *Communications of the ACM*, Volume 37, Number 7, New York, July 1994. ACM, ACM press.
- [18] The World Wide Web Consortium. URL: <http://www.w3c.org>, December 2001.
- [19] J. Punin and M. Krishnamoorthy. Extensible graph markup and modeling language (xgmml). In *The XML.org XML Standards Report*, URL: <http://www.cs.rpi.edu/~puninj/XGMML/>, 2001. Dept. of Computer Science, Rensselaer Polytechnic Institute, NY, USA.
- [20] Michael Himsolt. Gml : A portable graph file format. Technical report, Universität Passau, URL: <http://www.uni-passau.de/Graphlet/GML>, 1997.
- [21] Michael Himsolt. Graphlet. Technical report, Universität Passau, URL: <http://www.uni-passau.de/Graphlet/>, 1997.
- [22] Algorithmic Solutions Software GmbH, URL: http://www.algorithmic-solutions.com/as_html/products/leda/products_leda.html. *LEDA*, December 2001.
- [23] R. Wiese and M. Kaufmann et al. *GraVis*. Wilhelm-Schickard-Institut für Informatik, URL: <http://www-pr.informatik.uni-tuebingen.de/Forschung/GraVis/welcome.html>, December 2001.
- [24] Max-Planck-Institut für Informatik, URL: <http://www.mpi-sb.mpg.de/AGD/index.html>. *AGD*, December 2001.
- [25] Michael Fröhlich and Mattias Werner. *daVinci*. University of Bremen, Germany, URL: <http://www.informatik.uni-bremen.de/daVinci/>, December 2001.

- [26] I. Bruss and A. Frick. *Gem3Ddraw*. Fakultät für Informatik, Universität Karlsruhe, Germany, URL: <http://i44s11.info.uni-karlsruhe.de/~frick/gd/>, December 2001.
- [27] J. Ellson, E. Gansner, and E. Koutsofios et al. *Graphviz*. AT&T Labs-Research, URL: <http://www.research.att.com/sw/tools/graphviz/>, December 2001.
- [28] Úlfar Erlingsson and Mukkai Krishnamoorthy. *Interactive Graph Drawing on the World Wide Web*. Dept. of Computer Science, Rensselaer Polytechnic Institute, NY, USA, URL: <http://www.cs.rpi.edu/projects/pb/graphdraw/>, December 2001.
- [29] J. Berry, N. Dean, M. Goldberg, G. Shannon, and S. Skiena. Graph drawing and manipulation with link. In *Lecture Notes in Computer Science*, LNCS 1353, pages 425–437, Berlin Heidelberg, 1997. Springer-Verlag.
- [30] Dr. Carolyn McReary and Larry Barowski. *Drawing Graphs with VGJ*. Auburn University, U.S.A., URL: http://www.eng.auburn.edu/departments/cse/research/graph_drawing/graph_drawing.html, 1998.
- [31] G. Sander. Graph layout through the vcg tool. In R. Tamassia and I. G. Tollis, editors, *Lecture Notes in Computer Science, Graph Drawing, DIMACS International Workshop GD'94*, LNCS 894, pages 194–205, Berlin Heidelberg, 1995. Springer-Verlag.
- [32] FSF. *GNU GENERAL PUBLIC LICENSE, Version 2*. Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, June 1991.
- [33] J. Q. Walker II. A node-positioning algorithm for general trees. In *Software-Practice and Experience*, Vol 20(7), pages 685–705, July 1990.

Appendix A

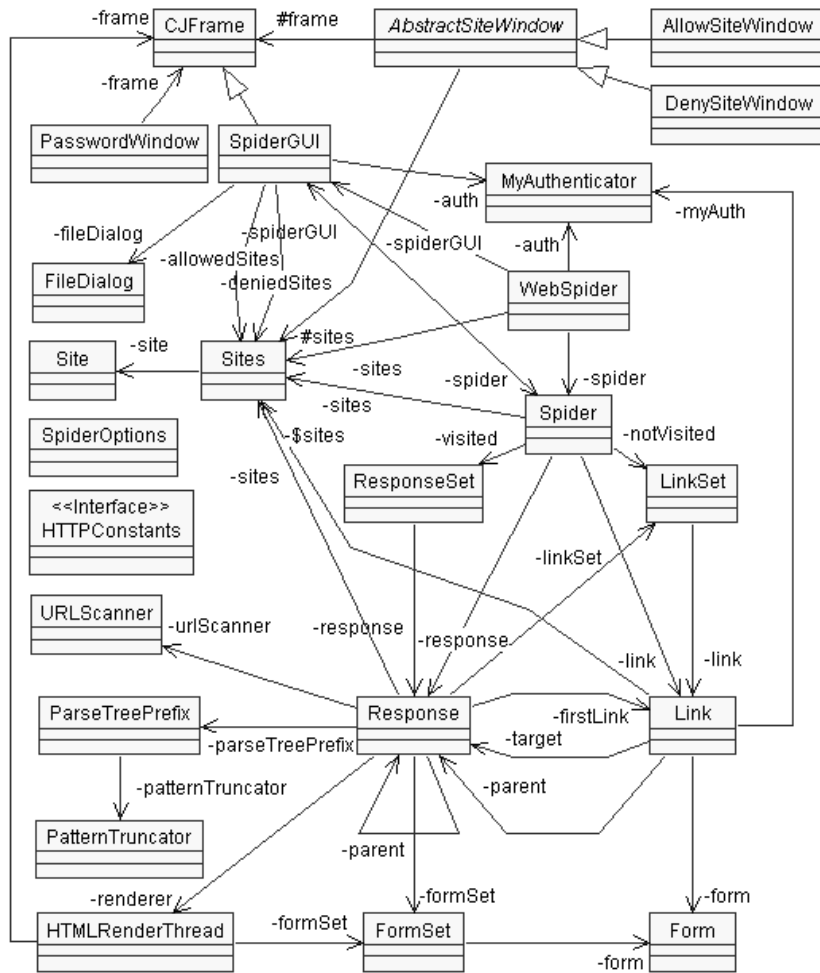


Figure A.1: The class diagram for the application.

This is a UML class diagram, showing the relations between the classes in the implemented system.