Dipartimento di Informatica e
Scienze dell'Informazione

Analysis, Testing and Re-structuring of Web Applications

by

Filippo Ricca

Theses Series **DISI-TH-2003-03**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

Analysis, Testing and Re-structuring of Web Applications

by

Filippo Ricca

September, 2003

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science

Submitted by Filippo Ricca
DISI, Univ. di Genova
ricca@itc.it

Date of submission: April 2003

Title: Analysis, Testing and Re-structuring of Web Applications

Advisor: Bruno Caprile
ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, Povo (Trento)
caprile@itc.it

Supervisor: Gerardo Costa
DISI, Univ. di Genova
costa@disi.unige.it

Ext. Reviewers: Cornelia Boldyreff
Dept. of Computer Science, Univ. of Durham, UK
Cornelia.Boldyreff@durham.ac.uk

Ira Baxter
Semantic Designs, Austin, USA
idbaxter@semdesigns.com

Giuseppe Di Lucca
Dip. di Ingegneria, Univ. del Sannio, Benevento
dilucca@unisannio.it

Abstract

Web applications are complex software systems providing access to hypermedia contents, services, computational facilities and data. The quality of Web applications is a complex, multidimensional attribute that involves several aspects, including correctness, reliability, maintainability, usability, accessibility, performance and conformance to standards.

We look at the current situation in the development of Web applications as reminiscent of the early days of software systems, when quality was totally dependent on individual skills and lucky choices. In fact, most Web applications have so far been developed without following a formalized process model: requirements are not captured, design is not considered and documentation is not produced; developers quickly move to the implementation phase and deliver the application without testing it. While this kind of practice was motivated by the relative simplicity of the first Web sites and by a pressing demand for short times-to-market, quality appears now to emerge as a primary factor of concern.

In this context, aim of the thesis is to investigate, define and apply a variety of conceptual tools, analysis, testing and re-structuring techniques able to support the quality of Web applications. The goal of analysis and testing is to assess the quality of Web applications during their development and evolution, while re-structuring points at improving their quality by suitably changing their structure.

The approach we have adopted moves from the observation that several methods for the analysis, testing and re-structuring of traditional software systems already exists, which have proved useful in a huge variety of environments and application domains. The idea is therefore that of resorting to these methods - suitably adapting them to the case of Web applications.

The work done in the thesis can be subdivided into three main parts: (1) development of abstract models for Web applications, able to accommodate static as well as dynamically generated entities; (2) application to Web models of a variety of analyses, testing and re-structuring techniques; (3) empirical analyses done on real-world applications.

More in particular, the Web models represent the starting point for several analyses, both in the assessment of the Web application structure and in the study of its evolution. Moreover, the Web models drive structural and statistical testing, as they are exploited to define the testing criteria and to provide semi-automated generation of the associated test cases. Finally, they represent the common framework for all the other analyses and re-structuring techniques, such as slicing, HTML transformations, and alignment of multilingual contents.

The investigation has been corroborated by a series of empirical studies carried out on selected corpora of existing web applications, encompassing a wide spectrum of technologies (frames; dynamic pages) and contents (multilingual; e-commerce). As a necessary component of the experimental work, a prototype toolkit has been built which implements a variety of analyses and supports the developers in testing and re-structuring activities.

Keywords: Web Engineering, Reverse Engineering, Re-structuring, Model Extraction, Evolution, Structural and Statistical Web Testing, Multilingual Websites, Slicing.

To my family and Betta

Acknowledgements

I am very grateful to the people that helped me during the thesis. First of all, I want to thank my advisor, Bruno Caprile, for his precious advice and continuous support.

I would like to thank Gerardo Costa (the supervisor), for the interesting discussion I had with him and for the help given to me, and Eugenio Moggi, the coordinator of my PHD program.

I would like to thank the reviewers of this thesis, Cornelia Boldyreff, Ira Baxter and Giuseppe Di Lucca, for their helpful and pertinent comments and suggestions.

Finally, I wish to express my deepest gratitude to Paolo Tonella, for his continuous and invaluable advice. Paolo introduced me to the literature on analysis and testing of software systems, and shared with me the work of the last three years. The present thesis would have never been conceivable without his help and support. Thanks Paolo.

Table of Contents

Chapter 1

Introduction

This chapter outlines the work done in the thesis, sketching motivations and problems tackled. The aim of the thesis and the approach followed are first introduced. The research path and the structure of the thesis conclude the chapter.

Web applications are among the fastest growing classes of software systems in use today. They pervade our life every day more, being crucial for a host of economic, social and educational activities. Associations, enterprises, governmental organizations, companies, scientific groups use the Web as a powerful, convenient and inexpensive way to make public and promote activities and products. Given the critical importance of such applications, poor quality can have heavy consequences on business, but also on national economies and social advancement at large. In this respect, relevance of supporting the reliability and quality of Web applications cannot be overstressed.

In the early days, Web sites were primarily static, i.e., composed only of Web pages stored in some file system, linked together through hyperlinks. Aim of Web sites was to provide information across the Web in a rather simple and intuitive manner. Quality assurance was therefore a relatively simple task – which many disregarded nonetheless. The rise of new technologies (e.g., server and client scripting languages) introduced the concept of computation in the Web applications realm, thereby allowing novel and much more complex human-Web interactions. Today, it is typical for Web applications to build dynamic pages on-the-fly, run server side programs whose outcome depend on user inputs. On one side, these novelties increase the usefulness of Web applications; yet, on the other, they increase their complexity. As a consequence, quality assurance (and the notion of quality itself) is deeply affected.

The first Web sites were developed without following a formalized process model: requirements were not captured, design was not considered and documentation was not produced.

Developers quickly moved to the implementation phase and delivered the "site" without testing it. This kind of practice, motivated by the relative simplicity of the first Web sites and by the pressure for short times-to-market, is no longer suited to the demands of customers and public. Today, many would acknowledge that development and maintenance of Web applications have grown too complex and multifaceted to be managed informally.

It is interesting to notice how the current situation is somewhat similar to the early development of software systems, before the advent of software engineering [Pre00], when quality was totally dependent on individual skills and lucky choices. The consequence of the lack of a formalized process model is that the quality of built Web applications is in general poor. In fact, authors refer this situation as the "Web crisis" [GM01].

Let us consider the issue from the perspective of quality factors. The typical user perceives a Web application as a set of Web pages on the screen monitor. The browser is the tool that permits the user to visualize the information and to navigate across the site. The user, through the browser, is exposed only to the *external factors of quality*, as, for example, correctness, reliability and usability. These quality factors are very important for a Web application because they impact directly on the appeal of the site, and, as consequence, on its success. People move away from Web applications that prove unable to satisfy external quality factors.

Yet, the user perception of a Web site does not always match the perception of the developer. The latter deals with internal components of the application such as the HTML, scripting languages, servers, databases, etc. Even if the *internal quality factors* of Web applications (i.e., maintainability and portability) are the main aspects for the Web developer, he/she must bear in mind also the external quality factors. Internal factors are important because they can influence indirectly the success of the site. In a Web application of good internal quality, evolution is easier, and changes are more rapid.

The situation can be synthesized in a line: *we are in need of better methodologies, techniques and tools for developing, maintaining and testing Web applications.*

There are different ways to improve quality factors of Web applications. Some of them are focused on the forward engineering step, proposing models and formalisms aimed at supporting the design of Web applications. Others assume that a Web application already exists and support its analysis, testing and re-structuring. While substantial effort was devoted to investigating models and formalisms aimed at supporting the design of Web applications only few works considered the problems related to Web applications analysis, testing and re-structuring. This is what the present thesis concentrates on.

1.1 Aim of the thesis

Aim of the thesis is to investigate, define and apply analysis, testing and re-structuring techniques to Web systems in order to assess and improve their quality.

The goal of analysis is to assess the quality of Web applications during their development and evolution. In particular, analysis results and views produced can be used to check whether the Web application satisfies suitable constraints and to detect possible anomalies. Analysis can be also useful to understand better the internal organization of Web applications, to support maintenance activities and to support restructuring interventions.

The goal of testing is to improve the Web application quality, detecting possible failures i.e. deviation of the application from the intended behavior.

With re-structuring we shall intend a transformation aimed at improving certain quality factors of the application, without changing the information provided and the external behavior of the Web application.

The approach we have adopted throughout the thesis is based on the **reverse engineering** paradigm. Differently from the more traditional forward engineering approach, we do not propose models and formalisms aimed at supporting the design of Web applications. Rather, we move from the assumption that a Web application already exists, and investigate techniques to support its analysis, testing and re-structuring. Our starting point is therefore the actual implementation of the Web application (Web pages, server programs, etc.), and the techniques investigated work on the *abstract models* recovered from the implementation (see Figure 1.1).

The approach we have adopted is based on the observation that several well established methods for the analysis, testing and re-structuring of traditional software systems already exist, that can be adapted to Web applications. The idea is therefore that of resorting to these methods, adapting them to the case of Web applications (see Figure 1.1).

Empirical validation of results is an integral part of our approach. In order to provide empirical support to the proposed models and techniques, a prototype toolkit implementing the analyses investigated and supporting developers in testing and re-structuring activities has been developed and field-tested on various corpora of existing Web applications (case studies).

The work done in the thesis can be subdivided into three main parts:

- study and definition of **abstract models** for Web applications, able to represent the various entities involved and their mutual relationships.
- application to Web models of a variety of analyses, testing and re-structuring techniques inspired to those already in use for **traditional software systems**.

- testing and validation of ideas, models and techniques proposed by means of a **prototype toolkit** applied on real-world Web applications.

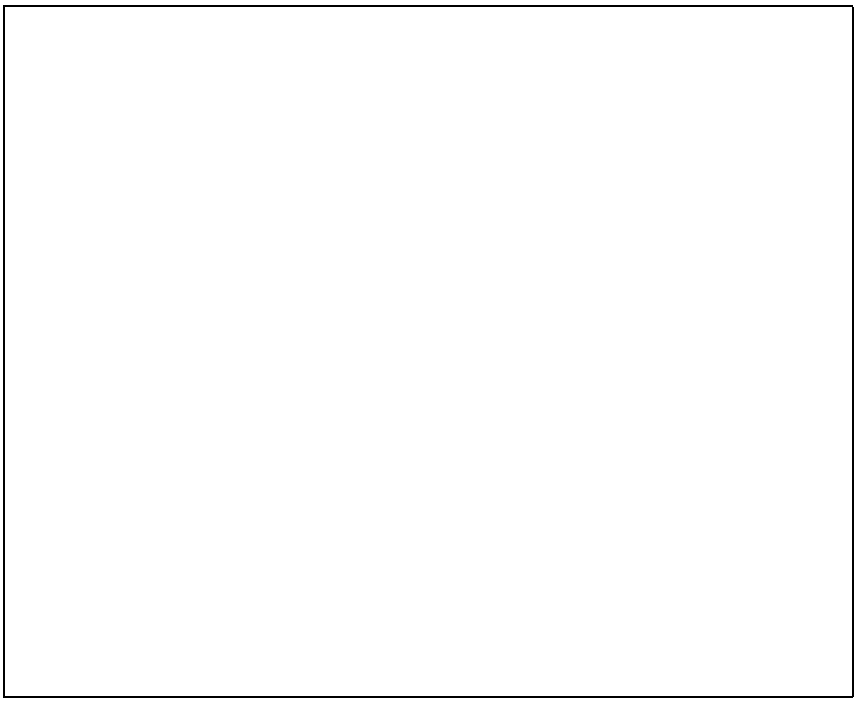

Figure 1.1: *Reverse engineering approach and resorting to software engineering techniques.*

1.2 Research path

The first **abstract model** developed was very simple and able to represent only some static components of a Web application. In this first model a Web site was represented as a graph where nodes corresponded to pages, and edges to hyperlinks. On this model, some simple structural (e.g., **unreachable pages**, **shortest path** and **reaching frames**) and evolution (**historical view**) analysis were proposed.

Later, entities such as forms, dynamic links and dynamic pages were added to the initial model. This model, which we choose to represent in the Unified Modeling Language (UML) format [BRJ98], turned out very useful for analysis of dynamic sites; yet, it proved inadequate for supporting other research lines of our interest – namely, Web application testing. In particular, the model called **implicit-state model** could not deal with the case of server programs that behave differently according to the interaction state. Another model was thus devised: the **explicit-state model**. The latter differs remarkably from the

previous one, in that it unrolls server programs and dynamic pages with different behaviors into actually different entities.

To the structural and evolution analysis proposed initially, other interesting analyses such as **slicing** and **multilingual Web sites consistency** have also been added. The slicing technique, well known for traditional software system, can be exploited during several activities such as testing, debugging and understanding. The problem of verifying the consistency between Web site portions devoted to different languages (e.g. Italian, English, French etc.) has been also investigated. Anomalies typically occurring in multilingual Web sites include absence of pages in some languages, differences in the page structure in different languages, missing information and parts not translated. Some algorithms for highlighting anomalies have been proposed along with a technique for restructuring multilingual Web sites.

A set of **(semi)-automatic transforms** improving the quality of Web applications has been proposed. These procedures work at two different levels: inter-page and intra-page transformation. In the first case the procedures exploit the model of the Web application in order to locate changes, and update the model when the transformations are applied. In the second case, the procedures transform a single HTML page, and the result is a new HTML page.

Investigation of **statistical testing** and **regression testing** of Web applications are the last research topics treated in the thesis.

Two research prototypes (**ReWeb** and **TestWeb**), were also developed which implement the extraction of models from existing Web applications (reverse engineering), thereby providing static analyses and the support to re-structuring and testing techniques. Both have been thoroughly employed to test and validate the research ideas, models and techniques proposed.

Structural problems and multilingual inconsistencies were detected in real-world Web application (e.g., www.ubicum.it and www.ien.it), for which re-structuring was done using the analysis and views provided by **ReWeb**. Some of the semi-automatic transforms improving the quality of Web applications (for example the design transform *reorganization into frames*) have as well been applied to real world case studies (e.g., www.apdt.net and www.simd.it). The testing techniques proposed were successfully applied to several real world Web applications, including Wordnet (www.cogsci.princeton.edu), Amazon (www.amazon.com), FS-online (orario.fs-on-line.com/orario.it.html) and ITC-publications (publications.itc.it).

1.3 Thesis structure

The thesis is structured as follows:

- **Background.** This chapter provides background information. Important terms used in the thesis are defined and explained.
- **Web application modeling.** Web application models, able to support analysis, testing and re-structuring are introduced and discussed.
- **Structural and evolution analysis.** The set of analyses investigated in the thesis is explained in detail.
- **Web application slicing.** An approach to Web application slicing is presented and motivated.
- **Structural Web application testing.** After surveying techniques for Web testing, structural testing is presented.
- **Statistical Web application testing and analysis.** Statistical Web application testing is described and some statistical analyses of Web applications are proposed which are based on the analysis of the access log.
- **Web application transformations.** After introducing the notation and the re-engineering toolkit used, a set of HTML transforms improving the quality of Web applications and handling complex re-structuring tasks is described.
- **Multilingual Website consistency.** A re-structuring process is proposed for making the multilingual parts of a Web application consistent with each other.
- **ReWeb and TestWeb.** The structure, the facilities, and roles of ReWeb and TestWeb in the analysis, testing and re-structuring process are described.
- **Conclusion and future work.** In this final chapter, main contributions of the thesis are summarized, the problems encountered are discussed and further developments are outlined.

Every chapter (except introduction, background and conclusion) is organized into three logical parts. The first is more theoretical in spirit, and introduces the problems, possible solutions and techniques used. In this part algorithms and heuristics used are explained in detail. The second part is more practical. It typically presents a case study related to the first part. In the last part, the pertinent literature is surveyed, highlighting difference between our techniques and other approaches.

Relevant background material of more general interest is collected in dedicated appendixes.

Chapter 2

Background

This chapter defines important terms used in the thesis and explains the object of our analyses by answering the question: What is a Web application? The meaning of the terms Web engineering, in general, and Web analysis, Web testing and Web re-structuring, in particular, is also explained. Some factors contributing to quality of Web applications are listed at the end of the chapter.

2.1 Introduction

The aim of this chapter is to clarify some terms used in the rest of the thesis. Terms such as Web application, Web site, Web engineering, Web testing and Web analysis are used in different context with slightly different meanings. Not all researchers agree on the set of quality factors of a Web application and different lists have been proposed in literature (for example [Con00, Mil98, Pow98]).

The chapter is organized as follows: Section 2.2 tries to answer the question what Web applications are and gives an overview of the overall Web application architecture. Section 2.3 focuses on Web engineering in general and analysis, testing and re-structuring of Web applications in particular. The last section lists the quality factors of a Web application as proposed in [Pow98].

2.2 What is a Web application?

A Web application is a special case of application, designed to be executed in a Web-based environment. More precisely a (dynamic) Web application is a mix of programs, that

dynamically generate hyper-documents (dynamic Web pages) in response to some input from the user, and static hyper-documents.

A (static) Web site is simply a collections of static hyper-documents (static Web pages).

2.2.1 Web application architecture

Figure 2.1: *Web application architecture.*

While a Web application is basically a program running on a Web server and a set of fixed Web pages, there is much more to be considered in the activities of Web analysis, Web testing and Web re-structuring. The behavior and the quality of a Web application depend on all its components.

Web applications contain many components that are linked together to deliver the functionality of the application. A typical generic Web application architecture is shown in Figure 2.1 (a similar schema is proposed in [ZK99]). Web applications are based on the client/server model: a browser (client) sends a request asking for a Web page over a network (Internet, via the protocol HTTP) to a Web server (server), which returns the requested page as response. Web pages can be static or dynamic. While the content of a static page is fixed and stored in a repository, the content of a dynamic page is computed at run-time by the *application server* and may depend on the information provided by the user (a similar distinction is proposed in [Con00] and [Eic99]). The programs that generate dynamic pages at run-time (called *server programs*), as for example CGI (Common Gateway Interface) scripts and servlets, run on the application server and can use information stored in databases and other resources.

HTML language.
Web pages are written principally in HTML language (HTML stands for the HyperText Markup Language). A Web page consists of text, which is an unstructured sequence of characters, and HTML elements. Elements are bracketed by a start-tag and an end-tag notation. This is an HTML element:

$< b > bbbbb < /b >$

Where the HTML element starts with a start tag (), the content of the HTML element is "bbbbb" and the HTML element ends with an end tag ().

The power of HTML is in its links to other HTML resources. In specific tags of an HTML document (for example the anchor <a> tag), the URL (Uniform Resource Locator) of another HTML document can be accessed. If the user clicks the mouse button on the text of an anchor element, the browser automatically retrieves a new Web page and then displays it. The language HTML is defined precisely in [Gro99].

Client/server interactions.
The HTML code can activate the execution of a server program by means of a `SUBMIT` input within an HTML element of type `FORM` or anchor. Data flows from a server program to the HTML code are achieved by embedding values of variables inside the HTML code, as the values of the attributes of some HTML elements. In the opposite direction, the basic mechanism for data value propagation is by means of form parameters. *Hidden* parameters are constant values that are just transmitted to the server (possibly recording the values of some previous computation), while non hidden input parameters are gathered from the user. Parameter passing is strictly by value and the invocation of the server program is a control transfer without return.

Server programs can exploit persistent storage devices (such as databases) to record values and to retrieve data necessary for the construction of the HTML page. Moreover, session specific data (*session variables*) can be stored at the server side and maintained across successive executions. *Cookies* act similarly to session variables, with the only difference of being stored at the client side instead of the server. Such data are used to identify the ongoing interaction and to record data that need to survive past the end of execution of server programs.

Finally, more dynamism can be included in a Web application by means of *client side code*, which can be either executed during the loading of the HTML page inside the browser, or in response to a graphical interface event (e.g., mouse click). In the latter case, callbacks are installed by declaring that they are reacting to a graphical event on a given HTML element. Some HTML attributes (e.g., `ONCLICK, ONBLUR`) are available for this purpose. Then, the occurrence of the graphical event triggers their execution, similarly to the graphical user interfaces of traditional software. An event loop dispatching the events to the respective callbacks is in fact implicitly activated. The client code (possibly executed in response to

an interface event) can read and write the values of the attributes of the HTML elements, by exploiting the *Document Object Model* (DOM), giving a standard interface to them. For example, the access path `document.Form0.x.value` can be used to access the value of variable `x` inside form `Form0`. This represents the basic data flow mechanism between HTML code and client code.

While usually callbacks return the control to the event loop after completing their execution, when the related interface event is a click on a `SUBMIT/A` HTML element associated with a server program, control is transferred to the server program and never returns back to the event loop. In other words, `SUBMIT/A` calls are no-return transfers of control. The typical usage of client side callbacks is form validation and computation of field values (e.g., conversions, totals, etc.). Instead of transferring any values entered by the user to the server programs, only valid values are transferred, thus delegating the validity check to a piece of code running on the client.

Several server side languages are available for the construction of a Web application (e.g., PHP, Java, Perl, VBscript, etc.). The same is true for the client side code (e.g., Java, Javascript, etc.).

2.3 What is Web engineering?

In 1998 the term *Web engineering* has been coined and efforts have been made to establish a new discipline to cover the life cycle of Web applications. Contrary to the general perception, Web engineering is not a clone of software engineering although both involve programming and software development [GM01]. While Web engineering adopts many software engineering principles, it incorporates new approaches and guidelines to meet the unique requirements of Web-based systems [GM01]. Web development is a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology [Pow98]. Building a complex Web applications calls for knowledge and expertise from many different disciplines such as: software engineering, hypermedia and hypertext engineering, human-computer interaction, information engineering and user interface development [GM01].

Web application process models.
Many process models are described in the Web engineering literature. In the book [Con00] the author considers as candidate process model for the development of Web applications an *incremental/iterative process model*. He lists some peculiar characteristics of Web applications which motivate the choice of an incremental/iterative process model. They are:

- Web applications have short delivery times.

- Web applications are subject to a tremendous pressure for change.
- Turn over of Web application developers is high.
- Complexity of Web applications is increasing.
- User needs evolve quickly.

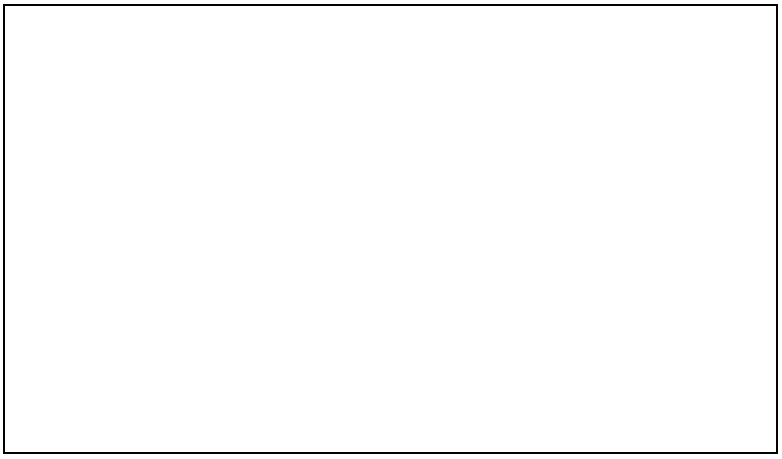

Figure 2.2: *Incremental/iterative process model.*

An incremental/iterative process model (see Figure 2.2) assumes that functionalities are delivered to the user incrementally, by traversing several times the different development phases. At each iteration an increment is considered (delta), which is sufficiently small to be completely implemented in a short time interval. After each iteration a complete Web application is delivered, which improves over the previous version because some functionalities are added, changed or enhanced.

This process model allows short delivery times and can easily incorporate the change requests coming from the user or from the support technologies. New business opportunities can also push for major changes in the application, which can be achieved by iterating several times through a sequence of finer grain modifications.

While it limits the overhead that may delay the production of a working release, this process model contains all elements necessary to approach the higher complexity of new Web applications. It assumes the existence of requirements and design, so that a new developer is not left only with the code of the application, but can find useful documentation about it. It also attacks the problem of ensuring the needed reliability of these applications by formalizing a testing phase.

In [RT02a] we have proposed another incremental/iterative process model for development of Web Applications. The most distinctive feature of this process is the central role of the

model of the Web application. Such a model is updated each time a design increment is introduced and is exploited for change assessment each time a requirement increment has to be mapped into the design. In fact, it allows estimating the portion of the application affected by the change, thus supporting the allocation of the change to a proper iteration and, if needed, the split of the change into smaller pieces. When moving to the implementation, a model of the "as is" system can be retrieved from the code by means of reverse engineering, and can be used to evaluate the impact of the modification in terms of the code portions to be updated. Developers are expected to benefit from the availability of a reference model of the application, since their main activities during the implementation of an increment are the understanding of the existing system and of the portions to be modified. Then, they have to implement the changes and avoid undesirable ripple effects. These activities become particularly difficult in a situation of high turn over, where the persons who developed the system are no longer available and new programmers are charged of understanding and changing it. Reverse engineering and impact analysis can assist programmers in this development phase and a high level of automation can be reached if a proper model of Web applications is adopted.

2.3.1 Web analysis

Web analysis is the automated inspection of a Web application to infer some property. The purpose of (static) Web analysis is to recover information about a Web application without "running" the application. Goals of Web analysis are similar to those of software analysis while the objects subject to analysis can be considered a superset. In the case of software the object of analysis is only the code, in the case of Web applications the objects of analysis are: the hypertext, the log file, the code (server and client side) and the multimedia objects. Techniques of Web analysis can be used, for example, to recover abstract views, to check whether the Web application satisfies suitable constraints and to detect possible anomalies.

2.3.2 Web testing

The general definitions used in the context of software testing can be applied with success also to Web applications. The goal of Web testing is the same of software testing: *to detect as many failures as possible*; where a *failure* can be considered as a deviation from the expected behavior. What is different, are the test objects and their features. The objects subject to testing in the case of Web applications are increasingly more multifaceted than they may be in conventional testing. Traditional testing can be considered as a fraction of Web testing, in fact in a Web application there are more key areas of testing beyond software testing [GM01]. Some of them are: Web user interface, hyperlink structure,

configuration and compatibility, security, databases, performance, load and stress.

2.3.3 Web re-structuring

In [CI90] code re-structuring is defined as a transformation from one form of representation to another at the same relative level of abstraction. The new representation is meant to preserve the semantics and external behavior of the original. Martin Fowler in [Fow99] defines refactoring (i.e., a particular kind of re-structuring) as: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. In this thesis the term Web re-structuring is meant as a transformation that aims at improving certain quality factors of a application without changing the information provided and the external behavior of the Web application.

2.4 Quality factors

For traditional software systems a number of quality factors are well known. In the hierarchical model ISO 9126 [fS91], a standard for measuring software quality, six factors contributing to quality of software systems are listed. They are functionality, reliability, usability, efficiency, maintainability and portability. Powell in the book [Pow98] proposes similar factors for Web applications. Web applications should be [Pow98]:

- **Correct** – a Web application is correct if it performs properly and is functionally and cosmetically error free;
- **Testable** – the Web application is simple to test;
- **Maintainable** – it is important that Web applications can easily be understood and extended;
- **Portable** – the Web application performs across platforms and is cross browser safe;
- **Scalable** – scalability is the system's ability to gracefully increase its capacity in order to accommodate future growth;
- **Reusable** – reusability is the degree to which a software module or other work product can be used in more than one Web application or software system;
- **Reliability** – reliability is defined as the probability of correct behavior within a given time interval;
- **Efficient** – the response time for the requested Web page is short;

- **Readable** – programming conventions should make sure that the source code (server and client side) and Web pages are readable;
- **Well Documented** – at the code level this means that the code and HTML pages should contain comments. At system level that system architecture, design and test documents should be documented (with diagrams, models, textual representations etc.);
- **Appropriately presented** – the presentation of information and services is very important in Web applications. It has to satisfy marketing objectives and usability issues.

Part I

MODEL

Chapter 3

Web Application Modeling

This chapter describes a Web application model able to support analysis and testing techniques proposed in the thesis. The presented model can be semi-automatically extracted from an existing Web application, by exploiting the algorithms detailed in this chapter. These algorithms are implemented in one module of the tool ReWeb, called Spider.

3.1 Introduction

The first step toward analysis, testing and re-structuring is the definition of a model representing the various entities involved in Web applications and their mutual relationships. The models proposed in literature (for example that proposed by Conallen [Con00]) usually aim at describing the Web application from a logical point of view at a high level of abstractions, as required when the application is being designed. On the contrary, we focus our model on the implementation of the site, which is the starting point for analysis and testing. In particular, we are interested in a Web applications model satisfying the following requirements:

- Emphasis is set on the navigational features of the Web application, i.e. on the relation among pages and links and not on the content of the pages;
- The model should be complete, i.e. the most important entities (as, for example, links, frames, forms and dynamic pages) should be explicitly represented in the model;
- It should be possible to provide (at least partially) automatic support for its extraction;

- It should be possible to apply to it some static analyses and testing techniques derived from those used with traditional software systems.

Modeling a static Web site – collections of static hyper-documents (static pages) encoded in the HTML language – is a simple task. The structure of a static Web site without frames can be easily modeled as a directed graph, where each node represents a single HTML page and an edge connects two nodes *n1, n2* if there is an HTML link from the page associated to *n1* to the page associated to *n2*. Some difficulties in the modeling are introduced by frames. In fact the basic model has to be extended to account for the composite structure of pages with frames. Also in this case the extraction of a model of a Web site is not a difficult task, it can be easily obtained by statically analyzing the HTML code composing the Web site.

Problems arise when we try to consider modeling and model extraction of dynamic Web applications – mix of server side programs, that dynamically generate hyper-documents (dynamic pages) in response to some input from the user, and static documents encoded in HTML. Modeling and model extraction are complicated by the presence of dynamic generation of the HTML code. In fact in this case the model and the internal organization (links, forms, frames etc.) of the pages are not fixed, being produced at run-time by server programs, and may depend on the user inputs.

Static analysis of highly dynamic Web applications is a difficult task. While in the simplest cases a fixed HTML skeleton is filled-in with values computed dynamically, in more complex applications even the structure of the resulting HTML page is not given a priori, and is constructed dynamically. In such a situation, a static analysis of the server programs generating the Web pages can hardly result in a useful model of the application. In fact, the problem of determining the HTML code produced by a server program is related to the problem of determining if a given execution path is feasible, which is known to be an undecidable problem. Moreover, a Web application involves several programming languages. On the server side, at least one programming language is used for the dynamic production of the HTML pages (e.g., PHP, Java, Perl, VBscript, etc.). If databases are accessed, a related query language, such as SQL, is also present. HTML statements are then generated, but typically they are not pure HTML code, and include client side code for form validation, client side computation, and graphical event handling (e.g., Javascript, Java applets, etc.). Static analysis of such a variety of languages – and of all their possible interactions – is a technological challenge.

In this chapter, we propose a technique for the extraction of a UML (Unified Modeling Language [BRJ98]) model of a Web application, obtained by statically analyzing the HTML code that is dynamically generated by the server programs. The analysis technique pro-

posed can be considered dynamic[1] in that it needs the execution of the Web application. More precisely, input values which cover all relevant navigations are pre-specified by the user, and downloaded pages are either unrolled or merged, in order to produce an abstraction over the set of HTML pages downloaded. The resulting model is computable in presence of high – even "extreme" – dynamism and requires the ability to parse just HTML. The problem of statically approximating the HTML code being generated is absent. On the other side, the model obtained may be partial, if the inputs used to produce it do not cover all relevant behaviors of the Web application. This is an intrinsic limitation of all dynamic analyses.

The chapter is organized as follows: section 3.2 presents the Web application model used throughout of the thesis. This model, that is the starting point for analysis and testing, is described in details in 3.2.1. The proposed model can be semi-automatically extracted from an existing Web application, by exploiting the algorithms detailed in 3.2.2. Two examples proposed in 3.2.3 and in 3.2.4 clarify the model in case of Web applications with frames and highly dynamic Web applications. In section 3.3 the model of a real world Web application is extracted using the technique explained in the section 3.2. Related works on Web application modeling concludes the chapter.

3.2 Web application modeling

The aim of the Web application model is that of describing a Web application in terms of composing pages and allowed navigation links. Both dynamic and static pages are to be properly modeled. Dynamic pages are the result of executing a program on the Web server in response to a request from the Web browser of the user. Important interactive features that are exploited by Web applications, like forms and frames, should be part of the model, being relevant to the navigation in the Web application. All paths between entities (i.e. static pages, dynamic pages, forms and frames) of the model can always be interpreted as navigation paths and represent a meaningful interaction of the user with the Web application. Other aspects as, for example, the logical structure of the underlying data base or the code which generates dynamic pages at run time, are considered as black boxes (except for slicing, chapter 5) and are reflected in the model only indirectly, in terms of the pages and links that are generated.

In the following, a Web application is identified with the set of pages that can be accessed from a given Web server. Documents accessed from different servers are considered external to the given application. Analysis and testing of a Web application as well as the extraction

[1] In literature dynamic analysis of traditional software is the analysis of the properties of a running program. In contrast to static analysis, which examines a program's text to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions.

of its model are supposed to be conducted in the development environment. Consequently, some information that cannot be accessed by external users browsing the site is considered available.

3.2.1 The model

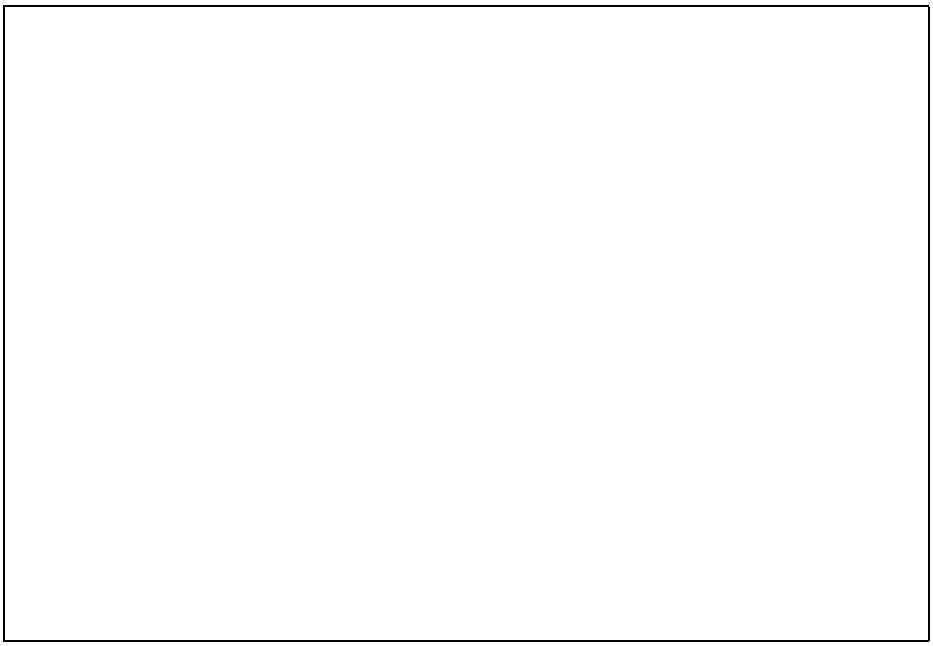

Figure 3.1: *Meta model of a generic Web application structure. The model of a given application is an instance of it.*

Figure 3.1 shows the meta model used to describe a generic Web application. The central entity in a Web application is the *HTMLPage*. An HTML page contains the information to be displayed to the user, and the navigation links toward other pages. It also includes organization and interaction facilities (e.g., frames and forms). Its URL is recorded in the attribute *url*. Navigation from page to page is modeled by the auto-association of class HTMLPage named *link*. Web pages can be static or dynamic. While the content of a *static* Web page is fixed, the content of a *dynamic* page is computed at run time by the server and may depend on the information provided by the user through input fields. The boolean flag *isDynamic* distinguishes the two cases. The class *ServerProgram* models the script/executable that runs on the server side and generates a dynamic HTML output. When the content of a dynamic page depends on the values of a set of input variables, the attribute *use* of class *ServerProgram* contains them. A server side program can be executed by traversing a *link* from an HTML page whose target is the server script/executable and whose attributes include a set of parameters, represented as pairs <*name, value*> or by

submitting a form. The server program can either redirect the request to another server program (auto-association *redirect*), build an output, dynamic HTML page (association *build*), or simply redirect to a static HTML page (association *redirect*). The latter two cases can be distinguished only because the resulting HTML page is respectively static or dynamic. When a server program builds a dynamic page, the input and hidden variable values that have been provided to it are stored in the attributes *input* and *hidden* of the resulting page, as sets of couples <*name, value*>. Field *altInput* stores alternative inputs that generate the same dynamic page (see page merging below).

A *frame* is a rectangular area in the currently displayed page where navigation can take place independently. Moreover, the different frames into which a page is decomposed can interact with each other, since a link in a page loaded into a frame can force the loading of another page into a different frame. This can be achieved by adding a target to the hyperlink. Organization into frames is represented by the association *split into*, whose target is a set of *Frame* entities. Frame subdivision may be recursive (auto-association *split into* within class *Frame*), and each frame has a unary association with the Web page initially loaded into the frame (absent in case of recursive subdivision into frames). When a link in a Web page forces the loading of another page into a different frame, the target frame becomes the data member of the (optional) association class *LoadPageIntoFrame*.

In HTML user input is gathered by exploiting a *Form* and is passed to a server program, which processes it, through a *submit* link. A Web page can include any number of forms; accordingly, the cardinality of this link is arbitrary. Each form is characterized by the input variables that are provided by the user through it (data member *input*). Additional *hidden* variables are exploited to record the state of the interaction. They allow transmitting pairs of the type <*name, value*> from page to page. Typically, the constant value they are assigned needs be preserved during the interactive session for successive usage. Since the HTTP protocol is stateless, this is the basic mechanism used to record the interaction state (variants are represented by the cookies or in certain languages by session variables).

In a Web application, the same server program may behave differently, according to the interaction state. To clarify this situation it is convenient to classify server programs into two categories:

- Server programs with state-independent behavior.
- Server programs with state-dependent behavior.

Server programs in the first category always exploit the same mechanism to produce the output (either auto-redirect, redirect or build) and generate a dynamic page whose structure and links are fixed. The behavior of these server programs is the same in every interaction state. On the contrary, server programs in the second category behave differently when executed under different conditions. A server program may, for example,

provide two completely different computations – and consequently different output pages – according to the value of a hidden flag recording a previous user selection.

In presence of server programs with state-dependent behavior, the paths in the model can still be interpreted as navigation sessions, provided that server program and related output page are replicated for all the possible variants. We call the resulting model an *explicit-state model*, differing from the alternative one, called *implicit-state model*, in that it unrolls server programs and dynamic pages with different behaviors into actually different entities, which are given a progressive identification number (*counter* in classes *HTMLPage* and *ServerProgram*). In this way, the page identity is not associated to a physical entity (page or server program), but is rather differentiated according to the behavior.

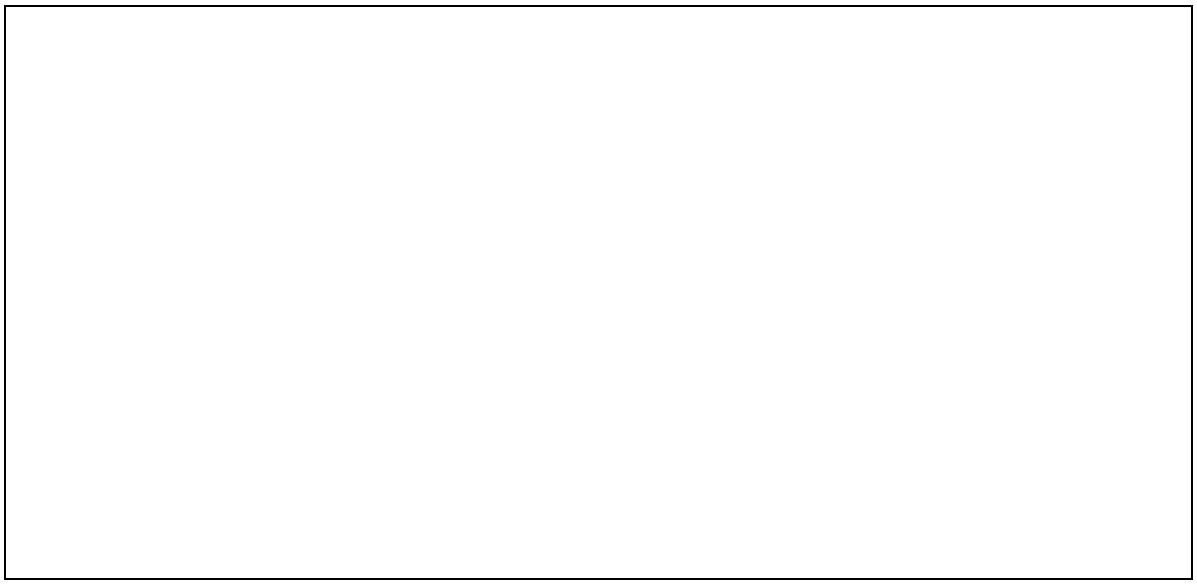

Figure 3.2: *Example of Web application model with implicit (left) and explicit (right) state. Nodes with grey background are server programs. Input_variables (with optional values) and hidden_variables (with values) are separated by ";".*

Figure 3.2 shows an example of Web application for which both implicit-state and explicit-state models are given. The application consists of an initial static page H (note that the name is underlined to indicate that it is a UML object and not a class), from which the user can navigate to a server program S through a link associated with a parameter, `state`, which is assigned the constant value 1. S builds a dynamic HTML page, the content of which depends on the value of variable `state` which is received by S. In particular, with `state = 1`, S builds a page containing one form which collects the values of variables `x` and `y` and transmits a value of `state` equals to 2 as a hidden variable. This is represented in the implicit-state model (left of Figure 3.2) as a *submit* link guarded by the condition `(state=1)`. Such a link is generated inside page D only when S receives a value of `state`

equals to 1. Then, the server program S is invoked for the second time, now with `state = 2`. The behavior in this situation is different from the previous one, and the output page contains two new forms, respectively devoted to collecting the values of `a` and of `b, c`, while it does not contain the previous form. This is the reason for the two *submit* links guarded by the condition `(state=2)`. Finally, the server program S is executed again, either by the first active form (gathering `a` as input) or from the second one (gathering `b` and `c`). The result of this execution is still different and the dynamic page D that is built now does not contain any form (`state` is equal to 3 and therefore all conditions are false). Its content varies also in the two cases where either `a` or `b, c` are filled in by the user. The explicit-state model of this example of Web application is provided on the right of Figure 3.2. The server program S and the dynamic page D have been split into 4 pages, associated to the 4 different behaviors that may occur during an interaction, corresponding respectively to `state = 1`, `state = 2`, `state = 3` and `a` gathered, and `state = 3` and `b, c` gathered. No condition has to be attached to the edges of this model, since all condition-dependent behaviors have been separated explicitly.

In order to define a set of analysis and testing techniques working on Web applications, it is convenient to re-interpret the model described above as a graph (*structural view*), whose nodes correspond to the objects in the model and whose edges correspond to the associations between objects. Labeled edges are used for the links having a *LoadPageIntoFrame, Form*, or *Parameter* relation specifier. Since the adopted model is aimed at representing navigation explicitly, the paths in the associated graph can be regarded as interaction sessions in which the user navigates inside the site, provides input data through forms and receives the results back through dynamic pages.

3.2.2 Dynamic model recovery

One modules of the reverse engineering tool **ReWeb**, called *Spider*, is responsible for the automatic extraction of the explicit-state model of a target Web application. The main difficulty in this operation is differentiating a same server program according to the different behaviors it may exhibit. Moreover, user input has to be simulated, so that dynamic pages be generated and navigation can proceed beyond the purely static part of the Web site. To accommodate the latter issue, a set of input values are specified before running the *Spider*, granting the traversal of all relevant site portions. Such values are used by the *Spider*, which provides them to the server programs during the downloading of the site pages. Among the hidden and input values that are used by the *Spider* for the download, some may affect the internal state of the Web application, and consequently the behavior of the server program being invoked. Formally, the domain of input and hidden variables can be partitioned into equivalence classes, and the same behavior is expected to be obtained when the inputs belong to a same equivalence class. The input values provided to the

Spider should be comprehensive enough as to cover all different behaviors of the server programs.

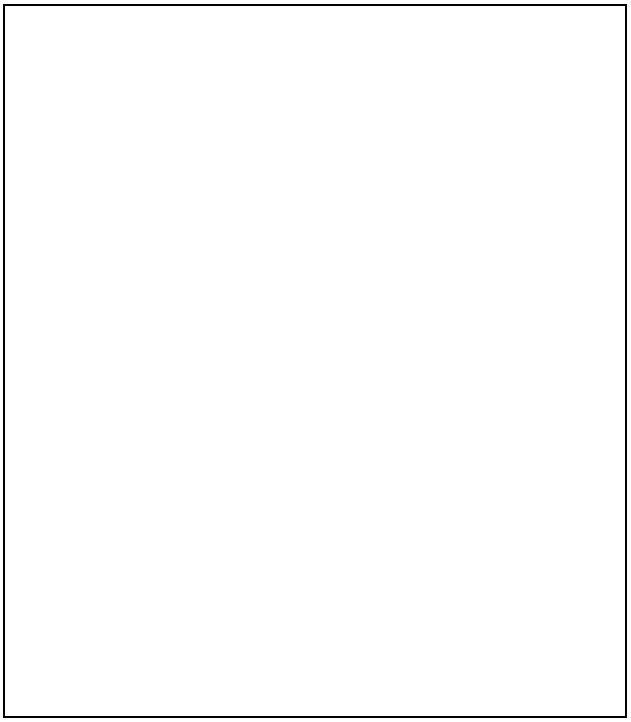

Figure 3.3: *Pseudocode of the SPIDER procedure, that downloads a target Web application and builds the related UML model.*

Figure 3.3 contains the pseudocode of the *Spider* module. A list L is maintained with all the pages still to be visited. Each page in the site is considered in the body of the most external loop. If the page is static, it is just downloaded (line 12), in case it has not been visited previously. Then, hyperlinks are examined within the downloaded page (line 14, *p.scanPage()*) and the referenced pages are added to L. The UML model is also updated in this phase (line 17). If the page contains forms, the related dynamic pages are downloaded (line 21) by means of a specific procedure, *DownloadDynamicPage*, described below. A dynamic page is considered to be already in the explicit state model (line 23) if its URL is the same of another page in the model and if it was obtained by passing input and hidden values to the server program which belong to the same equivalence class, i.e., which are associated to the same behavior of the server program. In this case, the dynamic page is not added to L and its input *d.input* is considered an alternative input for the page (q) already in the UML model. Otherwise, it is inserted into L for successive visit (line 29). When the equivalence classes characterizing the behavior of the server programs are not known a priori, some page merging heuristics, discussed below, can be employed to try to automatically recognize equivalent dynamic pages and thus equivalent instances of the

related server programs.

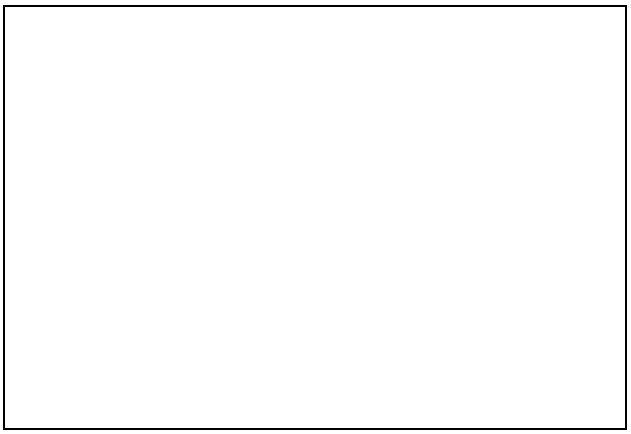

Figure 3.4: *Pseudocode of the* DownloadDynamicPage *procedure, that downloads a dynamic page providing proper input values to the server program.*

The procedure *DownloadDynamicPage*, sketched in Figure 3.4, extracts the input to be inserted into a given form from a persistent store (e.g., the text file `formInputFile.txt`), by looking for lines with a matching action (instr. 2: `line=f.action, ...`), and builds an object of type *HTMLPage*, inserted into the returned list D, for successive page scan. Such an object has the form *action* as URL, and the user specified inputs as *input* values. It can be noted that hidden values found in the form f are required to be the same as those specified in `formInputFile.txt` (instr. 3). This condition is necessary to ensure that inputs are used in the correct interaction state: if the same server program is activated in a different state, the input line from `formInputFile.txt` cannot be used. This is easily detected since the related hidden variables have different values.

Different inputs specified by the user may belong to a common equivalence class, being associated with the same behavior of the server program, or the same behavior may be obtained (and the same state may be reached) for different values of the hidden variables. This knowledge may not be available a priori. In such cases, an additional operation of page merging is required to unify equivalent instances of the server programs. Three increasingly weaker page merging heuristic criteria can be used to simplify the explicit state model downloaded by the *Spider*:

1. Dynamic and static pages that are identical according to a character-by-character comparison are considered the same page in the model.

2. Dynamic pages that have identical structure, but different texts, according to a comparison of the syntax trees[2] of the pages, are considered the same in the model.

[2]A compact representation resulting from the parsing of the page.

3. Dynamic pages that have similar structure, according to a similarity metric, such as the tree edit distance[3], computed on the syntax trees of the pages, are considered the same in the model.

While moving from criterion 1 to 3, the intervention of the user to validate the automatically identified page merges becomes increasingly important, since possible errors become more and more likely. Page merging is an integral part of the *Spider* procedure, to avoid the construction of multiple copies of a conceptually same page. This contributes to producing a finite model. In fact, a page already in the model may be downloaded an infinite number of times if not merged with the corresponding one in the model. With reference to Figure 3.3, page merging is checked at line 23, where potential page unification actions are detected by the *containsDynamic* method. In principle, the *Spider* procedure is not guaranteed to terminate, because server program nodes could be replicated an infinite number of times in correspondence with the occurrence of states that are not recognized as already encountered before. In practice, the usage of the page merging heuristics and the careful selection of the input values, as representative of the equivalence classes of inputs, results (according to the author's experience) in a finite model.

3.2.3 Example of model with frames

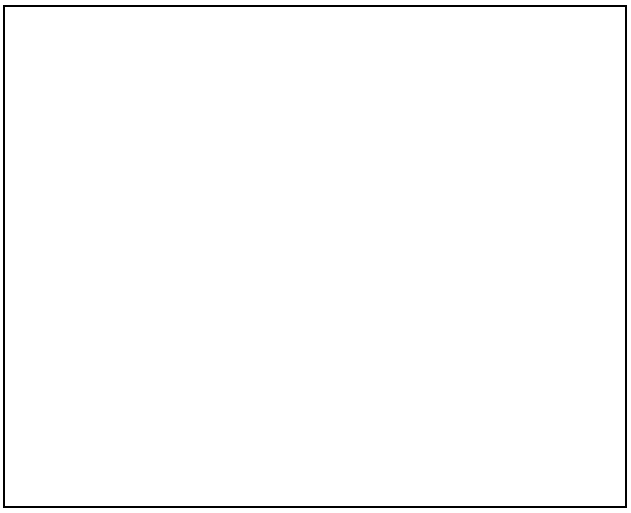

Figure 3.5: *Example of model with frames.*

Figure 3.5 shows an example of model for a web application with frames. The links between `p3` and `p5`, and between `p4` and `p5` are normal navigation connections between

[3]The smallest number of insertions, deletions, and substitutions required to change one tree into another (see 9.2.2).

HTML pages. Page `p1` is decomposed into the two frames `f1` and `f2`. The links between `f1` and `p2` and between `f2` and `p3` indicate that the pages initially loaded into `f1` and `f2` are respectively `p2` and `p3`. Finally, the association connecting `p2` to `p4` has an attribute specifying that the link in `p2` does not result in the navigation within `f1` toward a different page, but rather produces the loading of page `p4` into frame `f2`, with no regard to the page currently loaded into `f2`.

3.2.4 Example of model with dynamic pages

Let us consider a simple financial Web application, the model of which is shown in Figure 3.6. This application provides two different services, related to the stock market and to the exchange rates. A single server program S provides both services. The home page of this application, H, is a static page containing some descriptive material and two links to the two services. If the user is interested in the stock market, a form gathers the name of the corporation of interest as an input value, while the name of the currency is gathered in the form leading to the exchange rates service. The dynamic page generated in response to the user request allows repeating the request, specifying a different corporation within the stock market service, or a different currency within the exchange rates service. However, when a service type has been selected, it is not possible to switch to the other one without restarting from the home page. In other words, the state of the interaction remains fixed after the initial selection.

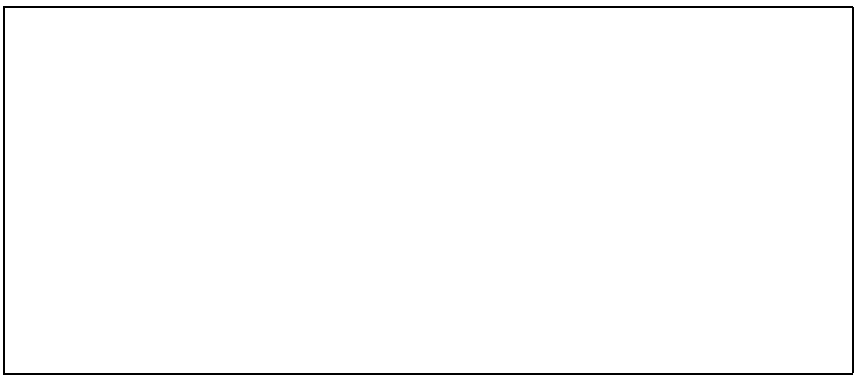

Figure 3.6: *Example of Web application model. Nodes with grey background are server programs. Input_variables and hidden_variables (with values) are separated by ";".*

In the model of this Web application the initial static page H is connected to two replications of the server program S, S_1 and S_2, corresponding to the two different behaviors that characterize it. Correspondingly two different dynamic pages are considered to be built in the two different interaction states, namely D_1 and D_2. When the *submit* association from H to S with `state=1` is followed, the input variable `corporation` is passed to the server

program, with the value provided by the user. To simplify the plot, the association class *Form* is not shown explicitly and its fields label the association edge. When the user selects the second service, the hidden variable `state=2` is propagated through the form, together with the input variable `currency`. The two target objects, S_1 and S_2, respectively build the two dynamic pages D_1 and D_2, which display the requested information and give the user the possibility to provide an alternative `corporation`/`currency` in input and see the related information, obtained by the same server program S. The hidden variable `state` is propagated unchanged to the server program S_1 from D_1, and to S_2 from D_2.

Figure 3.7: *Example of input file for the* Spider.

An example of input file to be used for the extraction of the model in Figure 3.6 is shown in Figure 3.7. The *Spider* module (see Figure 3.3) starts its computation by adding the initial page H to the list L of pages to download (*target_page* in *Spider* is H). Then, page H is extracted from L and downloaded (it is not a dynamic page). The *scanPage* and *retriveForms* methods scan its content, giving the set of referenced pages and of contained forms. In our case, two forms are retrieved. The procedure *DownloadDynamicPage* (see Figure 3.4) is invoked on both. The first form has a hidden variable `state=1`, so that only the first two lines of `formInputFile.txt`, shown in Figure 3.7,match the condition of having the same server program as action (S) and the same values of the hidden variables. When the first line is processed, the *Spider* requests the execution of the program S on the Web server with hidden variable `state=1` and input variable `corporation="atd"`. The resulting dynamic page is downloaded by the *Spider* and added to the list of downloaded dynamic pages D. A second dynamic page is generated for the second line of `formInputFile.txt`, with the same hidden variable value and `corporation="zpm"`. Such two dynamic pages are iteratively taken into consideration at line 22 of the *Spider* procedure. While the first page is surely *not* contained in the model, and it is therefore added to L and to the model, the second is already in the model, since the same behavior of S was already observed in correspondence with the first input line used. One possibility to recognize that the second page is equivalent to the first one is to know a priori that variable `state` determines the internal state of S and partitions the input domain into equivalence classes. As a consequence, the second page is unified with the first one, belonging to the same equivalence class. If this knowledge is not available, the two pages can be compared to recognize the possibility of merging them. The first merging criterion is expected to fail: since the two pages report data of two different corporations ("atd" and "zpm"), they are expected not to be textually equal. However, the second merging criterion is expected to succeed, in that a common page structure is expected to be used to provide the same kind of information

– even if referred to different corporations. The input used for the second dynamic page is stored as an alternative input for the page previously inserted into the model. The execution of *DownloadDynamicPage* on the second form of the home page gives two dynamic pages that are unified in a similar way.

After the first iteration of the loop at line 2 in the *Spider* procedure, the list L of model elements still to be considered contains D_1 and D_2, the two dynamic pages generated by S with the inputs taken from the first and the third line of `formInputFile.txt`. Both pages contain one form. The form of D_1 has a hidden variable `state=1`. Therefore, for the download of the associated dynamic page, lines 1 and 2 of `formInputFile.txt` can be used. The first resulting page is identical to D_1, while the second one has the same structure. As a consequence, both of them can be merged with D_1 (merging criteria 1 and 2 respectively). A priori knowledge about the possibility to discriminate the equivalence classes of inputs according to the value of `state` would simplify this task. The result is an association labeled *submit* from D_1 to S_1. Similarly, the input lines to be used for the form in D_2 are the third and fourth lines of `formInputFile.txt` and the dynamic pages obtained by form submission can be unified with D_2, giving rise to a link between D_2 and S_2. The final model produced by the *Spider* is exactly the one reported in Figure 3.6.

3.3 Case study

The Web application **ITC-publications** (`http://publications.itc.it/`), providing the on-line publications database of ITC-irst has been chosen as case study for explaining model extraction (other examples of model extraction are in [RT01a] and [RT01b]), structural testing (section 6.4) and statistical testing (section 7.5). The richness of its dynamic structure makes it an interesting case study for modeling and testing. Papers in the ITC-publications database can be accessed directly from the initial page of the site, `db.htm`, by means of three different forms collecting user inputs. The first form permits a search by **author**, **title**, **reference number** and **abstract**. The second permits a free text search, referred to all document fields (input variable **query**), while the third shows links to all the publications available in the database. After submitting the first form, filled-in with correct inputs (for example with abstract `slicing` or author `ricca` and title `web`), the dynamic page `ext-search.idc` appears, showing the list of papers matching the inputs inserted (for each paper, a button, implemented in HTML as a form containing a hidden variable with the reference number, is displayed in the Web page). In case of selection of a paper, the dynamic page `ext-view.idc` is loaded. This page contains details on the chosen paper, such as abstract and complete list of authors. It permits returning to the initial page and displaying a help page (`help.htm`). The results of submitting the second and third form are respectively the dynamic pages `ext-search2.idc` and `ext-search3.idc`. The structure of these pages is very similar to `ext-search.idc`. Like `ext-search.idc`,

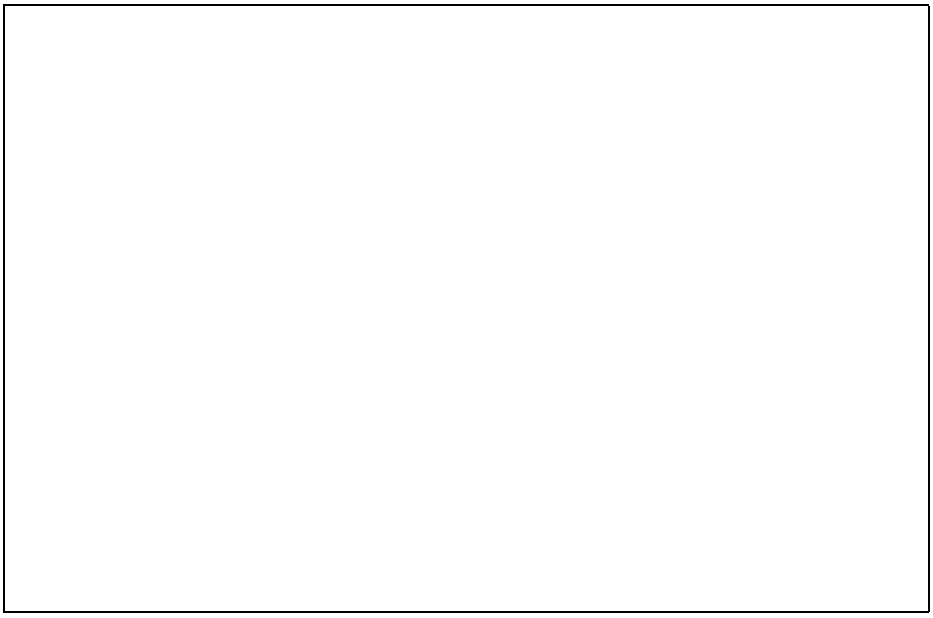

Figure 3.8: *Structural view of the Web application ITC-publications (nodes with grey background are server programs, labeled edges are used for the links having a* `Form` *relation specifier. The lists of couples input_variable=value and hidden_variable=value are separated by ";". The symbol "-" represents the empty list).*

they show the list of papers from which any one can be selected, resulting in the generation of the dynamic page `ext-view.idc`. In case the input values inserted into any of the three forms are not correct, the result is a blank page and the only way to return to input insertion is using the back button of the browser and going to the initial page `db.htm`.

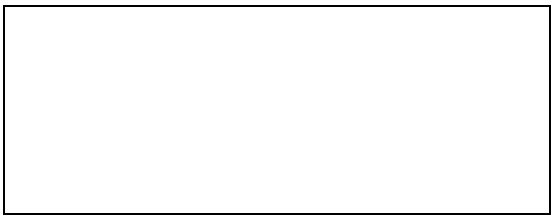

Figure 3.9: *File* `formInputFile.txt`*, used to download the Web application* **ITC-publications**.

Figure 3.8 shows the structural view of the Web application **ITC-publications** as recovered by **ReWeb** with the inputs indicated in Figure 3.9.

During the download of **ITC-publications**, three operations of page merging have been applied. Pages `/publications/html/db.htm` and `/dbp/html/db.htm` are identical according to a char-by-char comparison, thus they are considered the same page in the model

(page merging criterion number 1). The same happens for pages `/publications/html-/help.htm` and `/dbp/html/help.htm`. An operation of page merging of type 3 (similar structure) has been applied to the dynamic page `ext-search.idc`. Three inputs specified by the user belong to a common equivalence class (`author=ricca & title=web`, `abstract=slicing`, and `ref_number=0102-01`, see figure 3.9). The resulting pages have a similar structure, thus suggesting that the server program generated them in a similar internal state. Actually, these three dynamic pages are produced in exactly the same way by the server program, although under slightly different inputs.

3.4 Related works

Various Web application modeling methods have been proposed for different purposes, such as design, architecture recovery, re-structuring and testing. Each method emphasizes some particular aspect of a Web application. Many works have been dedicated to the problem of Web application design. They include languages, models and graphical notations useful for the specification of Web applications.

Among the design models [BN96, CFB00, Con00, IKK97] which have been proposed in support to Web development, those conceived for the specification of the navigation and presentation structure of the site [BN96, Con00] are closer to ours. Higher level abstractions (e.g., the entity relationship diagram of the RMM methodology [IKK97] and the structural model of WebML [CFB00]) are out of the scope of our current investigation.

The Web application model closer to ours is that proposed by Conallen [Con00]. Web pages are considered first-class elements, and are represented as objects, using UML. Similarly, all other architecturally relevant entities as, for example, links, frames, and forms, are explicitly indicated in the model. The main difference between Conallen's and our UML model of Web applications is in the emphasis given to design vs. analysis. In fact, the model by Conallen aims at describing the site from a logical point of view, as required when it is being designed. On the other side, we focused our model on the implementation of the site, which is the starting point for analysis and testing, and on the navigational features of the site. An example of this difference is the representation of frames. In Conallen's model a frameset contains two aggregations: an aggregation of targets (frame names) and an aggregation of the pages initially loaded into the different frames. On the contrary, in our model only the former aggregation, between a page and the enclosed frames, is retained, and in turn each frame is linked to the page loaded initially. In such a representation, each link sequence is a navigation path (with possibly some links traversed automatically), while in Conallen's one some links are used to express properties (namely, the list of available targets). Another difference is that UML itself is adopted differently in the two approaches. We chose not to extend UML and to exploit its standard features

(e.g., association classes to express link attributes, and association and attribute names to express the respective meanings). On the contrary Conallen extended UML with tagged values and stereotypes.

In [BN96] the tool W3DT Web-Designer, developed especially to facilitate the design of web sites, is presented. The web site model proposed in this paper introduces the concepts of dynamic page and dynamic link, while it does not consider the frames. The level of abstractions of this model is higher than ours.

The representation adopted by Di Lucca et al. [LPAC01] is similar to the one proposed by Conallen in [Con00]. Conallen's work is focused on forward engineering and is suited for the high level specification of a Web application, while the conceptual model proposed by Di Lucca et al. is focused on a reverse-engineering process. In fact, it better highlights the behavior of and the dynamic interaction between the elements in the Web application. In Di Lucca et al.'s representation, differences between static client pages and dynamic client pages, passive Web objects (e.g. images) and active Web objects (e.g. scripts) are remarked, and interface objects (i.e., objects that interface the Web application with a DBMS or an external system) are added. On the contrary, our model aims at explicitly representing user navigations, i.e., feasible paths, which are the starting point for analysis and testing. Consequently, internal entities such as the Web objects and interface objects are not considered.

The paper [LFP+02] presents the reverse engineering tool WARE. WARE automatically extracts information from the target Web application, builds a model of it (according to the representation presented in [LPAC01]) and help the user to recover some UML diagrams that depict static, dynamic and behavioral aspects of the application. WARE builds the model by means of static and dynamic analysis. In contrast to our approach the code of server side programs is analyzed (Vbscript, Javascript, ASP and PHP)) and input are provided to the Web application only if necessary (for example, for deducing a hyperlink that is built dynamically by script block). Automatic clustering algorithms are used by WARE in order to simplify the comprehension of large graphs. On the contrary, *focus* function and *system view* have been implemented in **ReWeb**.

The approach proposed by Hassan and Holt [HH02] is useful to extract the structure of dynamic Web applications and shows the interaction between the various components (databases, distributed objects and Web pages). The approach uses a set of specialized extractors which analyze Web pages, source code and binaries of Web applications. Authors use for their work the Portable BookShelf (PBS) [FHa02], an environment able to recover the architecture of traditional software systems. The level of abstraction of the architecture recovered in this approach is higher than ours.

Part II

ANALYSIS

Chapter 4

Structural and Evolution Analysis

In this chapter a set of analysis and abstract views of Web applications, based on the model explained in the previous chapter, are presented. Analysis results and views can be employed to detect possible defects and anomalies and in particular can be useful when the Web application enters maintenance and needs to evolve while retaining and possibly improving its quality. Some of the proposed analysis were derived from those used with traditional software systems. They are divided into two main categories: analysis of the structure and analysis of the evolution. These analyses are implemented in one module of the tool ReWeb, called Analyzer, while another module, the Viewer, displays the abstract views as well as the output of the analyses.

4.1 Introduction

Similar to the development of software systems, the production of high quality and reliable Web applications can be achieved only if proper methodologies and techniques are adopted. Even in presence of consolidated design practices, the possibility to analyze the implementation and to abstract views and models remains important.

Analysis results and views can be useful for different purposes:

- to detect possible defects and anomalies of the implementation;
- to highlight possible discrepancies between the initial Web application architecture and the implementation and make traceability links explicit;
- to understand better the organization of the various entities involved in a Web application (pages, forms, frames, etc.) and their relationships (*Web application understanding* phase);

- to support maintenance activities, in particular highlighting updates that may degrade the original structure;
- to answer queries about Web application organization and functions;
- to support restructuring interventions.

In this chapter several analyses and some views, based on the explicit-state model explained in the previous chapter, are proposed. Some of the analyses presented are derived from those used with traditional software. Analyses are divided into two main categories: analysis of the structure [RT00b, RT01c] and analysis of the evolution [RT00a, RT01c].

Since the structure of a Web application can be modeled with a graph (re-interpretation of the explicit-state model as a graph), several known analyses, working on graphs, such as flow analysis [ASU85] (see appendix 12.1 for a short summary on flow analysis), traversal algorithms and pattern matching can be applied. Some simple analyses may determine the presence of **unreachable pages** and **Ghost pages**. **Unreachable pages** are pages that are available at the server site but cannot be reached along any path starting from the initial page. They are obtained as the difference between the pages available in the Web server file system and those downloaded by the Spider. **Ghost pages** are associated with pending links, which reference a non existing page. They are obtained directly by the Spider during downloading phase, when it encounters the "404 error" message, also known as a "page not found" error. More advanced analyses can be derived from the general framework of flow analysis. An example of analysis which specializes the framework of flow analysis is the computation of the **reaching frames**, which determines the set of frames in which each page can appear. The outcome of this analysis can be useful to discover the presence of undesirable reaching frames. Examples are the possibility to load a page at the top level, while it was designed to always be loaded into a given frame, or the possibility to load a page into a frame where it should not be. Flow analyses can be employed in a more traditional fashion to determine the **data dependences**. Nodes of kind *Form* generate a definition of each variable in the *input* set. Such definitions are propagated along the edges of the Web application graph. If a definition of a variable reaches a node where the same variable is used (*use* attribute of a dynamic page), there is a data dependence between defining node and user node. Data dependences are useful to represent the information flows in the application. They may reveal the presence of undesirable possibilities, such as using a variable not yet defined or using an incorrect definition of a variable. Data dependences are also important for dynamic validation, when data flow testing techniques are adopted. When the pages of a site are traversed, it is impossible to reach a document without traversing a set of other pages, called its **dominators**. Sites in which traversing a given page is considered mandatory, e.g., because it contains important information, will have it in the dominator set of every node. Dominator analysis, also derived from the framework of flow analysis, automates the check. Other analyses, based on graph

traversal algorithms are: **shortest path** and **strongly connected components**. An interesting information about a Web application is the minimum number of pages that must be visited before reaching the target document. Information about the **shortest path** to each page in the site is an indicator of potential troubles for the user searching a given document (for example, when such a path is too long). In a Web application, the presence of **strongly connected components** containing several nodes suggests that the organization of the pages is aimed at promoting a navigation style that never gets stuck. Recurrent patterns are expected to be used in the design of Web applications. Matching a library of known patterns against a given Web application results in the identification of portions of the site that are compliant with a predefined structure from the pattern library (**pattern extraction**). Different navigation styles are enforced by the different patterns implemented within a Web application.

The evolution of Web application is another interesting object of investigation [RT00a]. Such an analysis requires the ability to compare successive versions of its pages and to graphically display the differences. Given two versions of a Web application, downloaded at different dates, their comparison aims at determining which pages were added, modified, deleted or left unchanged. It can be combined with the static analyses described above, since their re-computation over time allows controlling the evolution of the application quality. The assumption is made that the page name is preserved, when moving from a version to the next one. Determining the identity of pages by content, rather than by name, is much more complex. Such a problem is unavoidable in presence of dynamic pages which display totally different information under different interaction conditions, but have a unique name. When a same name page is found in the old version of the site for a page in the current site, it is classified either as modified or left unchanged according to the output of a character-by-character comparison. The model of the Web application can be enriched with information about its history by coloring the nodes and associating different colors to different time points. History visualization can also be applied to the links of a Web application.

In the following a graphical display of a Web application model will be referenced as *structural view*. In effect the structural view of a Web application differs slightly from the model. In fact, in the structural view frames are joined horizontally to intuitively suggest decomposition into frames, incoming edges are collapsed into a single edge and labeled edges are used for the links having a *LoadPageIntoFrame, Form*, or *Parameter* relation specifier. Figure 4.1 shows an example of structural view where frames `f1` and `f2` are joined horizontally. Contrast it with the explicit-state model depicted in Figure 3.5. The *history view* of a Web application is the structural view of it where the graph representing the model is enriched with information about its history (by coloring the nodes and the edges associating different colors to different time points). The structure of a Web application can be considered at different abstraction levels. The model described in the previous chapter is based on the pages and on the navigation from page to page as the basic entities

and relations. Higher level views can be defined by grouping pages into logically cohesive clusters and by abstracting the relations between pages into relations between clusters. Complex Web applications are usually organized by distributing the HTML pages associated to conceptually distinct portions of the site into different subdirectories. A typical example is the Web application organization usually employed by Internet providers. In order to give their clients the possibility to have their own space (a sort of *virtual* site), a natural choice is to assign a subdirectory to each virtual site. The graph representation of a Web application can be abstracted by grouping pages according to the directory containing them. If no organization into directories is present, such a view, in the following called *system view*, contains a single node representing the root directory. Otherwise, a node for each directory is added. An edge connects two nodes d_1 and d_2 of the system view if, in the initial graph W for the site, there is a page in the directory associated to d_1 connected to a page in the directory of d_2.

The chapter is structured as follows: section 4.2 describes the structural analysis that are performed on Web applications, while section 4.3 deals with the historical analysis of their evolution. The restructuring of a real Web application, based on abstract views, reaching frames, dominators and shortest path, is discussed in section 4.4. Discussion of related works on Web application analysis concludes the chapter.

4.2 Structural analysis

Flow analyses (appendix 12.1), traversal algorithms and pattern matching have traditionally been applied to solve problems related to the static analysis of computer programs. For example, flow analyses have been used to compute dominators, reaching definitions, reachable uses, available expressions and copy-constant propagation [ASU85]. Given the graph representation of a Web application proposed in this thesis, it is simple to extend the range of applicability of flow analyses, traversal algorithms and pattern matching to the Web applications.

4.2.1 Reaching frames

The computation of the reaching frames aims at determining the set of frames in which a page may appear. A page can be loaded into a frame as the initial page appearing in it, or it may be reachable from the initial page. Moreover there is the possibility that a page is loaded into a frame because a link in another page forces such loading. Pages reachable from it will also appear in the same frame. To compute the reaching frames of the pages in a Web application a specialization of the flow analysis framework can be employed. The propagation of flow information is the basic idea behind flow analysis. In the case of the

reaching frames, the information to be propagated is the identifier (name) of the frame, and the generators of such information are the nodes of type frame (F) and the edges forcing page loading into a specified frame (E_2). More specifically, the flow equations required for the computation of the reaching frames are the following:

$$
\begin{aligned}
GEN_n &= \{n\} \ if \ \ n \in F, && (4.1)\\
GEN_n &= \{f\} \ if \ \ \exists (m,n)_f \in E_2, && (4.2)\\
GEN_n &= \emptyset \ otherwise && (4.3)\\
KILL_n &= F \ if \ \ n \in F, && (4.4)\\
KILL_n &= \emptyset \ otherwise && (4.5)\\
IN_n &= \bigcup_{p \in pred(n)} OUT_p && (4.6)\\
OUT_n &= GEN_n \cup (IN_n \setminus KILL_n) && (4.7)
\end{aligned}
$$

A node n generates itself as flow information if it is a frame ($n \in F$). In such a case it replaces any incoming flow information ($KILL_n = F$), since it overrides any previous frame. If a node n is preceded by an edge of type E_2 forcing its loading into frame f, the flow information representing f is generated, but the incoming flow information is not killed ($KILL_n = \emptyset$). In fact alternative enclosing frames remain valid. The set IN_n of each node n collects the information about all possible enclosing frames as the union of the OUT sets of the predecessors of n ($pred(n)$), while its OUT set is obtained by subtracting the $KILL$ set and adding the GEN set. The root of the graph, associated to the first page provided by the site server, has empty IN set, having no predecessor. Its GEN set may contain a label (e.g., g) to be associated to the *global* frame, i.e., to the possibility to load pages at the top level, outside any enclosing frame.

Flow information is repeatedly propagated in the graph until a fixpoint is reached. It should be noted that no propagation occurs along the edges of type E_2, which do not represent navigation links, but rather page loading into an alternative frame. The result of the analysis, contained in the OUT set of each node, is the collection of all frames into which the page associated to the node may appear, i.e., its reaching frames.

Figure 4.1 shows the result of computing the reaching frames for the example in Figure 3.5. The initial page of this site, `p1`, has a label g in its GEN set to denote the top frame. Frames `f1` and `f2` generate themselves as flow information, overriding any incoming data. The dashed edge from `p2` to `p4` is the reason for the GEN of `p4`, containing `f2`, and for the empty $KILL$ set. After propagating the flow information until the fixpoint is reached, the IN and OUT sets in Figure 4.1 are determined. It can be noted that the only page visible at the top level is `p1`. All other pages are displayed inside a frame. This is a consequence of the $KILL$ sets of `f1` and `f2`. Page `p5` can only appear in frame `f2`, for a twofold reason:

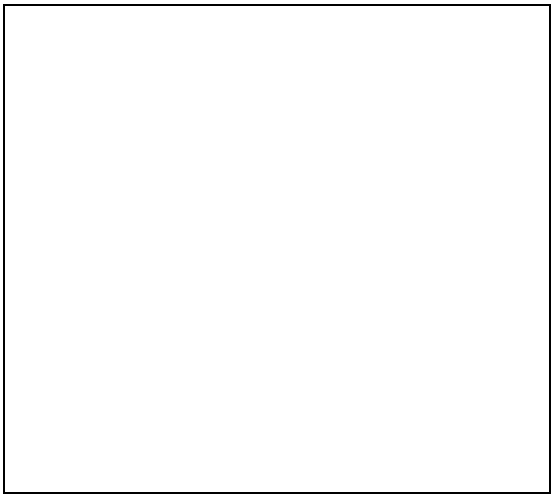

Figure 4.1: *Example of reaching frame computation.*

it is reachable from `p4`, which is forced into `f2` by `p2`, and it is reachable from `p3`, which is the initial page loaded into `f2`. If an edge is added from `p2` to `p5`, its IN and OUT sets become $IN = OUT = \{f_1, f_2\}$, thus indicating that `p5` can either be loaded inside frame `f1` or `f2`.

The outcome of the reaching frames analysis is useful to understand the assignment of pages to frames. The presence of undesirable reaching frames is made clear by this analysis. Examples are the possibility to load a page at the top level, while it was designed to always be loaded into a given frame, or the possibility to load a page into a frame where it should not be.

4.2.2 Dominators

When the pages of a Web application are traversed to reach a document of interest, several alternative paths can be chosen. Nevertheless, it is impossible to reach a given page without traversing a set of other pages, called the *dominators* of the page of interest. The page initially provided by the site server, say `index.html`, is a dominator of any other page in the site. It will be considered the *root* of the graph representation of the Web application. More formally, given a graph rooted at node r, a node m is a *dominator* of node n if every path from r to n traverses m. The computation of the dominator set for each node of a rooted graph can be achieved by exploiting a well known specialization of the flow analysis framework (see [ASU85]).

When applied to the example in Figure 3.5, the dominator analysis gives two dominators for `p3`: the root of the graph, `p1` and the frame `f2`. Any path leading to `p3` traverses all its

dominators. If page p5 is considered, its dominator set consists of just the first dominator of p3, i.e p1. In fact it is possible to reach p5 without traversing f2 and p3, by choosing the link in p2, loaded into f1, forcing page p4 into f2, and then navigating from p4 to p5.

Since frame loading is automatic, users of this analysis may be interested in filtering out nodes of type frame from the set of dominators of a given node. In fact, only nodes of type *HTMLPage* have to be *actively* traversed to reach the page of interest.

The knowledge about the dominators of a page is useful to understand the navigation constraints of a Web application. A Web application designed to provide several alternative navigation paths is expected to have only the root in the dominator set of its nodes, while applications in which traversing a given page is considered mandatory, e.g., because it contains advertising material or important information, will have it in the dominator set of every node.

4.2.3 Shortest path

Reaching a page of interest in a Web application may require traversing several pages. A useful information about a Web application is the minimum number of pages that must be visited before reaching the target document. Given a graph rooted at node r, the *shortest path* from r to each graph node n is the path from r to n with the minimum total weight associated to the edges. In the present application of the shortest path, edges are weighted 1 when they represent real user selections (*mouse clicks*), while they are weighted 0 when page loading is automatic. In particular, edges going from a node p of type *HTMLPage* to a node f of type *frame* have weight 0, since they represent page decomposition into frames, and edges connecting a frame f to its initial pages must be weighted 0, being the loading automatic.

With reference to the example in Figure 3.5, the shortest path from p1 to p3 has weight 0 and consists of the sequence: p1, f2, p3. In fact, page p3 is automatically loaded into frame f2, which is built inside p1 at the beginning of the interaction. The link selection number to go from p1 to p5 is 1, since navigation from p3 to p5 is required. If the link from p3 to p5 is removed, the link selections from p1 to p5 become 2, and consist of the link in page p2, forcing the loading of p4 into f2, and the navigation link from p4 to p5.

Information about the shortest path to each page in the site is an indicator of potential troubles for the user searching a given document, when such path is long. A well designed Web application should provide very short paths to the most relevant documents.

4.2.4 Strongly connected components

During navigation in a Web application, some pages may become no longer reachable from the current page, unless restarting from the beginning or from a previously visited page. Often regions with fully circular navigation facilities are present in Web applications, and in several cases the application is itself one such region as a whole. Given a directed graph, its *strongly connected components* are the equivalence classes of nodes that are mutually reachable from each other. In other words, if nodes n_1 and n_2 belong to the same strongly connected component, a path is assured to exist from n_1 to n_2 and another path exists from n_2 to n_1.

In the Web application depicted in Figure 3.5 each node corresponds to a strongly connected component containing just that node, since no loop is present in the graph. If a link from `p5` to `p2` is added, nodes `p2`, `p4` and `p5` become the elements of a common strongly connected component. In fact, when one of these three pages is loaded into the browser, it is always possible to reach the other two pages, and to circularly navigate across them.

In a Web application, the presence of strongly connected components containing several nodes suggests that the organization of the pages is aimed at promoting a navigation style that never gets stuck. Circular paths leading to previously visited pages allow the user to explore alternative pages.

4.2.5 Pattern extraction

Recurrent patterns are expected to be used in the design of Web applications. Matching a library of known patterns against a given Web application results in the identification of portions of the site that are compliant with a predefined structure from the pattern library. Since patterns are in general represented by graphs themselves, the algorithms employed for their matching are those used to detect the isomorphism of a subgraph with respect to a set of model graphs, although ad hoc solutions may be conceived for specific patterns. A general approach to this problem can be found in [MB98a].

Figure 4.2: *Examples of reference patterns.*

Examples of reference patterns to be matched against a given Web application are the *tree*, the *hierarchy*, the *diamond*, the *full connectivity* and the *indexed-sequence* (see Figure 4.2).

A portion of a Web application is organized according to a *tree* pattern if its nodes and edges form a tree, i.e., its graph representation is acyclic and each node has exactly one parent, except for the tree root, having no parent. A Web application region matches the *hierarchy* pattern if it is acyclic, but more than one parent is allowed. The *diamond* is a particular kind of hierarchy, characterized by a single entry point, a *top* node (root) with no parents, and a single exit point, a *bottom* node with no children. A *full connectivity* pattern is matched by a subgraph if each node in the subgraph is connected to all other nodes in the subgraph. A full connectivity subgraph is also a strongly connected component, but the inverse is not true. Pages are structured according to an *indexed sequence* if they are arranged into a singly or doubly linked list, so that from each page the next one is available, and optionally the previous one, and in addition an index page exists from which all other pages are directly accessible.

To discover the presence of a pattern in a site it may be necessary to remove some navigation links, aimed at facilitating operations such as going to the home page/moving backward, which do not contribute to the pattern structure.

Pattern extraction can be useful in Web application understanding exactly as happen in program understanding where automatic concepts recognition [KNE92] can help the designer to understand better the code under analysis. In fact different navigation styles are enforced by the different patterns implemented within a Web application. Tree and hierarchy patterns are associated to a search strategy based on the refinement of general notions into more detailed ones. At each step the user makes some choice, thus successively restricting the area of interest. The diamond organization enforces a single exit point, associated, for example, to a final operation, to be performed after selecting the items of interest, or to a final document, to be displayed before the user leaves the site portion. Fully connected regions are conceived to provide full access to every document from each page, without any need of query refinement. All information is available at every time from everywhere. Pages are made available sequentially, according to the indexed-sequence pattern, when there is a natural order in which pages are expected to be traversed, as for the chapters of a book. The index allows jumping into the middle.

4.3 Evolution analysis

Knowledge about the history of a Web application is useful to document the events leading to its current organization and to identify the reasons for potential structural problems. The analysis of a Web application evolution requires the ability to compare successive versions of its pages and to graphically display the differences. Works on history visualization for software systems [ESS92, GJR99] greatly influenced the chosen approach.

4.3.1 Difference computation

Given two versions of a Web application, downloaded at different dates, their comparison aims at determining which pages were added, modified, deleted or left unchanged. Tracing the evolution of an entity over time requires that a notion of identity be defined, so that the same object can be recognized at different times. In our case the problem is highly simplified if the assumption is made that the page name is preserved, when moving from a version to the next one. In this way a map exists between nodes in the graph representation of the previous site and nodes in the graph for the current site. It simply pairs nodes with the same name. All nodes in the old graph without corresponding node in the new graph represent pages that were deleted from the site, while all nodes in the new graph without counterpart in the old graph are added nodes. When a corresponding node can be found in the old graph for a node in the new graph, it is classified either as a modified node or a node left unchanged according to the output of a character-by-character comparison of the associated HTML pages.

The choice to compare pages with preserved name through successive versions simplifies history traceability, since the mapping between old and new pages is given, and has not to be reconstructed. The drawback is that an unchanged page, whose name is modified, is considered as a deleted page, in the old graph, and an added page, in the new graph. Since the name of a page is its unique identifier, used by the other pages to access it and possibly referenced by external pages from other sites, it is likely to be preserved among successive site versions. In fact, changing the name of a page has an impact on all referencing pages, which must update their links, with the risk of missing some update. Therefore the assumption of name preservation seems a reasonable one, and few exceptions are expected to be found.

4.3.2 Visualization

The graph representation of a Web site can be enriched with information about its history by coloring the nodes and associating different colors to different time points. In particular, a scale of colors ranging from the blue (B), going through the green (G) and reaching the red (R) can be employed to represent nodes added/modified in the far past, in the medium past or more recently.

Figure 4.3 gives a path, in the RGB representation of the colors, that can be uniformly divided into segments to be associated to different time intervals. If, for example, 5 points are needed to represent 5 different versions of a site, the associated RGB color codes will be: `"0 0 1"`, `"0 1 1"`, `"0 1 0"`, `"1 1 0"`, `"1 0 0"`. The far past is thus represented with pure blue (`"0 0 1"`), then some green is added (`"0 1 1"`). The pure green is traversed before moving toward the red (`"1 0 0"`).

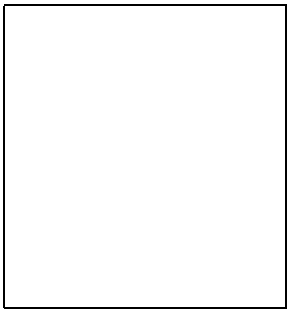

Figure 4.3: *Path in the RGB color cube, associated to the page introduction/modification date, going from the past to the present time.*

Several alternative paths have been evaluated, going from B to R in the RGB cube. Our choice was made by selecting a path providing a high number of intermediate colors, that can be visually distinguished, and at the same time ensuring a continuous scale from B to R, so that the distance in the past and the time proximity can be easily assessed by visual inspection. The path that was judged to be the best is the one depicted in Figure 4.3.

History visualization can also be applied to the links of a Web application. When a link from a page to another page is first introduced, the edge in the graph representation of the site assumes the color of the source node, i.e., of the enclosing page. Then such a color is maintained even when the page evolves, changing its color, if modifications do not affect the link.

4.3.3 Example of evolution

The site `www.cartercopters.com` (an aircraft manufacturer), used also in [WBM99b], is an interesting example because it had a high number of changed pages during the three months of monitoring. New pages and links were added, while others were removed. The significant versions of this site are four, at the dates: 21-10-1999, 24-11-1999, 23-12-1999, 25-12-1999. The tool **ReWeb** associates these dates to colors blue, light blue, yellow and red, thus permitting an immediate comparison between different versions. Colors are obtained by uniform sampling of the path in Figure 4.3. At the initial date the graph representation of the Web application is all blue.

When visualizing the second version, it is possible to note that there are nodes and edges light blue, i.e., the Web application evolved. Specifically, the pages `pressrel14.html, pressrel.html, contents.html, index.html` are light blue. In fact, the first one was added to the site, while the others changed, with respect to the previous versions. It can be noted that all new edges are links to the new page and that two edges were deleted. The tool **ReWeb** also provides a textual comparison between two versions. An example,

www.cartercopters.com compare: 21-10-1999 − > 24-11-1999
new nodes = [pressrel14.html] **deleted nodes** = [] **new edges** = [pressrel14.html − > pressrel13.html, pressrel14.html − > index.html, contents.html − > pressrel14.html, pressrel14.html − > contents.html, pressrel.html − > pressrel14.html] **deleted edges** = [contents.html − > NASA10TechGoals.html, contents.html − > pressrel13.html] **changed nodes** = [pressrel.html, contents.html, index.html]

Table 4.1: *Example of* **ReWeb** *textual comparison between two versions of the site www.cartercopters.com*

summarizing the explanation given above, is proposed in Table 4.1. The third version introduces eight yellow nodes plus several yellow edges. There are five new nodes, while three changed with respect to the version dated 24-11-1999. In the last graph it is possible to see five red nodes and several red edges. It is not practical to display the entire site in a window, because it is too large, but it is possible to see a portion of it by exploiting the **ReWeb** focus. If `pressrel13.html` is chosen as focus, with upward and downward depth equal to 1, and date 25-12-1999, the graph in Figure 4.4 is obtained (note that in this figure the shape of the pages, usually represented with boxes, is not consistent with the other pictures in the thesis. This is due to different implementation of **ReWeb**). Node `pressrel14.html` and its links were introduced on 24-11-1999. In fact their color is light blue. Page `pressrel.html` changed on 23-12-1999 and is consequently yellow. Pages `contents.html` and `index.html` changed on 25-12-1999, and are red, while the other nodes never changed and are blue, the color associated with the first date. In the structural design of this Web site the pattern indexed-sequence, introduced in the previous section, is often used. The history study of this Web application could not reveal phenomena of structural degradation. Even the largest insertion, occurred at date 23-12-1999, of the tree rooted at `pressrel15.html` preserved the previous structure of such portion: an indexed-sequence where each node can be in turn the root of another indexed-sequence.

4.4 Case study

In this section the analyses and the views proposed in this chapter are used as support to an understanding and re-structuring task of a real Web application. A Web application comprising 58 HTML pages, `www.ubicum.it` (an Internet provider), was analyzed and re-

Figure 4.4: *Example of* history *view depicting the site* `www.cartercopters.com` *at date* `25-12-1999`*, focused on node pressrel13.html.*

structured to improve navigability, to solve some structural problems and, above all, to make it easier to understand and maintain.

4.4.1 Understanding the current Web application

The process of understanding a Web application usually starts from the analysis of its overall, high-level organization. The system view is the most abstract view among those presented.

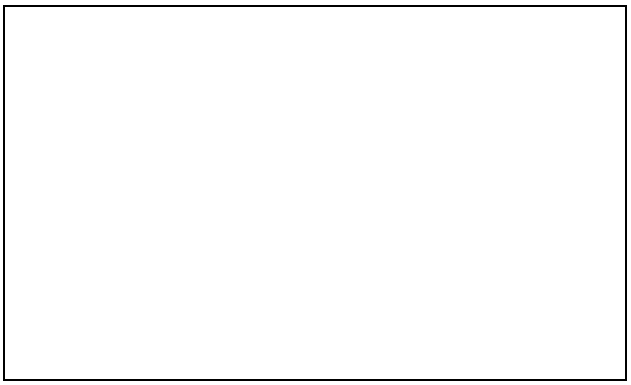

Figure 4.5: **ReWeb** system view *for the site* `www.ubicum.it`.

Figure 4.5 shows the system view of `www.ubicum.it`. This site hosts an Internet provider and is decomposed into eight virtual sites, physically associated with different directories,

serving different clients, and a directory, `bin`, containing a program to compute the Web server statistics. Directories `betty` and `len` are not reachable from the root of the site. By exploding the virtual sites into their structural views, some patterns introduced in the subsection 4.2.5 are apparent. In Figure 4.6, showing the entire site, it can be observed that the virtual site `piscini` matches the pattern diamond, `tigullio` matches a tree, `rigor` contains a large full connectivity region, while `madmaxpub`, `marco` and `merlo` have a structure that does not agree with any known pattern. The virtual site `betty` matches a tree while `len` is quite similar to a tree (two edges violate it).

Figure 4.6: *Example of view depicting the entire site* `www.ubicum.it`.

Figure 4.7 depicts a fragment of textual report for the site under analysis. The site does not contain pages with forms and applets, while there are pages decomposed into frames. Two pages are implemented with a scripting language (Javascript).

The Web application was downloaded every day by **ReWeb** for over six months. During this time, the page `bin/stat` changed its color every day in the history view (this page is a dynamic page that computes and visualizes the Web server statistics), the page `madmaxpub/madmaxpub.htm` changed once, at data 16-12-1999, and the other pages never changed (see Figure 4.8), i.e., no maintenance occurred for them.

Report of Web application www.ubicum.it
Total number of Pages = 87
Total number of HTML Pages = 58
Total number of links = 322
Number of error links = 0
Total Number of lines of code (LOC) = 3273
...
Pages that contain Applets = 0
Pages that contain Forms = 0
Pages that contain Frames = 4
http://www.ubicum.it/madmaxpub/index.html
http://www.ubicum.it/madmaxpub/frame1.html
http://www.ubicum.it/rigor/index.html
http://www.ubicum.it/rigor/index2.htm
Pages that contain Scripts = 2
http://www.ubicum.it/rigor/sommario.htm
http://www.ubicum.it/rigor/somcro.htm

Figure 4.7: *A portion of the report for* `www.ubicum.it`

4.4.2 Collecting change requirements

In this section all requirements for the re-structuring of `www.ubicum.it` are collected, after examining its internal organization and the way it supports navigation. Such *change requirements* are indicated with label CR in the following.

A potential problem can be noted in the system view (Figure 4.5) of this site by looking at the directories `marco` and `merlo`. The pages contained in `marco`, a personal site, cannot be reached from the root without passing through the virtual site `madmaxpub`. A better structure of the site would position `marco`'s directory at the same level as the others. An analogous argument can be made for the directory `merlo`. On the other hand, `betty` and `len` have another problem: they are not reachable from the root of the site.

CR 1 *New links from the root towards the virtual sites* `marco`*,* `merlo`*,* `betty` *and* `len` *should be added in such way that the users can reach them from the root.*

The search of a target virtual site in an Internet provider is a useful facility, especially if the provider manages a lot of clients. The root of this site does not contain it.

CR 2 *A form in the root of this site and a new dynamic page should be added to permit users inserting the name of the searched virtual site, receiving a link to it, if it exists.*

Figure 4.8: *The history view of the virtual Web application* `madmaxpub` *at date 3-2-2000.*

The decomposition into frames of the virtual site `madmaxpub` is shown in Figure 4.8. Page `madmaxpub/index.html` (the main page) is divided into two frames with identifiers `a` and `b`, and frame `a` is used as a menu to force the loading of pages into the other frame. The page `madmaxpub/frame1.html` fails to divide itself in two frames, since the edge from the second frame is missing. When loading the page `madmaxpub/index.html` into a browser, the page loaded in frame `b`, `madmaxpub/frame1.html`, remains grey and `madmaxpub/frame.html` results not reachable. In the HTML code of the page `madmaxpub/frame1.html` there is a syntactic error: the `FRAMESET` tag is not closed.

CR 3 *The syntactic error contained in the page* `madmaxpub/frame1.html` *should be fixed.*

The structural view of the virtual site `rigor` (see the respective portion in Figure 4.6) is, at first glance, very complex. There are two nodes of type `FRAME` and the connectivity among pages is high. Many edges are dashed and labeled with `main`. The reason for the high connectivity was further investigated. Every page contains two similar menus, with more or less the same links. One is implemented completely in HTML while the other one is implemented through Javascript buttons. A reasonable conjecture is that the designer implemented this virtual site in pure HTML in a first time, while in a second time, after a re-structuring operation, Javascript menus (pages `rigor/sommario.htm` and

rigor/somcro.htm) and frames were added, thus doubling, in some cases, the links among pages.

CR 4 *Only one menu, equivalent to the two menus present in the current version of the virtual site* rigor*, has to be available, so as to simplify and improve navigability.*

Then, results of the structure analyses were considered. The computation of the reaching frames revealed a major structural problem of the entire site. Pages of the virtual sites piscini, tigullio, marco and merlo and the main page of the site can appear in frame b of the virtual site madmaxpub. In fact, frame b is present in their reaching frame sets, making the multiple Web pages active and opened in different frames inconsistent. Figure 4.9 shows an example of this problem. The madmaxpub navigation menu, which is generated by the main page of the site when selecting the virtual site madmaxpub, is displayed concurrently with the main page itself, thus creating a potentially dangerous recursive loop and making index (left) and informative (right) frames not aligned. A similar problem occurs when the madmaxpub navigation menu is loaded together with any page from the virtual sites piscini, tigullio, marco and merlo. Information displayed on the right is not indexed by the menu on the left.

Another problem is that when the advertisement hyper-links contained in the menu (FRAME a) are clicked, the selected pages are loaded into FRAME a instead of the top level.

CR 5 *The menu of the virtual site* madmaxpub *loaded into left frame, and the information displayed on the right frame should always be consistent. Hyper-links in the menu of the* madmaxpub *virtual site should not force page loading into the left* FRAME (a)*.*

A similar problem with frames is also present in the virtual site rigor. This virtual site exploits two different menus to access internal pages. A menu loaded into the frame indice can coexist with a page associated to the other menu.

CR 6 *The menu of the virtual site* rigor *loaded into left frame and information displayed on the right frame should always be consistent.*

The computation of the shortest path for each node highlights a possible improvement. Pages merlo/germania.htm and merlo/inghilterra.htm have shortest path equal to 4.

CR 7 *The shortest path of the pages* merlo/germania.htm *and* merlo/inghilterra.htm *should be diminished.*

Figure 4.9: *Inconsistent page loading: the* `madmaxpub` *menu and the main page are displayed concurrently.*

As pointed out before, page `madmaxpub/madmaxpub.htm` changed at date 16-12-1999. The only difference is a new paragraph announcing the sale of the pub. The computation of the dominators highlights that this important information cannot be reached if the user navigates in this virtual site without visiting the page `madmaxpub/madmaxpub.htm`, which is not a dominator of the entire virtual site.

CR 8 *The sale announcement of the virtual site* `madmaxpub` *has to become a dominator of all other pages in the site.*

The virtual Web application `rigor` resembles a full connectivity pattern. According to the **ReWeb** facility computing the strongly connected components, this virtual site is composed of 19 nodes, of which 15 are strongly connected. Each of such 15 pages contains two menus providing access to the other pages, thus requiring the intervention described in CR 4.

Finally, the names chosen as page and `FRAME` identifiers were considered. The names of some pages are in Italian, but there are also names in English. Some names were chosen with care, trying to summarize the content of the page, while others were not (e.g. `madmaxpub/frame1.html` and `madmaxpub/frame.html`). The name of `FRAME` identifiers are also not very meaningful. In the virtual site `rigor`, for example, there are two different pages that contain `FRAMEs` with the same name: pages `rigor/index.htm` and `rigor/index2.htm` are both divided into `FRAMEs indice` and `main`. Such ambiguity and poor care in the name selection could have contributed to the `FRAME` problems explained above.

CR 9 *The names of the Web application pages and* `FRAMEs` *that are not informative should be made meaningful.*

The site `www.ubicum.it` contains a page named `rigor/merchan.htm` where `merchan` is the contraction of the word merchandising. The service offered by this page is very poor: if users want to buy an article, they must call the vendor by telephone or forward a message by e-mail.

CR 10 *The page* `rigor/merchan.htm` *should allow the customers to carry out their orders electronically.*

4.4.3 Re-structuring

Figure 4.10 presents the system view of the re-structured `www.ubicum.it` site. In accordance with CR 1, new links from the root towards the virtual sites `marco`, `merlo`, `betty` and `len` were added. In this way the users can reach also the virtual sites `marco`, `merlo`, `betty` and `len` directly from the root.

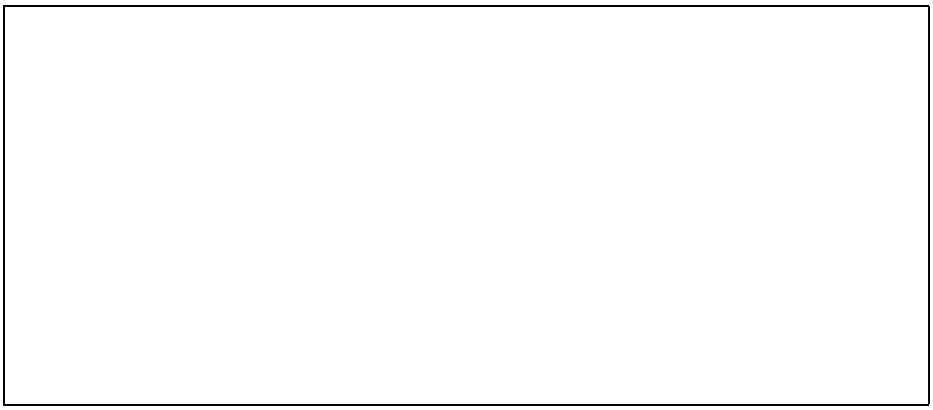

Figure 4.10: *Re-structured* **ReWeb** system view *for the site* `www.ubicum.it`.

A form was added in the root of the site which invokes a Perl script (`search.pl`), running on the server, in accordance with CR 2. The new pages permit users to search the virtual sites by account. After re-structuring, page `index.html` has a new link towards the dynamic page `search.pl` (see Figure 4.11).

The syntactic error contained in the page `madmaxpub/frame1.html` was fixed, as required by CR 3. In the new version, when the the main page of the virtual site `madmaxpub` is rendered, the page `madmaxpub/index.html` is correctly divided into two frames: `madmaxpub/elenco.html`, used as a menu to force the loading of pages into the other frame, and `madmaxpub/frame.html`, the presentation page of the site.

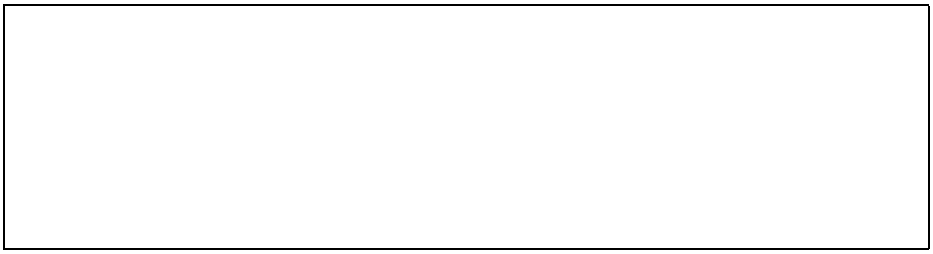

Figure 4.11: *A portion of the structural view of the re-structured site* `www.ubicum.it`.

As stated in CR 4, we simplified and improved the navigability in the virtual site `rigor`. Only one menu (the one implemented with Javascript buttons), equivalent to the two menus present in the old version of the virtual site `rigor`, remained. By comparing Figure 4.12, representing the structural view of the entire re-structured site (without dynamic pages and with `piscini, len, tigullio, merlo, betty` not expanded for space reasons), with the picture in Figure 4.6, the clearer and simpler organization of the virtual site `rigor` is apparent, with potential benefits on the site understandability and maintainability. The re-computation of the strongly connected components confirms this fact.

Figure 4.12: *Entire re-structured site* `www.ubicum.it` *without dynamic pages.*

In accordance with CR 5, we solved the problems associated with frames in the virtual site `madmaxpub`. In the new version of the virtual site, menu loaded into left frame and information displayed on the right frame are always consistent. This fact is confirmed

by the computation of the reaching frames. Our re-structuring intervention consisted in changing the targets of some commands `HREF`. In the previous version of the site the edges now labeled with `parent` were neither dashed nor labeled, i.e., the pages pointed by this hyper-links were reachable from pages inside frame `b` and were loaded into the same frame, instead of the entire browser window (see the related portion of Figure 4.12).

The problem with the hyper-links in the menu were also solved, and now all such link force page loading into the entire browser window.

A similar intervention was performed for the virtual site `rigor`. As for `madmaxpub`, the new version does not present inconsistencies between pages concurrently displayed in different frames. The computation of the reaching frames confirms it, and the intent of CR 6 is considered reached.

Adding new links from the root to the virtual sites `marco`, `merlo`, `betty` and `len` diminished the shortest path of the pages `merlo/germania.htm` and `merlo/inghilterra.htm`, now equal to 2. No page in the entire site has shortest path greater than 3. The intent of CR 7 is reached.

The sale announcement of the virtual site `madmaxpub` was moved to the page `madmax-pub/frame.html`, which is a dominator of all other pages. In the re-structured version, it is impossible to navigate in the `madmaxpub` site without encountering this important announcement. The goal expressed in CR 8 is fulfilled.

As required by CR 9 we re-structured the names of the Web application pages and `FRAMEs`, according to the following rules:

- The extension of the page must always be `.html`
- The letters in the name of the page must all be lowercase. An underscore is added to form composite identifiers.
- All name of the Web application pages and `FRAMEs` must be in Italian and meaningful.

To implement CR 10, we added a `FORM` to the page `rigor/merchan.htm` that gathers payment data (name, address, types and amounts of goods, and type of payment) from the user. A program that runs on the server and connects itself to a data base of orders should be added to the site to complete the intervention.

4.4.4 Lessons learned

The analysis of the user interactions with the tool **ReWeb** allowed the following lessons about Web application understanding and re-structuring to be learnt:

- During the understanding process, the system view is a good starting point for subsequent more fine-grained analyses. This view, besides providing an overall mental model of the site, can be useful to recognize some structural problems at a high level.
- A typical iteration in the understanding process with **ReWeb** includes: computing the system view, building a high level mental model of the site, exploding the virtual sites in the system view, using the structural and history analyses, using the report and trying to render some pages of the site on the browser. This process, or part of it, can be executed several times, until the user achieves a satisfactory knowledge about the target site.
- The structural view is one of the most frequently used. It gives a representation of the site in which it is possible to understand the navigation constraints, the page decomposition into frames and the transmission of data to the server. This view can be useful to recognize some structural problems at navigational and organizational level.
- The history view of a Web application allows an easy identification of the changes and of their impact on the structure. If the user can access only client-side information history analysis can also be used to recognize the dynamic pages.
- The report provides hints to understand the dimension of the selected Web application, the technology used and to reveal existing "pending" links.
- The reaching frames analysis highlights the presence of undesirable reaching frames. An example is the existence of the possibility to load a page where it should not be.
- Some navigation problems can be revealed by the dominator analysis. When traversing a given page has to be enforced, e.g., because it contains an important announcement, the dominator set of every node should contain it.
- The shortest path from the root to each page can be an indicator of potential search and navigation problems, in case it is long.
- Strongly connected components containing many nodes reveal a cyclic navigation style.
- All analyses and views are useful even after re-structuring. In fact, applying them to the re-structured Web application permits the verification of the change requirement implementation.

4.5 Related works

Various analyses of Web application have been proposed for different purposes, such as: link validation, navigability and usability issues, verifying integrity constraints, verifying design, and detecting duplicated portions of HTML code (i.e., clone analysis).

The paper [FFLS99] proposes a language useful for describing the structure of a Web site and for specifying some integrity constraints. Integrity constraints express properties that the Web site should have. An algorithm is used to verify the constraints against a site definition. The Web site is defined by means Horn clause while the constraints are specified by means a subset of formulas in first-order logic. An example of informal constraint, proposed in the paper, is: *all pages in the Web site are reachable from the root page.*

Adaptive sites are proposed in [PE97]. Adaptive Web sites are sites that improve themselves by learning from user access patterns. The log file is analyzed from autonomous algorithms, that using this information make improvements to the site's structure and presentation. Adaptive Web sites can make popular pages more accessible, highlight interesting links, connect related pages, and cluster similar documents together.

The papers [SDMP01, SDMP02] describe an activity in the field of automatic verification of Web sites design with model checking techniques. The verification of the site design consists in checking that the properties, expressed through the formal language CTL[CE81], are satisfied in the representation of the Web site. The representation of a Web site is similar to our model: a graph where nodes, that represent states, are pages, hyperlinks and applications and arcs connect pages to hyperlinks and hyperlinks to pages and applications. An example of an informal property presented in the papers is as follows: *Each text-string in a document that is associated to a link to another document will also be associated to the same document in any other document.*

The paper [LPFG01] proposes and compares two different approaches for detecting clones in Web sites. The approaches have been obtained by tailoring existing clone analysis methods in order to take into account the specific features of Web sites. In the first approach the *edit distance* is used as measure of similarity: two HTML pages are clones if the *edit distance* of the strings obtained from the pages collecting only HTML tags (text is not considered) is zero. Two HTML pages are *near missing clones* if the edit distance is greater than zero but less than a given threshold. The other approach is based on a frequency based method[May96].

To our knowledge the first authors to have started systematic studies on Web site evolution and Web maintenance were P. Warren, C. Boldyreff and M. Munro at the Department of Computer Science of the University of Durham. In [WBM99b] the authors recognize the similarity between software systems and Web based systems, the existence of problems,

in Web site development, similar to those encountered in software before the advent of software engineering and the importance of the maintenance phase. We share with them our model of Web site without frames and a common view of Web site development and evolution. In their work the evolution of Web sites is characterized by means of metrics, while we chose to represent it by means of the colors. In [WBM99a] the same authors presented some results on case studies. When analyzing the same sites, we reached the same conclusions as the authors of [WBM99a].

Chapter 5

Web Application Slicing

In this chapter an approach to Web application slicing is presented. The result of slicing a Web application is still a Web application (often simpler than the initial one), which exhibits the same behavior as the initial Web application with respect to some information of interest. The computation of slices on Web applications may be useful during debugging, when the amount of code to be inspected can be reduced, and during understanding, since the search for a given functionality can be better focused.

5.1 Introduction

Program slicing is a static analysis technique for the extraction of the program statements relevant to a specified computation. Program slicing was introduced for the first time by Weiser [Wei84] and has now many applications such as debugging, code understanding, program testing, reverse engineering, software maintenance, reuse, software safety and metrics [CLM95, DFHH00, GL91, GHS96]. Two excellent surveys on program slicing are [BG96, Tip95] (see appendix 12.2 for a short summary on program slicing).

A Web application consists of a set of static HTML pages displayed to the user and (possibly) of server side programs, which perform some computation, finally resulting in the production of dynamic pages transmitted to the browser for display. Moreover, HTML pages may embed client procedures which are either executed during page loading or in response to graphical interface events. As a consequence, slicing can be extended to Web applications.

In this chapter we extend the notion of slicing to Web applications, and describe the dependences that have to be taken into account. In particular, nesting control dependences for the HTML code and data flows from HTML code to server programs, as well as call

and parameter-in dependences, are considered. We focus on a real world language, widely used in the development of Web applications: PHP. Moreover, we consider also the case in which an HTML page includes client side code, written in Javascript. Several problems, have to be handled, such as the representation of the graphical events triggering execution of client code, or transfer of control to the server, the reference to HTML elements by client code through access paths, and the usage of session variables. Dynamic generation of HTML code will be also discussed, leading to a novel notion: *dynamic slicing* applied to Web applications. In fact, in presence of "extreme" HTML code generation, a purely static analysis is very limited, while dynamic slicing remains still applicable.

Web application slicing [RT01d, RT02b] is conducted on an extension of the program representation used with traditional software, the System Dependence Graph (SDG) [HRB90] (see appendix 12.2 for further information on SDG). Its basic elements are similar to those introduced in [LKHH01] for data flow testing. We describe an approach for solving the difficulties in the construction of the SDG for Web applications, based on the definition of Web specific dependences and nodes. Call dependences are categorized either as returning or non-returning, and graphical interface events and event loop are explicitly represented in the SDG.

The chapter is organized as follows. Section 5.2 introduces our approach to static slicing of Web applications. This section defines a SDG valid for Web application and is ended with a concrete example: slicing a real online travel agency Web application. Section 5.3 presents a novel notion, dynamic slicing applied to Web applications. A discussion of related works concludes the chapter.

5.2 Static slicing

According to its original definition [Wei84], a *program slice* is a reduced, executable program obtained from a given program by removing statements, so that it replicates part of the behavior of the initial program. The main requirements in this definition are that a slice be still a legal program and that its behavior be preserved with respect to a computation of interest.

Similar constraints are enforced in introducing the notion of slice for Web applications. Slicing a Web application results in a Web application which exhibits the same behavior as the initial Web application in terms of information of interest displayed to the user, programs generating it (in the case where it is dynamic) and interaction with the user (client side code).

Definition: *a* Web application slice *is obtained from a given set of Web pages and server programs by removing HTML and server/client code statements, so that part of the behavior of the initial Web application is replicated.*

The criterion for slice computation is an information item of interest displayed in a given Web page, and the resulting slice reproduces the same information item, if the user performs the same navigation actions and provides the same input in the original and in the sliced Web application.

Program slices can be computed by incrementally adding consecutive sets of transitively relevant statements. Statements that directly affect the computation at another statement are connected to the latter by a dependence relation, which can be explicitly represented in a graph called System Dependence Graph (SDG) [HRB90]. Different kinds of dependences are distinguished in the SDG (control, intraprocedural flow, call, parameter-in, etc.) and the problem of static slicing can be restated in terms of a reachability problem in the SDG. In the following, we discuss the kinds of dependences that hold for Web applications and we define an SDG valid for Web applications, that can be used to slice them.

5.2.1 Nesting/control dependences

Control dependences for traditional programs connect the predicate tested at a conditional or loop statement to the instructions the execution of which directly depends on the truth value of the predicate. This continues to hold for the server/client side code, but Web applications are characterized by an additional kind of control dependence (called nesting dependence), which is found in the HTML code. In fact, an HTML statement can be interpreted correctly by the browser only if all enclosing HTML tags are available. Therefore a nesting dependence holds between a tag and all directly enclosed statements.

Definition: *a* nesting dependence *holds between two HTML statements if the latter is nested inside the former.*

Figure 5.1 contains a fragment of PHP and HTML code, in which the first 6 lines are executed by the server, being included within the tags `<?  ?>`, while the last 4 lines are HTML statements directly passed to the browser. The server execution consists of testing the value of variable `$x`, and if it contains the string `"xx"`, the two variables `$y` and `$z` are assigned a new value. The HTML code inserts a form in the page being constructed. Such a form includes a text field, named `w` and initialized with `"ww"`, for user input, and a submit button which executes the page `aa.php`.

Control dependences for this code fragment are shown at the bottom. Node `cd.php` is connected to the top level instructions. When the execution of a statement is conditional to another one (e.g., the assignment at line 3 and the `if` at line 2), there is a control dependence between the two. Statements which close a scope (e.g., the closed bracket at line 5) have no associated nodes, in that the control dependence relation makes them unnecessary.

Figure 5.1: *Fragment of PHP/HTML code and related nesting/control dependences.*

5.2.2 Data dependences

Data dependences for traditional programs hold when a statement defines the value of a variable, which is used at another statement after being propagated to it along a definition clear path (i.e., a path containing no redefinition of the given variable). In addition to such a situation, which still can be found in server/client code, data flows may occur in a Web application from server/client side statements to the HTML code. On the other side, a variable definition at an HTML statement can reach a server/client instruction only through a procedure invocation, as a submission parameter, or through a DOM element access.

Definition: *a* data dependence *holds between two statements if the former defines the value of a variable which is used by the latter, and a definition clear path exists between the two.*

Figure 5.2 contains an example of two kinds of data dependences: a data dependence internal to PHP code, and two other ones from two PHP statements to an HTML statement. The server side statement 2 defines the value of `$x`, which is subsequently used (without intermediate redefinition) at statement 3. The two server side assignments to variable `$y` at lines 4 and 6 have a data dependence with the HTML statement at line 10, where the value of the input variable `w` is obtained from the value of `$y`. The effect is that a text field is shown to the user inside the dynamic page `dd.php` with an initial value equal to `$y`. The collected input is then stored inside variable `w`. It should be noted that this is the main

Figure 5.2: *Fragment of PHP/HTML code and associated control dependences (straight lines) and data dependences (curve lines).*

communication mechanism between server side statements and HTML variables, allowing the propagation of values from code executed on the server to Web pages. A variant of this mechanism is often exploited by Web applications to transmit a value from a server program to another server program via the generated Web page. It is obtained by means of a *hidden* HTML variable, the value of which is filled in by a first server program, and is transmitted to another server program, invoked via form submission. In the example of Figure 5.2, this can be achieved by replacing the input type at line 10 (`Text`) with `Hidden`. The form will not prompt the user for any input, and the value inserted by `dd.php` will be transmitted to `aa.php`.

Figure 5.3 shows the case of a data dependence between an HTML statement and a client code statement, achieved by means of the DOM. The Javascript code fragment from line 7 to line 9 is executed by the browser when these lines are loaded. The input variable `x` of the form `Eform`, loaded by the browser previously, is both used and defined at line 8. It is referenced as a DOM element, reached from `document`, the DOM element associated with the whole page. Then, the value of `x` is passed to `bb.php` as an input parameter, when the user clicks on the related submit button (the event loop, handling interface events is explained below).

Figure 5.3: *Data dependence mediated by the DOM.*

5.2.3 Call dependences

When a server side program is invoked from an HTML statement (e.g., via form submission or `HREF`), control of execution is transferred to the server and never returns back to the Web page issuing the call. In other words, a server program invoked from an HTML statement cannot produce any effect on the variables of the calling page, since the invocation is a no-return invocation. A consequence is that the *calling context problem*, i.e., the problem of keeping the data flows associated to the different call sites separated, is absent, since call sites are not affected by the invocation.

Calls issued within server (or client) code, such as from a PHP procedure to another PHP procedure (or from a Javascript function to another Javascript function), similarly to traditional software, are subject to the calling context problem (see [HRB90] for an algorithm handling such cases).

Invocations within the event loop can be issued before page loading is complete, if the user clicks on some interface widget while page loading is in progress. This means that it is not possible to assume that all HTML and client statements interpreted during page loading have been encountered when a callback is activated. The consequence in the construction of the SDG is that data dependences having a callback parameter as target need be computed in a non-killing (flow insensitive) way. That is, when a statement redefines the value of a variable, it is not possible to consider the previous definitions as invalid for the callback parameters, since execution could be interrupted by the user before the invalidating statement is loaded, and the actual parameter value is the previous one. The visible result in the SDG consists of some extra data dependence edges, which end at parameter-in nodes, and are due to the non-killing nature of the related statements. An

example of such an edge connects node 2 to the parameter-in node `x_in` in Figure 5.3.

Definitions: (1) *a* call dependence *holds between each statement of type* call *and the server/client program or procedure invoked. (2) A* parameter-in *dependence holds between any actual parameter of a call and the respective formal parameter of the invoked program or procedure. (3) A* parameter-out *dependence holds between any value returned from an invocation and the related variable in the call site.*

Figure 5.4: *Fragment of PHP/HTML code and associated control/data dependences. Call/parameter-in dependences are depicted with dashed lines.*

In Figure 5.4, the HTML form at line 4 installs `bb.php` as the response to a click on the `Submit` button of this form. Consequently, the event loop of this page contains only one call, triggered by the click event (indicated generically with `E`). Since this is a no-return invocation, it is indicated with a curved dashed line. Straight lines are used for normal procedure invocations. Two input variables are submitted by the form and accessed by the server program invoked: `x` and `y`. They are associated with two parameters (`x_in` and `y_in`), and with two parameter-in dependences (straight dashed lines in the figure).

5.2.3.1 Session variables

Session variables hold values that are maintained across different executions of PHP server programs, provided that such executions are triggered during a same Web navigation session. They can be handled similarly to global variables of traditional programs. Their

Figure 5.5: *Invocation with session variables.*

representation in the SDG is obtained by means of additional input and output parameters, making the transmission of values possible from procedure to procedure. Output parameters are necessary only when the invocation returns to the call site, where the effect of the call has to be propagated. On the contrary, for no-return invocations (curved dashed lines) only input parameters (and parameter-in dependences) have to be created.

In Figure 5.5, a session is started at line 2, and variable `$GlobalX` is declared to be a session variable. Consequently, the access to this variable from another server program (`bb.php`, line 11), successively invoked, results in the value set at line 4. In the associated SDG, a data dependence connects node 4 to the input parameter `GlobalX_in`, and a parameter-in dependence connects the latter to the respective parameter inside `bb.php`. No parameter-out dependence is necessary, since the call is non-returning.

5.2.3.2 Document object model (DOM)

Figure 5.6 is related to the possibility of having data dependences that cross the boundaries of the procedures, without being associated to parameters or global variables. Interface events can trigger either the execution of the Javascript function `set_zero` on the client (loss of focus from the text field at line 7), or the transfer of control to the server program `bb.php` (click on the `Submit` button, line 6). Correspondingly, the event loop contains two call nodes. The input variable `w`, which is assigned the value 3 at line 7, is redefined in-

Figure 5.6: *Access to an invisible variable via the DOM.*

side the Javascript function `set_zero` (line 3). Although here this variable is out of scope (*invisible*), it can still be accessed by means of the DOM (`document.Eform.w.value`). Invisible variables that are accessed via the DOM can be handled by introducing an additional input and an additional output parameter to transmit their values into and out of the called procedure. This is the reason for the two additional parameters, `w_in` and `w_out`, of function `set_zero` indicated in the SDG. In this case, resolution of the invisible variable reference based on the DOM can be achieved by means of a simple static analysis of the source. The two names, `w` at line 7, and `document.Eform.w` at line 3, refer to a same variable, because line 7 is inside a top level form named `Eform`. More generally, HTML variables referenced by client code can be located by traversing the document elements listed in the DOM access path. This may become difficult in presence of "extreme" HTML code generation (see below).

5.2.3.3 Double callback

Interface events may trigger the execution of more than one callback in sequence. To represent such cases in the SDG, an additional node is introduced for the graphical event, instead of the edge labeled *E*. Its children are the related calls. The example in Figure 5.7

Figure 5.7: *Double call in the event loop.*

associates the Javascript function `set_zero` and the PHP program `bb.php` to the click of the mouse on the `Submit` button (respectively, at lines 10 and 8). The effect of such double association is that the Javascript function will be executed first, and the PHP program will receive the control of the execution then. The node labeled E in the event loop is connected to such two calls, properly ordered. Note that there is no data dependence between node 9 and the input parameter `w_in` of `bb.php`, because the call to `bb.php` is always preceded by a call to `set_zero` which redefines the value of `w`, and no alternative path exists from 9 to the call of `bb.php`.

As with slicing of traditional software, the need of including variable declarations for the variables included in a slice is not represented explicitly with a specific dependence, and is assumed implicitly. This applies to the declaration of `x` at line 3 of Figure 5.7, in case `x` is used in the slice, but there are other situations reducible to this one. The declaration of the session variables and the start-up of the session (lines 2-3 of Figure 5.5) are another example.

5.2.3.4 HTML code generation

Figure 5.8: *Generation of HTML code from PHP.*

Figure 5.8 shows an example where the HTML code of the dynamic page produced in output has not got a fixed structure. There are some fixed parts, the form opened at line 1 and closed at line 8, and the submit button created at line 7, but the input text fields of the form are created dynamically at lines 2-6, within a loop iterating a number of times, `$N+1`, which is also computed at run time (not shown). The names of the variables associated with these inputs are constructed dynamically from the loop index (`x0, ..., xN`). Their initial value is equal to the loop index (`0, ..., N`). Consequently, the number of parameters passed to the server program `bb.php` is variable, function of `N`. On the server side, the program `bb.php` constructs dynamically the names of the parameters from `$N`, assigning them to the array `$z` (line 15). Then, parameter values are retrieved (line 16) and accumulated in variable `$sum` (line 17).

Construction of the SDG for this code fragment is remarkably difficult, due to the dynamic generation of a part of the code itself, namely the input elements of an HTML form and the parameters of a server program. Figure 5.8 shows what an "ideal" code analyzer is expected to generate, that is, a variable number of parameters (ellipsis) with an associated set of data and parameter-in dependences. In its most general form, the problem is that the code under analysis generates the code actually interpreted by the browser. Such a generation process includes constant strings (such as `" VALUE="`), concatenated with strings computed dynamically. Consequently, the code generation instructions (lines 1, 5,

7, 8 of Figure 5.8) build the strings that are the real object of the analysis. For a static analysis of the source, the variable parts of such strings have to be reduced to constants, whenever possible, or otherwise conservative assumptions have to be made about them. A safe approximation of the set of constants that possibly replace each string variable can be obtained, for example, by means of copy-constant propagation or symbolic execution techniques. In the example of Figure 5.8, the names of the input variables, `$y[$i]`, cannot be determined by copy-constant propagation. They are therefore labeled as *nonconstant* values, and the names of the associated variables are *unknown*. A conservative choice about such variables is to assume that their name could collapse with any other name in scope, and to represent them with a single parameter node in the invocations. Better approximations could be obtained with ad-hoc analyses.

5.2.4 An online travel agency Web application: case study

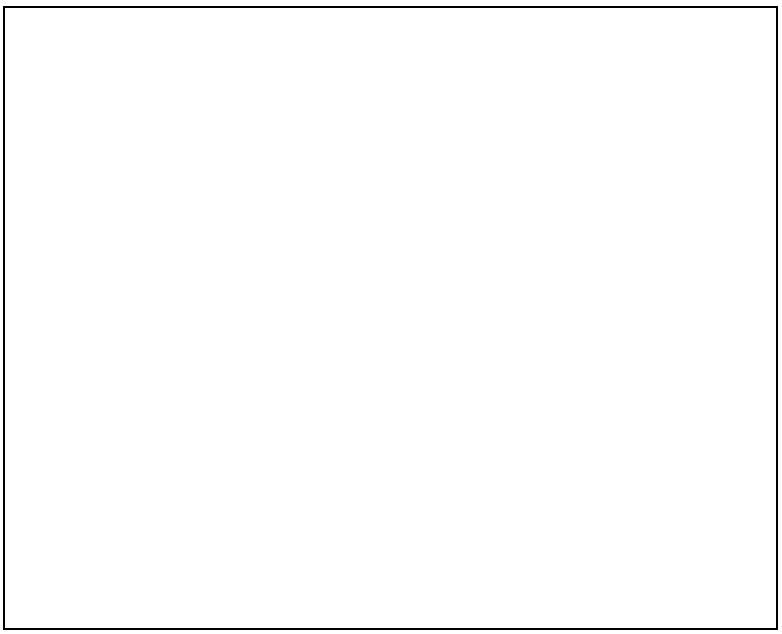

Figure 5.9: *An on-line Travel Agency application.*

Let us consider the Web application for a travel agency shown in Figure 5.9. This simplified Web application, obtained by abstracting the main characteristics present in a set of real travel agency applications downloaded from the Web, was implemented by the author using PHP and Javascript.

In this Web application, users can select hotels, cars and flights and reserve/rent/book them on-line. For the sake of simplicity, the pages displaying error messages are not

reported. When users select an option of interest from the page `index.php` (Figure 5.9.a), a dynamic page `choose.php` is created, as shown in Figure 5.9.b. If, for example, the option *Hotels* was selected, users are asked to input the city where they want to stay and the arrival and departure dates. After clicking the submit button, the dynamic page `selectHotel.php` (Figure 5.9.c) appears, showing a list of available hotels in the chosen city. Users can select one of them or change the option of interest passing from hotels to cars or flights. If users decide to select a hotel and press the related submit button, the dynamic page `hotelPrice.php` is created, as shown in Figure 5.9.d. This page contains the chosen hotel, its daily price and the total cost for the entire stay. In the same page the user can insert name, surname and payment type to reserve a room in the selected hotel. The dynamic page `reserveHotel.php` (Figure 5.9.e) is then displayed. It contains a simple report confirming the reservation. Now users can return to the homepage and select cars or flights.

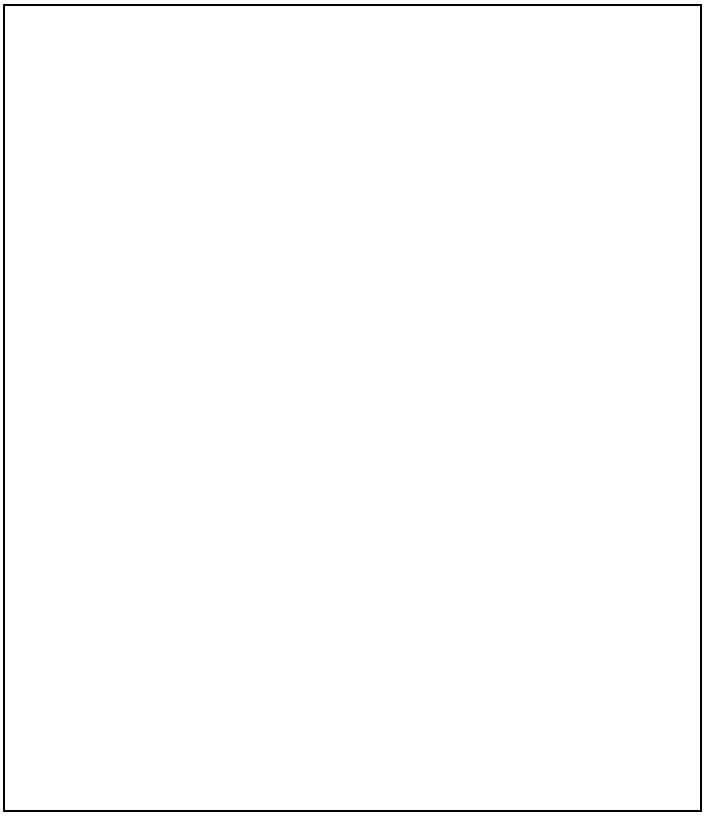

Figure 5.10: *Page* `choose.php`.

The values inserted by the users inside input forms are stored in session variables and are used by the Travel Agency application to propose default values in successive insertions. For example, dates (arrival and departure) and city inserted by the users to reserve a hotel are proposed as a default in the reservation of cars or flights.

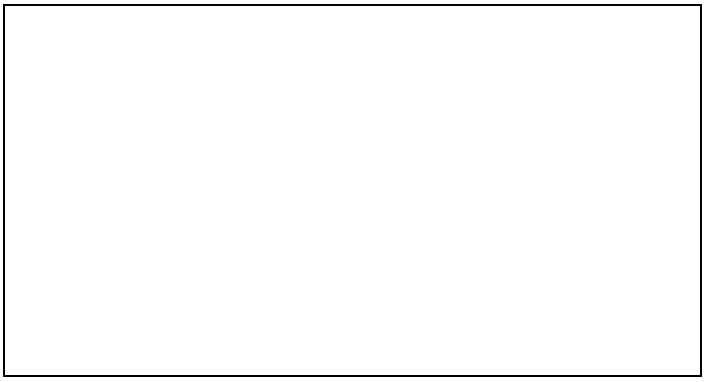

Figure 5.11: *Page* `index.php`.

Figure 5.10 contains a portion of the PHP source code of the server program `choose.php`. The content of the dynamic page produced by this server program depends on the selection made in the previous page, `index.php` (see code in figure 5.11). The value for such a selection is stored in variable `$x`. If the option *Hotels* was selected, the test at line 30 succeeds, while tests at lines 68 and 90 fail. Lines between 32 and 46 recover (when available) the values inserted previously, to rent a car or book a flight. They become the default values of the form generated at lines 54-64. Before invoking (line 54) the server program `selectHotel.php`, the dates inserted by the user are checked by the Javascript function `check_date.php` (see Figure 5.12), called when there is a loss of focus from the text fields (lines 58 and 60).

Figure 5.13 contains the PHP code of the server program `selectHotel.php`. In this program a connection with the MySQL database *TravelAgency* is established (lines 151-153). The array `$result` (line 155) is assigned the list of hotels available in the city chosen by the user in the previous page. This list is recovered by the SQL query at line 154, on the table *Hotel*, consisting of the columns *name, city* and *price*. The session variables `$Globalcity`, `$GlobalinitDate` and `$GlobalendDate` are assigned their values at lines 162-173, after converting the dates to timestamps. The list of available hotels is printed inside a pull-down menu of the page being generated (lines 201-204). The two forms, respectively for submitting the chosen hotel to `hotelprice.php`, and for changing option and passing to cars or flights, are created at lines 199-207 and 209-218 (portion not shown).

Figures 5.14 and 5.15 contains the PHP instructions of the server programs `hotelPrice.php` and `reserveHotel.php`. The first program recovers the daily price of the hotel chosen by the user, by means of a query (line 224) on the database *TravelAgency*, table *Hotel*, and stores it into variable `$priceForDay` (lines 226 and 242). This value is used at line 244 to compute the total cost of the stay. At line 245, the total cost of the hotel is stored into the session variable `$GlobalCostHotel`. The server program `reserveHotel.php` accesses the session variables stored previously to build a report confirming the reservation.

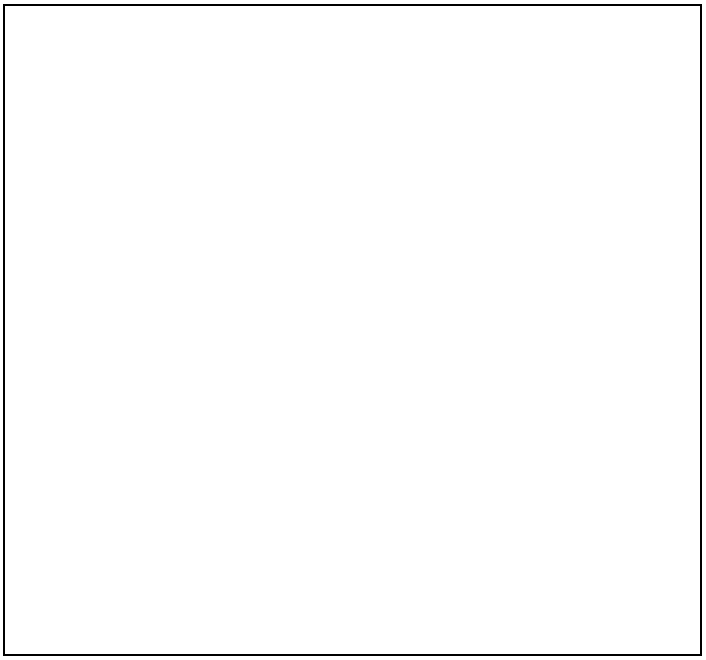

Figure 5.12: *Javascript Function* `check_date`.

The code associated with the pages `selectCar.php` and `selectFlight.php` is similar to that of `selectHotel.php`. `carPrice.php` and `flightPrice.php` are similar to `hotelPrice.php`, while `rentCar.php` and `bookFlight.php` are similar to `reserveHotel.php`.

5.2.5 Slice on variable `$GlobalCostHotel`

The SDG of the Travel Agency application has been built (a portion of it is shown in figure 5.16), and a slice at the output node 296 on variable `$GlobalCostHotel` of `reserveHotel.php` has been determined. The resulting PHP/HTML/Javascript/SQL statements have grey background in the Figures 5.10, 5.12, 5.13, 5.14 and 5.15. At the end of each user interaction, this slice displays the same value of variable `$GlobalCostHotel` as the initial Web application. All statements in the original Web application involving the variables `name`, `surname`, and `payment` have been removed, being not relevant for the computation of the total hotel cost.

It is interesting to note that the SQL query at line 224 (Figure 5.14) contributes to the slice. In fact, the hotel daily price (line 242) depends on the result of the query (dependences: 224 $\rightarrow$ 225, 225 $\rightarrow$ 226, 226 $\rightarrow$ 242). In this simple example, the code for inserting values into the database accessed at line 224 is not included. Consequently, slicing can stop at line 224, as regards the database values, assuming that the database is already in place, with

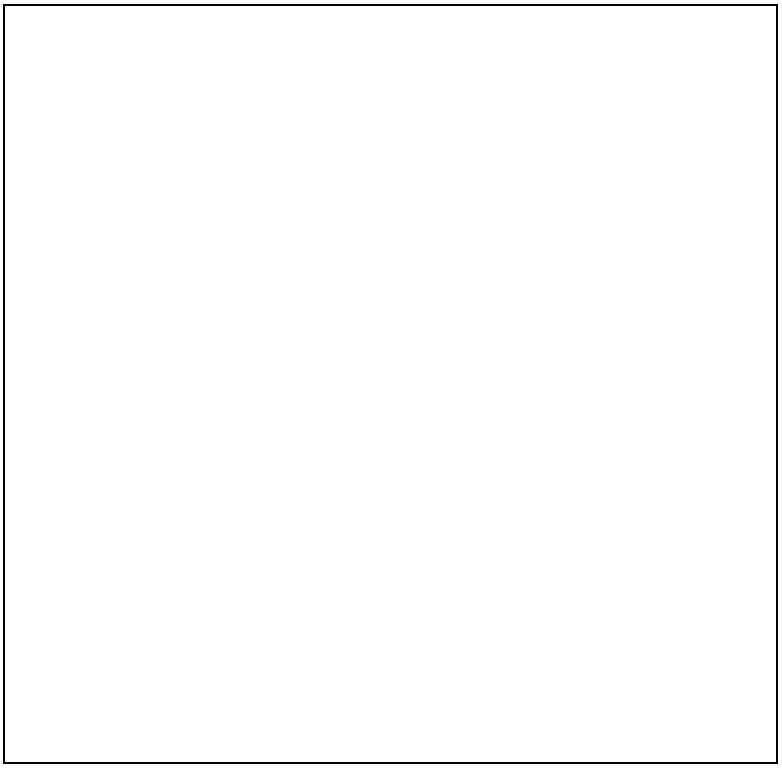

Figure 5.13: *Page* `selectHotel.php`.

all necessary tables filled in. A more complex Web application may include also dynamic pages for the insertion of values into a database. Slice computation has to take database insertion statements into account, since they originate additional data dependences. The SDG representation of the application does not change, since it is possible to conservatively treat each database table column as a global variable, without distinguishing among the different records. The only additional problem is that the SQL code has to be analyzed as well. Since it is generated dynamically, in a similar way as the HTML code, its analysis may be troublesome (and in general infeasible), and approximations obtained by constant propagation or symbolic execution may be necessary. With reference to the SQL query at line 224, a slicer which handles also the SQL code should detect the usage of the column `name` of table `Hotel`. Such a value can be considered equivalent to a global variable (for example named `_db.TravelAgency.Hotel.name`), and as such it becomes an additional input parameter of `hotelPrice.php`. The SQL statements inserting records into table `Hotel` have an indirect (mediated by parameter-passing) data dependence with the query at line 224, and thus would be included in the slice. A similar argument can be made for the SQL query at line 154.

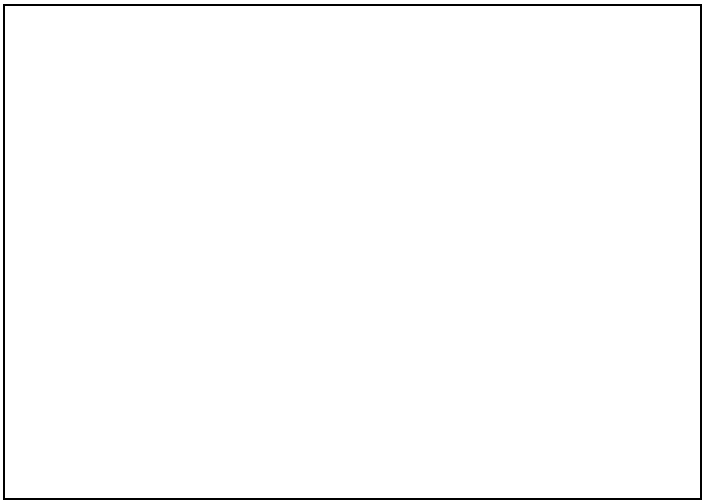

Figure 5.14: *Page* `hotelPrice.php`.

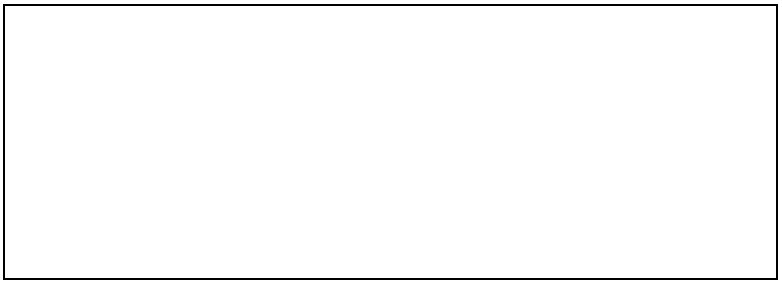

Figure 5.15: *Page* `reserveHotel.php`.

5.3 Dynamic slicing

Dynamic slicing [KL88] is achieved by considering an execution trace in addition to the source code of an application. The transitive closure of the dependences between elements in the trace gives the dynamic slice, which can be eventually mapped to a subset of the original source statements. In order to produce the trace, an input vector with values for all input variables has to be selected. The result of slicing is a subprogram computing the same value of a given variable at a given statement for the input vector associated with the execution trace.

For a Web application, the execution trace consists of the sequence of statements executed on the server/client, plus the set of HTML pages encountered during the navigation. Some of them may be static, but some may also have been generated dynamically. In contrast to static slicing, when dynamic slices are computed the HTML code generated by the server programs needs not be approximated, as it is completely known. Therefore, all of

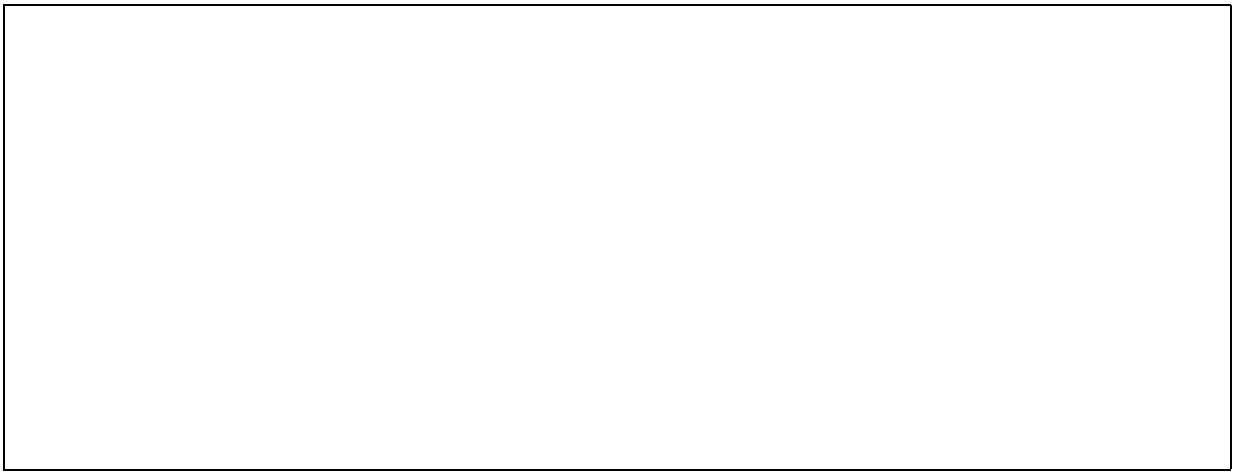

Figure 5.16: *SDG of a portion of Travel Agency Web application.*

the problems outlined in the previous section with reference to the dynamic generation of the HTML code (see example of Figure 5.8) disappear with dynamic slicing. The price to pay is a loss of generality. In fact, the slice reproduces the original behavior of the Web application only for the considered input vector.

To map the execution trace of a Web application to its source code, it is necessary to introduce an additional dependence, associated with HTML code generation. When an HTML statement is reached during dynamic slicing, the original source code statement that built it has to be included in the dynamic slice.

Definition: *a* build dependence *holds between a server program statement and an HTML statement if the former generates the latter.*

The tracer to use for dynamic slicing of Web application will have to be able to trace server/client statement execution, to store the HTML pages traversed during the navigation, and to trace the association between each server statement generating HTML code and the code generated.

Figure 5.17 shows the execution traces obtained by invoking `aa.php` (its code is given in Figure 5.8) with `$N=1`, inserting the values 12 and 13 in the two input text fields of the HTML page produced by `aa.php` and displayed by the browser, and finally submitting the form to `bb.php` (the code of which is also given in Figure 5.8). The execution trace of `aa.php` is enriched with the build dependence information, while the trace of `bb.php` is shown with information on the variables defined at each statement and with the input parameters retrieved by each statement. The dynamic HTML code generated by `aa.php` is in Figure 5.17.

If a dynamic slice on variable `$v[0]` at the end of the execution of `bb.php` is determined,

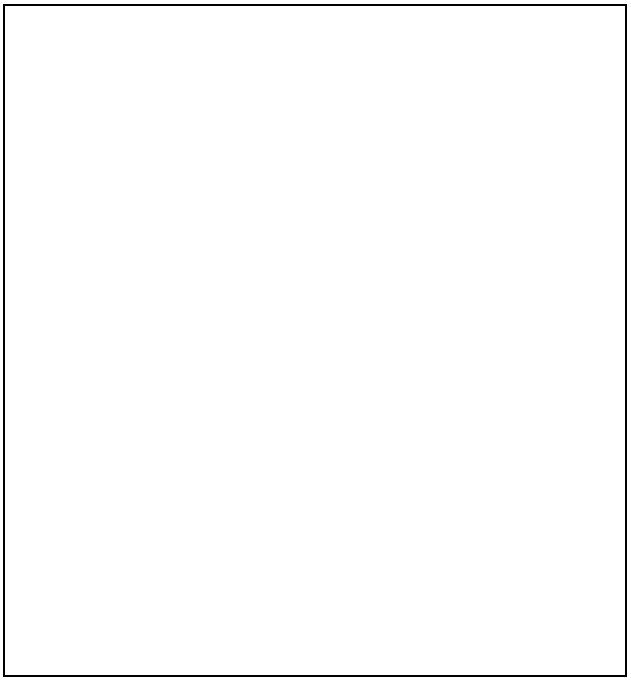

Figure 5.17: *Computation of a dynamic slice.*

the statements encircled in Figure 5.17 are obtained. The first statement to include in the slice is 16, since it defines the value of `$v[0]`, and no redefinition occurs until the end of execution. Data and control dependences lead to including statements 14 and 15. Moreover, statement 16 accesses the parameter `x0`, which is passed to `bb.php` at statement 20. This statement is generated by `aa.php` at statement 5, which is thus also included in the dynamic slice. Data and control dependences of `aa.php` lead to the inclusion of statements 2, 3, 4. Finally, the analysis of the HTML code generated by `aa.php` results in the need for the slice to build also lines 19, 22 and 23, because of nesting control dependences in the HTML code (line 22 is necessary to make the form executable). Correspondingly, the statements 1, 7, 8 of `aa.php` are included in the slice, being responsible of generating the HTML lines 19, 22, 23.

5.4 Related works

To the author knowledge, Web application slicing has not been considered in the literature. Slicing of traditional Software systems was proposed for the first time by M. Weiser [Wei84]. Since then it was the object of several studies and several variants of the initial idea have been considered.

Conditioned program slicing [CCL98, DFHH00] allows the specification of constraints on the input variables to further restrict the code portions isolated in a slice. Amorphous slicing [HD97] allows performing additional simplifications on the resulting slice by means of transformation rules. Other extensions of the original idea deal with the usage of dynamic information in addition to the static one [KL88], with the possibility to identify reusable functions through program slicing [CLM95] and with testing [GHS96, HD95, KFS93].

Other works are related to the difficulties of computing slices according to the syntactic constructs permitted by the programming language of choice. Some examples are the problem of computing slices on programs with *goto* statements [CF94], the problem of computing interprocedural slices [HRB88], of slicing in presence of pointers [LB93, TAFM97].

The problem of applying the slicing at other language paradigms (e.g functional, logic and object oriented) has been extensively faced. An approach of object oriented code slicing is in [LH96]. Slicing has been applied also to concurrent languages.

Part III

TESTING

Chapter 6

Structural Web Application Testing

This chapter describes a technique useful for testing Web applications. This technique, derived from structural testing of traditional software systems, is based on the Web application model presented in chapter 3. Structural Web application testing is implemented in a tool called TestWeb.

6.1 Introduction

Web applications can involve a complex interaction among Web browser, operating systems, plug-in applications, communicating protocols, Web servers, databases, server programs (for example CGI programs) and firewalls. Such complexity makes the testing of Web Applications a great challenge [Mil98]. Ideally all components and functionality of a Web application on both client and server sides should be completely tested. However, this is rarely possible in modern Web application projects because of the extreme time pressure under which Web systems are developed. Available testing techniques [Mil98] differ on the features of the Web application we want to test. For example it is possible to execute (the list is not complete):

- **functionality testing**, i.e. searching behavioral deviations from specifications and/or user expectations;
- **link testing**, i.e. searching pending links, which reference a non existing page;
- **HTML validation**, i.e. checking the HTML syntax, in particular checking HTML documents for conformance to W3C HTML and XHTML recommendations and other HTML standards;

- **performance testing**, i.e. assessing the response times (the time necessary to obtain the required page on the browser);
- **load testing**, i.e assessing the number of clients that can be simultaneously served;
- **security testing**, i.e testing firewalls, encryption, user authentication, financial transactions, access to databases and in general testing the overall protection of a Web application against unauthorized internal or external access.

We concentrate our attention on functionality testing, in particular on *dynamic validation*. In general, dynamic validation methods aim at exercising the system by supplying a vector of input data (*test case*) and comparing the expected outputs with the actual ones after execution. We consider *structural* (white box) *testing* of Web Applications: the internal structure of a Web application is accessed to measure the coverage that a given *test suite* (collection of test cases) reaches, with respect to a given *test criterion* (stating the features to be tested). Some white box testing criteria, derived from those available for traditional software [Bei90], are: *page testing, hyperlink testing, definition-use testing, all-uses testing and all-paths testing.*

The chapter is structured as follows. Before considering our approach of structural testing (section 6.3), integration testing is positioned with respect to the various aspects involved in Web application testing (section 6.2). Section 6.4 presents a case study in order to clarify our structural testing approach. A discussion of related works concludes the chapter.

6.2 The testing process

In [Con00] the three most common architectural styles for the design of Web applications are classified as follows:

Thin Web Client Only the standard facilities of the client-side browser are exploited (including forms) and all business logic is executed on the server.

Thick Web Client A portion of the business logic is delegated to the client, where applets, Javascript, etc. can be used, but the communication protocol between client and server remains HTTP.

Web Delivery Executable programs running on the client establish with the server a communication based on an ad-hoc protocol different from HTTP.

Actually, a continuum exists between the extreme points of the above classification. At one end, the Web Delivery style becomes a traditional client-server application, defining

its own communication protocol, and the Web is just the underlying infrastructure. At the other end, applications adhere completely to the HTTP protocol, the server provides HTML pages without executable parts and all information exchange is mediated by the standard facilities provided by HTML (e.g., forms, links with parameters).

The position that a Web application occupies in this continuum makes relevant differences in the way testing can be conducted. As noted above, the main distinctive feature of Web applications (those closer to the Thin Web Client) is the usage of the HTTP protocol for message exchange. This is also the main feature that makes testing these applications different from testing traditional programs. The HTTP protocol can be exploited to automate test execution, and the organization of the interaction according to the standard facilities of HTML simplifies test case generation. While departing from this style and introducing more elements that are typical of client-server applications, the benefits of HTTP are lost and the testing techniques become those employed for traditional client-server software.

In the next section we will describe an approach to integration testing of Web applications that exploits the HTTP protocol to increase the level of automation. Before considering our approach, integration testing is positioned with respect to the various aspects involved in Web application testing and the different activities conducted during the testing processes.

As with traditional software, testing can be performed to verify either the functional or the non functional requirements. Among the non functional requirements that are typical of Web applications, performance is maybe the most important one.

The test of the functional requirements can be conducted by considering the Web application as a black-box. In this case it is exercised by partitioning legal inputs into equivalence classes and one or more test cases for each class are defined. Output pages obtained by navigating the application and providing such inputs are compared with the result expected from that interaction according to the system specifications. Each deviation from the expected behavior will be called an *failure* in the following. Boundary values are inputs that deserve special attention and therefore specific test cases have to be defined for them. Alternatively, the Web application is considered as a white-box and testing exploits knowledge about the internal functions of the application. In particular, server and client side code, HTML pages and messages exchanged via HTTP are assumed to be known and can drive the definition of the test cases. A widely used example of this kind of test is called *link validation*. It aims at verifying that every link in every page generated by the application is a valid link, i.e., no error is reported when it is traversed. In the next section other white-box testing techniques are described, based on the notion of coverage.

Different *levels* of testing can be conducted on a Web application, similarly to a normal software system:

Unit Test Singular components are tested and stubs/drivers replace missing parts. For

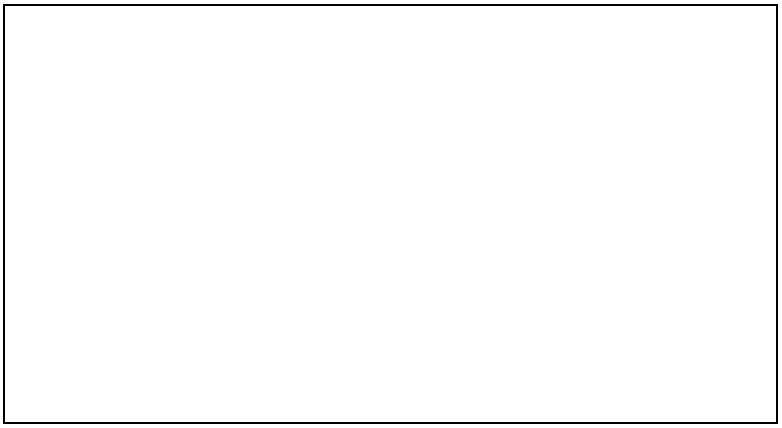

Figure 6.1: *Capture/replay of graphical events vs. HTTP messages.*

example, a server script is invoked by a driver which simulates the browser and receives the HTML page as the resulting output. Another example is a Javascript program which validates fields in a client side page and sends a request to a fictitious server simulated by a stub.

Integration Test Pages are composed and integrated with server programs. The tester can now navigate from page to page and requests can be passed from the browser to the Web server via HTTP. This testing phase is strongly based on the protocol exploited, and, when this is HTTP, it can be supported by ad-hoc techniques (see next section).

System Test The system is validated as a whole in an environment as similar as possible to the real, target environment.

Acceptance Test The customer installs and runs the application in its own environment.

Regression Test During evolution, the preservation of previous functionalities is checked by re-running the test cases defined for them. Exploitation of the HTTP protocol can positively affect also this testing activity.

Among the various testing levels, those for which the difference with respect to traditional software is more relevant are *Integration Test* and *Regression Test*, in that these levels heavily depend on the communication protocol used for client-server data exchange. Web applications have their own protocol (HTTP) and the testing methods used can take advantage of it. For the other levels, the methods usually adopted for traditional software systems can still be used in the context of Web applications.

During Integration/Regression Test of a traditional application the interaction with the user is simulated by generating the graphical events that trigger the computation from

the application interface. One of the main methods used to obtain this result is based on capture/replay tools, which record the interactions that a user has with the graphical interface and repeat them during regression test (Figure 6.1, top). Although this can still be done with Web applications, and is recommended for applications closer to the Web Delivery than to the Thin Web Client, availability of the communication protocol HTTP, which separates user interactions with the browser from server side computations, offers the opportunity to define ad-hoc techniques. HTTP messages can replace the graphical events generated by a capture/replay tool during test case execution (Figure 6.1, bottom).

6.3 Structural Web testing

When performing *structural testing* [Bei90] (see appendix 12.5 for a short summary on traditional software testing) of a Web application, its explicit-state model is accessed to measure the coverage that a given collection of test cases reaches, with respect to a given *test criterion*. A *test case* for a Web application is a sequence of pages to be visited plus the input values to be provided to pages containing forms. Therefore it can be represented as a sequence of URLs specifying the pages to ask and, if needed, the values to assign to the input variables. Execution consists of requesting the Web server for the URLs in the sequence and storing the output pages. Differently from traditional software, branch selection can be forced by choosing the associated hyperlink, without having to track conditions back to input values. Some white box testing criteria, derived from those available for traditional software [Bei90] and implemented in **TestWeb**, are:

- **Page testing**: every page in the site is visited in at least one test case.
- **Hyperlink testing**: every hyperlink from every page in the site is traversed at least once.
- **Definition-use testing**: all navigation paths from every definition of a variable to every use of it, forming a data dependence, are exercised.
- **All-uses testing**: at least one navigation path from every definition of a variable to every use of it, forming a data dependence, is exercised.
- **All-paths testing**: every path in the site is traversed in some test case at least once.

The definition-use and all-paths criteria are often impractical, since there are typically infinite paths in a site, if loops are present. They can be satisfied if additional constraints are imposed on the paths to be considered. Examples are loop k-limiting and path independence. In the first case, loops are traversed at least k times, while in the second case

only *independent paths* are considered. A path is *independent* from a given set of paths if its vector representation is linearly independent from that of any other path in the set.

When designing and executing test cases for a Web application, not all pages hold the same interest. Static pages not containing forms can be disregarded, since they do not collect user input, process it or display results. They contain fixed information that need not be examined during dynamic validation. The site can thus be reduced, while retaining only the relevant entities. Given the graph representation of a Web application (*structural view*), a *reduced graph* can be computed for the purposes of white box testing in the following way: each static page without forms is removed from the graph and all its predecessors (if any) are linked to all its successors (Cross-Term step, see Figure 6.2). In the resulting graph, a fictitious *entry* node is added, connected with all nodes with no predecessor, and a fictitious *exit* node is directly reachable from all *output* nodes, i.e., dynamic nodes generated by server programs with non empty *use* attribute. In fact, in a Web application the end of a computation is reached when some result is displayed to the user, but no intrinsic notion of termination for a navigation session exists. Therefore, dynamic pages whose content depend on the user input are good candidates for ending meaningful computations.

Figure 6.2: *Step of the Cross-Term algorithm at a node selected for removal.*

Satisfaction of any of the white box testing criteria involves selecting a set of paths in the Web site graph and providing input values. Since path selection is independent of input values, given the specific nature of branching in Web applications, it can be based exclusively on the graph structure of the explicit-state model. Specifically, test cases can be generated by computing the path expression [Bei90] of the reduced Web site graph. A *path expression* is an algebraic representation of the paths in a graph. Variables in a path expression are edge labels. They can be combined through operators + and $*$, associated respectively with selection and loop. Brackets can be used to group subexpressions.

Let us consider the Web application model in Figure 3.6, reported in Figure 6.3 with an additional exit node X, reachable from the two dynamic pages that end meaningful computations (D_1 and D_2), and with labels on the edges. The related path expression is:

$$e_1 e_3 (e_5 e_3)^* e_7 + e_2 e_4 (e_6 e_4)^* e_8$$

The initial selection of which edge to traverse leads to two alternative sub-paths, accounting for all possible navigations. In each sub-path, the possibility to loop through the edges

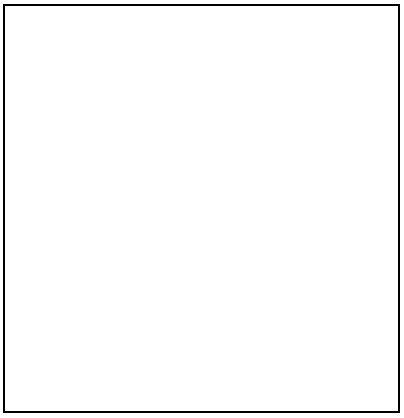

Figure 6.3: *Dynamic pages D_1 and D_2 are connected to the additional exit node X.*

e_5, e_3 and $e_6 e_4$ is respectively provided. Finally, edges e_7 or e_8 lead to the exit node.

Figure 6.4: *Reduction algorithm.*

Computation of the path expression for a site can be performed by means of the Node-Reduction algorithm described in [Bei90] and depicted in Figure 6.4. The lines 1-3 of the Reduction algorithm initialize the process and put the graph in normal form. The body of the command *while* is executed until the number of nodes in the graph is greater than 2. Line 5 assigns a node of the graph different from the initial or final node to variable n, while line 6 executes a Cross-Term step on node n. This step eliminates the node n and transforms the graph according to the diagram shown in Figure 6.2. Lines 7-10 combine all serial and parallel links and remove self-loops. At the end of the execution, the path-expression of the input graph is obtained.

Since the path expression directly represents all paths in the graph, it can be employed to generate sequences of nodes (test cases) which satisfy any one of the coverage criteria. Determining the minimum number of paths, from a path expression, satisfying a given criterion is in general a hard task. However, heuristics can be defined to compute an approximation of the minimum. The heuristic technique adopted for this work is based on the scheme of Figure 6.5 (the alternative at line 2 for a loop is whether to re-iterate

Figure 6.5: *The heuristic technique adopted to obtain the paths satisfying a criterion.*

or not). Definition-use and all-uses testing can be achieved by considering, for each data dependence, the definition as *entry* node and the use as *exit* of the subgraph to be tested.

Once test cases are generated from the path expression of the site, the input values for the variables collected through forms can be obtained from those inserted into *formInputFile.txt* (either those stored by the *Spider* in the field *input* or those in *altInput*, see chapter 3). Otherwise, the test engineering can insert new input values provided that they belong to the same equivalence class of values stored into *formInputFile.txt*.

The resulting test cases can be executed automatically and, after downloading, the output pages can be inspected to assess whether the test cases have been passed or not.

Regression testing highly benefits from the automation described above, since each test case can be re-executed unattended on a new version of the Web application, and its output pages can be automatically compared with those obtained from a run of the previous version.

With reference to the example in Figure 6.3, the number of independent paths is 4 and their coverage can be achieved for example by traversing $p_1 =< e_1, e_3, e_7 >$, $p_2 =< e_1, e_3, e_5, e_3, e_7 >$, $p_3 =< e_2, e_4, e_8 >$ and $p_4 =< e_2, e_4, e_6, e_4, e_8 >$. The four associated test cases ensure the all-paths coverage, restricted to independent paths, which implies hyperlink and page coverage. The input values that can be used to traverse the four independent paths are easily obtained as follows: in p_1 one edge, namely e_1, and in p_2 two edges, namely e_1 and e_5, have a server program, S_1, as target. The dynamic page built by S_1 is D_1, which has <`corporation, "atd"`> as *input* couple and <`state, "1"`> as *hidden* couple. Therefore, the test case which allows traversing p_1 is:

```
(S, corporation="atd", state=1)
```

while for p_2 it is possible to use – exploiting also *altInput*:

```
(S, corporation="zpm", state=1)
(S, corporation="atd", state=1)
```

From the initial page H the server program S is requested for the stock market service (`state=1`) with proper input. The result is the dynamic page D_1. Up to this point, the edges e_1 and e_3 have been traversed: p_1 is covered, since navigation is assumed to stop here (implicit traversal of e_7). In the second test case, by invoking S for a second time, edges e_5 and e_3 are traversed, and D_1 is regenerated: p_2 is also covered. The test cases for p_3 and p_4 can be generated and executed similarly.

Once the dynamic pages D_1 and D_2 have been manually checked for correctness, they can be recorded as the oracles of these four test cases, so that regression testing can be fully automated.

6.4 Case study

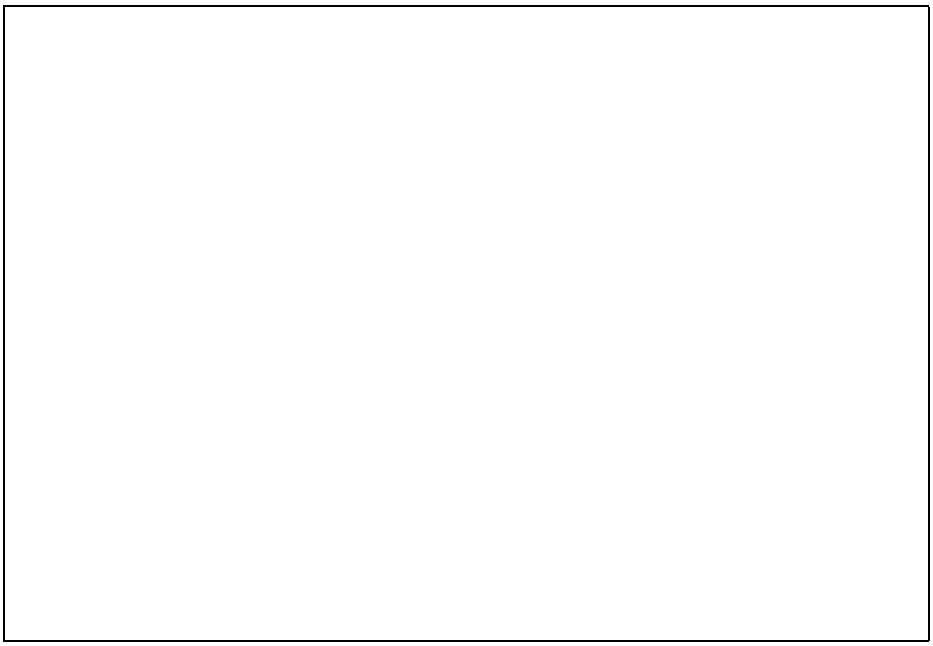

Figure 6.6: *Structural view of the Web application ITC-publications with an additional exit node X, reachable from the dynamic pages. Nodes with grey background are server programs.*

Let us consider the Web application **ITC-publications** (already presented in section 3.3) and in particular the model in Figure 3.8, reported in Figure 6.6 a little simplified, with an additional exit node X (only reachable from the dynamic pages that end meaningful computations) and with labels on the edges.

The first step of the structural testing activity is the construction of the path expression,

that is automatically computed by **TestWeb**. The paths represented in the path expression go from the initial page `db.htm` to the fictitious exit node X. 11 loop (*) and 5 alternative operators (+) are present (see Figure 6.7).

Figure 6.7: *Path expression of the* **ITC-publications** *model (edges Ei lead to the exit node X, ':' represents concatenation).*

Four independent, feasible paths were computed by **TestWeb** from the path expression (see Figure 6.8) and associated with 4 corresponding test cases. This set of 4 test cases provides 100% coverage according to the page and hyperlink testing criteria.

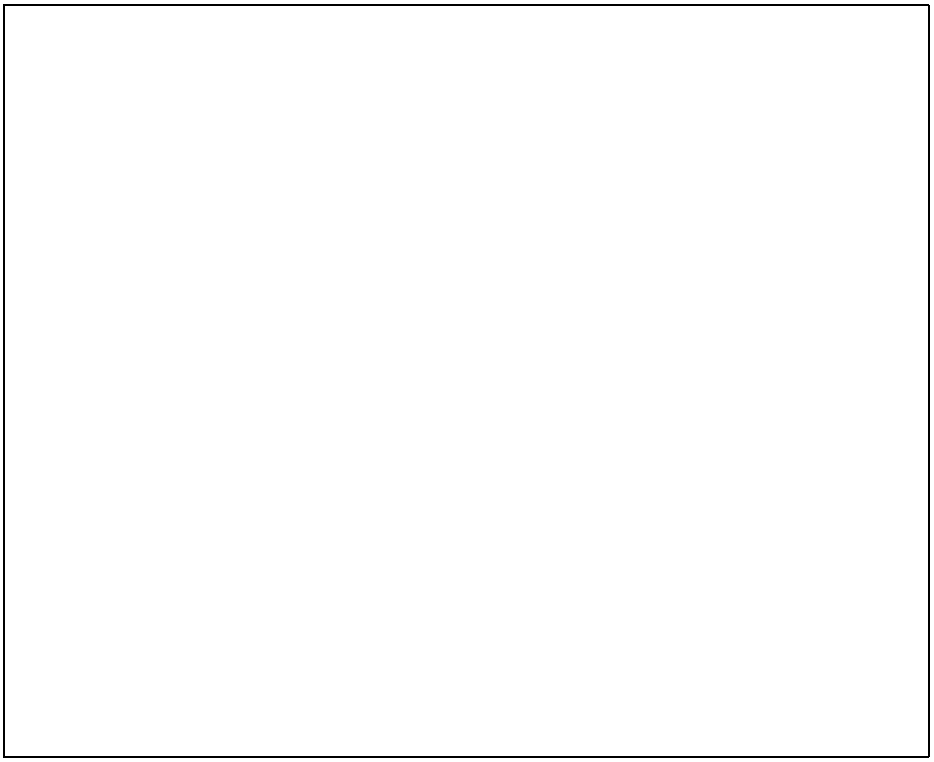

Figure 6.8: *Independent paths computed by* **TestWeb** *from the path expression of Figure 6.7.*

Execution of the test case number 1 highlighted an anomalous behavior, which may be regarded as a defect of the server program `ext-search.idc` used by the **ITC-publications** application. In fact, the first form of the initial page (searching by **abstract**) does not work at all. For every input of the variable **abstract**, it gives the entire database in output.

Another possible improvement area, highlighted by the test case number 1 with alternative input `ref_number=0102-01`, is associated with the following behavior: the search by **reference number**, filled-in with correct input, produces the dynamic page `ext-search.idc`, showing a list consisting of just one button (leading to the paper matching the reference number). At this point, the user has to click the button to load the dynamic page `ext-view.idc`, that shows the details on the searched paper. A better solution would be to redirect the input directly to the server program `ext-view.idc`, without generating the dynamic page `ext-search.idc`.

The execution of the other test cases did not highlight other defects or improvement areas of the server programs used in **ITC-publications**.

6.5 Related works

Recently, a few testing tools have been proposed to support functional testing of Web applications (e.g., [Mil98]). These black-box testing tools are based on capture/replay facilities: they record the interactions that a user has with the graphical interface and repeat them during regression testing. Our approach is a useful complement to functional testing: it permits exercising a selected set of feasible paths.

Liu et al. [LKHH01] extended traditional data flow testing techniques to Web applications. The data flow information is captured using various flow graphs. Control flow graph and interprocedural control flow graph are used to discover def-use chains present in the scripting portion of a client Web page or present in a server program, while object control flow graph and composite control flow graph permit to compute two more types of def-use chains. The first one is associated with different function invocation sequences, depending on the user interaction (scripting events), while the second captures def-use chains between different pages, where a variable is defined in a page and is used in a server program. Data flow testing for Web applications can be accomplished in five levels: *function level*, used to test a function for the variables that have def-use chains occurring within the function, *function cluster level*, used to test a cluster of functions within a client page or a server program for the variables whose def-use chains involve more than one function, *object level*, used to test the interactions of all functions in a client page, *object cluster level*, used to test a cluster of pages for the variables that have the *definition* in a page and the *use* in another page (server program), *application level*, used to test the shared variables, manipulated concurrently by the clients of the Web application. Our approach differs from Liu et al.'s [LKHH01], in that it is based on the model paths instead of the data flows. Restricted to data flow testing, our technique is equivalent to their *object cluster level*; we draw attention to the chains between different pages, where a variable is defined in a form or in a scripting portion of a page and is used in a server program. The approach we

propose takes advantage of some features of the Web applications (e.g., data transmission between client and server pages via the HTTP protocol), thus resulting in a higher level of automation than that of Liu et al. [LKHH01].

The tool described in [JL01] supports the specification of test cases in XML and provides powerful facilities to encode the related oracles. Operations such as retrieving a node in the abstract syntax tree of the resulting page and checking a value (also by means of regular expressions) are available.

The authors of [LFFC02] exploit a slightly different format for the specification of test cases, based on decision tables that include input values for each execution variant and expected outputs. Execution is automated by their tool **WAT**. We share with these works the ability to automate test case execution and the focus on functionality testing.

The paper [EKR03] proposes a Web application testing approach that utilizes data captured in user sessions (stored in a modified log file) to create automatically test cases. The authors describe two similar implementations of this approach, and a hybrid implementation that combines this approach with our structural testing. The idea is simple, instead of insert manually inputs in testcases, as proposed in our technique, the inputs are recovered automatically from the modified log file. The coverage is still guaranteed and the process is completely automatic. This approach can be considered an improvement of our proposal. Moreover Elbaum et al. in [EKR03] report results of an experiment comparing the different approaches proposed, assessing both the adequacy of the generated tests and their ability to detect faults.

Chapter 7

Statistical Web Application Testing and Analysis

This chapter describes statistical Web application testing, an alternative technique to structural testing approaches. This chapter proposes also some statistical analyses of Web applications derived from the analysis of the access log. Statistical testing is implemented in TestWeb while the statistical analyses proposed are implemented in a module of ReWeb.

7.1 Introduction

In functional (black box) testing of traditional systems software the program or system is subjected to inputs and its outputs are verified for conformance to specified behavior [Bei90]. Statistical testing follows the black box testing approach with some extensions and differences. In particular, inputs are chosen in different way with respect to functional testing. In statistical testing, sequences of input are stochastically generated based on a probability distribution that represents a profile of (anticipated) use of the software. The main benefit of statistical testing is that it allows the use of statistical inference techniques to compute probabilistic aspects of the testing process, such as: reliability (R), mean time to failure (MTTF) and mean time between failures (MTBF). Statistical software testing has been discussed in various forms in literature, a good paper on it is [WT94].

Statistical testing of Web application [TR02, TR03a] is a useful complement to structural testing in that it accounts for the typical interactions with the Web application, rather than its structure and data flows. In order to apply statistical testing to a Web application it is necessary building its *usage model*. The usage model is the structural view (chapter 4) of a Web application where every link toward other pages is labeled with a probability.

Values associated with links are computed using historical information, e.g. such as that contained in the log file, and represent the (conditioned) probability of navigating, at the next step, toward another page.

The usage model can be also exploited for statistical analysis. Using the usage model of a Web application is possible to determine the most/least accessed pages and the average number of clicks to be performed before reaching each Web page.

The chapter is structured as follows. The next section defines the usage model of a Web application. Statistical analysis and testing that are based on it are, respectively, presented in 7.3 and 7.4. Section 7.5 clarifies on an example our statistical testing approach. Related works concludes the chapter.

7.2 Usage model

In order to apply statistical analysis and testing to a Web application, it is convenient to build its usage model. The *usage model* is a representation of the statistics involved in the executions of a given Web application and in the input values provided. A natural choice of usage model for a Web application is a Markov chain [CT91]. In fact, each HTML page can be seen as a state and hyperlinks in the page can be regarded as Markov chain edges leading to other states. The usage model of a Web application can therefore be obtained from its structural view. The only missing information to make it a Markov chain is an estimate of the *transition probabilities* to be associated with the edges. Values for such probabilities can be computed by using historical information, such as that contained in the log file. *Transition probability* represents the (conditioned) probability of navigating, at the next step, toward another page.

Let us consider the Web application model in Figure 3.6 (also in Figure 6.3, with exit node and edge labels added). An example of related access log, here slightly simplified for the sake of presentation, is given in Figure 7.1. In particular, time stamps are assumed to be associated to each entry of the log, although not shown. The first column contains the name of the host requesting a Web page. The next column contains the name of the requested page followed by the input provided to the Web server (via `GET`). When requests coming from the same host are found within a proper time interval, it is assumed that navigation from a previously accessed page to a new one is taking place. Otherwise, a direct request of a page is considered to occur. Thus, the second request of page S made by `h1.d1.com` is considered to be issued from the page downloaded with the previous request. In other words, while the first request made by `h1.d1.com` causes the traversal of the edges e_1 and e_3 (see Figure 6.3), the second request comes from D_1 and results in the traversal of e_5 and e_3. When a request from a host is not followed by any other request from the same host, it is assumed that the edge leading to the exit node is followed. This corresponds

```
h1.d1.com  H
h1.d1.com  S & state=1 & corporation="ttt"
h2.d2.com  H
h2.d2.com  S & state=2 & currency="euro"
h1.d1.com  S & state=1 & corporation="bb"
h2.d2.com  S & state=2 & currency="dollar"
h3.d3.com  H
h3.d3.com  S & state=1 & corporation="cc"
h4.d4.com  H
h4.d4.com  S & state=1 & corporation="acb"
h5.d5.com  H
h5.d5.com  S & state=2 & currency="pound"
```

Figure 7.1: *Example of simplified access log.*

to termination of the navigation session. With reference to the log in Figure 7.1, after the second request of S from `h1.d1.com`, the edge e_7, leading to page X, is marked as traversed, in that no further request issued by the same host is present in the access log (within a reasonable time interval).

Edge	Count	Prob.
e_1	3	3/5
e_2	2	2/5
e_3	4	1
e_4	3	1
e_5	1	1/4
e_7	3	3/4
e_6	1	1/3
e_8	2	2/3

Table 7.1: *Estimation of the transition probabilities.*

The simple analysis of the access log described above allows marking each edge in the Web site model with the number of traversals resulting from the access log. With reference to the example in Figure 6.3 and the access log in Figure 7.1, the edge traversal count reported in the second column of Table 7.1 is obtained. Such values can be normalized into relative frequencies, approximating the probabilities of the related Markov chain, by dividing by the sum over the outgoing edges of each node. The result is an estimate of the probabilities of a Markov chain modeling the usage of the site, reflecting the *real world* requests arriving to the Web application. In practice, some page sequences in the access log may be not

recognizable as following legal connections between pages (infeasible sequences), due to the absence of intermediate pages which are cached, or to the usage of browser commands such as *back*, *forward* and *go-to*. As a first approximation, they can be skipped during the analysis of the access log. Another more complex solution could be obtained by extending the usage model with new edges (*back*, *forward* and *go-to*) and computing probabilities considering navigation through cached pages.

7.3 Statistical analysis

Properties of the Markov chains can be exploited to analyze a Web application model from a statistical point of view. Specifically, it is possible to determine the probability that the user be in a given Web page (stationary probabilities) and the average length of a path leading to each Web page from the initial one.

Stationary probabilities account for the probability of being in a given state, in an arbitrary number of Markov chain traversals. They are obtained by solving the following set of equations:

$$\pi_j = \sum_i \pi_i U_{ij} \tag{7.1}$$

where U_{ij} is the transition probability from state i to j.

The mean number of state transitions that occur before reaching a given state j from state i, in an arbitrary number of interactions, is obtained by solving the following set of equations:

$$m_{ij} = 1 + \sum_{k \neq j} U_{ik} m_{kj} \tag{7.2}$$

$$m_{jj} = \frac{1}{\pi_j} \tag{7.3}$$

With reference to the example in Figure 6.3 and the transition probabilities in Table 7.1, the stationary probabilities and the average paths reported in Table 7.2 are determined. Page H has the highest stationary probability (5/17) (equal to that of X), giving the probability that during an arbitrary interaction the user be in page H. The average number of transitions to be performed before reaching pages S_1 and D_1 (they are not distinguished since the former generates the latter, without any user interaction) is 10/3 (approx. 3.3). It is associated to a set of interactions that can either start from H and proceed along edge

Page(s)	Prob.	Avg	Short.	Ratio
H	5/17	17/5	0	-
S_1, D_1	4/17	10/3	1	3.3
S_2, D_2	3/17	6	1	6.0
X	5/17	12/5	2	1.2

Table 7.2: *Stationary probabilities; average and shortest paths.*

e_2, never reaching S_1, or start from H and reach S_1, traversing exactly 1 edge before S_1 is encountered. The average is 3.3 transitions (corresponding to mouse clicks) for S_1, and 6 for S_2. The shortest path to both pages is 1. As expected, it is lower than the average path.

The statistical analysis described above can be useful to determine the most/least accessed pages (stationary probabilities). Moreover, the comparison between average and shortest path to a page (see Table 7.1, last column) may suggest improvement interventions on the site structure. In fact, if the ratio between average and shortest path is high, navigation toward the given page appear to be requested less frequently than designed in the site structure. Possible actions are moving the page far from the home page or increasing its visibility in the home page, so that more users reach it. On the other side, a low ratio between average and shortest paths for pages which have a medium/long shortest path (say, more than 3-4 clicks) indicate that navigation toward such pages is frequently performed along the shortest path. If such pages are requested often (see stationary probabilities), it may be appropriate moving them closer to the home page (or introducing shortcuts to them).

Further interesting statistics that can be derived from the analysis of the access log are relate to the navigation modes followed by the user. In fact, the presence of requests of pages not connected by a hyperlink from a same host (and within a reasonable time interval) indicates that the user performed part of the navigation by means of browser commands (such as *back, forward* and *go-to*). Then, a new page was requested to the Web server. The presence of a high number of navigation sequences exploiting browser commands may indicate that the structure of the site does not include important hyperlinks to some of the previous/related pages.

7.4 Statistical testing

The structural view of a Web application (see chapter 4), enriched with transition probabilities, can be exploited for statistical testing with two purposes:

1. Estimating the reliability of the Web application.

2. Prioritizing the execution of test cases.

To the first aim, test cases are automatically generated according to the statistics encoded in the Markov chain. This is easily achieved by stochastically visiting the chain, i.e. by choosing which edge to traverse in accordance with the transition probabilities of the outgoing edges. The resulting test suite complies with the statistics of the usage patterns and simulates a real usage of the Web application. When test cases are executed and failures occur, the classical measures of reliability can be made, by determining the Mean Time Between Failure (*MTBF*) and estimating the probability R of correct behavior within a given time interval (reliability models) [Mus98]. Usually, reliability is expressed as numbers between 0 (unreliable) and 1 (completely reliable). To derive this measure we have to capture failures of the system (suppose i-1 failures) and times to failure (t1, t2, ... , ti-1). The average of these values is the MTBF. Such measures can be useful to decide when to stop testing. For the present work, we have used a very simple reliability model, based on the following formulas:

$$R = 1 - \frac{f}{n} \qquad\qquad MTBF = \frac{n}{f}$$

where n is the total number of test cases executed and f is the number of failures. Additional reliability estimates, such as those proposed in [WT94] are also possible, since a Markov chain is adopted as usage model. They include stopping criteria based on the discriminant, the probability of a failure-free realization of the testing chain and the expected number of steps between failure states.

Path	Prob.
$e_1e_3e_7$	9/20
$e_2e_4e_8$	4/15
$e_1e_3(e_5e_3)e_7$	9/80
$e_2e_4(e_6e_4)e_8$	4/45
$e_2e_4(e_6e_4)^2e_8$	4/135
$e_1e_3(e_5e_3)^2e_7$	9/320
$e_2e_4(e_6e_4)^3e_8$	4/405
$e_1e_3(e_5e_3)^3e_7$	9/1280
$e_2e_4(e_6e_4)^4e_8$	4/1215
$e_1e_3(e_5e_3)^4e_7$	9/5120

Table 7.3: *Paths sorted by decreasing probability.*

To the second aim, paths in the Markov chain are ordered according to their probabilities, given by the product of the probabilities on the traversed edges. With reference to the example in Figure 6.3 and the probabilities in Table 7.1, the paths following e_1 as first edge and looping n times through e_3 and e_5 have probability $3/5(1/4)^n 3/4$, while the paths following e_2 as initial edge and looping n times through e_4 and e_6 have probability $2/5(1/3)^n 2/3$. The sorted list of the top ten paths is given in Table 7.3. The next path, not included in the Table, has probability 4/3645 (approx. 0.1%), while the overall probability of the top ten paths is approximately 99.78%. This means that after executing 10 test cases for the paths in Table 7.3, the probability that the user exercises a path not seen during testing is 0.22%.

7.5 Case study

Let us consider the Web application **ITC-publications** presented in section 3.3. In order to apply statistical testing to it, it is necessary to build its usage model. The server log file of **ITC-publications** has been analyzed to estimate the transition probabilities associated with the model edges. The log file analyzed contains user connections during about two years, from 18-04-2000 to 12-02-2002. Each line of the log file provides, for each user connection, the following information: date, time, name of the host requesting the Web page (IP-number), name of the requested page, HTML status code (404 for resource not found, 200 resource found) and User-Agent (browser) used.

The first task was a preprocessing to remove all log entries with suffix gif, jpeg, mpeg etc., i.e. images, sounds and video files. Then all log entries with HTML status code equal to 404 have been deleted as well. The next task was session identification. A session can be approximated, in a log file, as a set of subsequent accesses to the Web site by a user with the same IP-number and the same User-Agent, within a given time period (time-out). In the session identification phase this value is quite important. In our implementation the time-out is fixed to 10 minutes, a reasonable time in order for a session to be closed. It has been determined by manually estimating the sessions, using the access log information and trying to avoid the loss of a session tail (too short time-out) as well as the mix of two or more separated sessions (too long time-out).

The analysis of the log file highlighted the presence of several consecutive page requests, within a same session, that are not feasible according to the edges in the extracted model. 869 out of 2046 sessions exhibit such a problem. For example, many (precisely, 445) requests of the page `ext-view.idc` are followed by requests of the page `ext-search.idc`, although in the model (see Figure 3.8) there does not exist an explicit hyperlink between these two pages.

A first usage model (Figure 7.2) has been computed, ignoring infeasible pairs of requests

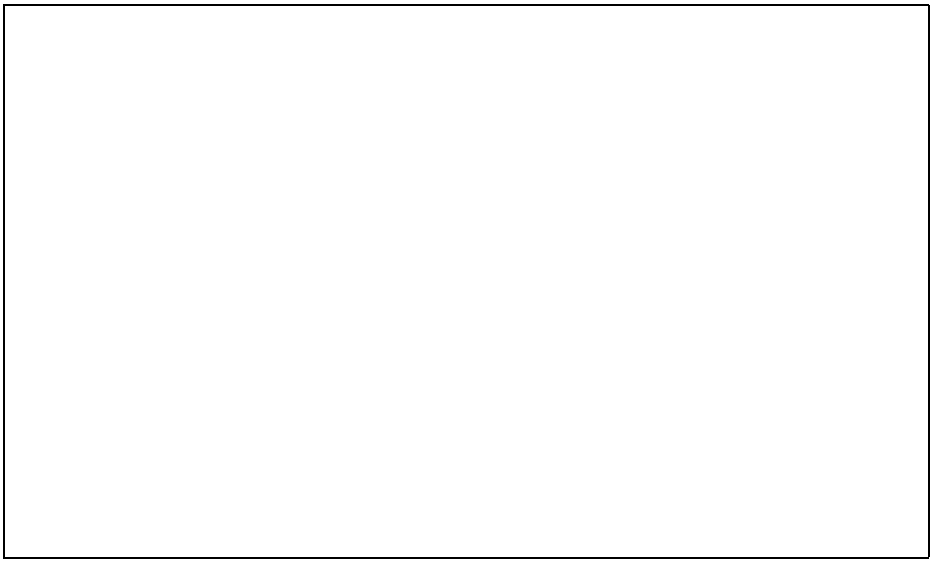

Figure 7.2: *Usage model of the Web application* **ITC-publications**, *computed ignoring navigation through cached pages.*

in the access log. Node X in the usage model represents the end of a user interaction (end of a session). In the usage model, server programs (nodes with grey background in the model) are not represented, since they generate the resulting dynamic pages (shown in the usage model) with probability 1.

By manually inspecting the infeasible pairs of consecutive pages, it was evident that in such cases the navigation performed by the user comprises one or more accesses to the previously visited page (*back* button in most browsers). Since this page is stored in the browser cache, it does not generate any request on the Web server. Correspondingly, no entry is produced in the access log. A simple algorithm has been defined and implemented to estimate the pressing of the back button by the user during the navigation. It works in the following way: when two consecutive pages appear in a same session of the access log for which no edge exists in the model, the preceding pages in the same session are examined, moving backward from the infeasible pair. As soon as a page is encountered that contains a link toward the second page of the pair, the backward traversal is stopped and all traversed pages are used as a completion of the session. Correspondingly, the usage model is augmented with edges that connect a page to (one of) its predecessors, each time a backward traversal is discovered by the algorithm described above. Considering the possibility of backward navigation, a plausible explanation can be given for most of the infeasible sequences of pages in the access log (777 out of 869). For example, the infeasible sequence `db.htm`, `ext-search.idc`, `ext-search.idc` becomes feasible by introducing a backward edge from `ext-search.idc` to `db.htm`, before the last request of `ext-search.idc`. In fact, it is very likely that after a first search (`ext-search.idc` following `db.htm`), the user pushed the

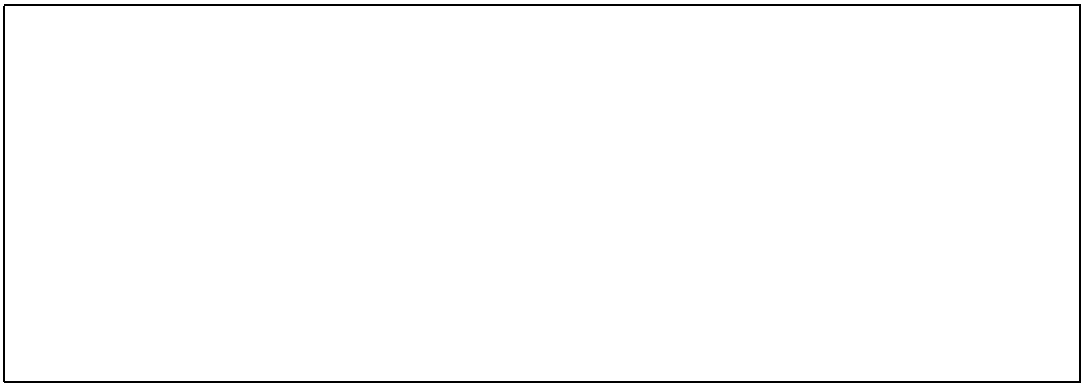

Figure 7.3: *Usage model of the Web application* **ITC-publications** *with back edges (dotted).*

back button, with the browser loading `db.htm` from the cache and no request traced in the log file, to perform a new search. Then, after the insertion of the parameters for the new search, the dynamic page `ext-search.idc` is requested for a second time. The (feasible) path followed by the user becomes: `db.htm`, `ext-search.idc`, `db.htm` (back pressed), `ext-search.idc`. Remaining infeasible sequences, not explained by the algorithm above, may be due to direct accesses to a page (*go-to* button or complete address specification) or to imprecise session identification.

Figure 7.3 shows the new usage model, inclusive of backward edges. The values of the transition probabilities associated with the edges are changed, with respect to the previous usage model, in particular in the case of edges connected to the exit node X. In fact, several sessions terminate with one or more infeasible pairs of pages. Backward edges are new outgoing edges, whose probability was previously zero. Being now greater than zero, it produces a reduction of the probabilities associated with the other outgoing edges (such as the termination edges).

Table 7.4 shows the ten most probable paths in the usage model, sorted by decreasing probability. The overall probability of the top ten paths is approximately 46.57%. The most probable navigation (topmost path) consists of the traversal of e_5 only, the edge that connects the initial page to X. It accounts for the sessions in which the initial page is loaded and then no further page is accessed. The second topmost path accounts for a paper search (performed by means of `ext-search.idc`), followed by the selection of one entry in the search result, visualized by means of `ext-view.idc`. Then, the session terminates. The third topmost path consists of the same navigation as the second one, without entry selection.

The Total Probability (TP) of a set of paths (such as 46.57% for those in Table 7.4) is the sum of the probabilities of the paths. If paths are interpreted as test cases for the Web

Num.	Path	Prob.	Num.	Path	Prob.
1	e_5	16.49%	6	$e_2e_{13}e_{20}$	1.95%
2	$e_1e_7e_{20}$	9.99%	7	e_2e_{14}	1.95%
3	e_1e_8	4.57%	8	$e_1e_7e_{17}e_8$	1.83%
4	$e_1e_7e_{17}e_7e_{20}$	4%	9	$e_1(e_7e_{17})^2e_7e_{20}$	1.6%
5	$e_1e_6e_5$	2.61%	10	$e_1e_6e_1e_7e_{20}$	1.58%
				Total	46.57%

Table 7.4: *Paths sorted by decreasing probability.*

Test cases (n)	TP	Test cases (n)	TP
10	46.57%	1000	83.33%
20	55.33%	1500	84.94%
30	59.67%	2000	86.01%
50	64.61%	2500	86.77%
100	70.23%	5000	88.90%
200	74.99%	10000	90.68%
300	77.38%	20000	92.18%
500	80.14%		

Table 7.5: *Total probability of the n most probable paths.*

application, a set of paths with a high TP is a test suite that provides a statistically more complete test than a test suite with lower TP. Correspondingly, test cases are generated from paths sorted by decreasing probability. By adding more test cases, the TP increases and with an infinite number of test cases this value tends to 100%. Table 7.5 shows the values of TP for an increasing number of test cases (n).

For example, if we execute the 500 most probable test cases, instead of the most probable 20, the TP becomes 80.14% (from 55.33%). This means that after executing the 500 most probable test cases, the probability that the user exercises a path not seen during statistical testing is 19.86% (i.e., 100% - 80.14%). With 20000 test cases the TP becomes 92.18%.

Figure 7.4 gives a graphical representation of the data in Table 7.5. The abscissa represents the number of test cases, while the ordinate gives the TP value. It is apparent from the plot that beyond 500 test cases, the growth rate of the curve becomes smaller and smaller. Correspondingly the extra effort necessary to produce, execute and check additional test cases is not justified in terms of statistical test thoroughness.

To compute the reliability R of **ITC-publications**, one thousand test cases ($n = 1000$) have been generated, by performing a stochastic visit of the Markov chain model of the Web

Figure 7.4: *Plot of the total probability vs. number of test cases.*

application. Correspondingly, the (estimated) number of failures f is 415.25 (see below), and the resulting reliability R is 58.47% ($R = 1 - f/n$). This means that the existing Web application is executed 58.47% of the times without failures. In the remaining cases, one or more failures occur, resulting in a user dissatisfaction. The presence of failures indicates that defects are still present in this Web application. Their impact in terms of impossibility to satisfy a user request is remarkable: according to our user model only 58.47% of the sessions can be completed successfully. In all other cases, the defects in the Web application produce a user observable failure and the session terminates without providing the searched information.

Let us consider the observed failures in more detail. Execution of the test case number 2 in Table 7.4 highlights an anomalous behavior, associated to a defect of the server program `ext-search.idc`, used by the **ITC-publications** application. When the search is made by **abstract**, a text, expected to appear in the searched paper abstract, is specified in the first form of the initial page. The text specified by the user is assigned to the variable **abstract**. The anomalous behavior occurs each time this variable is assigned a non empty value, regardless of the values of the other variables of this form (collecting author, title and reference number, as other possible inputs). If **abstract** is a non empty string, and no other search criterion is specified, the entire database of publications is reported in the output (a very long list), with no filtering in place at all. If other search criteria are specified in addition to the **abstract**, the latter is ignored: the presence of search data about the abstract is not handled and results in an incorrect output, with no regard to the other form fields.

The main problem with the estimation of the reliability R for this Web application is that a failure occurs each time a search is made through the first form *and* the abstract is specified as search criterion. While our usage model provides an accurate estimate of the probability of a search made through the first form, we have no data from which to estimate the number of times a search includes the specification of the text expected to appear in the abstract. In fact, the values submitted by forms (via `POST`) are not saved in the access log, and no additional logging facility was in place in the time interval considered. We have consequently made a very rough estimate of the probability that a user inserts some data in the **abstract** field. Since the form permits a search by **author**, **title**, **reference number** and **abstract**, there are 16 different combinations of inputs (no field specified, field 1 specified, field 2 specified, ..., all four fields specified). Eight combinations out of 16 have the field **abstract** filled in, while the other 8 do not. If the probability of these 16 cases is considered to be uniform, the probability to fill in the field **abstract** is 50%. The main limitation of this estimate is that it might be the case that the 16 possible combinations are not equally probable at all. If for example, the insertion of data about the abstract is less likely than the other data, a lower probability of having a failure would be obtained.

Among the considered 1000 paths, 678 submit the first form at least once (in these paths the edge e_1 is contained at least once). When e_1 appears in a path exactly once, the probability of a failure is $p = 0.5$, i.e., the probability of filling in the **abstract** field. For paths with two occurrences of e_1, the probability of *not* producing a failure is $(1-p)^2 = 0.25$ (both occurrences of e_1 do not contain the field **abstract**). Its complement, 0.75, is the probability of a path with failure. With 3 occurrences of e_1 a failure has a probability $1 - (1 - p)^3 = 0.875$, and in general, when a path contains k occurrences of e_1, the probability of observing a failure is

$$P[Fail] = 1 - (1 - p)^k$$

The mean number of observed failures for the 1000 test cases can thus be computed as the sum of the probabilities $P[Fail]$ over all test cases (with $P[Fail] = 0$ for test cases not including e_1, as results also from the formula above with $k = 0$). This sum is equal to 415.25. If used as the estimated value of f (total number of failures observed), it produces a reliability equal to 58.47%. This means that the analyzed application is expected to fail providing the requested information to the user about half of the times, due to the presence of e_1 in the navigation path and to the specification of the field **abstract**.

Discussion

The structure of a Web application is typically associated to a highly connected graph, which is often one strongly connected component as a whole. Back edges and hyperlinks

providing alternative navigations are quite common, making the resulting structure close to a fully connected graph. On the contrary, state/structural models used for statistical testing of traditional software are typically simpler graphs. The main consequence of this difference is that the number of paths that can be followed, with a proper upper bound for the loops, tends to be much higher for Web applications than for traditional applications, since several (unstructured) ways to go back to a same starting point are available in a Web application. When the graph representation of the Web application is used for statistical testing, such a difference has a very negative impact on the ability to reach a high value of the total probability (TP) of the considered paths. In fact, if a loop is traversed k times in a path, the overall probability of the path is equal to the loop free probability multiplied by q^k, where q is the probability of traversing the loop. Since the loop factor q^k decreases exponentially with k, as soon as loop traversals are included in the considered paths, the contribution of additional paths to the TP becomes small. In other words, when the structure of the Web application is strongly connected, loop probabilities tend to make the contribution of new test cases to the TP small. The result is that it is difficult to achieve high values of TP and the user is expected to live with a Web application containing paths that have a non negligible probability of being traversed, but have never been exercised during testing. For Web applications with an internal state having dependences on the specific path followed, this results in a limited efficacy of the test phase. It is thus preferable to limit as much as possible the dependences of the internal state on the path followed. Ideally, only the immediate predecessor should affect the internal state of the application.

For the present work, the usage model constructed from the access log consists basically of an estimate of the probabilities associated with the possible navigation paths. However, when the user navigates along a given path, input values are inserted into the forms encountered during the navigation. A more complete usage model should include also an estimate of the probabilities that a given input (or an input from a given equivalence class) be inserted into a form. In fact, the occurrence of a failure depends both on the followed path and on the inserted inputs. The absence of information about the probability of the inputs affects the statistical significance of the test cases that are produced, and the estimate of the reliability of the Web application. As regards the test cases, they should cover the most probable paths as well as the most probable inputs, while in our work we have been able to pursue only the first objective. As regards reliability, if inputs are considered uniformly distributed, both an overestimate and an underestimate are possible. If the inputs generating the failures are less likely than estimated according to a uniform distribution, reliability will be higher than estimated. Vice versa, it will be lower if failure inputs are more likely than in the uniform distribution.

7.6 Related works

Statistical Web testing has been already proposed by Chang and Hon in [CH00] and Kallepalli and Tian in [KT01].

In [CH00] the number of error links encountered along the test paths allows estimating the site reliability, i.e., probability that a user completes the navigation without errors. Differently from our proposal, this approach is completely automatic but the failures discovered do not include behavioral deviations from the user expectations (functionality testing).

The main differences between our method and [KT01] are in the model extracted and in the type of failures considered. Kallepalli and Tian's approach is based on UMMs (Unified Markov Models), generated using the **FastStats** log file analysis tool. This tool analyses the data in the access log and produces the *hyperlink tree view*, a tree-structured graph that shows the Web site architecture as well as the direct hits (edge traversal count) from parent pages to child pages. In contrast, we construct an explicit-state UML model, which captures also the dynamic aspects of a Web application (ignored in [KT01]). The usage model of our approach, similar to their UMMs, is obtained in a second time, by decorating the UML model with probabilities deduced from the log file. The failures considered by their method are those present in the access log and error log, i.e., failures such as "permission denied" (accounting for unauthorized access to a Web resource) or "file does not exist" (the requested file was not found in the system). Our technique permits the discovery of these types of failure, but also the discovery of the behavioral deviations from the user expectations not reported in log files: the output pages are inspected by the test engineer to assess whether the test cases have been passed or not, going beyond the information about static failures reported in the log files.

Part IV

RE-STRUCTURING

Chapter 8

Web Application Transformations

In this chapter a set of HTML transforms improving the quality (re-structuring) of Web applications is described. HTML transforms works at two level: intra-page transforms change a single HTML page while inter-page transforms restructure an entire (or a portion of it) Web application.

8.1 Introduction

During the evolution phase, the structure (pages and links) of a Web application, according to Lehman's laws [Leh80], tends unavoidably to degrade. A solution to reverse this degradation can be re-structuring the Web application, but this work may take a lot of time and effort if conducted without appropriate methodologies and tools.

Re-structuring a Web application is a difficult task and little work has been conducted in this area. Better tools are needed to analyze hyperdocuments and to help Web designers in the critical activities of Web site evolution and re-structuring.

A Web application consists mainly of a set of static/dynamic Web pages, usually written in/generating HTML code. For this reason, there are several programs and online services that validate the HTML syntax (i.e. W3C HTML validator [CG01]). Most of them check HTML but do not correct it. Manual correction may take a lot of time and effort. Web designers need tools that not only check HTML files but also fix the problems they find. Tidy [Nes00, Rag98] is one of them. Tidy fixes a number of common errors in HTML files such as: missing or mismatched end tags, tags in the wrong order, missing quotes around attributes, and so on. According to its documentation, this tool can also convert HTML to XHTML and replace the layout and presentation tags with Cascading Style Sheet (CSS), making the markups compliant with the HTML standard. However, tools like Tidy do not

handle any complex re-structuring or correction tasks.

Transforms[1] have been used with success in many real re-structuring works on traditional software. Our idea is trying to apply transforms to Web applications with the aim of re-structuring them. The set of pages which constitute a Web application and are downloaded by **ReWeb**, as well as the related site model, are the starting point for re-structuring. Transforms are based on the site structure described by the model presented in chapter 3 and are applied to the downloaded pages.

In this chapter, we sketch some Web application transforms handling complex re-structuring and correction tasks [RTB01, RTB02], and we provide details about the implementation of one of them, aimed at the reorganization of menus into frames. Results of its application to a case study is also discussed.

The next section introduces the DMS Software Re-Engineering Toolkit(TM) [BP97], the tool used to implement transforms, and gives the syntax of the transforms used in this chapter. Section 8.3 sketches a portion of HTML grammar and describes some examples of HTML transforms. Section 8.4 presents in details the implementation of the *reorganization into frames* transform. In this section we provide a case study about it. Related works concludes the chapter.

8.2 The DMS software re-engineering toolkit(TM)

The DMS Software Re-engineering Toolkit(TM) enables the construction of analysis and modification tools for large scale software systems. Typical applications include: source code generation from a specification (program generation), clone detection and removal [BYM+98], porting legacy languages to more modern platforms (for example Jovial to C), and refactoring Object Oriented systems.

Many re-engineering tools are implemented as assemblies of other existing tools, and are generally difficult to build because the components are not really designed to work together. On the contrary, DMS provides an integrated tools infrastructure (see the following subsection) so that engineers can concentrate on the original re-engineering problem, rather than on the struggle to compose a number of components.

[1] A transform is a (arbitrary) map between program representations while a transformation is the actual application of a transform. A transform can be implemented in many ways. For example procedurally or by means rewrite rules (see appendix 12.3 for a short summary on program transformation).

8.2.1 Domain definition tools

DMS enables transformation and analysis of arbitrary languages by accepting a **domain definition** for the language consisting of:

- Domain Grammar
- PrettyPrinter Rules
- Transforms
- Analyzers

A transform is any method for effecting a mechanical change to a program representation. Typically, transforms are conditionally enabled by analyzers. Analyzers and transforms in general require the full power of procedural programming. DMS provides means to encode both of these as procedures (written in PARLANSE[2] language), or more easily as attribute evaluator rules for analyzers, as (sets of) source-to-source rewrite rules for transforms, or any composition as appropriate to achieve the desired re-engineering effect.

In particular, DMS provides the following generalized components:

- A Unicode capable lexer generator, handling multiple lexical modes, comments, and providing support for preprocessors;
- A Tomita-style [Tom86] parser generator, capable of producing parsers that automatically build abstract syntax trees (ASTs) for any context free language;
- A generalized symbol table, capable of capturing a wide variety of scoping rules and type information;
- An attribute grammar definition system providing for parallel execution, enabling name and type information to be captured by the symbol table, or encoding arbitrary other analyses;
- A pattern and rule specification language, allowing the specification of conditional source-to-source rewrite rules on trees;
- An associative/commutative rewriting engine, able to apply such rules in a single language or refining one language to another;
- A prettyprinter generator, capable of printing out modified ASTs as legal source text;

[2]http://www.semdesigns.com/Products/Parlanse/index.html

- A parallel procedural language, PARLANSE, useful for different purposes, such as creating custom tools (gluing the components), encoding procedural transforms and transform conditions.

Other interesting characteristics of DMS are the ability to mix representations for multiple languages, and the possibility of handling domain variants (i.e., dialects).

8.2.2 Parser and prettyprinter generators

DMS uses a very straightforward BNF formalism. Tree-building operations are not needed, as DMS determines a tree representation automatically from the grammar.

Syntax rules have the following form:

```
lhs '=' rhs_1 rhs_2 ... rhs_N ';'
```

where `lhs` represents a non-terminal symbol while `rhs_i` can be terminal or non-terminal symbols (= and ; are meta-syntax characters, and are written unquoted). There are two kinds of terminal tokens, literals and terminals representing values. Terminal literal tokens are, by convention, written within quotes, as character strings (e.g., '<FORM>'). Terminal tokens representing values, defined by regular expressions (not shown here), are by convention written in the grammar in uppercase (e.g., STRING and PCDATA) and represent a variable sequence of characters. DMS automatically constructs both a conventional tree-building parser, and a pattern parser (used in rewrite rules) from the grammar. Dialects are handled by means of conditional preprocessor directives embedded in the BNF rules.

For each syntax rule, a prettyprinter rule can be given, which produces the text in the unparsing phase by composing text boxes similarly to TeX. Each prettyprinting rule is of the form:

```
'<<PrettyPrinter>>:' '{' box_expression ';' '}'
```

Box expressions are either tokens mentioned in the grammar rules, or are of the form:

```
box_type '(' box_expression_list ')'
```

where box_type is "H" for Horizontal, "I" for indent, "V" for vertical, etc. Grammar rules without any prettyprinter rules use default prettyprinting.

With this information, DMS can parse and then prettyprint the language. Usually, in a reengineering process a program is parsed, transforms are applied, and then the transformed result is prettyprinted.

8.2.3 Rule specification language and rewriting engine

Source-to-source rewrite rules are coded using the following format:

'rule' NAME '(' parameters ')' ':' token1 '− >' token2 '='
lhs_pattern '− >' rhs_pattern 'if' pattern_condition '.'

parameters specifies a comma-separated list of pattern tree variables and their syntactic class in the form: *name ':' token*. Each rule is a map from a syntactic class *token1* to another syntactic class *token2*; for most intra-domain rewrites, the token classes are identical. *lhs_pattern* and *rhs_pattern* specify pattern trees over the parameters, either by providing mixtures of direct syntax tree-building constructs, such as calls to procedures that build trees from parameters (e.g., $add\_operands(term1, term2)$), or quoted text from the domain. Escape ("\") inside quoted domain text is used to distinguish meta-pattern syntax, such as parameter names (\X), or tree-builders (\add_operands\(term1,term2\)) from literal domain text. Rewrites may be conditional; a pattern condition is an expression over pattern variables, which can check for exact equality between terms, equivalence via rewriting, do sub-pattern matching checks, or execute arbitrary predicates coded in PARLANSE.

For example, in the following rule that transforms absolute URLs into relative ones, the pattern *"<A HREF=\h> \b </A>"* specifies a tree of type A, having an arbitrary child of type string (parameter h) as `HREF`, and an arbitrary body (parameter b) within open and closed tag `A`.

rule transform_absolute_URL(h:string, b:A_body): A − > A =
"<A HREF=\h> \b </A>" − > "<A HREF=\extractFile\(\h\)> \b </A>"
if is-absolute-URL(h).

The tree-builder procedure *extractFile(h)* (which reduces an absolute path to a relative one) and the predicate *is-absolute-URL(h)* are coded in PARLANSE (not shown here).

Rewrites involve matching *lhs_pattern*, binding parameter variables, instantiating *rhs_pattern* with pattern variables bound, and replacing *lhs_pattern* with *rhs_pattern*. Patterns involving a grammar non-terminal declared as associative-commutative (AC) can match trees that are not adjacent (see Fig. 8.1 for a further explanation). Rules are named and can be invoked individually, or can be grouped into named rulesets and invoked as a set. One often creates rulesets whose rules collectively implement some larger useful goal (e.g., a ruleset to simplify an equation, comprised of many individual simplification rules). Invocation of a ruleset results in a bottom-up repeated rewriting. More sophisticated control strategies (e.g, avoiding applying some rules repeatedly because they are non-terminating) can be implemented procedurally in PARLANSE.

Figure 8.1: *Associative-commutative rewriting property of DMS.*

8.3 HTML transformations

The basic components for HTML re-engineering, are an HTML grammar, a prettyprinter, and a set of HTML transforms.

8.3.1 HTML grammar

Parsing HTML may seem an easy task at first sight, since HTML is defined precisely by its DTD [Gro99]. Unfortunately, only a few real Web pages are compliant with it. To resolve this problem, browsers tend to use a super-set of the language defined in [Gro99] and tolerate non compliant HTML. For example, the non HTML-compliant constructs (with respectively a syntax error and a proprietary tag):

```
<B> bold <I> italic and bold </B> italic </I>
<BLINK> blink <BLINK>
```

are parsed (and rendered) without problems by popular browsers (like Netscape and Microsoft's). Our solution to this problem has been to implement in DMS an HTML grammar

Figure 8.2: *Mixed tree example.*

compliant with the DTD [Gro99], and then to create dialects to handle proprietary tags and browser-acceptable syntax errors. The final HTML grammar (used to parse real Web pages) is the union of the standard HTML grammar (*HTMLstandard*) with all the dialects built (*HTMLnetscape, HTMLmicrosoft* and *HTMLerrors*).

Another important aspect to consider in the parsing phase is that a Web page can contain different languages: HTML, scripting languages (e.g., Javascript) and style sheet languages (e.g., CSS). The solution requires working with multiple languages in the same session. In fact, DMS permits changing lexical modes, defining grammars for the different languages that may appear in a same source file, and working with "mixed trees" (see Fig. 8.2).

Fig. 8.3 gives a portion of the HTML grammar and prettyprinter specification implemented in DMS. Non-terminals are in boldface, literal terminals are within quotes, while terminal tokens representing values are uppercase. The attributes of tags are implemented as a list (note that a list can be void, for a tag without attributes) of non-terminals `attr`. The nonterminal `attr` can derive into a terminal `ID` or into an assignment `ID = STRING`.

8.3.2 A taxonomy of the HTML transforms

HTML transforms work at two levels: there are **intra-page transformations** and **inter-page transformations**. In the first case, transforms work on a single HTML page and the result is a new HTML page. There is no impact on the Web application model. In the second case the transforms take more pages – the entire Web application (or a portion of it) – as input, change some links/pages, add/delete some pages, and, at the end of the

Figure 8.3: *Portion of (DTD-compliant) HTML grammar.*

transformation, return the changed pages. These transforms exploit the model of the Web application to locate changes and update the model when they are applied.

HTML transforms can be further characterized by means of the ISO 9126 [fS91] quality model for software systems. The transformations proposed for Web applications and given below aim at improving 3 of the 6 quality factors described in [fS91]: **maintainability, usability** and **portability**.

We have so far identified six classes of HTML transforms: **(1) syntactic clean up, (2) page re-structuring, (3) style renovation and grouping, (4) improving accessibility, (5) update to new standards,** and **(6) design re-structuring**.

Classes 1, 2, 6 are associated with **maintainability**, 3 and 4 with improving **usability**, and 5 is related to **portability**. Among the identified classes, only **design re-structuring** includes a set of **inter-page transformations**, with all the others related to a single page

(**intra-page transformations**).

When editing HTML code, sometimes syntax errors are accidentally introduced. Popular browsers (like Netscape and Microsoft's) are tolerant to faults and resolve these errors in many cases, rendering the page as if it were correct. The consequence is that on the Web there are a lot of documents with syntactic errors which, from the users' perspective, work perfectly. A set of transforms, named **syntactic clean up**, correct the HTML code so that it matches, where possible, the rendering observed in popular browsers. These transforms increase the robustness of the Web application with respect to the target browsers and prevent unpredictable side effects resulting from cumulative syntactic errors.

The purpose of the **page re-structuring** transforms is improving the quality of a single HTML page, which is already syntactically correct (compare with **syntactic clean up**). Examples are: transforming absolute URLs into relative URLs by means of the tag `BASE` (after this transformation it is possible to move the page between directories and even servers without problems), adding the part `NOFRAMES` to a page split into frames (to permit a correct navigation also in browsers which do not handle frames), and adding a description for non-loaded objects (e.g. applets, movies).

The W3C Consortium recommends to use style sheets, for example CSS, to control layout and presentation, instead of using presentation markups. **Style renovation and grouping** transforms change (if possible) an HTML page with presentation markups into a page with style sheets, and organize style sheets in a compact way.

The Web Accessibility Initiative (WAI) promotes a high degree of usability for people with disabilities. The document [CVJ00] provides some techniques to satisfy the requirements defined in [CVJ99]. This document contains a set of guidelines that can be expressed in terms of (automatic or semi-automatic) transforms. Examples are: removing the blinking effect, providing the ability to stop the refresh, providing redundant text links for each active region of a server-side image map and providing keyboard shortcuts to important links. The **improving accessibility** class contains a set of transforms that implement some of these guidelines.

The W3C Consortium recommends, when designing a Web application, to use the tag set of the latest version of HTML (HTML 4.01 or XHTML 1.0). Even if popular browsers will continue supporting deprecated tags for reasons of backward compatibility, it is better to avoid using them. If present in old documents, deprecated tags can be eliminated by making the enclosing page compliant with the new standard via transforms. Moreover, support tools which are commonly used to create HTML documents for publication on the Web often insert deprecated markups and use tables for layout purposes. Consequently, the content of these automatically generated pages have to be purified. **Update to new standards** transforms substitute the deprecated tags with tags of the latest version of HTML.

The transforms for **design re-structuring** are related to re-structuring a portion of the structure of a Web application. Examples are the automatic *reorganization into frames*, *page extraction* and *page insertion*. In the first case, a frame containing an index replaces all mutual references of a set of pages. Page extraction and page insertion can be useful transformations respectively when a page contains too much information and it is better to split it into more pages or when every page in a given set contains little information and it would be better joining them.

8.3.3 Examples of HTML transforms

Some of the transforms proposed above are presented in the following. When they are too large and complex to fit a page, an example of their effect on an input HTML page is shown, by providing the HTML code before and after the transform.

8.3.3.1 Syntactic clean up

These transforms correct simple markup errors (e.g missing or mismatched end tags, tags in the wrong order, etc.) in a way that matches, where possible, the observed rendering. Three possible syntactic clean up transforms follow.

T 1 *rule balance-tag-H1(X: inline): inline − > inline =*
"<H1> \X" − > "<H1> \X </H1>".

T 2 *rule correct-order(X,Y,Z: inline): inline − > inline =*
"<B> \X <I> \Y </B> \Z </I>" − >
"<B> \X <I> \Y </I> \Z </B>".

T 3 *rule HR-extrusion(X: inline): inline − > inline =*
"<H1> <HR> \X </H1>" − >
"<HR> <H1> \X </H1>".

By applying T1, T2 and T3 to the following fragment of HTML code:

```
<H1> <HR> Example1 </H1>
<B> bold <I> bold italic </B> bold </I>
<H1> <HR> Example2 </H1>
... other HTML lines ...
<H1> Example3
```

we obtain:

```
<HR>
<H1> Example1 </H1>
<B> bold <I> bold italic </I> bold </B>
<HR>
<H1> Example2 </H1>
... other HTML lines ...
<H1> Example3 </H1>
```

The two fragments of HTML code are rendered in the same way in a Netscape browser.

8.3.3.2 Page re-structuring

Let us consider the following HTML code, working only in browsers that manage frames:

```
<HTML><HEAD><TITLE>Example</TITLE></HEAD>
 <FRAMESET cols="50%, 50%" title="Document">
  <FRAME src="main.html" title="Content">
  <FRAME src="contents.html" title="Index">
 </FRAMESET>
</HTML>
```

The transform **add_noframes** (not shown here) ensures that the document is readable also in browsers without frames. It adds the part `NOFRAMES` to a page split into frames:

```
<HTML><HEAD><TITLE>Example</TITLE></HEAD>
 <FRAMESET cols="50%, 50%" title="Document">
  <FRAME src="main.html" title="Where
   the content is displayed">
  <FRAME src="contents.html" title="Table">
   <NOFRAMES>
    <A href="main.html">main</A>
    <A href="contents.html">contents</A>
   </NOFRAMES>
 </FRAMESET>
</HTML>
```

`BASE` tags ensure that any relative URL within the document will be resolved into a correct document address. This is particularly important when there exist groups of documents

logically related that are stored in a common directory. By placing the correct BASE tag in each document, it is possible to move the entire collection between directories and even servers without breaking all of the links within the documents: only the BASE tag needs be updated in each document, while links need not be modified.

By applying the transform **add_base** to the following code fragment:

```
<HTML><HEAD><TITLE>Example</TITLE></HEAD>
<BODY>
 <A HREF="http://www.itc.it/index.htm">HP</A>
</BODY>
</HTML>
```

we obtain:

```
<HTML><HEAD>
<TITLE>Example</TITLE>
 <BASE HREF="http://www.itc.it/">
</HEAD>
<BODY>
 <A HREF="index.htm">HP</A>
</BODY>
</HTML>
```

8.3.3.3 Style renovation and grouping

By applying the **style renovation** transform to the following fragment of HTML code:

```
<HTML><HEAD><TITLE>Example</TITLE>
 <BODY>
 <H1 Align=center>How to Carve Wood </H1>
 </BODY>
</HTML>
```

we obtain:

```
<HTML><HEAD><TITLE>Example</TITLE>
  <STYLE type=text/css> H1 {
      TEXT-ALIGN: center
  }
```

```
  </STYLE>
 <BODY>
  <H1>How to Carve Wood </H1>
 </BODY>
</HTML>
```

where the alignment of the text inside the `H1` tag is obtained by means of a CSS style specification, thus separating style from content.

In designing CSS brevity is a goal. Short style sheets load faster than longer ones. Transform **Grouping style sheets with equal declaration** reduce the size of style sheets.

For example, let us consider these three style sheet rules:

```
H1 { font-weight: bold }
H2 { font-weight: bold }
H3 { font-weight: bold }
```

Since all three declarations are identical, we can group the selectors into a comma-separated list inside a single declaration:

```
H1, H2, H3 { font-weight: bold }
```

The order of the tags in a style list is commutative. So we would mark the CSS grammar rule for the style list (not shown) as AC, and consequently DMS will be able to handle the following case:

```
H1 {fw:bold} H2 {fw:italic} H3 {fw:bold}
```

and rewrite it as:

```
H1,H3 { fw:bold }  H2 {fw:italic}
```

8.3.3.4 Improving accessibility

The following transform, that improves accessibility for people with disabilities, aims at removing the *BLINK* effect.

T 4 *rule remove-blink(X: inline): inline − > inline =*
"<BLINK> \X </BLINK>" − > "\X".

The transform **control refresh** searches the HEAD part of an HTML document for a pattern of this type:

```
<META http-equiv="refresh" content=\N URL=\U>
```

that implements the refresh of the current page, loading the URL $\backslash U$ after $\backslash N$ seconds. If this pattern is found, the transform **control refresh** removes it from the page and adds a hyperlink to the URL $\backslash U$ to the *BODY*.

By applying this transform to the following fragment of HTML code:

```
<HTML>
  <HEAD>
   <META http-equiv="refresh" content=15 URL="http://www.example.it">
   <TITLE>Refresh Example</TITLE>
  </HEAD>
 <BODY>
  ... example body ...
 </BODY>
</HTML>
```

we obtain:

```
<HTML>
  <HEAD>
    <TITLE>Refresh Example</TITLE>
  </HEAD>
 <BODY>
  <A HREF="http://www.example.it">refresh</A>
  ... example body ...
 </BODY>
</HTML>
```

8.3.3.5 Update to new standards

To include applets, HTML code developers should use the `OBJECT` tag[3], since the `APPLET` tag is deprecated. The transform **applet_to_object** changes the HTML code by substituting each `APPLET` tag with an `OBJECT`.

For example, by applying this transform to the following HTML code fragment:

[3]HTML objects should not be confused with objects in object oriented programming.

```
<APPLET code="Bub.class" width="500" height="500">
  Java applet that draws animated bubbles.
</APPLET>
```

we obtain:

```
<OBJECT codetype="application/java" classid="java:Bub.class"
        width="500" height="500">
  Java applet that draws animated bubbles.
</OBJECT>
```

8.3.3.6 Design re-structuring

T 5 *Automatic reorganization into frames.*

Figure 8.4: *Automatic reorganization into frames.*

As depicted in Fig. 8.4, this transform allows separating index (in the picture represented with h-1-2-3) and content in a site portion where the index is inserted and replicated in all pages. A reorganization of the pages into frames may simplify navigation and may improve maintainability.

This transform is not a direct rewrite rule. It requires procedural support. We need a procedure producing the index page (split into two frames L and R), another one collecting the list of URLs implementing the index (h-1-2-3) and producing the menu page, and a final one applying a rewrite rule that removes the index from all pages.

T 6 *Page insertion/extraction.*

Fig. 8.5 presents the two symmetric situations in which a consecutive piece of information (B) is extracted from/inserted into a given Web page (A).

Figure 8.5: *Page insertion/extraction.*

This transform is mostly procedural. If we consider page A as a tree, we need a procedure to extract a set of adjacent subtrees from A. Once the subtrees for B have been extracted, a reference (in page A) to a newly created page B can be constructed using DMS patterns. Another rewrite rule can be used to update the URLs present in A and B that must be modified after the operation of extraction/insertion of B (e.g., URLs contained in page B toward A).

8.4 Reorganization into frames

At present, the standard HTML grammar (prettyprinter included), as well as some common dialects, have been implemented in DMS, together with some transforms. In particular, we have implemented the intra-page transformations *add_noframes, add_base, applet_to_object, add_missing_tags* and the inter-page transformation *reorganization into frames*. In this section we describe the implementation of the *reorganization into frames* transform in DMS, and we provide a case study and some experimental data about it.

8.4.1 Reorganization into frames: implementation in DMS

Differently from typical intra-page transformations, where input and output are a page, inter-page transformations are not direct rewrites. They require procedural support for parsing the entire Web application (mass-parsing phase) and applying transforms to each page (transformation phase). Moreover, an analysis phase doing pattern matching, devoted to the identification of the subtrees to be transformed, is conducted before the transformation. In particular, the *reorganization into frames* transformation requires the modules

Figure 8.6: *Reorganization into frames tool.*

depicted in Fig. 8.6. Since this transformation cannot be completely automated, the user is required to intervene in the final transformation phase.

After downloading the site, the PARLANSE module for mass-parsing parses every Web page present in the input Web site and builds a set of ASTs (one for each page). The module Menu-Marker applies a set of rules to identify each list of URLs implementing a menu (pattern matching phase) to the ASTs produced by the mass-parsing module. If a sub-tree of a page matches a pattern belonging to a library of menu patterns, (see Fig. 8.7 for examples), the rule decorates the related sub-tree by inserting, as a parent node, a tag that identifies the nature of the pattern found (*HorizontalMenu_PlusTR, HorizontalMenu_OneTD, VerticalMenu_PlusTR*)[4]. An alternative implementation, exploiting the DMS attribute evaluators instead of the rules, would be also possible.

Fig. 8.8 shows the rule *HorizontalMenu_OneTD* (the simplest among those implemented), written in the rule specification language provided by DMS. *OKTD* and *OKTDend* are intermediate tags, used in the transformation, while *Horizontal_MenuTD* is the tag that

[4]The horizontal vs. vertical layout of the menu characterizes the enclosing table (columns vs. rows), not its display, which can be modified, for example, by adding some `BR` tags.

Figure 8.7: *Examples of menu patterns.*

identifies the (possible) presence of a menu of type *Horizontal_MenuTD*. The conditions *Contain_list_HREF* and *ContainTD* are implemented externally in PARLANSE (keyword *external*) and respectively control if the tree e contains a list of URLs (>=3) and if it contains a *TD* element.

The rules *createListHREF1* and *createListHREF2* identify a *TD*-block containing a list of `HREF` tags (the second rule handles the case of a *TD* tag with attributes). Then, the rule *mark* is used to attach the tag *Horizontal_MenuTD* to all identified subtrees.

After executing the rules in Fig. 8.8 on all pages of a Web site, a set of candidate menus associated with each analyzed page is produced. The next step is deciding which menus are to be moved to an *index* frame, forcing the loading of its URLs into a *content* frame. Every page is possibly associated with several menus that could become the index frame. Moreover, the same menu could be replicated in different pages with modifications. A typical situation is that the current page is removed from the URLs in the menu. Consequently, it is not possible to identify the menus to be restructured by looking for a common sequence of URLs replicated in a group of pages, because slight modifications may invalidate the search of exact replications.

A possible solution to these problems is based on concept analysis [GW96] and its clustering capability. Concept analysis (see appendix 12.4 for further explanation on concept analysis) allows determining maximal groupings of objects sharing common attributes. For our purposes, the objects we are interested in are the Web pages, and the attributes they possess, that suggest the possibility of an index frame, are the URLs in their menus. Execution of concept analysis in this context results in maximal sets of pages sharing common URLs in their menus. These are exactly the desired candidate indexes. Maximality of the objects and attributes of each concept provides the required robustness with respect to

Figure 8.8: *DMS implementation of rule* HorizontalMenu_OneTD.

holes or variants in the retrieved menus.

The module *Concept-analysis* analyzes the set of menus detected by the *Menu-Marker* for each page of the site. Specifically, the relationship associating each HTML page to the set of URLs in its menus is the input for the *Concept-analysis* module. Its output is a set of concepts, that can be interpreted as the alternative candidates for the transformation. This information is printed into a textual *Report*, which is inspected by the user who is charged with selecting the candidate to be transformed.

For example, by applying the modules in Fig. 8.6 to the Web application `www.sestese-pallavolo.it`, we obtain the following (simplified) report, consisting of four concepts. Every concept corresponds to a possible re-organization into frames. The concept number 002, for example, indicates that the pages `under20.html, societa.html, sconto.html, organico.html, news.html, dove.html, date.html, classifi.html` share the set of URLs `index.html, squadra.html, sponsor.html` and `societa.html`.

```
Pattern found: HorizontalMenu_PlusTR

001 ({under20.html; sponsor.html; societa.html; sconto.html;
      organico.html; news.html dove.html date.html classifi.html},
     {squadra.html; sponsor.html; societa.html})

002 ({under20.html; societa.html; sconto.html; organico.html;
```

```
      news.html; dove.html date.html classifi.html},
     {index.html; squadra.html; sponsor.html; societa.html})

003 ({sponsor.html; societa.html; sconto.html; organico.html;
      news.html; dove.html; date.html; classifi.html},
     {squadra.html; sponsor.html; societa.html; news.html
      dove.html; classifi.html})

004 ({societa.html; sconto.html; organico.html; news.html;
      dove.html; date.html; classifi.html},
     {index.html; squadra.html; sponsor.html; societa.html;
       news.html; dove.html; classifi.html})
```

After reading the *Report*, the user can choose a concept (a set of concepts, when more than one menu can be reorganized into frames). For each concept *(Object_Pages, Attribute_URLs)* chosen by the user, the PARLANSE *Transformation* module produces an initial page (split into two frames), a menu page (consisting of the URLs in *Attribute_URLs*) and applies a transform (not shown here) that removes the menu from all pages in *Object_Pages*.

8.4.2 Case study

The web application `apdt` (`www.apdt.net`, an anglers' association in Trentino) is a good example to illustrate an intervention of frame reorganization. The structure of this site is typical of medium size (about 50-100 pages) sites, offering a wide spectrum of products and services (other examples are: `www.anfilazio.com`, `www.sestesepallavolo.it`, `www.sitaserre.com`). In these Web applications a portion of HTML code is devoted to implementing a menu (usually by means of the construct `TABLE`) repeated in a lot of pages.

Figure 8.9: `apdt` *Web application before (left) and after (right) transformation.*

In apdt (see the structure shown in Fig. 8.9, left) the original menu (following piece of HTML code, slightly simplified and reduced) is implemented using the construct TABLE and a structure that matches the pattern *VerticalMenu_PlusTR*. This menu is repeated in every page of the apdt Web application.

```
<table width="182" border="0" cellspacing="0" cellpadding="0" vspace="5">
 <tr>
  <td height="27" width="182">
   <div align="center"><a href="index.htm" onMouseOver="..."></a></div>
  </td>
 </tr>
 <tr>
  <td width="182" height="21">
   <div align="center"><a href="chi_siamo.htm" onMouseOver="..."></a></div>
  </td>
 </tr>
 .
 .
 .
</table>
```

The application of the *reorganization into frames tool* to apdt gives a report with only one concept (all the pages of the site contain a VerticalMenu_PlusTR pattern with the same URLs):

```
({regolamenti.htm; noce.htm; le_nostre_acque.htm; iniziative2.htm;
  iniziative.htm; index.htm; fersina.htm; chi_siamo.htm; avisio.htm;
 attivita.htm; adige.htm},
{chi_siamo.htm; index.htm; iniziative.htm; attivita.htm;
 le_nostre_acque.htm; regolamenti.htm})
```

By applying the transformation associated with this concept, we obtain the Web application shown in Fig. 8.9 (right). The transformation module creates the page start.html (split into two frames, *MENU* and *DISPLAY*), produces the page menu.htm, and removes the portion of HTML code implementing the menu from each page of the Web site. By comparing the left and right part of Fig. 8.9 the clearer and simpler organization of the re-structured apdt application (right) is apparent, with potential benefits on the site understandability and maintainability.

8.5 Related works

Different evaluation and transformation tools (freeware or commercial) of Web applications are available on the net.

For example, the document [CK01] provides information about evaluation, repair and transformation tools that make the Web more accessible [CVJ99]. Tools in this document are classified into three different categories: **evaluation tools** – performing a static analysis of pages or sites, focused on their accessibility, and returning a report or a rating; **repair tools** – once the accessibility concerns of a Web page or site have been identified, these tools assist the author in making the pages more accessible; **filter and transformation tools** – these tools assist Web users rather than authors to either modify a page or handle a support technology.

Bobby and *Doctor HTML* are two evaluation tools. *Bobby* helps authors determine if their sites are accessible for users who have some form of disability and it also analyzes Web Pages for compatibility with various browsers. *Doctor HTML* performs minimal accessibility checking, verifies links, performs spell and syntax checking. The W3C HTML validation service, based on another evaluation tool, checks HTML documents for compliance with the *W3C HTML Recommendations* and other HTML standards. *Demoronizer* and *Tidy* are two repair tools. *Demoronizer* removes vendor specific HTML conventions and extensions, while *Tidy*, already discussed in the introduction of the thesis, repairs errors, improves style and converts HTML to XHTML. The filter and transformation tools *LaTeX2HTML*, *PDF-to-HTML Converter* and *RTF-to-HTML* permit transforming LaTeX, PDF and RTF documents into HTML

A tool for interactively creating and updating HTML documents, preserving the DTD compliance, is described in [BR96]. This tool, called Tamaya, is based on a transformation language tailored for HTML documents (presented in the paper), and can be used to restructure existing HTML pages which are non DTD-compliant.

In [GFKF01] an automated transformation converts traditional interactive programs into Web applications based on CGI programs. The authors extend a programming language with primitives that support saving and restoring the interaction state across successive interactions.

The main difference between the work described in this dissertation and the cited tools and approaches for HTML transformation is that we consider also design re-structuring, involving high level changes that affect the entire organization of a site, (potentially) improving its overall usability and maintainability. On the contrary, available tools approaches work at a purely syntactic and intra-page level. Our experimental work on the automatic reorganization into frames consists of a complex transform requiring pattern matching in each HTML page of the site and concept analysis to cluster information at the inter-page level.

Chapter 9

Multilingual Web Site Consistency

In this chapter a set of techniques and algorithms that can be used to support re-structuring of multilingual Web sites (sites where the information is supplied in more than one language), so as to align the information provided in different languages and make it consistent across languages, are proposed. The target representation of the re-structuring process is an extension of XHTML, called MLHTML. MLHTML is designed to simplify the evolution of a site while maintaining pages in different languages consistent.

9.1 Introduction

To be accessible to a larger audience, Web sites are often required to be multilingual, i.e., sites where the information is supplied in more than one language (e.g. Italian, English, French etc.). For such sites the design, the check of the quality and the evolution phase are more complicated that in monolingual sites: Web pages provided in more than one language introduce new problems for developers.

In a multilingual site every page in a language is expected to have a corresponding page in every other language of the site, and a change in a language has to be propagated to all other languages. In other words, the presented information as well as the site structure should be *consistent* across different languages. Not only the available information should be the same and should be displayed with the same format, but intra and inter-language hyperlinks should comply with the overall site organization, leading to the requested information in the right language.

Current practice of Web site development does not address explicitly the problems related to multilingual sites and there aren't available tools that can help developers to design and maintain them. The current lack of automatic support to multilingual site development typically leads to personal choices on the physical and logical organization of the pages

in different languages. In some sites a directory for each language is accessible from the top level and inside each directory the same structure is replicated. In other sites, each page name has a suffix identifying the language (e.g., `-en` for English, `-it` for Italian). Others prefer to translate the name of each page into all supported languages, and the correspondence relies on the translation. Finally, some sites adopt a chaotic mix of all options above, with possibly additional variants (numbering the pages in different languages, etc.). In all these cases, the task of maintaining the different site portions aligned is under the responsibility of the developer and is performed manually.

In this chapter we propose a set of techniques and algorithms that can be used to support re-structuring of multilingual Web sites so as to align the information provided in different languages [TRPG01, TRPG02]. The target representation of the re-structuring process is an extension of XHTML, called MLHTML. A valuable property of MLHTML is that it centralizes the language dependent variants of a page in a single representation, where shared parts are not duplicated. This property is fundamental during the evolution of a Web site. Existing Web sites can be migrated to MLHTML according to the process depicted in Figure 9.1.

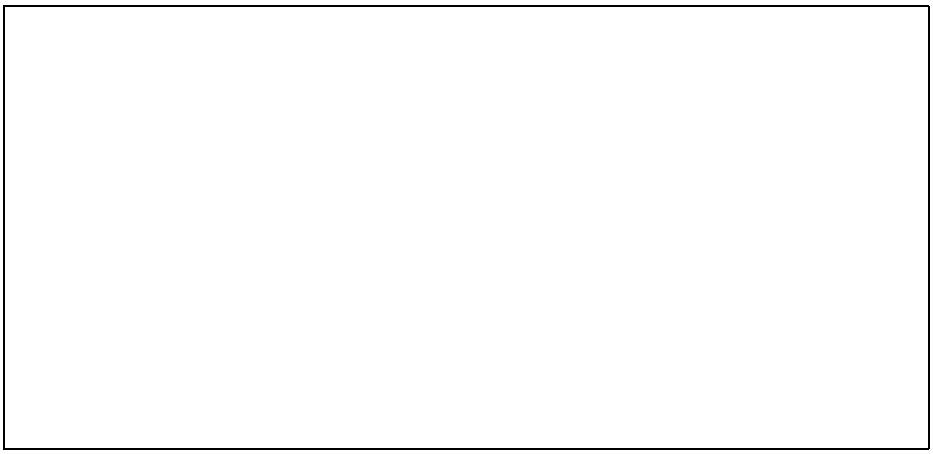

Figure 9.1: *Phases of the re-structuring process for a multilingual Web site.*

The first operation in this process aims at aligning the pages in the different languages. Page alignment is conducted semi-automatically, in that several tools can support the identification of the correspondences and of the inconsistencies, but the developer is still required to fix the problems discovered. Then, aligned pages can be merged into a single MLHTML page, from which the published site can be automatically generated. Alternatively, MLHTML pages can be directly visualized by the browsers, provided that the related plug-in is installed. Every maintenance operation can now be performed on the MLHTML representation of the pages, thus ensuring a consistent propagation of the changes to all site versions in different languages. Tools have been developed to support all phases of the process.

It can be noted that the page alignment operation is desirable even if MLHTML is not adopted, since it produces a Web site in which the provided information does not contain differences depending on the chosen language. Moreover, the same re-structuring process, with similar tool support, can be adopted for a target representation different from MLHTML. The advantages of this re-structuring are a simplification of the maintenance phase and a higher quality of the site (all portions are consistent). Of course, a site that is developed from scratch can be directly coded in MLHTML. The experience we made with real world sites, downloaded and analyzed by means of our tools, indicates that a high amount of multilingual Web pages is maintained consistent manually and the availability of a tool-supported re-structuring process to migrate them would be very beneficial.

The chapter is organized as follows. Section 9.2 describes the page alignment procedure that is constituted by three sub-task: language identification, page matching and text comparison. Then, page merging, page extraction and page evolution are considered. In particular, MLHTML is described in section 9.3 and its features are discussed with respect to potential alternatives. The re-structuring of a real-world multilingual Web site is presented in section 9.4, where the usefulness of the proposed techniques is commented. A discussion of related works concludes the chapter.

9.2 Page alignment

The purpose of page alignment is to determine the correspondences between pages in different languages belonging to a multilingual site and to resolve inconsistencies (missing pages, tags, links or translations). A first problem is how to partition the pages in the site according to languages. In some cases it is easy to infer this information from the naming convention adopted for the pages, but more generally it may be useful to support this operation with a language identification tool, which analyzes the content of the pages. In this way it is possible to recognize pages that were not translated but respect the convention and pages that do not conform to the naming convention.

After assigning pages to different languages, the correspondences between pages have to be retrieved. For such a purpose the *structure* of the pages is exploited: the syntax tree of the HTML code provides information on the page formatting and on the location of text, images and other objects contained in the page. A structural comparison algorithm will be described to infer the matching between corresponding pages. A relevant contribution to this algorithm comes from the comparison of attributes of the syntax tree nodes, and, among these attributes (node labels, etc.), of the text. A natural language processing method is employed to determine if the text attribute of a node is the translation of another.

9.2.1 Language identification

The problem of language identification was extensively investigated in the Natural Language Processing (NLP) research community, and some methods and tools have been proposed to address it [CT94, Gre95].

We decided to adopt a simple solution to this problem based on the entries available in a set of mono-lingual dictionaries, associated with the words in the page to be classified. Our tool determines the number of words in the page text that do not have an entry in each dictionary. The dictionary which gives the minimum number of errors (unrecognized words) for the given page is selected. If the ratio between number of errors and number of words is greater than a proper threshold (in our implementation set to 0.5), the language automatically assigned to the page is *unknown*, i.e. the high number of unrecognized words makes the language identification procedure unreliable. This case may occur when a page is internally multilingual, so that no dictionary can give a low number of errors on its words, or when a page contains language independent information (e.g., a list of person names and phone numbers). There is a second situation in which this algorithm cannot classify the page. This is when the same minimum number of errors is produced by more than one dictionary. The output of our tool is an *ambiguous classification* and the operator is then required to solve the ambiguity. In all other cases the algorithm can assign a language to each page in the site, thus partitioning it into mono-lingual portions, without having to rely on the naming conventions.

The dictionaries employed for this work are a subset of those distributed with the UNIX utility `ispell` (more specifically, the Italian, American and British English, Spanish, German and French dictionaries have been used). More dictionaries could be found in the public domain if needed.

9.2.2 Page matching

The basic property which is exploited to determine the matching of pages in different languages is the page structure, as coded in HTML. Given two HTML pages, p and q, their structure can be represented by the syntax trees resulting from parsing the two pages. Two examples of such trees are provided in Figure 9.2, where the pseudo-HTML tags `A, B, C, D` are used. In the syntax tree for a page, information nesting is mapped onto the parent-child relation. Moreover, nodes are assumed to be decorated with a label, equal to the HTML tag, and with a set of attributes[1] (not shown in the figure), as, for example, the *text* attribute, containing the text within the given tag, and the *image* attribute, containing the file name of the included image, if any.

[1]Node attributes should not be confused with attributes inside tags.

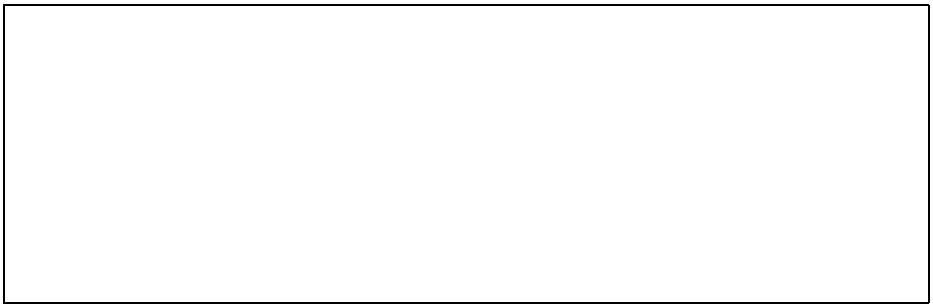

Figure 9.2: *The structural comparison of the pages* p *and* q *gives an edit distance equal to 2.*

In order to compare the syntax trees associated with two pages, a tree visit is performed, during which a list representation of the tree is generated. When a node is entered during the visit, the associated tag is inserted into the list, while the corresponding closing tag is put into the list when the node is left, after completing the visit of its children (this is done even if in the HTML source code the tag is not closed). The lists associated with pages p and q are given in Figure 9.2 below the syntax trees. It can be shown that trees and lists are isomorphic with each other, so that there is no information loss when considering the lists instead of the trees.

Dynamic programming algorithms for the comparison of sequences [Ata99] can be employed to determine the number of edit operations that transform the list representation of a page p into the list associated with page q. The typical set of allowed edit operations on sequences consists of element deletion, insertion and update. In the example in Figure 9.2 it is easy to see that the sequence associated with p can be transformed into that associated with q by deleting two elements, `D` and `/D`. Therefore the two pages have a structural edit distance of 2.

Using the sequence edit distance for page comparison is very appealing because the resulting edit operations have a direct interpretation in terms of elementary actions that are executed by the developer to modify the first page and make its structure correspond to that of the second page. In fact, a Web site developer is expected to delete, insert or update the same tags as obtained from the distance computation in order to perform the transformation. When two pages are considered as matching, because they are at minimum distance, the result of the comparison provides information on the internal features of the two pages, by indicating explicitly the operations that must be performed to update the first page and make it consistent with the second with respect to the structure.

The algorithm sketched above can be extended to incorporate any node attribute, other than the node label, into the comparison procedure. With reference to the *text* attribute, an edit operation will not be necessary to transform a first node into a second one, provided that they have the same label, only if the text associated with the first node is a translation

of the text associated with the second node. In the example in Figure 9.2, the three nodes labeled `B, C` and `B` are considered matched, and thus require no edit operation, only if `text1`, `text2` and `text4` are respectively translations of `text5`, `text6`, and `text7`. If some of these pairs are not translations of each other an update edit operation needs be executed, and the associated cost will increase the final value of the edit distance. The comparison of node attributes can be extended to other attributes (e.g., *image*). The comparison of the *text* attribute, aimed at determining if a text is the translation of another text, is the subject of the next sub-section.

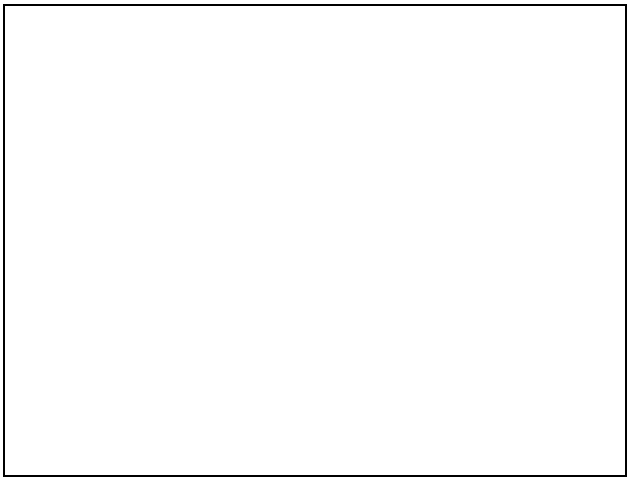

Figure 9.3: *Pseudocode of the algorithm to determine the match between two sets of pages, S_1 and S_2.*

Given two sets of pages, S_1 and S_2, associated with two different languages, it is possible to match each page p from S_1 with a page q from S_2 by choosing the page from S_2 at minimum edit distance from p. Since edit distance determination is computationally expensive ($O(n^2)$, where n is the number of tree nodes), an algorithm was devised to reduce the number of page comparisons to be performed (see Figure 9.3). This algorithm exploits the difference between the number of nodes in the two sequences associated with the pages as a lower bound for the edit distance. Such a measure is actually a lower bound for the edit distance because all nodes contributing to it have necessarily to be deleted, if belonging to the first page, or inserted, if belonging to the second page. Additional edit operations may be required for the other nodes. An initial subset of S_2, S, is first determined, containing all pages for which the distance lower bound is minimum (the minimum value is assigned to m). With reference to the example in Figure 9.4, the minimum lower bound is equal to 4, and 3 pages, r, q_1 and q_2, are in S. The edit distance computation (DIST) is then executed on the elements of S, which is expected to be a small subset of S_2. In Figure 9.4, the distance follows the lower bound (a slash separates them). If the minimum edit distance (assigned to d) between p and the elements of S is equal to m, the algorithm stops, since

all nodes in $S_2 \setminus S$ have necessarily a greater edit distance, being the related lower bound greater than $m = d$. If $d > m$ (this is the case for the example in Figure 9.4), the subset of S_2 inside which the minimum distance is looked for has to be enlarged, so as to include nodes with lower bound greater than m and lower than d. This new set, indicated as S' in Figure 9.4, contains all pages with lower bound equal to 5, since the minimum distance on S is equal to 6. The minimum distance d is recomputed on this set (augmented with the previous minimum distance page). The resulting value is a global minimum. In fact, each node outside S has a distance lower bound greater than or equal to d, since S contains all elements with a lower bound between m and d, (elements with lower bound equal to m have already been considered) and the new value of d can only be lower than, or equal to, the previous one. Therefore, no node with distance lower than d can be found outside S. With reference to Figure 9.4, the minimum distance page is r', with a distance equal to 5. Such a distance is lower than 6 (the previous minimum, r) and outside S' distances are surely greater than or equal to 6, since nodes outside S' have lower bound greater than 5.

The two pages at minimum distance are considered matched, and the related pair is added to the relation MATCHED, if such a distance is below a given threshold. All non matched pages are inserted into the sets DEL_1 and DEL_2. Pages having no counterpart in the second language belong to DEL_1, while the pages that are missing in the first language are those in DEL_2.

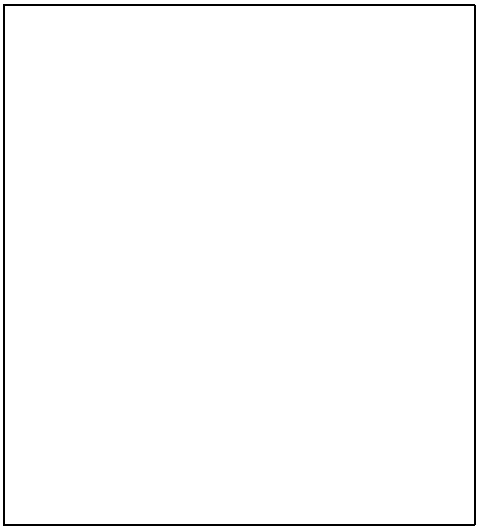

Figure 9.4: *Lower bound and distance between p and each page in S and S′ are shown on the edges, separated by a slash.*

The result of executing this algorithm on a multilingual site is a set of corresponding pages in different languages as well as a set of pages with no counterpart. Moreover, for each pair of matching pages the outcome of the edit distance computation includes the *edit script*, i.e., the set of edit operations that have to be performed to update the pages so as

to achieve full alignment of structure and node attributes (including the text). The Web site developer can exploit this information to update the site and make its multilingual portions consistent with each other.

After aligning pages in the multilingual portions of the site, hyperlinks can be analyzed to identify potential problems. In fact, the evolution of the site could have led to a mismatch of the pages as well as of the hyperlinks. Two categories of hyperlinks should be considered: links internal to each monolingual site portion and links crossing the monolingual boundaries.

If a hyperlink **internal** to a monolingual site portion connects page *p1* to page *p2*, then the two corresponding pages in every other monolingual portion, say *q1* and *q2*, obtained by the alignment procedure described above, should also be connected by a hyperlink. If this is not the case, a warning is issued on the presence of an *internal hyperlink inconsistency* which needs be fixed.

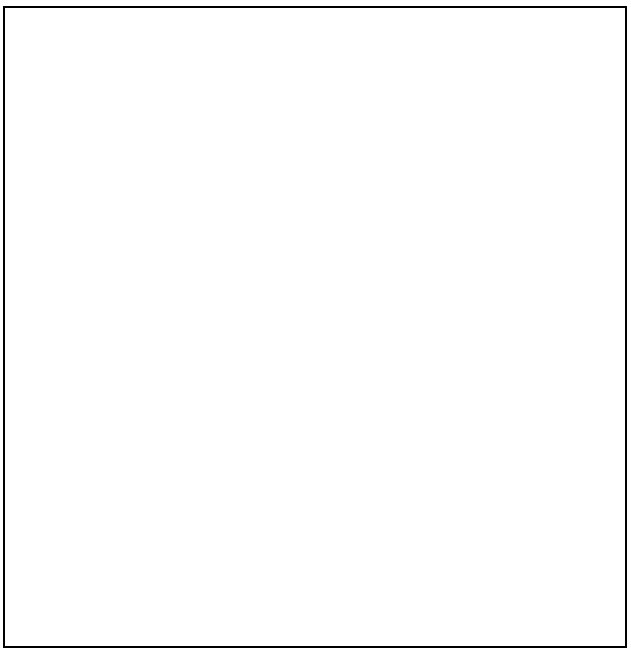

Figure 9.5: *If a cross-language hyperlink connects two pages, a hyperlink between the corresponding pages must also exist (dashed edges).*

If **cross-language** hyperlink connects a page *p_it* to a page *p_en*, and *p_it* and *p_en* are corresponding nodes according to the page match algorithm (see Figure 9.5 (a), dotted edge), a symmetric hyperlink from *p_en* to *p_it* is expected to exist. If this is not the case, the developer is warned on the presence of an *external hyperlink inconsistency*, which can be fixed by removing the hyperlink from *p_it* to *p_en* or adding a hyperlink from *p_en* to *p_it* (dashed link in Figure 9.5 (a)).

A second case occurs when *p_it* and *q_en* are connected by a cross-language hyperlink, but are not translations of each other. For example, the hyperlink may lead from a nested page in a language to the top page in another language (see Figure 9.5 (b)). In this situation, the two corresponding pages, say *p_en* and *q_it*, should be connected by a hyperlink from the former to the latter (dashed link in Figure 9.5 (b)). Otherwise, an *external hyperlink inconsistency* is signaled.

9.2.3 Text comparison

The algorithm for page matching described in the previous sub-section relies on the possibility of deciding whether two texts are one the translation of the other or not. This problem, which we will call the *translation pair problem*, is a natural candidate for the use of Natural Language Processing (NLP) techniques. Such techniques can be quite expensive from a computational point of view, and require large repositories of linguistic knowledge, such as lexicons, grammars, knowledge bases, etc. However, in this section we describe a low cost NLP technique able to solve the problem at stake with acceptable confidence, given the aims of the overall application.

Some preliminary evaluations [TRPG01] showed that elementary techniques based on the comparison of the number of characters in the two texts [Res99] are not satisfactory in our case. In fact, even if it can be shown that if T_1 is the translation of T_2 the difference in the number of characters between the two texts is very likely to be lower than, or equal to, 5 percent, the inverse implication is much less likely to be true. Also, the correlation between having a comparable number of characters and being a translation pair is particularly weak when short texts are considered. Note that the texts attached to the nodes under comparison can be very short indeed: for instance a text can be the title of a page or a phrase enclosed between some formatting tags or even a word which has some hyperlink attached to it.

To improve on the character based approach we tried a different algorithm making use of larger linguistic knowledge (e.g., WordNet). The new algorithm is based on the hypothesis that if a text T_1 is a translation of a text T_2, then it is highly probable that any word in T_1 will correspond to a word in T_2 which is its translation. In this case the inverse implication is much more likely than the inverse implication applied to the comparable number of characters [Res99]. That is, we assume that if all the words of a text T_1 have a translation in T_2 and vice versa, then there is a non negligible probability that T_1 and T_2 form a translation pair, independently from the truth of other conditions.

Unfortunately deciding whether a word of T_1 translates a word of T_2 is not a trivial task, as it requires that the meanings of the two words in the two texts are disambiguated. For this reason we considered a simpler problem, that is finding whether a word of T_1 *can* be

a translation of a word of T_2. We assume that the answer to this question is yes if one of the possible lemmas of the word in T_2 is reported as translation equivalent of one of the lemmas of the word in T_1, according to some bilingual linguistic resource, such as a bilingual dictionary. To check the possible translation correspondences between T_1 and T_2 we also used an English and an Italian morphological analyzers, able to derive all the possible lemmas for the words of the two texts. Moreover, we had to treat a number of special cases in which the translation correspondence cannot rely on bilingual dictionaries. For example numbers, dates, and proper names tend to occur unchanged in texts of different languages. This assumption is often true but with exceptions. In fact certain names are translated (e.g *Florence* = Firenze). Something similar can also happen to numbers which are prices (e.g. *200.000 Lire* = 100 Dollars).

To maximize the significance of the translation correspondence we concentrated on content words, ignoring frequent functional words such as propositions, articles, particles etc. This allowed us to calculate for any candidate translation pairs the *Translation Correspondence Index* (TCI), defined as the ratio between the number of content words that have a possible translation correspondent and the number of content words in the two texts, i.e.:

$$TCI = \frac{content\ words\ with\ translation}{total\ number\ of\ content\ words} \tag{9.1}$$

In an ideal situation the TCI of any translation pair is 1.0. In reality, very often the TCI calculated by our algorithm is lower. This happens because of limitations in the available resources (morphological analyzers, Collins bilingual dictionary) or because of free or approximate translations. After a preliminary analysis of a corpus of Italian-to-English translation pairs, we arrived at the conclusion that in most translation pairs the TCI is equal or higher than *0.3*. Using this threshold we checked the reliability of the inverse implication, that is, we checked how often a pair that has a TCI equal or higher than 0.3 is a translation pair. The algorithm solves the translation pair problem with a precision (correct pairs over pairs retrieved) of *0.24* and a recall (correct pairs over pairs to be retrieved) of *0.73* on a corpus of translation pairs extracted from a set of Web pages [TRPG01]. Although precision is apparently low, the combination of this algorithm with the structural page matching algorithm gives good performances (more detailed comparison data between the alternatives are provided in [TRPG01]).

From a qualitative analysis of the cases in which the proposed algorithm fails, it is clear that using more sophisticated techniques and larger linguistic resources would improve the performance of the algorithm. For example the algorithm would gain from the application of sense disambiguation techniques and from the ability to recognize and translate multiword lexical units instead of only single words (*agriturismo* = holiday flats in a farm house). However we think that the technique based on the possible translation equivalents presented here is a good compromise between complexity and efficiency.

9.3 Multilingual XHTML

MLHTML is an XML representation of multilingual Web pages, with information in all languages inserted into a single source file. The physical proximity of the information is the key to ensure consistency. In fact, document format and structure are shared among languages, and thus cannot be inconsistent. The multilingual content is inserted inside adjacent lines, so that content inconsistencies can be detected easily by visual inspection.

MLHTML adds just one new tag, `<ml lang="`L`">`, to XHTML. The tag includes the mandatory indication of a language identifier, L, assigned to the `lang` attribute, and is matched by the associated closing tag, `</ml>`. The text between opening and closing tags is assumed to be in language L. For example, a multilingual page providing information in English (`en`) and Italian (`it`) may include the following lines, mixed with normal XHTML code:

```
<title>
<ml lang="en"> Publication List </ml>
<ml lang="it"> Lista delle pubblicazioni </ml>
</title>
```

MLHTML pages can be transformed into HTML code statically (offline) or dynamically (by the server/browser). In the former case, multilingual references to objects (HTML pages, images, etc.) have the form:

```
<ml lang="en">
<a href="ref-page.en.html"> en-text </a></ml>
<ml lang="it">
<a href="ref-page.it.html"> it-text </a></ml>
```

while in the latter case the language becomes a parameter to be handled dynamically:

```
<ml lang="en">
<a href="ref-page.ml?lang=en"> en-text </a></ml>
<ml lang="it">
<a href="ref-page.ml?lang=it"> it-text </a></ml>
```

Once a multilingual Web page *p.ml* has been coded in MLHTML, HTML pages in any requested language can be produced in three different ways:

offline: HTML pages are produced statically, once for all, and published by the Web server, similarly to any other static HTML page;

by a server script: the request of a multilingual page is recognized (e.g., by the extension) and redirected to a server side script that generates dynamically the HTML page with the content in the requested language;

by the browser: XML enabled browsers can directly process multilingual pages, accompanied by a stylesheet for the selection of the portion in the requested language.

For the offline generation of static HTML in all supported languages, it is possible to use any of the available standard XSL processors (for example, *Saxon*[2]). Their input is the XML document (in our case, *p.ml*), and the stylesheet to use for its processing. We have developed (and made publicly available) the XSL stylesheet *ml.xsl* to be used in conjunction with the MLHTML format. Language specification is obtained by passing the parameter `lang=L` on the command line to the XSL processor.

For the dynamic generation of the HTML pages by the Web server, the PHP script *ml.php* was developed. Every file with extension `ml` has to be preprocessed with *ml.php*. This can be declared to the Web server by modifying some of its configuration files (with Apache, `.htaccess`).

Finally, XML enabled browsers can directly process *p.ml* (possibly renamed *p.xml*), accompanied by the same stylesheet (*ml.xsl*) used for offline HTML generation.

Maintenance of the multilingual pages can now be centralized. Each time a change has to be made, only the MLHTML version of the page (`p.ml`) is modified. Then, the pages in the different languages are re-generated automatically, either offline or dynamically. Consistency of the evolved site is thus granted.

Given an existing site in which different pages were written for the different supported languages, it is an easy task to restructure it into MLHTML, provided that pages in different languages are aligned. A further requirement is that the input is compliant with XHTML. If this is not the case, automatic conversion tools such as *Tidy*[3] can perform the job of updating the HTML page to the new standard. Then, since the structure of the corresponding pages is the same and the only differences are in the content, provided in different languages, it is possible to automatically merge the multilingual contents into a single page by exploiting the `<ml lang="`*L*`">` tag and to retain the XHTML code for the shared page structure. This work is performed by our tool *PageMerger*. The result of tool execution is an MLHTML page ready for publication on the Web. To improve its readability, pretty printing tools for XML documents (*Tidy* does also this job) can be used.

[2]http://saxon.sourceforge.net/
[3]http://tidy.sourceforge.net/

After completing the re-structuring process, the old multilingual pages can be discarded and only MLHTML documents are evolved.

Web pages are often maintained at a higher level than pure HTML code. Graphical tools with nice formatting facilities are available for the creation of Web pages. A new generation of such tools can be devised embedding support for multilingual sites. Their graphical presentation of the page under edit may include support to switch from a language to another, with the effect of changing the content and retaining the formatting. Similarly to current tools, which internally reference the HTML code of the displayed page, multilingual tools could internally reference the associated MLHTML page.

MLHTML pages can also be exploited when a site generates its pages dynamically. In this case, typically, when a page is requested, the server side program that generates it inserts a content in a language depending on some parameter, associated with the selected language. This complicates the code of the server side program, in that every print instruction must include a conditional statement to produce the text in the appropriate language. All such switches can be eliminated if MLHTML is adopted. The server side program can simply generate an MLHTML page including the content in all supported languages. Then, its output is filtered by *ml.php* before being sent to the client browser, in order to allow only the selected language to pass through. This operation is not even required if the browser is assumed to be XML enabled. Such a solution has the great advantage of completely separating page generation from language selection. The current practice mixes these two operations, thus giving rise to complex server programs. MLHTML gives the opportunity to simplify them: language selection is done once for all by *ml.php* or *ml.xsl*.

Re-Structuring an existing dynamic site is far more complex than re-structuring a static site. It is necessary to update the server programs by identifying the code portions the execution of which is language dependent and by replacing them with the production of MLHTML statements. Such a transformation can be partially automated, for example by exploiting pattern matching, but this investigation is out of the scope of the present work.

9.4 Case study

The tools developed to support the re-structuring process described before have been applied to the Web site `www.ien.it`, the official site of the National Electro-technical Institute (IEN) "Galileo Ferraris". This Institute performs important duties in the metrological field regarding time and frequency, electromagnetic, photometric and acoustic quantities. Specifically, IEN carries out studies and researches oriented to the realization of the primary standards of the IS measurement units and to the determination of fundamental physical constants. Among the other services, IEN holds the clock giving the official Italian Standard Time, which is also accessible from the Web site.

Original		Portion restructured	
Pages	857	Pages	440
HTML LOC	165887	HTML LOC	51674

Table 9.1: *Features of the IEN site.*

The considered site was downloaded by means of the Spider module of the Web site analysis tool **ReWeb**. The static portion of the site includes 857 HTML pages for a total of about 165 kLOC (thousands Lines Of Code). It is a bilingual site, providing the same information in Italian and English. The initial page contains two links to the Italian and English version respectively, and (almost) the same page organization is replicated within the two sub-sites.

Figure 9.6 shows a portion of the site, with pages split between English and Italian (the initial page, `index.html`, is bilingual). The two initial pages of the two sub-sites reachable from the initial page, `index_i.html` and `index_e.html` are connected by a bidirectional cross-language hyperlink. A visual inspection of this site portion immediately reveals some inconsistencies. Page `Pagina4.html` is available only in Italian, but it is referenced also by English pages. Symmetrically, pages `history.html`, `present.html` and `staff.html` are only in English, but are referenced also by Italian pages. A language-related inconsistency visible in Figure 9.6 is particularly awkward: if `Pagina4.html` – an Italian-only page – is reached from an Italian page (for example `services_i`) and then the standard time is accessed, the English version of the related page is showed (`stittime.shtml`), although the Italian version does exist (`stittime_i.shtml`). The presence of several inconsistencies even in this small site portion suggests that re-structuring is highly desirable.

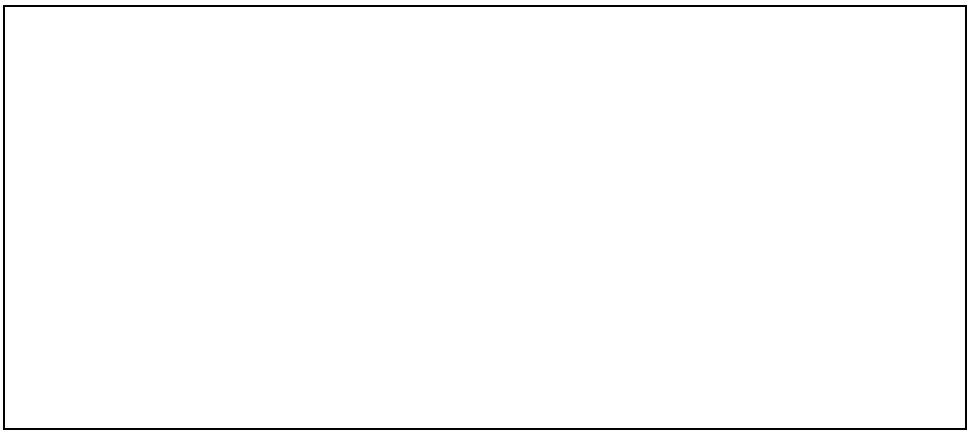

Figure 9.6: *Graphical view of a portion of the bilingual site* `www.ien.it` *computed by* **ReWeb**.

The JavaCC HTML parser written by Brian Goetz (`www.quiotix.com`) has been employed to obtain the syntax tree associated to each page of the site. Syntax trees are decorated

with attributes, such as *text* and *image*, which are used by subsequent analysis phases. The text attribute is used to recognize the page language, while the tree structure, combined with node attributes, is the basis of the page matching algorithm.

English	Italian	Unknown	Ambig.	Errors
268	167	5	0	10 (2.2%)

Table 9.2: *Page classification according to the language. Last column gives the number of classification errors.*

The re-structuring process aimed at migrating the site to MLHTML was applied to a site portion consisting of 440 pages (51 kLOC). Personal directories of the Institute's staff and directories with no page translated were excluded from re-structuring. The first step in the page alignment procedure was language identification. The tool for language identification produced the classification summarized in Table 9.2. Only 2.2% of the language labels produced by the tool are incorrect. An example is page `staff.html`, which contains a long list of names and only a few English words. The number of errors from both dictionaries is high and the page is classified as *Unknown* instead of *English*. Other incorrect classifications are associated with pages which are internally bilingual. The score produced by the tool is lower than 0.5, so that the language label is not *Unknown*, but it is usually close to 0.5 (e.g., 0.49 for `disclaimer.html`). Consequently, these errors are easy to detect since the score produced by the tool hints a possible classification error (i.e., it is an indicator of confidence in the classification). The overall automatic support to language identification was considered good and manual interventions were very limited (a few minutes in total).

Matched pairs	Errors	Not transl.	Biling.	Errors
122	3 (2.4%)	193	3	0

Table 9.3: *Pairs of aligned pages and nonaligned pages.*

The next step of the re-structuring process was page alignment. Our tool was able to match 122 Italian pages with the respective translations in English, with 2.4% of alignment errors (see Table 9.3). 193 English-only or Italian-only pages were automatically detected by the tool, with no error. For example, page `present.html`, giving a presentation of the Institute, is in English even if accessed from the Italian sub-site. Bilingual pages are recognized as such during language identification. The page matching procedure marks them as unmatched, similarly to those not translated.

Aligned pages with edit distance greater than 0 were manually updated so as to become consistent with each other. The total number of edit operations in the edit scripts automatically produced by our tool is provided in Table 9.4. Operations are classified as tag

Totals			
Edits	Tags	Transl	Errors
622	212	410	348
Average values per page			
Edits	Tags	Transl	Errors
5.0	1.7	3.3	2.8

Table 9.4: *Edit operations for the alignment of the pages, classified into tag updates and missing translations. Last column refers to unrecognized translations.*

updates (tags or tag attributes are modified, reordered, inserted or deleted to become consistent), and translations (the tool signals that a text pair is not a translation pair because of a TCI lower than 0.3). Errors in the specification of edit operations are always related to the *Transl* category. They are associated with translated text pairs that were not recognized by the tool, due to the presence of free translations, technical terms, abbreviations or contractions (e.g., "CNR"). As apparent from Table 9.4, such cases are quite frequent: 348 over 410. The time necessary to manually update a page and make it consistent with its translation strongly depends on the edit distance from its translation. Typically a few minutes are sufficient to complete the update of a page with about 5 edit operations to be performed, but this may increase up to half or one hour if the number of edit operations is higher and/or they require the translation of long texts that were not originally translated.

After aligning all pages, the *PageMerger* tool generated the MLHTML pages and the original site was (locally) replaced with the new one in MLHTML. Published pages are automatically generated by the server script *ml.php* on demand. A few MLHTML pages required some minor edit interventions, related to the cross-language hyperlinks. In the following code fragment, the language selection had to be changed, since the referenced page is in a language different from the current one. The following line was therefore modified manually:

```
<ml lang="it">
<a href="/index.ml?lang=en">
English version </a> </ml>
```

The resulting MLHTML site was navigated in parallel with the original one, providing the same page organization and information (when initially available), with the noticeable remark that the new site does not contain differences in presentation or content depending on the language chosen. Maintenance of the new site is expected to be simpler, since the number of pages is halved, structural consistency during evolution is automatically granted and texts in different languages are adjacent, thus enabling simultaneous updates.

An example of page alignment is provided in Figure 9.7, while Figure 9.8 shows the ML-HTML page obtained re-structuring the Italian and English pages of Figure 9.7.

9.5 Related works

To the author knowledge, the problem of re-structuring an existing Web site to ensure consistency among its multilingual portions has not been considered in the literature. Works on Web site re-structuring focused on model-based re-engineering [ACCL00] and on migration to dynamic content [BK01].

An experience of Web site re-engineering is described in [ACCL00], where the target representation of the reverse engineering phase is based on RMM [IKK97]. The recovering process and the following re-design activity were conducted almost completely manually. On the contrary, our approach to Web site re-structuring is highly supported by tools and manual interventions are minimized, with a positive impact on the cost/benefit trade-off.

The re-structuring process described in [BK01] aims at removing the content from the HTML code and at storing it in a database. When a page is requested, its content is generated dynamically by accessing the database and restoring the requested information. The main advantages of this intervention are a higher maintainability of the resulting dynamic site and the possibility to detect duplications of content, that are eliminated in the database. The basic techniques exploited in this work are similar to ours, in that the syntactic structure of the HTML pages is analyzed to extract the information of interest. Moreover, a text comparison utility is used in both cases to increase the consistency of the information provided. The main difference is that we consider cross-language consistency, while in [BK01] duplication removal and database storage ensure a higher level of intra-language consistency and accuracy. Consequently, our text comparison method is not just based on string equality, but has to revert to dynamic programming and NLP techniques.

Figure 9.7: *English (left) and Italian (right) HTML sources of pages* `depts.html` *and* `depts_i.html`. *Dashed lines indicate item correspondences.*

Figure 9.8: *MLHTML page obtained after re-structuring* `depts.html` *and* `depts_i.html` *(see Figure 9.7).*

Part V

TOOLS

Chapter 10

ReWeb and TestWeb

In this chapter the toolkit architecture implemented for supporting the analysis, testing and re-structuring of Web applications is considered. Some indications on possible algorithms and solutions, based upon our experience in the development of the two research prototype tools **ReWeb** *and* **TestWeb***, are provided. The two research tools have been developed to test and to validate our techniques and to support analysis, re-structuring and testing of Web applications.*

10.1 Toolkit architecture

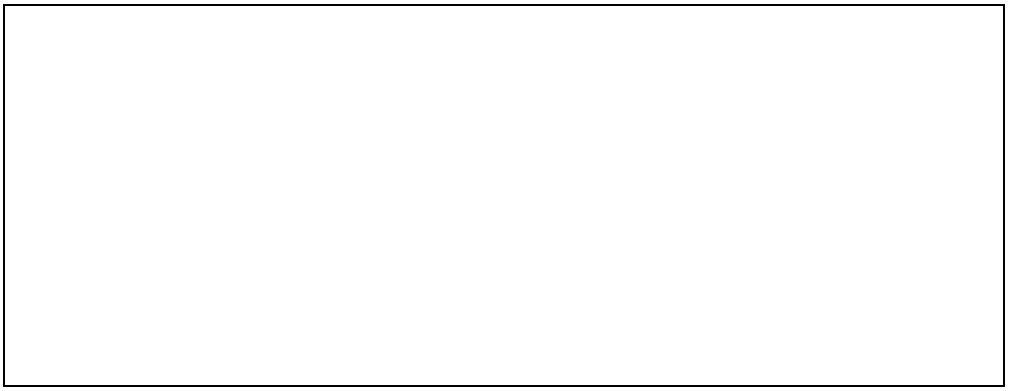

Figure 10.1: *Roles of ReWeb and TestWeb.*

The two research prototype tools **ReWeb** and **TestWeb** have been developed to test and to validate, on real Web applications, our techniques and to support complex tasks such as: analysis, re-structuring and testing of Web applications. Their relative roles are

schematized in Figure 10.1 (to see in conjunction with the Table of symbols in Figure 10.2). **ReWeb** and **TestWeb** are composed of independent software modules, written mainly in Java language (but also C and Perl), that communicate by means internal files. **ReWeb** and **TestWeb** are thought as a toolkit able to help the user (i.e web master, test engineer, maintenance engineer etc.) in the difficult tasks of analysis, re-structuring and testing. The whole process is semi-automatic: the interventions of the user are indicated within ellipses while user inputs in diamonds (see Figure 10.1). Both tools perform their operations on an abstraction of the given Web application, indicated in Figure 10.1 as UML model.

ReWeb downloads and analyzes the pages of a Web application with the purpose of building a UML model of it (chapter 3) and producing some analyses, views and reports (chapter 4). Other modules of **ReWeb** are devoted to re-structuring (chapters 8 and 9) the Web application taken in input and compute slices (chapter 5). These last outputs are indicated in Figure 10.1 with the generic term *Additional outputs*. **ReWeb** contains a module called *Spider*, which downloads all pages of a target Web application starting from a given URL (by means a connection to Internet). The HTML documents outside the Web application host are not considered. The pages of a site are obtained by sending the associated requests to the Web server. For dynamic pages, the user has to specify the set of inputs that the *Spider* will provide to the server programs generating them (*input file* in Figure 10.1). To obtain a model representing all behaviors of the server programs, more inputs can be provided by the user, associated with the same dynamic page. The resulting model is an explicit-state model in which server programs and dynamic pages have been unrolled according to all different conditions specified by the user in terms of different input values. The operations of state merging of type 2 and 3 (see chapter 3) are currently performed manually (*manual activities* in Figure 10.1), while type merging of type 1 is automated. If no input is specified for a given dynamic page, the *Spider* will not expand the model beyond the related server program. This feature of **ReWeb** is useful to analyze the functioning of portions of a Web application.

The Spider can also behave in local. In this case, it uses as input the *Web server file system*, instead of the connection to Internet.

TestWeb implements structural testing and statistical testing of Web applications, described respectively in chapters 6 and 7. **TestWeb** produces a set of paths from the UML model, according to a user defined testing criterion (such as, for example, page testing and hyperlink testing for structural testing and prioritization by likelihood and stochastic visit for statistical testing) and to generate test cases from it. The usage model, useful for statistical testing, is built by **TestWeb** adding to the UML model the transition probabilities computed using the log file of the target Web application. Generated test cases are sequences of URLs which, once executed, grant the achievement of the selected criterion. Input values in each URL sequence are those specified in the *input file* (see dotted arrow in Figure 10.1), but the user can modify them (*Input values* diamond in Figure 10.1).

Figure 10.2: *Table of symbols.*

TestWeb can now provide the URL request sequence of each test case to the Web server, attaching proper inputs to each form. The output pages produced by the server are stored for further examination. After execution, the user (e.g., test engineer) intervenes to assess the pass/fail result of each test case (ellipse *Check*, Figure 10.1). A second, numeric output of test case execution is the level of coverage reached by the current test suite.

10.2 ReWeb

The **ReWeb** tool consists of six modules: a Spider, an Analyzer, a Viewer, a Slicer, a Web application Re-structuring tool and a Multilingual alignment tool (see Figure 10.3). Grey modules are not yet completely implemented (at the time of the thesis).

10.2.1 Spider

The Spider (chapter 3) downloads all pages of a target Web application, starting from a given URL, using a Internet connection (alternatively recovering pages directly from the *Web server file system*) and providing the input required by dynamic pages. Moreover, it builds a UML model of the downloaded site. Each page found within the site host is downloaded and marked with the date of downloading. The user has to specify the set of inputs for each page that contains Forms (Input file, Figure 10.3).

Problems encountered in the construction of the Spider.
Web pages are not actually written using only HTML. They can be rather regarded as multilanguage documents, where code fragments in languages different from HTML can be loaded (e.g. Applets) or interpreted (e.g. Javascript). Libraries for the construction of Spider programs are available for programming languages such as Perl, C/C++, or Java. Examples of Java class libraries are WebSPHINX[MB98b] and those written by Brian

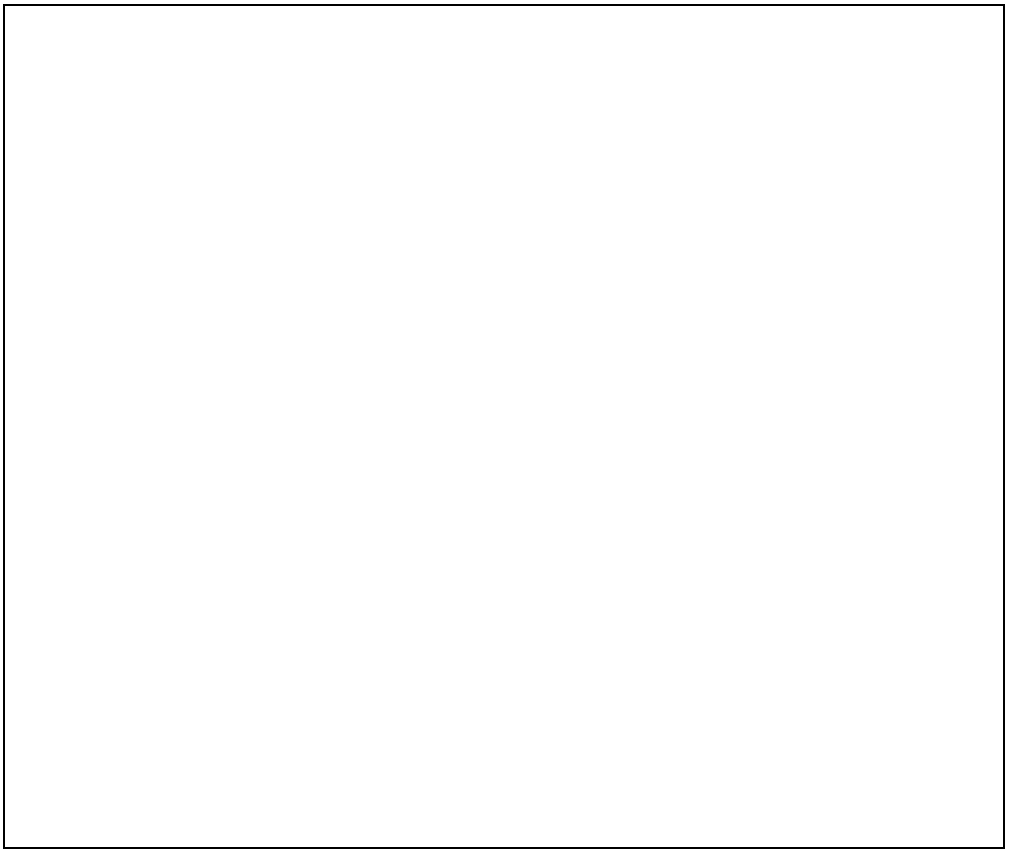

Figure 10.3: *High level architecture of* **ReWeb**.

Goetz (an HTML parser based on JavaCC[1]). We decided to implement our Web Spider just exploiting the Java language and its standard library, starting from scratch, in order to have total control on the multilingual aspects of the downloaded pages. We developed an extractor (*URLs and FORMs extractor*, Figure 10.3) which recognizes both HTML and Javascript code fragments, and extracts the needed information (such as links, forms and frames) from them. Figure 3.3 (chapter 3) shows the pseudo-code of the Spider.

Problems encountered in the construction of the Spider are due to irregularities and ambiguities present in HTML code, also noted in [WBM99b], and to the current state of the Web technology, offering a large spectrum of alternatives to implement a Web application. Our solution was to implement an extractor (a scanner that recognize only the needed information without building a complete parse tree of the page) instead of a parser, so that it could accept a superset of HTML including the main irregularities commonly recognized and properly interpreted by available browsers.

[1] www.webgain.com/products/java_cc/documentation.html

Dynamic pages pose additional problem to the activity of the Spider. Since the content of these pages is decided at run time, it may in general depend on the input previously provided by the user. In particular, the structure of a dynamic page may change when it is encountered in a different interaction. Since the model of a Web site encompasses all possibilities, the Spider has to recover all variants of a dynamic page, and has to merge them into a single representative object. This can be achieved by specifying the input values to be provided before downloading the dynamic page of interest. Moreover, the same dynamic page has to be downloaded several times, with different inputs, when the different conditions generate a different page structure. All sequences of input values to be provided before each page download are specified in a file which is read by the Spider. All dynamic pages specified in the file are downloaded after providing the Web server with the given inputs. Finally, all versions of the same page are merged. Although the number of inputs to be provided may explode combinatorially, our experience suggests that in practice few alternatives are sufficient to cover all variants.

An additional input that may affect the content displayed in a dynamic page is the *cookie* that the browser provides to the Web server. A *cookie* is a user identifier that is stored by the browser in the local file system and is provided to the Web server to allow user identification each time a new connection with a given server is established. After recognizing the user, the Web server can provide a customized version of the dynamic pages in the site. Since their structure may depend on the cookie, the Spider needs the ability to send a cookie to the Web server, in order to obtain also the pages that are generated when the user is identified.

10.2.2 Analyzer

The **Analyzer** consists of six Java modules (see Figure 10.4) devoted to different analyses. Grey modules are not yet completely implemented.

Since that the UML model of a Web site can be interpreted as a graph (by associating objects with nodes and associations with edges) algorithms of *flow analysis* and *graph traversal* are applicable on it. The module devoted to structural analyses uses algorithms of *flow analysis* and *graph traversal* to compute the analysis already described in chapter 4. The *History analyzer* produces the data useful to compute the evolution analysis. Such an analysis requires the ability to compare successive versions of pages of the same Web application and to graphically display the differences. Given some versions of the same Web application, downloaded at different dates, their comparison aims at determining which pages were added, modified, deleted or left unchanged. The *History analyzer* stores in the database *History data* the first version of the Web application downloaded (corresponding to the older date) and the successive variations. *Views generator* and *Reports generator* generate the files representation of the views produced (e.g., *structural*, *history* and *system*)

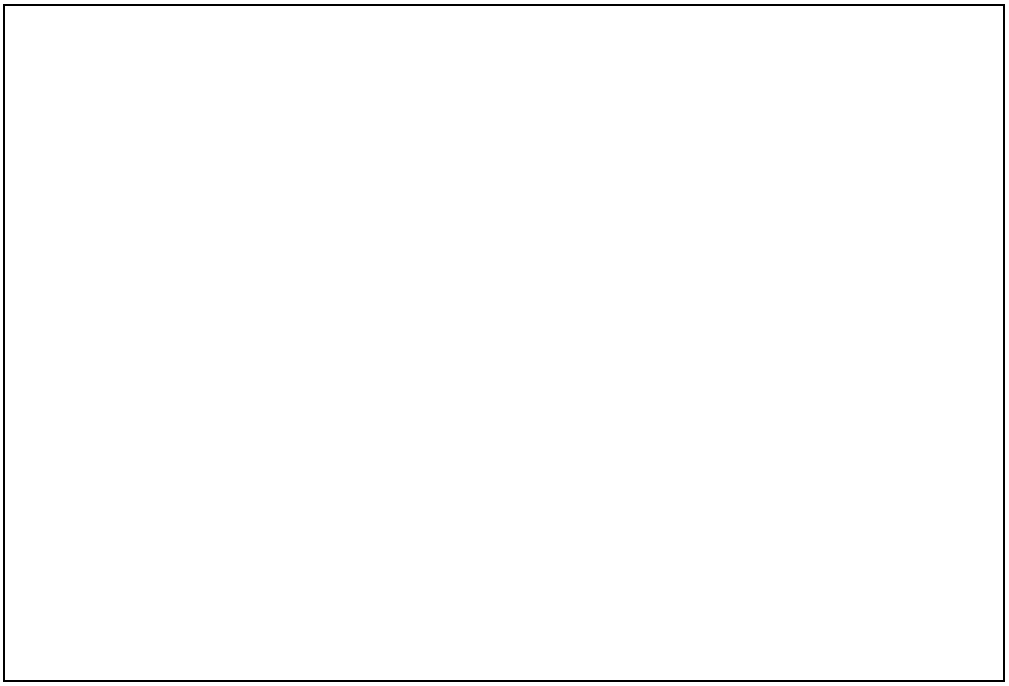

Figure 10.4: *Spider and Analyzer.*

and of the reports (chapter 4). The *Statistical analyzer* (not yet completely implemented) computes, using historical information contained in the *log file*, the statistical analyses of *stationary probabilities* and *average paths* described in chapter 6. The *Pattern matcher* (not yet completely implemented) matches a library of known patterns (*Patterns library*) against a given Web application and identify portions of the application that are compliant with a predefined structure from the pattern library (process described in chapter 4).

10.2.3 Viewer

The *Viewer* reads the files representation of the views, generated from the *Analyzer*, and produces the graphic representation of the *structural*, *history* and *system* views (see Figure 10.3). Examples of such views are in Figure 4.11, Figure 4.4 and Figure 4.5 (chapter 4). The *Viewer* is based on Dotty, a customizable graph Editor developed at AT&T Bell laboratories by Eleftherios Koutsofios and Stephen C.North.

The *Viewer* uses the algorithm explained in [GKNV93] for drawing directed graphs. The aesthetic principles followed by the algorithm are: to expose hierarchical structure (if any) in the graph, to avoid edge crossings (if possible) and sharp bends, to keep edges short, to favor symmetry and balance. The layout algorithm of the *structural view* of a Web application is very important to understand its structure, especially when the application

is very complex.

A problem connected with the visualization of a Web application is the fact that also small applications (e.g. with 100 pages) can have an entangled structure difficult to understand. A way to improve the Viewer display is to use techniques to abstract, to simplify, to extract a portion of, or to see only a part of the *structural view*. Another possibility can be to add other views of a Web application as for example the birdeye and overview diagrams or to display only the depth-first tree without return edges (solution adopted in [MB98b]).

The views and facilities we propose follow. The *system view* represents the organization of pages into directories; the *data flow view* displays the read/write accesses of pages to variables, respectively through incoming and outgoing edges linking pages to variables; the *history representation with percentage bars* describes, in compact way, the percentages of nodes with the same color. Among the provided facilities, the viewer supports zoom, search, deletion of incoming or outgoing edges, and focus. The facilities for focusing on and searching a node are useful when the visualized graphs are very large. By exploiting the focusing facility it is possible to display only a limited neighborhood of a selected node. Another possibility to access the *structural view*, not yet implemented, is the identification and extraction of a portion of a Web site by means of pattern matching techniques.

10.2.4 Slicer

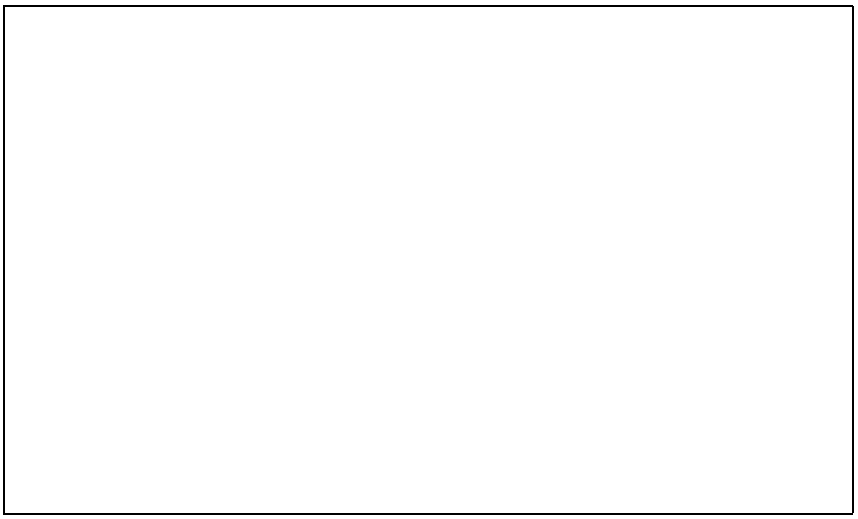

Figure 10.5: *Slicer.*

The Slicer is composed of two modules: one devoted to compute static slices (see Figure 10.5, bottom) and one dynamic slices (see Figure 10.5, up).

The module devoted to compute static slices is composed of the *SDG builder* and the *Static slicer*. The *SDG builder* procedure (not yet completely implemented) takes in input the *Web server file system* of a dynamic Web application, composed of HTML and PHP files (at the present the only server side language treated) and builds the System Dependence Graph (SDG) of the target application (see chapter 5). After that the user inserts where to slice the application (i.e. *line* and *variable*), the *Static Slicer* can compute the slice. The *Static slicer* obtains the slices from the SDG by starting from the node of interest (corresponding to the *line* inserted by the user) and traversing backward all dependences until the transitive closure is reached. The Slicer module is based on a preliminary prototype tool that has been implemented in TXL [CDMS02]. TXL is a programming language designed to support computer software analysis and source transformation tasks. Since each TXL program is formed by a description of the structures to be transformed (grammar) and by a set of transformation rules, the first step in the *Static slicer* implementation has been draw up in TXL language the HTML grammar and the PHP grammar. The problem of statically building the SDG, that is in general undecidable, (see subsection 5.2.3) has been considered in [TR03b], where some automatic transformations has been devised and implemented to limit it.

Dynamic slicing (section 5.3) has not been yet implemented. To implement a module computing dynamic slices we are in need of a *Tracer* and a *slicer* that works on traces instead of the SDG. The *Tracer* will have to be able to trace server/client statement execution, to store the HTML pages traversed during the navigation, and to trace the association between each server statement generating HTML code and the code generated. The *Dynamic slicer* will have to obtain the slices starting from the statement of interest (corresponding to the *line* inserted by the user) in the trace and traversing backward all data, nesting/control and build dependences until the transitive closure is reached.

10.2.5 Re-structuring tool

The *Re-structuring* tool is thought as a toolkit able to help the user in the task of Web application re-structuring. This tool takes in input the HTML files downloaded by the *spider* (it can also behave with the *Web server file system*) and the UML model of the target Web application and applies to them the transformation inserted by the user. At present, only some of the transformations proposed in chapter 8 have been implemented. In particular, we have implemented the intra-page transformations: *add_noframes, add_base, applet_to_object, add_missing_tags* and the inter-page transformation: *reorganization into frames*. In chapter 8 the implementation of the *reorganization into frames* transform, developed in DMS, has been described in great detail (Fig. 8.6).

10.2.6 Multilingual alignment tool

Figure 10.6: *Multilingual alignment tool.*

The operations performed by the tool we have developed (see Fig. 10.6) aim at determining the correspondences between pages in different languages and at resolving inconsistencies (missing pages, tags, links or translations).

A first problem to be addressed is how to partition the pages in the site according to the language. The *language identification* procedure, which analyzes the content of the pages, is a useful support at this operation. Two different solutions to this problem have been adopted. The related performances have been contrasted on a benchmark of multilingual Web sites in [TRPG03]. The first algorithm (not explained in the thesis) uses an open source software available on-line, named *Textcat*[2], that implements the text categorization algorithm presented in [CT94]. This algorithm compares the N-gram (bi-gram, tri-gram, etc.) frequencies of the input with the N-gram frequencies that represent the various languages. The result of the algorithm is the language associated to the frequency distribution at lowest distance from the input. The second algorithm (explained in chapter 8) is based on the entries available in a set of mono-lingual dictionaries, associated with the words in the page to be classified.

After assigning pages to different languages, the correspondences between pages are retrieved (*Page matching* phase). For such a purpose the *structure* of the pages is exploited: the syntax tree of the HTML code provides information on the page formatting and on the location of text, images and other objects contained in the page. A structural comparison algorithm has been described (see chapter 8) to infer the match between corresponding pages. A relevant contribution to this algorithm comes from the comparison of attributes of the syntax tree nodes, and, among these attributes (node labels, etc.), of the text (algo-

[2] `http://odur.let.rug.nl/~vannoord/TextCat/`

rithm of *text comparison*, Figure 10.6). Two different methods, one based on the empirical observation [Res99] that two texts are the translation of each other if they have a comparable number of characters (method not explained in the thesis), and the other based on the availability of a bilingual electronic dictionary, are employed to determine if the text attribute of a node is the translation of another (see [TRPG03] for a comparison). The result of the *Page matching* phase is a set of corresponding pages in different languages as well as a set of pages with no counterpart. Moreover, for each pair of matching pages the outcome of this phase includes the set of operations that have to be performed to update the pages so as to achieve full alignment of structure and node attributes (including the text). The Web site developer can exploit the output of *Page matching* to add/update any missing/misaligned translations or tags, and to modify the site, making its multilingual portions consistent with each other (*Page alignment* phase).

After aligning pages in the multilingual portions of the site, hyperlinks can be analyzed to identify potential problems (*Hyperlinks analysis* phase). A common awkward hyperlinks inconsistency is an unintended "jump" between different languages, i.e. a cross-language link that connects two pages p (language L) and q' (language L'), while the intention of the designer was an intra-language link from p to q, where q is the translation of page q' in language L.

The *Page merging* phase follows the phases of *Page alignment* and *Hyperlinks analysis*. This phase transforms automatically a set of corresponding HTML pages into a unified format, MLHTML (explained in chapter 8) being one possible choice, based on XML. It can be noted that all operations supported by the toolkit are desirable even when a unified format different from MLHTML is adopted, or no unified format is adopted at all, since they produce a Web site in which the provided information does not contain differences depending on the chosen language. When the format is different from MLHTML, the only toolkit module to be changed is the one supporting *Page merging*.

10.3 TestWeb

TestWeb is composed of three modules (*Test generator, Usage model generator* and *Test executor*, see Figure 10.7) completely implemented in Java language.

Test generator and Usage model generator
The *Test generator* generates test cases from the UML model of a Web application. The user has to add some information to the model produced by **ReWeb** to complete it for testing purposes and furthermore the user has to choose a test criterion. The user specifies the page type when the distinction between static and dynamic pages cannot be obtained automatically (e.g., dynamic pages with no input). The user also provides the set of used variables, *use*, for each dynamic page whose content depends on some input value.

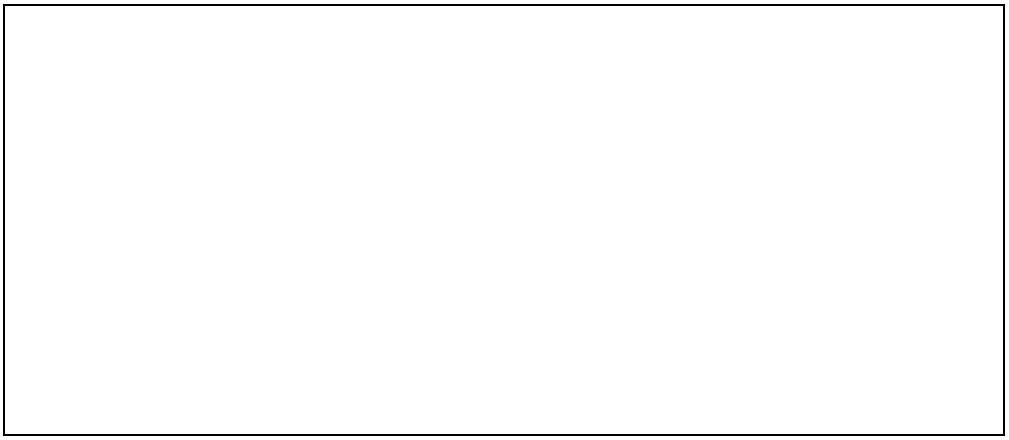

Figure 10.7: *Architecture of the tool TestWeb.*

Additional manual interventions, related to *state unrolling* and *page merging*, have been already described in chapter 3.

The criterion chosen by the user discriminate between structural and statistical testing. In the case of white box testing criteria (such as page testing, hyperlink testing, definition-use testing, all-uses testing and all-paths testing - chapter 6) a module of the *Test generator* computes the path expression of the model (see reduction algorithm in chapter 6) and another module determines from it the paths satisfying the criterion inserted.

In case of prioritization by likelihood or stochastic visit criteria inserted by the user, the *Usage model* (chapter 7) produced by the *Usage model generator* is used by the *Test generator*. Given the number of paths wanted (n) and the *usage model*, two different modules of the *Test generator* are devoted to compute the n most probable paths (prioritization by likelihood) and the n paths randomly generated (stochastic visit, useful for computing reliability, MTTF and MTBF).

The first module, computing the n most probable paths, is based on a algorithm that uses a priority queue. Consider the application of this algorithm to the example of Figure 10.8. It starts (step 1) by putting in the priority queue P the couples (*probability*, *path*) with *path* starting from the initial node, i.e., in our example:

(3/5, [(H,S1)]) and (2/5, [(H,S2)])

and proceed as follows. The couple with most high probability (the first) is extracted (step 2). For each edges starting from the arrival node of the *path* of the extracted couple (i.e. S1) a new couple (only one in our example, that is (3/5, [(H,S1),(S1,D1)])) is computed and inserted in P. The path of the new couples is up-dated and the probabilities are computed by multiplying the probability of the edges in the path. At step 3 of our example, P is equal to:

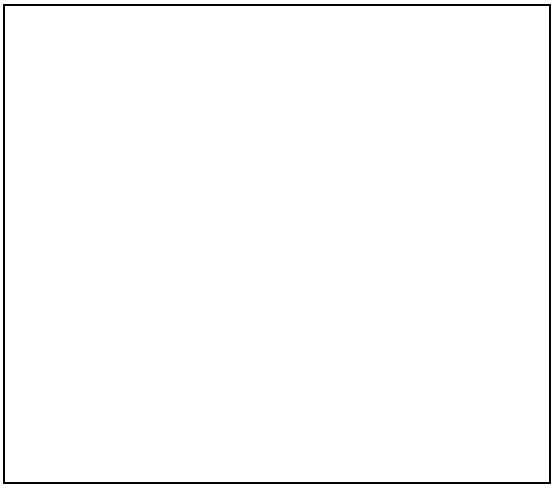

Figure 10.8: *Example of* Usage model.

[(9/20, [(H,S1),(S1,D1),(D1,X)]), (2/5, [H,S2)]), (3/20, [(H,S1),(S1,D),(D1,S1)])]

When a path in the priority queue has arrival node equal to X, case of (9/20, [(H,S1),-(S1,D1),(D1,X)]) the couple is extracted and printed in output. The process continues until the number of paths printed is less than n.

The second module, that generates n random paths, performs a stochastic visit of the *usage model* by choosing which edge to traverse in accordance with the transition probabilities of the outgoing edges.

Once paths are generated, test cases can be produced adding to them input values. Input values for the variables collected through forms can be obtained from those inserted into the *Input file* (see Figure 10.7), or the test engineer can insert new input values provided that they belong to the same equivalence class of values stored into the *Input file*. Generated test cases are sequences of URLs which, once executed, grant the coverage of the selected criterion.

Test executor
The *Test executor* simply provides the URL request sequence of each test case to the Web server and stores the resulting HTML pages in a file (*Output pages*). After execution, the test engineer intervenes to assess the pass/fail result of each test case. For such an evaluation, she/he opens the output pages on a browser and checks whether the output is correct for each given input (*functionality testing*). During regression check such user intervention is no longer required, since the oracle (expected output values) is the one produced (and manually checked) in a previous testing iteration. Of course, a manual intervention is still required in presence of discrepancies.

A second, numeric output of test case execution is the level of coverage reached by the

current test suite, equal to TP (i.e. Total Probability, see chapter 7) in case of statistical testing, and giving the percentage of entities (pages, hyperlinks, data flows, etc.) covered, in case of structural testing. In the case of statistical testing the user can compute the probabilistic aspects of the testing process, such as: reliability (R), mean time to failure (MTTF) and mean time between failures (MTBF).

Part VI

CONCLUSION

Chapter 11

Conclusions and Future Work

This chapter concludes the dissertation by summarizing the contributions of the thesis, and indicating potentially interesting areas for future research.

The problem faced in the thesis is the study of methods and tools for the improvement of quality of Web applications. To achieve this aim, we have taken an approach common in the **reverse engineering** community – i.e., we start from the assumption that a Web application already exists, and we have investigated techniques to support its analysis, testing and re-structuring. In other words we have concentrated our effort on the evolution phase and not on design – the latter being typical of the forward engineering approach.

The first step toward analysis, testing and re-structuring has been the definition and the extraction of a model representing the various entities involved in Web applications, and their mutual relationships. The analysis, testing and re-structuring techniques proposed here and based on the extracted model have been developed and evolved from the software engineering toolkit used for traditional software systems; such techniques, have been applied to well identified problems related to the quality of Web applications. This kind of adaptive development and evolution is not straightforward: it has often involved radical and creative rethinking of models, techniques and already known algorithms. It has also been corroborated by a series of empirical studies carried out on selected corpora of "real world" Web applications. A necessary component of this experimental work has consisted in the development of a prototype toolkit implementing the techniques investigated.

11.1 Main contributions

Main contributions of the thesis extend along several lines:

- **Modeling and model extraction of dynamic Web applications**. A major contribution of the thesis is the Web application model: the basis for understanding, analysing, testing and re-structuring Web applications. A Web application model able to manage high dynamism has been defined and an algorithm able to extract the model was implemented. The central problem in these tasks was the dynamic generation of the HTML code on the server side. In presence of dynamic generation of HTML the internal organization (links, forms, frames etc.) of the pages is not fixed, being produced at run-time by server side programs. Static analysis is therefore insufficient to recover the model of a dynamic Web application. This is due to the fact that the problem of determining the HTML code produced by a server program is related to the problem of determining whether a given execution path is feasible, which is a well known undecidable problem.

  The solution proposed resorts to dynamic analysis techniques. The model has been obtained by statically analyzing the HTML code that is dynamically generated by the server programs. From a conceptual point of view the most difficult task was handling the case in which the same server program behaves in different ways, depending on the interaction state. To tackle this situation we found it convenient to classify server programs into two categories: server programs with state-independent behavior and server programs with state-dependent behavior. From the implementation point of view the most difficult task was to realize and apply, in the model construction, the operation of *page merging*.

- **Static slicing**. To the best of our knowledge, static Web application slicing has been never considered in the literature before our proposal. Our work mainly consisted in extending the notion of slicing to Web applications, and in describing the dependences that have to be considered. Web application slicing has built upon a model that is a novel extension of the program representation used with traditional software – the System Dependence Graph (SDG). Another interesting contribution of the thesis was showing (by exhibiting an example) that the problem of statically building the SDG for Web applications is in general undecidable (see subsection 5.2.3). This is another effect of the dynamic generation of the code. Two different solutions have been proposed to limit the problem: dynamic slicing and creating approximations of SDG by means transformations [TR03b].

- **Structural testing**. Many types of Web application validation techniques have been proposed in the literature and many tools have also been developed. Most of these techniques and tools, however, do not directly address validation of functional requirements. The tools that focus on functional requirements primarily provide infrastructure to support capture-replay only. While the first approach of structural testing was [LKHH01], the first that builds the Web model in (semi)automatic way from inspection of code and handles dynamism was that presented here. In [LKHH01]

the problems connected with dynamism are not considered while **TestWeb**, our research tool, copes with it. A major problem in testing was to devise a Web application model adequate to support Web application testing when the behavior of the server program depends on the interaction state. The idea that solved the problem was to unfold the server program and the dynamic pages exhibiting different behaviors into actually different entities. Other authors have expanded on our technique [EKR03], by recovering automatically the inputs from a (modified) log file.

- **Inter-page transforms**. A variety of transformation tools (freeware or commercial) of Web applications have been developed and are available on the net. The novelty of our approach is that we consider also high level transformations, such as design re-structuring, which involve changes affecting the entire organization of a site. In contrast to this, available tools and proposed transformations work at a purely syntactic and intra-page level. Identifying and getting familiar with a toolkit suited to the problem was a relevant investment. In fact, toolkits like DMS (our choice) are the exception rather than the rule in today's code transformation tools panorama. DMS indeed provides a parser generator, a rule specification language and a rewriting engine. Moreover, DMS gives the possibility of implementing transformations procedurally, supports mass-parsing, dialects creation and allows users to work with more than one languages in the same session. Learning to use the toolkit and implementing our design re-structuring ideas was the second step. Writing the HTML grammar took a considerable effort, because to implement the HTML language as defined by its DTD [Gro99] turned out insufficient. In fact, (mainly for commercial reasons) browsers tend to use a super-set of the standard HTML and tolerate non-compliant dialects of HTML. For re-structuring real Web applications it was necessary to implement (in DMS) an HTML grammar compliant with the DTD, and then to define appropriate dialects to handle extra tags and browser-accepted syntax errors (e.g., tags matching).

- **Multilingual Website re-structuring**.

  The idea of resolving inconsistencies such as missing pages, tags, links or translations not correct in a multilingual Web site is new. We have proposed a set of techniques and algorithms that can be used to support restructuring of multilingual Web sites so as to align the tags structure and the information provided in different languages. The target representation of this restructuring process is an extension of XHTML, named MLHTML, where the suffix ML stands for MultiLingual. A valuable property of MLHTML is that it allows to centralize the language dependent variants of a page in a single representation, where shared parts (usually tags) are not duplicated. This property is very important during the evolution of a Web site because it facilitates a consistent propagation of changes to all site versions in different languages. The problem of language identification was solved easily by resorting to well known results

from the field of Natural Language Processing (NLP). It was more difficult to solve the *translation pair problem*, i.e., deciding whether two texts are one the translation of the other or not. After a first tentative solution , based on the comparison of the number of characters [Res99] the solution adopted was the *Translation Correspondence Index* (see subsection 9.2.3). The computation of TCI requires large repositories of linguistic knowledge, such as lexicons, grammars and knowledge bases (e.g., WordNet). This part of the work was done in collaboration with the NLP research group at ITC-irst.

11.2 Future work

We have proposed analyses, testing approaches and a set of transforms suited to Web applications. Many interesting developments and extensions of the techniques and tools proposed are possible, however. Let us briefly outline those we regard as the most promising.

- **Pattern extraction**. The problem is that of identifying, in the structure of the Web application, certain recurrent, meaningful organizational patterns. Since patterns are in general represented by graphs, the algorithms employed for matching are typically those used to detect the isomorphism of subgraphs with respect to a set of model graphs. A general approach to this problem can be found in [MB98a]. In this direction, we have already done some preliminary work, and we look forward to merge this approach with our ideas on pattern matching.

  Another interesting direction of the research is to extend the concept of pattern, now limited to the shape of the graph, to e-commerce schema – thereby giving more emphasis to the functional units of the application. From the Web application understanding point of view it could be useful to be able to extract some e-commerce patterns automatically or semi-automatically.

- **Slicing**. Even if the problem of building the SDG statically (in general, undecidable) has been already considered in [TR03b], more work is necessary in order to have a Web slicer able to cover as many real Web applications as possible (given the nature of the problem, a portion of them shall ever be out of reach of any automated procedure). Moreover, additional empirical studies are necessary to assess the real usefulness of this technique.

- **Statistical testing**. A critical review of the results obtained on the case study (section 7.5) highlights some directions for our future research. A test case is associated to a path in the Web application model. In the Web application we considered for testing (ITC-publications), the ability of a test case to reveal a defect depended only

and exclusively on the presence (absence) of a given edge of the model in the path, and on the input values inserted into the source page of such an edge. No regard was paid either to the path length and to the input previously inserted values. In such cases, it is possible to split the given Web application into a set of smaller sub-applications, consisting of three pages each: a page with a form to be filled in for the search (with the form varying according to the search criterion), the dynamic page with the search result, and the dynamic page produced by selecting an entry from the search result.

Testing the sub-applications is of course much less expensive than testing the whole application (full coverage of all paths is easily achieved for the sub-applications). The presence of a very local state of a Web application is, in our opinion, a quite general feature of several existing applications. This is partially explained by the nature of the interactions mediated by the Web, whose form typically consists of an input submission followed by a resulting page, with the possibility to interrupt the navigation at any moment. Moreover, ensuring the expected behavior in case of long chains of dependences is a difficult task, due to the (typically) high connectivity of the Web application structures (see 'Discussion' in the section 7.5). Identifying independent portions of a Web application that can be tested separately seems to be an important issue, and will be the subject of our future work.

Another aspect where the present work can be improved is the usage model constructed for a Web application. Currently, we are not modeling the statistics of the input values inserted by users into forms, in case of submission via `POST`, since such values are not recorded in the access log. However, the results obtained on our case study indicate that an accurate estimate of the reliability can be obtained only if the usage model includes the statistics of the input values. In order to obtain them, it is necessary to develop an instrumentation tool to automatically log the inputs submitted by the user when requesting the generation of a dynamic page. This is also a topic of our future research.

- **Transformations**. In this direction our future work will be devoted to investigating other design restructurings, with the purpose of supporting a high level reorganization of a Web site, similarly to refactoring in the Object Oriented domain. Automation and formalization of design-level transforms allow reasoning at a higher abstraction level than the single HTML page. Moreover, improvements at this level are expected to impact positively on both the usability and the evolvability of the site as a whole. Examples of such transforms include the migration toward a more dynamic site, presenting most information extracted at run-time from databases, and representation of the data in a format independent from the presentation, and can thus be handled by content management systems.

- **Multilingual consistency**. Future work will be devoted to trying to apply the

approach proposed to dynamic sites, for which code analysis techniques, such as program slicing, could help identify the language dependent portions of the script code that generate the page content. The target representation will still be MLHTML.

Many are the new topics that appear to offer other interesting research opportunities.

- **Web application understanding: producing abstract views.** The major problem in Web application understanding is that the views available, based on pages and hyperlinks and recovered with spiders or other tools, are too complex for humans to manage. The automatic identification of conceptual clusters of Web pages, based both on the structure of the site (page structure and connectivity) and on its content, may help build abstract and more compact views. Clustering algorithms can be used for this aim. The approach of connectivity clustering has been already investigated in [LFC+02] while clustering based on contents has not been considered so far. A possible solution for clustering Web pages based on contents can consist in considering keywords associated to the content of pages. Pages can then be clustered together as they share a large number of keywords. Another interesting research line in this context, though more methodological in spirit, is working on evaluation methods for Web application clustering.

- **Migrating static Web site toward dynamic Web application.** Re-structuring techniques could also be employed to migrate a static Web site toward a dynamic Web application. The idea is to migrate the contents of similar structural static Web pages (for example representing product-cards) into a database and the common HTML structure to a template. As template and database are generated, all static pages migrated to the database can be replaced by a server side script that fill the template and generates them at run time.

- **Automatic generation of form input.** This technique could be used in the production of test cases, and in the browsing of the so called "hidden web". Instead of manually inserting inputs in test cases, the inputs could be generated automatically from an "expert" algorithm. The aim of this approach shares many similarities with the one in [EKR03], where inputs for testing purposes are recovered automatically from a modified log file. For building the "expert" algorithm, that fills forms, we have to resort to machine learning techniques.

Chapter 12

Appendices

12.1 Flow analysis framework

Flow analysis of a software program is based on the notion of **Control flow graph (CFG)**. The Control flow graph G of a program P is a directed graph (N, E) where:

- Nodes set N is the set of statements of P.
- Edges set E is the set of pairs of statements (n, m) such that m is possibly executed just after n.

Figure 12.1 show the CFG of the following C program.

```
main()
{
1   scanf("%d %d", &a, &x);
2   if (a == 3)
3      x = a;
4   else if (a == 4)
5           a++;
6        else while (x)
7                x--;
8   printf("%d %d", a, x);
}
```

The **flow analysis framework** is a general description of a procedure that can be used to determine properties that hold for a given program by propagating proper information

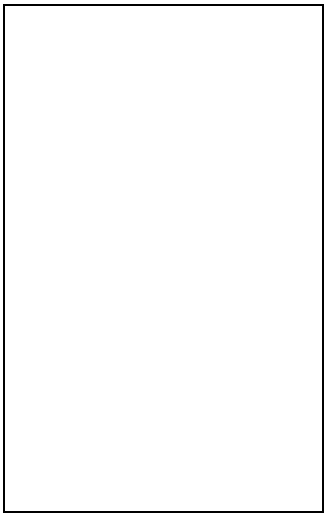

Figure 12.1: *Example of control flow graph.*

flows inside the control flow graph. These information flows are altered during propagation according to the computation performed by each program statement.

A flow analysis framework consists of:

- V, the set of *values* that can be propagated in the graph. Elements of V will be assigned to IN[n] and OUT[n], for each node n of the control flow graph.
- F, the set of the transfer functions. Every node n is associated with a transfer function f_n: $V \rightarrow V$. Such a function represents the computation performed by node n: OUT[n] = f_n(IN[n]).
- The confluence operator, $\bigoplus$, used to join values coming from the OUT sets of the predecessors (or successors), into the IN set of the current node: IN[n] = $\bigoplus_{p \in pred(n)}$ OUT[p].
- The direction of propagation (*forward* or *backward*), which determines whether information is joined from predecessors or successors.

A simple algorithm that can be used to compute flow analysis is showed in Figure 12.2. This algorithm propagates flow information inside the CFG until the fix-point is reached. The kind of flow information to be propagated depends on the purpose of the analysis being performed. The confluence operator exploited at line 8 to collect outgoing information from predecessor nodes is also dependent on the analysis, and is typically either the intersection or the union. After initializing the input and output sets of each node (IN_n and OUT_n,

Figure 12.2: *Pseudo-code of the flow analysis algorithm.*

lines 1-3) with the initial flow information, propagation is achieved inside a fix-point loop (line 5) by subtracting the destroyed information ($KILL_n$ set) and adding the generated information (GEN_n set) to the incoming information (line 10) for each node n in the graph.

Examples of static analysis, used in particular for code optimization [ASU85], which specializes the algorithm in Figure 12.2 are: dominators, reaching definitions, reachable uses, available expressions and copy-constant propagation.

12.2 Program slicing

Program slicing is a technique for restricting the behavior of a program to some specified subset of interest. A **static** slice S(x, n) of a program P on variable x at statement n yields the portion of a program that contributed to the value of x just before statement n is executed. A program slice has the property of being an executable program. The pair (x, n) is called *slicing criterion*.

Given a program P, a static slice $S(x, n)$ can be computed as the transitive closure of the data dependences[1] of P (DU) at n on x, and of the control dependences[2] of P (CD) on n:

[1] a data dependence holds between nodes n and m on variable x if n defines x, m uses x and a path exists in the control flow graph from n to m along which x is not redefined.

[2] a node m holds a control dependence on n if m is a conditional or loop statement and the execution of n directly depends on the truth value of the predicate.

$S(x,n) \leftarrow \{n' : (n',n,x) \in DU \vee (n',n) \in CD\}$
while $S(x,n)$ increases
$S(x,n) \leftarrow S(x,n) \bigcup \{n'' : \exists n' \in S(x,n), \exists y \in V : (n'',n',y) \in DU$
$\vee \ \exists n' \in S(x,n) : (n'',n') \in CD\}$
end while

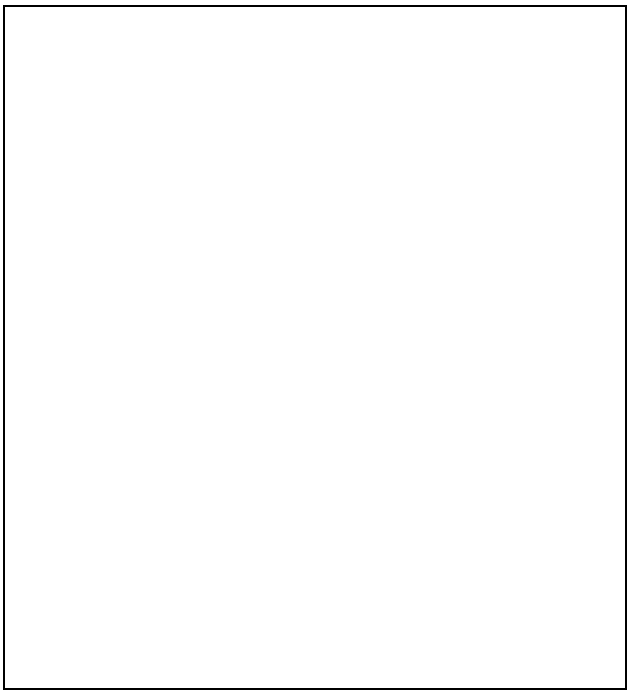

Figure 12.3: *Program (to be sliced) WordCount.*

Figures 12.4, 12.5 and 12.6 illustrate slicing on the C program wordCount[3] of figure 12.3.

An alternative mode to compute a slice of a program P is construct a Program Dependence Graph (PDG) from P, slice the dependence graph and obtain a sliced program from the sliced graph. A PDG is a graph where nodes are program statements and edges are either control or data dependences between statements. Data dependence edges can be labeled with the variable on which the dependence holds, while control dependence edges can be labeled with the truth value which determines the execution of the target node. A fictitious *entry* node is connected via (true) control dependences to each node that is not controlled by any other different node. Slices can be easily obtained from the SDG by starting from a

[3]a simplified version of the Unix utility **wc** that counts the number of characters, words and lines in a text file.

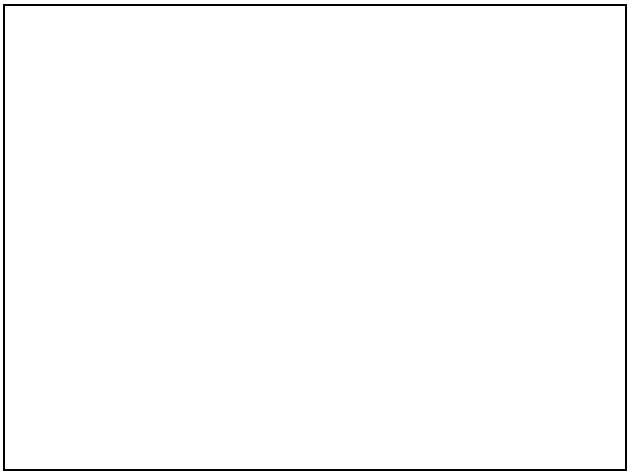

Figure 12.4: *Slice on (nw, 17): word counter.*

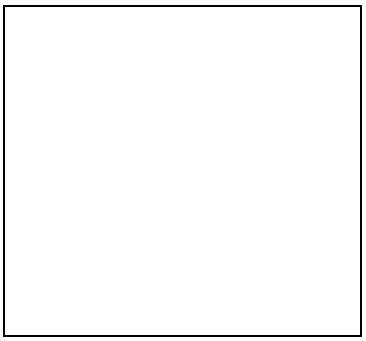

Figure 12.5: *Slice on (nc, 18): character counter.*

node of interest an traversing backward all dependences terminating at the selected node, until the transitive closure is reached.

Slice computation on the PDG is straightforward:

$S(x, n) = \{m \in P | \exists p =< m, ..., n >$ in PDG $\}$ with USE(n)[4] $= \{x\}$.

Figure 12.7 shows the PDG of the program wordCount (figure 12.3). In boldface is shown the slice at line 18 on variable *nc* (Slice on (nc, 18): character counter).

A **dynamic slice** differs from a static slice in that it makes use of information about a particular execution of a program. A dynamic slice contains "all statements that actually affect the value of a variable at a program point for a particular execution of the program". Dynamic slices are computed with respect to an *execution trace* i.e. the list of statements

[4]USE(n) is the set of variable used in the statement n. For example USE$(x = y + z)$=$\{y, z\}$ and USE(if $x < y$ then $z = 0$)=$\{x, y\}$

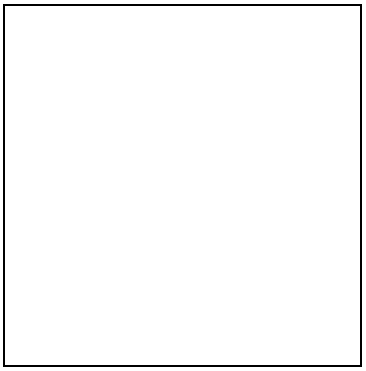

Figure 12.6: *Slice on (nl,16): line counter.*

(labeled by an occurrence counter) as the program executes.

Given a program P, a dynamic slice $S_d(x, n, I)$ (x is the variable, n is the statement and I is the input) can be computed as the transitive closure of the dynamic data dependences[5] of P (DU_p) at n on x, of the dynamic control dependences[6] of P (CD_p) on n and of the identity relationship[7](ID) .

12.3 Program transformation

Program transformation[8] is the act of changing one program into another. A **transform** is a (arbitrary) map between program representations while a transformation is the application of a transform. The language in which the program being transformed and the resulting program are written are called the source and target languages, respectively.

Program transformation is used in many areas of software engineering, including compiler construction, software visualization, documentation generation, and automatic software renovation. In all these applications we can distinguish two main scenarios, i.e., ones where the source and target language are different (*translations*) from scenarios where they are the same (*rephrasings*).

In a translation scenario a program is transformed from a source language into a program

[5]a dynamic data dependence holds between nodes n and m on variable x if n defines x, m uses x, n appears before m in the execution trace T and no intermediate node in T defines x.

[6]node n holds a dynamic control dependences on node m if there is a static control dependences between n and m, and if there is a path in the execution trace T from n to m such that n holds a transitive dynamic control dependence on all intermediate nodes in the path.

[7]node n^i is identical to node m^j if n and m are two successive occurrences of the same node in the execution trace T

[8]http://www.program-transformation.org

Figure 12.7: *PDG of the program WordCount. Slice(nc, 18) is shown in bold.*

in a different target language. Usually translations aim at preserving the semantics of a program. Examples of translation scenarios are synthesis (for example, compilation or program refinement) and migration i.e. translation from one language to another (e.g., porting a Pascal program to C).

Rephrasings are transformations that transform a program into a different program in the same language, i.e., source and target language are the same. In general, rephrasings try to say the same thing in different words thereby aiming at improving some aspect of the program, which entails that they change the semantics of the program. The main sub-scenarios of rephrasing are normalization (normalization reduces a program to a program in a sub-language, with the purpose of decreasing its syntactic complexity), optimization (examples of optimization are inlining, constant propagation, common-subexpression elimination, and dead code elimination) and refactoring (transformation that improves the design of a program by restructuring it such that it becomes easier to understand and maintain).

Transforms
A transform can be implemented in many ways. For example procedurally (an algorithm that implements program transformation) or by means rewrite rules.

In the first case the algorithm manipulate directly the AST of the program changing some nodes. Usually the languages for implementing transforms in procedural way providing some features for navigating in the AST.

A rewrite rule is a rule of the form

$lhs_{pattern} - - > rhs_{pattern}$ if *cond*

where $lhs_{pattern}$ and $rhs_{pattern}$ are term patterns and the condition is optional. It declares that any instance of $lhs_{pattern}$ rewrites to the corresponding instance of $rhs_{pattern}$ if the condition is true. The application of a rule to a term succeeds if the term matches (pattern matching) with the $lhs_{pattern}$ pattern. The result is the instantiation of the $rhs_{pattern}$ pattern.

For example, if we have the term:

$3 + 0$

and applying to it the rewrite rule:

$x + 0 - - > x$

the result is 3 (the pattern variable x match the number 3).

Rewrite rules define basic transformation steps while **strategies** combine set of rules into full transformations. Strategies are programs that determine the order and the place in the term for application of rules. Examples of simple strategies are: leftmost bottom-up rewriting and rightmost top-down rewriting.

12.4 Concept analysis

Concept analysis is a mathematical technique that allows determining maximal grouping of objects sharing common attributes. This technique was laid by Birkhoff in 1940 and introduced to software engineering recently [SR97, ST00, Ton01].

Concept analysis is based on a binary relation R between a set of objects O and a set of attributes A. A *context* is a triple $C = (O, A, R)$.

Let $X \subseteq O$ and $Y \subseteq A$. The set of **common attributes** σ is defined as:

$\sigma(X) = \{a \in A | \forall o \in X : (o, a) \in R\}$

while the set of **common objects** τ is defined as:

$\tau(Y) = \{o \in O | \forall a \in Y : (o, a) \in R\}$

If we consider the binary relation shown in table 12.1 (the example stems from Lindig and Snelting [LS97]), the following equations hold:

$\sigma(\{o_1\}) = \{a_1, a_2\}$

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
o_1	√	√						
o_2			√	√	√			
o_3			√	√		√	√	√
o_4			√	√	√	√	√	√

Table 12.1: *Example relation*

$\tau(\{a_7, a_8\}) = \{o_3, o_4\}$

A pair (X, Y) is called a **concept** if all objects share all attributes, i.e. :

$Y = \sigma(X)$ and $X = \tau(Y)$

X is called the **extent** of the concept while Y is the **intent**.

For example table 12.2 contains the concepts for the relation in table 12.1.

C_1	$(\{o_1, o_2, o_3, o_4\}, \emptyset)$
C_2	$(\{o_2, o_3, o_4\}, \{a_3, a_4\})$
C_3	$(\{o_1\}, \{a_1, a_2\})$
C_4	$(\{o_2, o_4\}, \{a_3, a_4, a_5\})$
C_5	$(\{o_3, o_4\}, \{a_3, a_4, a_6, a_7, a_8\})$
C_6	$(\{o_4\}, \{a_3, a_4, a_5, a_6, a_7, a_8\})$
C_7	$(\emptyset, \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\})$

Table 12.2: *Example of concepts for table 12.1*

The set of all concepts of a given context forms a partial order via:

$(O_1, A_1) \leq (O_2, A_2) \iff O_1 \subseteq O_2$

or equivalently with

$(O_1, A_1) \leq (O_2, A_2) \iff A_1 \supseteq A_2$.

The set L of all concepts of a given context and the partial order defined above form a complete lattice, called **concept lattice**.

The **infimum** of two concepts in this lattice is computed intersecting their extents, while the **supremum** is determined by intersecting the intents. The most general concept is called the top element and is denoted by $\top$, while the bottom element is denoted by $\bot$.

Figure 12.8 shows the concept lattice for table 12.2.

Figure 12.8: *Concept lattice for table 12.2.*

A free software that, given a binary relation objects-attributes, computes all concepts and their lattice structure is available on the net [9]. This program, written by C. Lindig, is in portable C code and is used by many people working on concept analysis.

12.5 Software testing techniques

The goal of software testing is to discover the presence of failures in a target program. The most intuitive mode to discover failures in a program is to consider it as a black box and test it for all possible input cases to see if it will produce the correct outputs (*exhaustive test*). Unfortunately exhaustive test is impractical, because usually the number of test cases involved is too large.

A randomly selected set of test cases is statistically insignificant, but an accurate selection of test cases based on specification (called **functional testing** or **black box testing**) or on the structure of the program (called **structural testing** or **white box testing**) can lead to a high probability of failure detection.

In **functional testing**, given a program P with intended function F (the specification) and input domain D, a sequence of input values is drawn from D and applied to P. The output is compared with the expected behavior indicated by F. Any deviation from the intended function is designated as a failure.

[9] http://www.eecs.harvard.edu/~lindig/software/

Structural testing requires complete knowledge of the program's structure and it is based on the use of the **Control flow graph (CFG)** (see subsection 12.1). It indicates a family of testing techniques based on the selection of a set of test paths through the program (CFG). A path through a program (CFG) is a sequence of instructions or statements (nodes) that starts at the entry point and ends to the exit point. The aim is reaching a given level of *coverage*, for example, selecting enough paths so as to assure that every source statement has been executed at least once.

There are different coverage criteria for structural testing:

Statement coverage: every statement in the program is executed at least once in some test case.

Branch coverage: for every decision point in the program, each outgoing branch is chosen at least once in some test case.

Condition coverage: every boolean expression appearing in a decision is made both true and false in some test case.

Path coverage: every path in the control flow graph is traversed at least once in some test case.

In the following C program, **statement coverage**, but not **branch coverage** – branch (3, 7) is not covered, can be reached with this set of test cases $\{(4,1),(4,0)\}$. For reaching also **branch coverage** we have to add a new test case, for example (0, 1).

```
main() {
1    scanf("%d", &a);
2    scanf("%d", &b);
3    if (a == 4) {
4       a = a - b;
5       if (a == 3)
6          a--;
     }
7    a--;
8    if (b)
9       while (a)
10         a--;
11   else a = 2;
12   printf("%d\n", a);
13   printf("%d\n", b);
```

Bibliography

[ACCL00] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Web site reengineering using RMM. In *Proc. of the International Workshop on Web Site Evolution*, pages 9–16, Zurich, Switzerland, March 2000.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1985.

[Ata99] Mikhail J. Atallah (editor). *Algorithms and Theory of Computation Handbook*. CRC Press, Boca Raton, Florida, USA, 1999.

[Bei90] B. Beizer. *Software Testing Techniques, 2nd edition*. International Thomson Computer Press, 1990.

[BG96] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[BK01] C. Boldyreff and R. Kewish. Reverse engineering to achieve maintainable WWW sites. In *Proc. of the 8th Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2001.

[BN96] M. Bichler and S. Nusser. Developing structured WWW-sites with W3DT. In *Proc. of WebNet*, San Francisco, California, USA, 1996.

[BP97] I. D. Baxter and C. W. Pidgeon. Software change through design maintenance. In *Proc. of the International Conference on Software Maintenance*, pages 250–259, Bari, Italy, October 1997.

[BR96] S. Bonhomme and C. Roisin. Interactively restructuring html documents. In *Proc. of the 5th International World Wide Web Conference (WWW5)*, Paris, France, 6-10 May 1996.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language – User Guide*. Addison-Wesley Publishing Company, Reading, MA, 1998.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the International Conference on Software Maintenance*, pages 368–377, Bethesda, Maryland, USA, November 1998.

[CCL98] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology*, 40:595–607, 1998.

[CDMS02] J.R. Cordy, T.R. Dean, A.J. Malton, and K.A. Schneider. Source transformation in software engineering using the txl transformation system. *Information and Software Technology*, 44(13):827–837, 2002.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*. Springer Verlag, May 1981. Lecture Notes in Computer Science No. 131.

[CF94] J. D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transaction on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[CFB00] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (webml): a modeling language for designing web sites. In *Proc. of the 7th International World Wide Web Conference (WWW7)*, May 2000.

[CG01] Dan Connolly and Mark Gaither. *W3C HTML Validator*. http://validator.w3.org/, last modified Sept 13, 2001.

[CH00] W. K. Chang and S. K. Hon. A systematic framework for ensuring link validity under web browsing environments. In *Proc. of the 13th International Software/Internet Quality Week*, San Francisco, California, USA, 2000.

[CI90] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[CK01] Wendy Chisholm and Len Kasday. *Evaluation, Repair and Transformation Tools for Web Content Accessibility*. http://www.w3.org/WAI/ER/existingtools, last modified 28/6/2001.

[CLM95] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 124–133, Opio(Nice), 1995.

[Con00] J. Conallen. *Building Web Applications with UML*. Addison-Wesley Publishing Company, Reading, MA, 2000.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, 1991.

[CT94] W. B. Cavnar and J. M. Trenkle. N-gram-based text categorization. In *Proceedings of Third Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, Las Vegas, CA, April 1994.

[CVJ99] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. *Web Content Accessibility Guidelines 1.0*. http://www.w3.org/TR/WCAG10/, W3C Recommendation 5 May 1999.

[CVJ00] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. *Techniques for Web Content Accessibility Guidelines 1.0*. http://www.w3.org/TR/WCAG10-TECHS/, W3C Note 6 November 2000.

[DFHH00] S. Danicic, C. Fox, M. Harman, and R. Hieros. Consit: A conditioned program slicer. In *Proc. of ICSM 2000, International Conference on Software Maintenance, San Jose, California, USA, 11-14 October*, pages 216–226, 2000.

[Eic99] D. Eichmann. Evolving an engineered web. In *Proc. of the International Workshop on Web Site Evolution*, Atlanta, GA, USA, October 1999.

[EKR03] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proc. of ICSE 2003, International Conference on Software Engineering, May 3 - 10, Hilton Portland, Oregon USA*, page (to appear), 2003.

[ESS92] S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft – a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[FFLS99] M. Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Verifying integrity constraints on web sites. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

[FHa02] P. Finnigan, R. C. Holt, and al. The software bookshelf. In *IBM Systems Journal*, pages 36(4):564–593, 1997, 2002.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[fS91] International Organization for Standardization. *Information Technology–Software Product Evaluation: Quality Characteristics and Guidelines for their Use, ISO/IEC IS 9126*. ISO, Geneva, 1991.

[GFKF01] P. Graunke, R.B. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the web. In *Proc. of the 16th International Conference on Automated Software Engineering (ASE 2001)*, Paris, France, 26-29 November 2001.

[GHS96] R. Gupta, M.J. Harrold, and M.L. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6(2):83–112, June 1996.

[GJR99] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: the use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance*, pages 99–108, Oxford, England, August-September 1999. IEEE Computer Society press.

[GKNV93] E.R. Gasner, E. Koutsofios, S. North, and Kiem-Phong Vo. A technique for drawing directed graphs. In *IEEE-TSE 1993*, March 1993.

[GL91] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[GM01] A. Ginige and S. Murugesan. Web engineering: An introduction. *IEEE Multimidia*, 8(1):14–18, April-June 2001.

[Gre95] Gregory Grefenstette. Comparing two language identification schemes. In *Proceedings of JADT 1995, 3rd International Conference on Statistical Analysis of Textual Data*, Rome, Italy, December 1995.

[Gro99] HTML Working Group. *Transitional Document Type Definition, HTML 4.01.* http://www.w3.org/TR/html4/sgml/loosedtd.html, W3C Recommendation 24 December 1999.

[GW96] B. Ganter and R. Wille. *Formal Concept Analysis.* Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[HD95] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5:143–162, September 1995.

[HD97] M. Harman and S. Danicic. Amorphous program slicing. In *Proc. of IWPC'97, International Workshop on Program Comprehension*, pages 70–79, Dearborn, Michigan, May 1997.

[HH02] A. E. Hassan and R.C. Holt. Architecture recovery of web applications. In *Proc. of International Conference on Software Engineering*, Orlando, Florida, USA, May 19-25 2002.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Proc. of the ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pages 35–46, June 1988.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transaction on Programming Languages and Systems*, pages 26–61, 1 1990.

[IKK97] T. Isakowitz, A. Kamis, and M. Koufar. Extending RMM: Russian dolls and hypertext. In *Proc. of HICSS-30*, 1997.

[JL01] Xiaoping Jia and Jordan Hongming Liu. Rigorous and automatic testing of web applications. In *Proc. of the DePaul CTI Research Symposium (CTIRS)*, Chicago, USA, November 2001.

[KFS93] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the International Conference on Software Maintenance*, pages 386–395, Montreal, Quebec, Canada, September 1993.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[KNE92] V. Kozaczynski, J. Q. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, Dec 1992.

[KT01] C. Kallepalli and J. Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, November 2001.

[LB93] J. R. Lyle and D. Binkley. Program slicing in the presence of pointers. In *Proceedings of the Third Software Engineering Research Forum*, pages 255–260, Orlando, Florida, November 1993.

[Leh80] M. M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[LFC+02] G. A. Di Lucca, A. R. Fasolino, U. De Carlini, F. Pace, and P. Tramontana. Comprehending web applications by a clustering based approach. In *Proc. of the 10th International Workshop on Program Comprehension (IWPC)*, pages 261–270, Paris, France, June 2002. IEEE Computer Society.

[LFFC02] Giuseppe A. Di Lucca, Anna Rita Fasolino, Francesco Faralli, and Ugo De Carlini. Testing web applications. In *Proc. of the International Conference on Software Maintenance (ICSM)*, Montreal, Canada, October 2002. IEEE Computer Society.

[LFP+02] G. A. Di Lucca, A.R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. Ware: a tool for the reverse engineering of web applications. In *Proc. of the 6th European Conference on Software Maintenance and Reengineering(CSMR2002)*, Budapest, Hungary, March 2002.

[LH96] L. Larsen and M.J. Harrold. Slicing object-oriented software. In *Proc. of the Int. Conf. on Software Engineering*, pages 495–505, Berlin, Germany, March 1996.

[LKHH01] C-H Liu, D. C. Kung, P. Hsia, and C-T Hsu. An object-based data flow testing approach for web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, April 2001.

[LPAC01] G. A. Di Lucca, M. Di Penta, G. Antoniol, and G. Casazza. An approach for reverse engineering of web-based application. In *Proc. of the 8th Working Conference on Reverse Engineering (WCRE)*, Stuttgart, Germany, October 2001.

[LPFG01] G. Di Lucca, M. Di Penta, A. R. Fasolino, and P. Granato. Clone analysis in the web era: an approach to identify cloned web pages. In *Seventh IEEE Workshop on Empirical Studies of Software Maintenance (WESS)*, 9 November 2001.

[LS97] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. of the 19th International Conference on Software Engineering*, pages 349–359, Boston, Massachussets, USA, May 1997.

[May96] Merlo E. MayrandJ., Leblanc C. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the International Conference on Software Maintenance*, pages 244–253. IEEE Computer Society, 1996.

[MB98a] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–503, May 1998.

[MB98b] Robert C. Miller and Krishna Bharat. Sphinx: A framework for creating personal, site-specific web-crawlers. In *Proc. of WWW7, Brisbane Australia*, April 1998.

[Mil98] Edward Miller. The web site quality challenge. - companion paper: "website testing". In *Proc. of QW'98, 11th Annual International Software Quality Week*, San Francisco, CA, USA, May 1998.

[Mus98] J. D. Musa. *Software Reliability Engineering.* McGraw-Hill, NY, 1998.

[Nes00] Scott Nesbit. *HTML Tidy: Keeping it clean.* on-line journal webreview, http://www.webreview.com/2000/06-16/webauthors/06-16-00-3.shtml, 2000.

[PE97] Mike Perkowitz and Oren Etzioni. Adaptive web sites: Automatically learning from user access patterns. In *In Proceedings of the Sixth International World Wide Web Conference(WWW6)*, April 1997.

[Pow98] T.A. Powell. *Web site engineering: Beyond Web Page Design.* Prentice-Hall, Englewood Cliffs, NJ, 1998.

[Pre00] R. S. Pressman. What a tangled web we weave. *IEEE Software*, 17(1):18–21, 2000.

[Rag98] Dave Raggett. *Clean up your Web pages with HTML TIDY.* http://www.w3.org/People/Raggett/tidy/, 1998.

[Res99] Philip Resnik. Mining the web for bilingual text. In *37th Annual Meeting of the Association for Computational Linguistics (ACL'99)*, College Park, Maryland, June 1999.

[RT00a] F. Ricca and P. Tonella. Visualization of web site history. In *Proc. of the International Workshop on Web Site Evolution*, pages 30–33, Zurich, Switzerland, 2000.

[RT00b] F. Ricca and P. Tonella. Web site analysis: Structure and evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 76–86, San Jose, California, USA, 2000.

[RT01a] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proc. of ICSE 2001, International Conference on Software Engineering, Toronto, Ontario, Canada, May 12-19*, pages 25–34, 2001.

[RT01b] F. Ricca and P. Tonella. Building a tool for the analysis and testing of web applications: Problems and solutions. In *Proc. of TACAS 2001, Tools and Algorithms for the Construction and Analysis of Systems, Genova, Italy, 2 - 6 April*, 2001.

[RT01c] F. Ricca and P. Tonella. Understanding and restructuring web sites with reweb. *IEEE MultiMedia*, 8(2):40–51, April-June 2001.

[RT01d] F. Ricca and P. Tonella. Web applications slicing. In *Proceedings of the International Conference on Software Maintenance*, pages 148–157, Firenze, Italy, 2001.

[RT02a] F. Ricca and P. Tonella. Testing processes of web applications. *Annals of Software engineering*, (14):93–114, 2002.

[RT02b] Filippo Ricca and Paolo Tonella. Construction of the system dependence graph for web application slicing. In *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 123–132, Montreal, Canada, October 2002. IEEE Computer Society.

[RTB01] F. Ricca, P. Tonella, and I. Baxter. Restructuring web applications via transformation rules. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 150–160, Firenze, Italy, 2001.

[RTB02] F. Ricca, P. Tonella, and I. Baxter. Web application transformations based on rewrite rules. *Information and Software Technology*, 44(13):811–825, 2002.

[SDMP01] E. Di Sciascio, F.M. Donini, M. Mongiello, and G. Piscitelli. Verifying integrity constraints on web-based systems using model checking. In *Atti Aica 200*, september 2001.

[SDMP02] E. Di Sciascio, F.M. Donini, M. Mongiello, and G. Piscitelli. Anweb: a system for automatic support to web application verification. In *In Proceedings of International Conference on Software engineering and Knoledge engineering (SEKE 2002)*, july 2002.

[SR97] M. Siff and T. Reps. Identifying modules via concept analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 170–179, Bari, Italy, Oct. 1997.

[ST00] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, May 2000.

[TAFM97] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. *Proc. of the Int. Conf. on Software Engineering*, pages 433–443, 1997.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[Tom86] M. Tomita. *Efficient Parsing for Natural Languages – A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.

[Ton01] P. Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, April 2001.

[TR02] Paolo Tonella and Filippo Ricca. Dynamic model extraction and statistical analysis of web applications. In *Proc. of the International Workshop on Web Site Evolution (WSE)*, pages 43–52, Montreal, Canada, October 2002. IEEE Computer Society.

[TR03a] Paolo Tonella and Filippo Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution*, (submitted), 2003.

[TR03b] Paolo Tonella and Filippo Ricca. Web application slicing in presence of dynamic code generation. *Journal of Automated Software Engineering*, (submitted), 2003.

[TRPG01] Paolo Tonella, Filippo Ricca, Emanuele Pianta, and Christian Girardi. Recovering traceability links in multilingual web sites. In *Proc. of WSE 2001, International Workshop on Web Site Evolution*, pages 14–21, Florence, Italy, November 2001.

[TRPG02] Paolo Tonella, Filippo Ricca, Emanuele Pianta, and Christian Girardi. Restructuring multilingual web sites. In *Proc. of the International Conference on Software Maintenance (ICSM 2002)*, pages 290–299, Montreal, Canada, October 2002. IEEE Computer Society Press.

[TRPG03] Paolo Tonella, Filippo Ricca, Emanuele Pianta, and Christian Girardi. Automatic support for the alignment of multilingual web sites. *Journal of Software Maintenance and Evolution*, (submitted), 2003.

[WBM99a] P. Warren, C. Boldyreff, and M. Munro. Characterising evolution in web sites: Some case studies. In *Proc. of the International Workshop on Web Site Evolution*, Atlanta, GA, USA, October 1999.

[WBM99b] P. Warren, C. Boldyreff, and M. Munro. The evolution of websites. In *Proc. of the International Workshop on Program Comprehension*, pages 178–185, Pittsburgh, PA, USA, May 1999.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[WT94] J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, October 1994.

[ZK99] Y. Zou and K. Kontogiannis. Enabling technologies for web-based legacy system integration. In *Proc. of the International Workshop on Web Site Evolution*, Atlanta, GA, USA, October 1999.