Université de Montréal

Formal Checking of Web Based Applications

Par

Doina Mirela Barburas

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Mémoire présenté a la Faculté des Études Supérieures

en vue de l'obtention du grade de

Maître ès Science (M. Sc.)

en Informatique

Juin, 2006

Université de Montréal

Faculté des Études Supérieures

Ce mémoire intitulé

Formal Checking of Web Based Applications

Par

Doina Mirela Barburas

A été évalué par un jury composé des personnes suivantes :

Jean - Yves Potvin
président – rapporteur

Houari Sahraoui
directeur de recherche

Alexandre Petrenko
codirecteur

Michel Boyer.
membre du jury

Mémoire accepté le : 13 octobre 2006

Résumé

La plupart des outils de test examinent seulement un ou quelques aspects spécifiques des applications Web. La vérification de la syntaxe des documents HTML, la confirmation de l'intégrité des liens d'un ensemble de documents HTML, l'examen des composants GUI inclus dans les browsers et la mesure de l'exécution de l'application en sont de bons exemples. Dans cette thèse de maîtrise, nous définissons des méthodes d'extraction d'un modèle de l'application Web à tester qui permet d'examiner, à l'aide d'un outil de vérification de modèle, si l'application possède certaines propriétés définies par l'utilisateur. Cette approche permet, par exemple, de vérifier la structure de navigation d'une application Web. Notre outil est un prototype qui permet ces vérifications en utilisant des méthodes formelles.

Elles peuvent être exécutées avec un vérificateur de model suivant la création d'un modèle « machine à états finis » (FSM) extrait à partir de l'application en question. Nous employons une méthodologie basée sur la théorie de machine à états finis, en particulier, la théorie du vérificateur de modèle. Développant un environnement de vérification pour des applications Web, nous réutilisons un vérificateur de modèle SPIN utilisé dans les laboratoires de recherche. La méthode est intégrée à un prototype qui automatise le processus d'extraction du modèle de machine à états finis à partir de l'application à vérifier et automatise également la transformation du modèle dans un langage reconnu par le vérificateur de model choisi.

Mots clés: vérificateur de model, application web, machine à états finis, vérification des liens

Abstract

Most Web application test tools test only one or some specific aspects of Web applications, such as verifying the syntax of HTML documents, confirming the hyperlink integrity of a set of HTML documents, testing the GUI components embedded in the browsers, or measuring the performance of the Web application.
In this work, methods are defined for extracting a model from a given web application so that a commercial or academic model-checking tool can be used to verify whether the application possesses certain properties defined by the user. This approach allows us to check, for example, the link structure in hyperdocuments or more generally the navigation structure of a web application. Our prototype tool checks the properties of a web application using formal methods. This is achieved with an existing model-checker once a state machine model is extracted from the application in question.

We use a methodology based on finite state machine theory, in particular, the theory of model checking, and test derivation from state machine models. Then developing a verification environment for web applications, we reuse an academic model-checker and testing tool called SPIN. The method is implemented in a prototype tool that automates the process of extracting a state machine model of the application to be verified and the model transformation into the input recognized by the off-the-shelf model-checker chosen.

Keywords: model checker, web application, finite state machine, link verification

Table of contents

1 Introduction 1

2 Motivation 8

2.1 Methodology Overview 10

2.2 Detailed Methodology 11

3 Model Checking 16

3.1 The Process of Model Checking 16

3.1.1 Modeling 17

3.1.2 Specifications 17

3.1.3 Verification 17

3.2 The model 17

3.3 Formalizing browsing properties 18

3.4 SPIN Model Checker 19

3.4.1 PROMELA (PROcess MEta LAnguage) 21

3.5 Formal Properties (LTL) 22

4 State of the Art 24

4.1 Stotts approach 24

4.1.1 Links-Only Document Behaviour 25

4.1.2 Temporal logic and dynamic properties of systems 28

4.2 Stotts model 29

4.3 Formal modeling in WWW multimedia 32

4.3.1 Interactive temporal behaviour 32

4.3.2 The abstract temporal synchronization control architecture 35

4.3.3 The EFSM model 37

4.3.4 Link synchronization 38
4.4 Other Related work 42
4.5 Our approach 44
5 Formal Checking of Web Based Applications 46
5.1.1 Communication interception 46
5.1.2 Java HTTP Proxy Server 46
5.2 The Model 51
5.2.1 FSM in XML 51
5.2.2 DTD 55
5.2.3 FSM in Promela 62
5.2.4 Formal Properties 62
5.2.4.1 Browsing Properties 63
5.2.4.2 Connectivity Properties 67
6 Web Model Extractor/Manipulator 68
6.1 Functionalities 68
6.2 GUI 71
7 Case Study 75
7.1 The "Beethoven" Radio Station Web Site 75
7.2 Properties 77
7.3 Formal Model 78
7.4 Formal Properties (LTL) 85
7.4.1 Property 1 86
7.4.3 Property 3 89
7.4.4 Property 4 90
7.5 Limitations of the Extractor/Manipulator Tool 91
8 Conclusions 93
8.1 Summary of the Results 93
8.2 Future Work 93
References 95

Table of Figures

Figure 1 The growth of Internet from January 1994 to January 2004 1
Figure 2 Framework 6
Figure 3 Framework detailed 10
Figure 4 Fragment from the log file 12
Figure 5 Fragment of the FSM Model 13
Figure 6 SPIN Model Checker 20
Figure 7 Traditional view of the hypertext document 26
Figure 8 Automaton view of the hyperdocument 27
Figure 9 FSM encoding of the links-automaton for Trellis document 30
Figure 10 A distance course in several linked web pages 33
Figure 11 Multimedia presentation schedules for page P0 34
Figure 12 The abstract temporal synchronization control architecture 37
Figure 13 The media server EFSM 39
Figure 14 Added states and transitions in Pager EFSM for processing 40
Figure 15 HTTP Request 48
Figure 16 HTTP Response 49
Figure 17 HTTP Request/Response for an Image 50
Figure 18 A Simple Site Graph 54
Figure 19 The DTD file for the implemented model in XML 55
Figure 20 The XML Partial Model 59
Figure 21 XML to Promela with Web Model Extractor/Manipulator 62
Figure 22 CRIM's home page 63
Figure 23 CRIM's Services page 64
Figure 24 One page web site 64
Figure 25 Enable/disable objects 65
Figure 26 CRIM's Members page 66
Figure 27 Class Diagram 69
Figure 28 Processing of the proxy server log file 70

Figure 29 Web Model Extractor/Manipulator Window 71
Figure 30 Option Window. 73
Figure 31 Show the file button. 73
Figure 32 Build the model button 74
Figure 33 Home Page from www.beethoven.com site. 76
Figure 34 Explore page from www.beethoven.com site 78
Figure 35 Web Model Extractor/Manipulator 79
Figure 36 Option Window 80
Figure 37 The model – XML Format 81
Figure 38 Finite State Machine in Promela 83
Figure 39 The Model – Graphical Representation 85
Figure 40 Linear Time Temporal Logic Formulae. 86
Figure 41 The Never Claim for LTL: [] occur 87
Figure 42 Guided Simulation Output – Property 1 88
Figure 43 Guided Simulation Output – Property 4 91
Figure 44 Scalability 92

Abbreviation list

DOM	Document Object Model
DTD	Document Type Definition
XML	Extensible Markup Language
FSM	Finite State Machine
PROMELA	PRocess MEta LAnguage
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
LTL	Linear Temporal Logic
URL	Uniform Resource Location
WWW	World Wide Web

Acknowledgements

I would like to express my honest appreciation to my supervisor, Professor Alexandre Petrenko for his constant guidance, constructive criticism and encouragement during all the period of my internship at CRIM and afterwards.

I also want to thank my thesis director, Professor Houari Sahraoui for his valuable courses during my studies at Université de Montréal and his constant encouragements and valuable advice.

Special thanks to May Haidar and Dr. Serge Boroday for their collaboration. I benefited greatly from formal and informal discussion with them.

I thank CRIM (Centre de recherche informatique de Montréal) who provided a wonderful research environment and financial support for this thesis.

I wish to thank the "Département d'informatique et recherche opérationnelle", Université de Montréal for the graduate courses and the research environment, and Mariette Paradis for easing the procedure of dealing with Department.

I would like to express my honest appreciation to my fiancé Charles Younghusband for his support and constant efforts to improve my English.

Chapter 1

1 Introduction

Since its inception in 1991 the World Wide Web (WWW) has experienced a rapid growth, and become the dominant Internet user application. As shown in Figure 1 below, drawn from the Internet Domain Survey [URL1], an increase in the number of hosts was close to one hundred percent per year and occasionally exceeding that number. Today the rate has tapered off to approximately 33% annually.

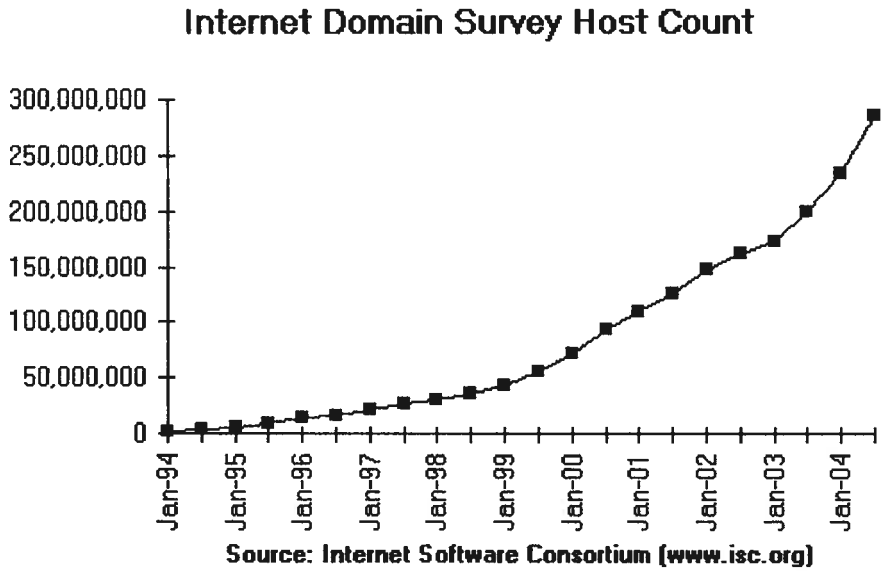


Figure 1 The growth of Internet from January 1994 to January 2004

The increase in the number of personal computers, combined with the rapid growth of digital networking technologies, has dramatically increased the accessibility of the Internet and the World Wide Web. This new way of accessing information made the Internet more attractive to companies, which in turn stimulates its growth by creating more elaborate and attractive web sites appealing to both personal and business users. Thanks to the Web, companies are now able to reach almost all possible target groups and it has become an excellent advertising medium and interactive channel for new businesses or adapted business models. New products and services, in turn, drive new features to optimize the Web based communications, adding to the complexity of web sites while not necessarily improving the robustness and quality.

Presently, the connection between different Internet resources is almost exclusively done by means of hypertext links, in which case the resources are called standard resources: file-based, static and read-only. Up to now these standard resources have been used very successfully within the current Internet environment. However, they have a number of shortcomings, which, in the light of the immense growth rates experienced, could make the use of the hypertext within the future Internet environment problematic. The problem of broken links between Internet resources, caused by the lack of referential integrity within the current hypertext implementation, is a good example of such a shortcoming. This problem will scale exponentially with the ongoing growth in the number of resources.

A classification by Thomas A. Powell [15] based on the degree of interactivity offered by a web site divides web sites into five categories:

- Static Web Sites

They are the most basic form of Web site, such as presentation of HTML-documents. No interactivity is offered except for the choice of pages by clicking links.

- Static with Form-Based Interactivity

A web site containing forms is used to collect information from the user, including comments or requests for information, but the content they deliver is static. The

primary purpose of this category is document delivery, implementing data collection mechanisms.

- Sites with Dynamic Data Access

The web site is used as a front-end for accessing a database. Users can search a catalogue or perform queries on the contents of a database, through a web page, and the results are displayed in HTML format.

- Dynamically Generated Sites

These sites are generally based on static content, but which were generated in a personalized fashion to suit the needs of individual users.

- Web-Based Software Applications

These are Web sites that are used as an interface for software applications such as inventory-tracking programs or sales force automation tools. They often have more in common with traditional client/server applications than with static web sites.

This classification arose from the need for methodology during the development of web sites, but is useful also for the testing process, as different types of web sites require more or less extensive testing. For instance, the requirements for a static web site are relatively few: the only things that need to be checked are that the information is correct and up-to-date, that the source HTML is correct and that the load capacity of the server is large enough, i.e. that it can handle a sufficiently large number of visitors at the same time. At the other end of the spectrum, the demands of Web-Based Software Applications are much higher, security and code integrity being merely two of several relevant issues.

For the purpose of this thesis, we choose to define web-based application as any HTTP application available on the Internet.

There are an estimated 1 billion pages accessible on the World Wide Web, with 1.5 million pages being added daily. These resources have been created by large teams of human editors which make the formatting of the content impossible to measure.

However other web site properties can be analyzed and evaluated, such as the navigation properties.

In this thesis we focus on the formal modeling of interactive temporal behaviour of the hyperlinks. This research is an exploratory study and the first step of an on going PhD thesis [7] focusing on test and quality assurance of web-based applications. Reference [7] presents an approach for modeling an existing web application using communicating finite automata model based on the user-defined properties to be validated. A method to automate the extraction from a recorded browsing session of such a model is generated. The obtained model is used to verify properties with a model checker.

In this thesis, our goal is to find a method to extract the information from a browsing session from which we extract a finite state model. The finite state model should be in a standard format in order to be easily transformed into a model accepted by a model checker that was determined during the research period. The next step was to define the properties that we wanted to test and to formalize them to be accepted by the chosen model-checker.

We developed a prototype tool which extracts a recorded browsing session into a finite state model in XML format then from the XML format into the finite state model formatted properly for the model-checker SPIN. We used the model-checker to verify the properties defined by the user.

This paper presents the steps for developing the prototype tool for web-testing. We will define methods to extract a model from web applications. The model type chosen is a finite state machine, which has been widely used in protocol engineering including protocol validation/verification [5], [9], implementation [3], and testing [7], [12]. We will show how we transformed the model in a standard format which is flexible for further transformation in accordance with the model-checking tool chosen. The format chosen to represent the model is XML, which can be easily

be transformed in a FSM accepted by the out of the box model-checking tool SPIN used for research purposes. The next step is to formulate the properties to be tested in a language understood by the model checking verification tool, SPIN, to test and interpret the result.

We will conduct a case study to demonstrate the applicability of the developed tool, stepping through all the necessary phases:

- intercepting the communication between the client and the web server.
- logging the web pages visited in a flat file, in text format; the fact that we work with the output from the web server to the browser makes our tool programming language independent. For example, on the server side the code which generated the web page could be C# , ASP.Net or javascript, but we are logging only the HTML code from the browser side.
- parsing the text file in order to transform the given site into XML FSM; this gives the flexibility in the selection of the model-checker;
- demonstrating the model checking by transforming the XML FSN to PROMELA FSM, which is the input of the model checker SPIN;
- formulating the properties to be tested in the linear temporal logic language (LTL) understood by the model checking verification tool;
- verification of the properties;
- interpreting the result.

The following figure represents the steps described above:

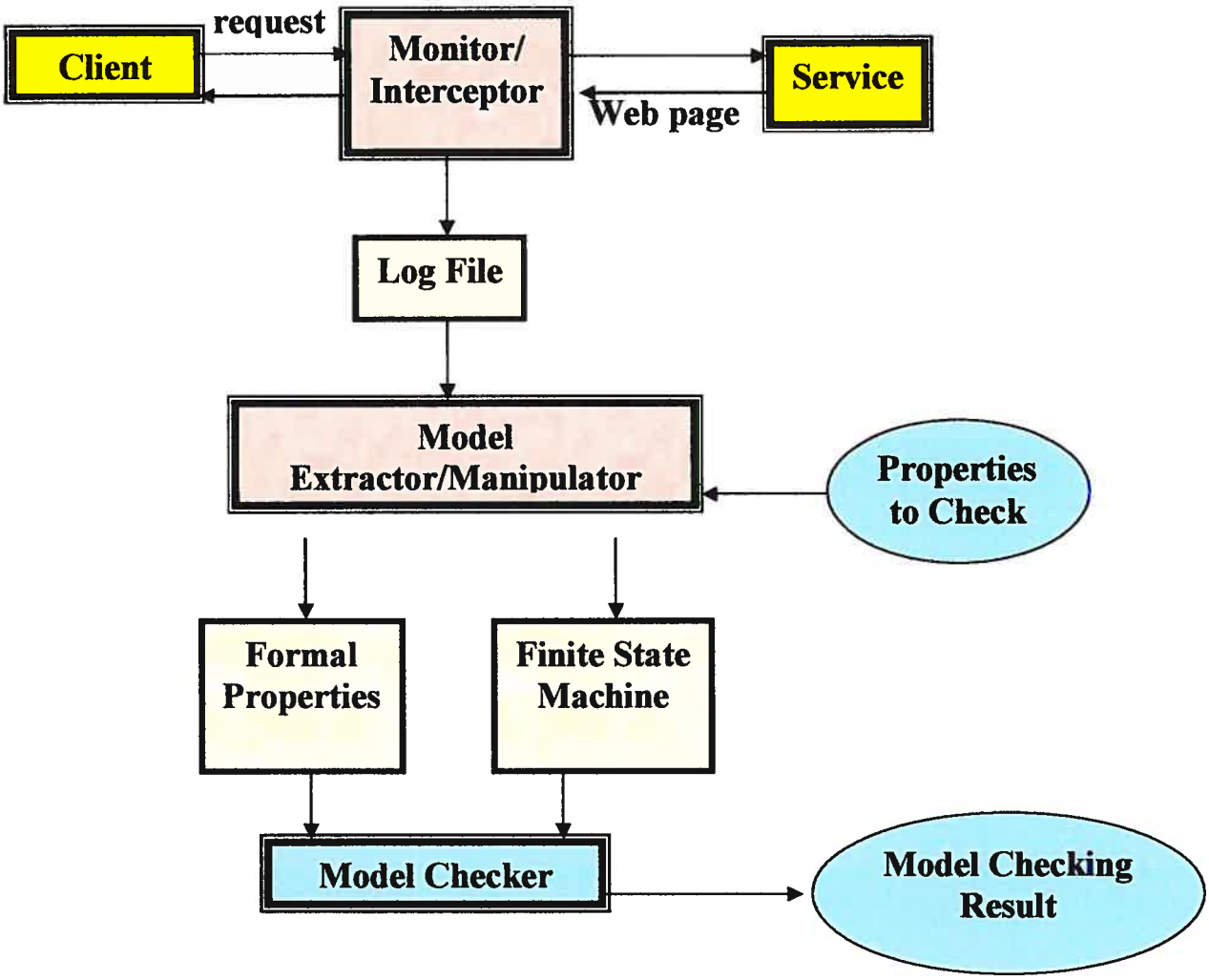


Figure 2 Framework

The remainder of this thesis is organized as follows.

Chapter 2 gives the motivation of our work conducted on the applicability of model checking technique for the web verification.

Chapter 3 presents the basics of the theory of model checking, model checking principles and the process of model checking verification technique. We finalize this chapter with the introduction of model checking tool SPIN.

Chapter 4 it is a short review of the previous research related to formal verification of hyperdocuments: specifically model checking. We present in particular the work of Stotts et al., which is the approach and methodology on which our work is based.

Chapter 5 describes the functionalities of the Web Model Extractor/Manipulator, the tool developed to extract the model from a given web site.

Chapter 6 presents a case study on which we have applied our approach and framework, namely the web site of a classical music radio station "Beethoven".

In Chapter 7 we present the conclusions and future work.

Chapter 2

2 Motivation

Existing Web testing tools [URL14] generally verify the syntax in HTML documents, confirm the hyperlink integrity of a set of HTML documents, test the GUI components embedded in the browsers and measure the performance of the Web application. Most of the tools test only one or some aspects of Web applications.

There is a need for formal checking of properties of a web application. This could be achieved with an existing model checker once a finite state machine model is extracted from the application in question.

In this case the purpose of the formal modeling is to undertake the development of a tool capable of testing the navigation of a web site according to the properties specified by the user. For example, a user may want to test if they can reach a Web site's home page from any page in a particular web site; or if a site has orphan pages, which prevent back and forth navigation unless using the browser's features.

This masters project is a part of an ongoing Ph.D. research project [9] related to the development of a prototype tool that could be used by the web based application developers and testers to improve the quality of their products. The main project is about modeling an existing web application into a FSM model based and validate user-defined properties. The obtained model is used to verify properties with the model-checker as well as for regression testing and documentation. The project plan includes complex multi-window/frame applications testing.

This master thesis is the first step in the realization of such a complex testing tool. Several activities were undertaken during the course of this project: the architecture was defined, a solution was implemented to record a browsing session to a log file, and a JAVA application was developed to create, manipulate and export a model of a web site for analysis. The JAVA application creates a FSM coded in XML and it can transform the XML FSN into the PROMELA language FSM – which can be interpreted by the model-checker SPIN. The JAVA application can also check the logged web-pages for requested information by the user. The end result of these activities is a methodology and set of software tools which can transform an arbitrary web site into a standard model as a finite state machine, which can then be imported into a model-checking tool. The use of finite state machines provides a formal model on which we based the testing framework.

To effectively use the tool developed by this project, the user must define what properties are to be tested and formulate them in a language understood by the model checking verification tool.

The next section presents the methodology of extracting the relevant information from a browsing session, recording it into a log file, transforming the web site in a FSM and then using the model checker to verify the user-defined properties.

2.1 Methodology Overview

Figure 3 is the detailed representation of Figure 2 where the proxy server represents the Monitor/Interceptor step from Figure 2. The Model Extractor/Manipulator is made of three steps: Processing Log File, FSN in XML and Processing FSM in XML. The Finite State Machine from Figure 2 is replaced with FSM in PROMELA and the Model Checker is specified as the SPIN Model Checker.

Referring to the Figure 3, we will describe the methodology used to extract a web site or a partial website and model it into a XML FSM.

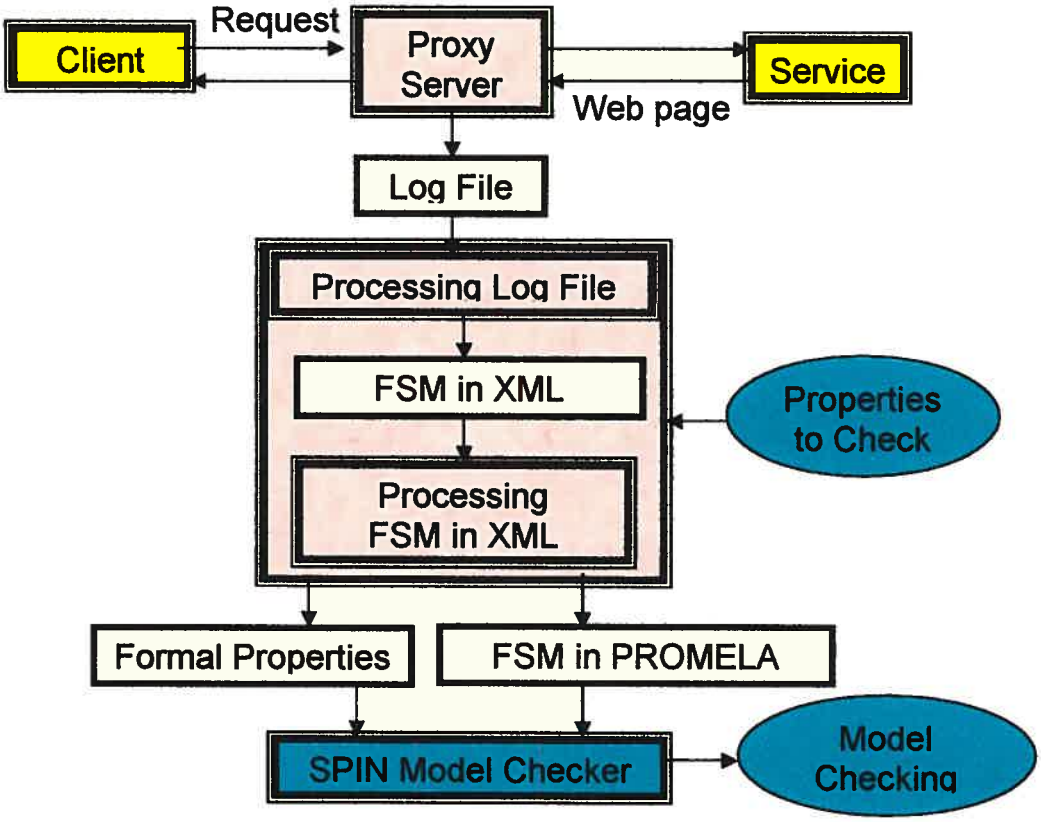


Figure 3 Framework detailed

These individual steps will now be described in more detail.

2.2 Detailed Methodology

The first step is to extract a formal model from the web application. In order to extract the model offline, we have used a proxy server that is able to intercept the data transfer between the web client and the web server: the client's requests and the server's responses. Moreover this proxy server is logging the intercepted data to a text file, referred in Figure 3 as "Log File".

The proxy server used is an open source project developed by Stefan and Emily Reitshamer, available from their web site [URL3]. This product performs typically as an HTTP proxy: it reads a client's HTTP request, forwards the request to the referred server, reads the HTTP response from the server, and forwards it to the client and optimally creates the log file of all the data transfers that occurred. The requests and responses are logged as they arrive.

The Log File contains the requests and the responses with their header, body and details as they would appear if we would look at the source of the web page from the browser's View\Source menu. The next figure is a fragment of the log file.

```
GET http://www.crim.ca/index.epl?selec=2300&href=/rd/telecom.htm HTTP/1.0
Host: www.crim.ca
Accept: application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, image/gif,
image/x-xbitmap, image/jpeg, image/pjpeg, */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 4.0)
Accept-Language: en-us
----------END OF HTTP REQUEST----------
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 18316
Server: Apache/1.3.9 (Unix) mod_perl/1.21 mod_ssl/2.4.9 OpenSSL/0.9.4
Date: Wed, 10 Apr 2002 19:40:02 GMT

<HTML>
<HEAD>
<LINK rel="stylesheet" href="/styles.css">
...
<TITLE>CRIM : Développement de réseaux de télécommunications</TITLE>
...
<A href="/index.epl?selec=2700&href=/rd/personnes/alexandre_petrenko.htm">
Alexandre Petrenko</A>
....
<AREA href="/index.epl?href=/visite/index.htm" coords="79, 11, 143, 33" shape="rect">....
</HTML>
----------END OF HTTP RESPONSE----------
```

Figure 4 Fragment from the log file

Having the log file of the proxy server as a starting point we consider the data extraction to create a finite state machine model.

As previously discussed, the prototype tool is based on the finite state machine theory, in particular, the theory of model checking. The development of a verification environment for web applications is relying on the model checker SPIN, an efficient verification system for models of distributed software systems.

The finite state machine (FSM) model consists in a set of states (including the initial state), a set of input events, and a state transition function. The function takes the

current state and an input event and returns the next state. Some states may be designated as "terminal states", which are end states with no more transitions.

In order to create the FSM the log file in text format is parsed and the client's requests and the server's responses are extracted.

In the Figure 5, an example of a FSM fragment is shown. The states are the web pages, the initial state is the Home Page of CRIM's web site, and the input events are the clicks on a hyperlink which creates a request to the web server. One of the terminal states in our example is the *error state* marked in red in the Figure 5.

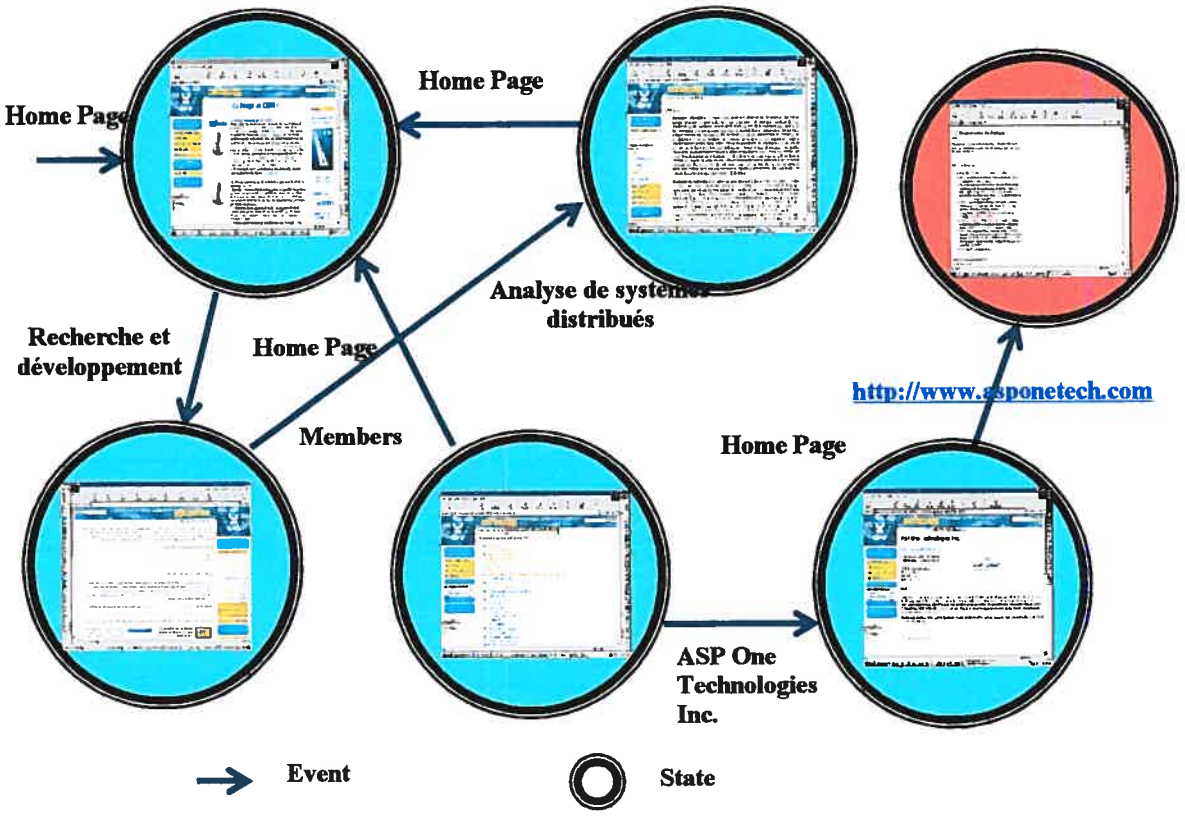


Figure 5 Fragment of the FSM Model

With the data extracted from the log file, the corresponding FSM model in XML format is then created. We have chosen the XML format due to its increasingly flexible data format, easy to transform in another format if necessary and easy to understand by anyone with or without XML knowledge.

Our goal is to transform a chosen web site into a FSM model and then as an input of the model checking tool. The model checking tool chosen for our work is SPIN. SPIN is a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems and the language accepted by SPIN is PROMELA [URL13]. The next step is to transform the FSM formatted in XML into an FSM formatted in PROMELA. If for any reason there is a need to use another model checking tool, the FSM in XML can be easily transformed into another format of FSM.

The transformation of the log file into the XML FSM and from XML FSM into PROMELA FSM is automated with the prototype tool that we have implemented. In Figure 3 those steps are represented as "Processing Log File", "FSM in XML" and "Processing FSM in XML".

As shown in Figure 3, before the verification of the FSM model with the model checking tool, we need to formulate the properties to be verified and to translate them in the language used by the model checking verification tool (*Formal Properties* step). The language in which the properties are formulated is the *linear temporal logic* (LTL), which will be described in more detail later in this paper.

Finally, after applying the verification of the properties on the FSM model – (*SPIN Model Checker* step) we interpret the results (*Model Checking Results)* to see if the model respects the properties or not.

The steps *Processing Log File*, *FSM in XML* and *Processing FSM in XML* are implemented in our prototype tool that automates the process of extracting a state

machine model of the application to be verified, so that formal verification could be performed using an off the shelf model checker.

Chapter 3

3 Model Checking

In this chapter we make a short introduction of model checking theory in the first two sections and then continue with the presentation of the model checking browsing properties and introduce the SPIN model checker.

Recently, many tools were developed to verify the correctness of a system. The basics of those tools rely on **formal methods,** which are methods based on mathematical models describing the systems that are to be tested. The automation of this reasoning allows large systems to be analyzed efficiently. **Model checking** method is a successful attempt in this direction.

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient algorithms are used to traverse the model defined by the system and to check if the specification holds or not. Large state-spaces can often be traversed in minutes. The technique has been applied to several complex industrial systems such as the Futurebus+ and the PCI local bus protocols. [URL5]

3.1 The Process of Model Checking

The process of model checking was first introduced by Edmund M. Clarke [4]. It consists of three main tasks: modeling, specifications and verification.

3.1.1 Modeling

In order to use the model checking technique, the first task is to model a system using a formalism accepted by a model checking tool. In many cases, this can be straightforward in other cases it can be quite involved. Modeling a system may require the use of abstraction to eliminate irrelevant or unimportant details. In general this task consist of transforming the system in an automaton with the aim to produce the appropriate finite state machine for the Kripke structure [4] required by the model checker.

3.1.2 Specifications

The next step is to state the properties that the model must satisfy. A property in the model checking theory is described as a temporal logic formula named automata.

3.1.3 Verification

Ideally the verification is fully automatic, but, in practice it often involves human assistance. The analysis of the verification result is done manually. In case of a negative result, the model checker provides us a counter example for the checked property so we can detect where the error occurred.

An error trace can result from incorrect modeling of the system or from an incorrect specification (also called *false negative*). The error trace can be useful in identifying and fixing these problems.

3.2 The model

The first step in verifying correctness of the system is specifying the properties that the system should or must have. Once system related properties are identified, the second step is to construct the formal model for the system. In order to be suitable for verification, the model should capture those properties that must be considered to establish correctness. On the other hand, it should abstract away those details that do not affect the correctness of the checked properties and make verification more complicated.

In the model checking theory, the system to be analyzed is modeled using some framework based on defining a set of states and a transition relation determining how the system may change over time. This model can then be checked against a specification, written in some logic specifying desirable properties of the system's dynamic behaviour. [4]

A **state** is a representation of the characteristics of the system in a particular instance of time. The state of the system can change in time as a result of some internal or external events. Those changes can be described by giving the state before the event occurs and the state after the event occurs. Such a pair of states determines a **transition** of the system.[4]

3.3 Formalizing browsing properties

For a better understanding of properties to be checked with a model checker on a hypertext document model, we present a short example of how the browsing properties can be formalized. Given a hypertext document with elements X and Y, it contains the link anchors B and C and the information I. We would like to formalize statements like:

- "all browsing sessions must encounter X, but only sometime after seeing Y, and information I is encountered on this path not more than 3 times"
- "there is at least one browsing session encountering Y with the information I,"
- "there is a browsing session in which at some point B and C are both encountered and the information I is present."

More general specifications include:

- "for every node X there must be a path back to the index"
- "every node Y must have at least one link out."

The browsing behaviour of a hypertext document can be formalized as event traces which can occur in a browsing session. Traces can be described with a temporal logic language.

Temporal Logics are descriptive languages for the specification of ordering relationships characterizing the execution sequencing of time-varying systems. Temporal logics are often classified according to whether time is assumed to have a linear or a branching structure. The concept of time implied by "temporal" is not duration but rather the relative ordering of events in a sequence. [16]

Symbolic logicians and philosophers, for reasoning about ordering of events in time without mentioning the time explicitly, first conducted work in temporal logic under the name of tense logic. In the last decade, temporal logic has become a convenient formalism used in program verification. [16]

More recently, temporal logic has been applied successfully to the specification, verification, and analysis of reactive systems. Reactive systems are based on concurrent computations that maintain a relationship with their environments. [16] Hypertext documents bear some resemblance to reactive systems, that is why we can successfully apply on them model checking method.

3.4 SPIN Model Checker

SPIN is a widely distributed software package that supports the formal verification of distributed systems. The software was developed at Bell Labs in the formal methods and verification group starting in 1980.

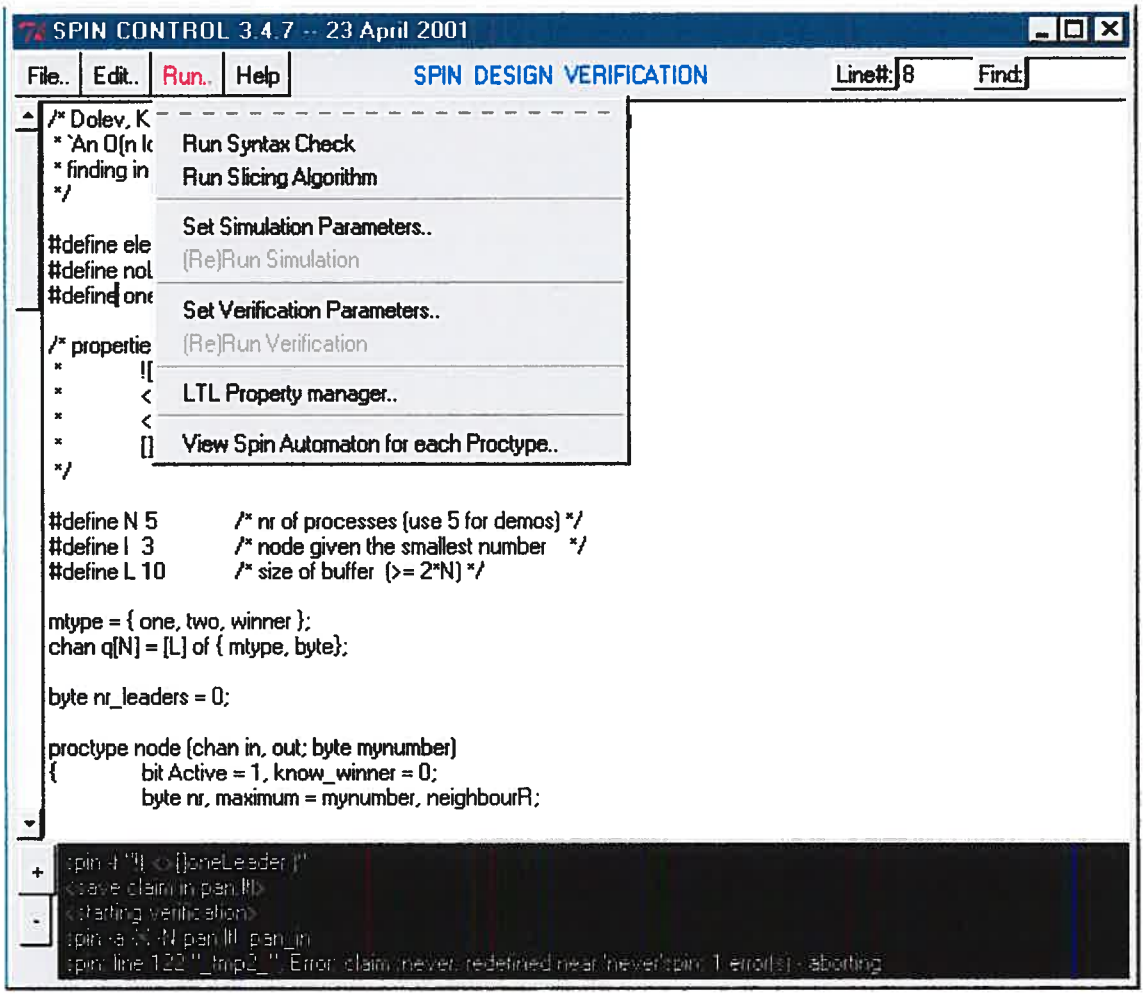


Figure 6 SPIN Model Checker

Some of the features that set this tool apart from related verification systems are:

- SPIN targets efficient software verification, not hardware verification. SPIN uses a high level language to specify system descriptions, called PROMELA (PROcess MEta LAnguage). SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signalling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks and unspecified receptions, it flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

- SPIN works **on-the-fly**, which means that it avoids the need of constructing a global state graph, or a Kripke structure, as a prerequisite for the verification of any system properties.
- SPIN can be used as a full **LTL model checking** system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), or as general linear temporal logic requirements (LTL), either directly in the syntax of LTL, or indirectly as **Büchi Automata** (called **never** claims).
- SPIN supports random, interactive and guided simulation, and both exhaustive and partial proof techniques. The tool is meant to scale smoothly with problem size, and is specifically designed to handle even very large problem sizes.
- To optimize the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques. [4]

3.4.1 PROMELA (PROcess MEta LAnguage)

PROMELA is a verification modeling language. The intended use of SPIN is to verify fractions of process behaviour that for one reason or another are considered suspect. The relevant behaviour is modeled in PROMELA and verified. A complete verification is typically performed in a series of steps, with the construction of increasingly detailed PROMELA models at each step. Each model can be verified with SPIN under different types of assumptions about the environment. Once the correctness of a model has been established with SPIN, this fact can be used in the construction and verification of all subsequent models. [URL6]

3.5 Formal Properties (LTL)

Linear temporal logic (LTL), specifying correctness requirements, expresses features of one individual (possibly infinitely long) execution of a system. In this thesis we consider LTL, because we want to verify an individual run that occurs during a simulation of the system. The execution of a system can be observed as an (infinite) discrete sequence of boolean values that can be evaluated in every state of the system. Such values can for example correspond to the fact that a certain information or transition has been found. Without specifying how these observable features are defined, we will call them atomic propositions and denote them with the letters p, q, etc. [12]

LTL formulae consist of atomic propositions, boolean connectives and temporal operators. Temporal operators are:

- always, or henceforth (symbol: □)
    - We say that $\Box\varphi$ is true at a moment i, if φ is true from moment i onwards.
- eventually (symbol: ◊)
    - We say that $\Diamond\varphi$ is true at moment i, if φ will eventually be true at moment i or later.
- strong until (symbol: U)
    - We say that $(\varphi U \psi)$ is true at moment i, if ψ eventually becomes true, and until then φ is true.
- weak until (symbol: W)
    - Like Strong Until, but without the requirement that ψ eventually becomes true.
- next time (symbol: O)

- We say that Oφ is true at the moment *i*, if φ will be true at moment i+1.

Chapter 4

4 State of the Art

In this chapter we describe work related to formal verification of hyperdocuments and in particular the use of model checking. Several approaches have been proposed to validate aspects of hyperdocuments, such as navigation-based verification by de-emphasizing browser features and emphasizing inherent document structure with browsing semantics [16], [17].

In section 4.1 and 4.2 we will present Stotts et al. [16] approach and methodology. In section 4.3 we will talk about Huang and Jang [9] work in multimedia domain. Section 4.4 is a brief presentation of other related research papers. With that background, in section 4.5 our approach is described.

4.1 Stotts approach

We will start with a resume of the work completed by Stotts et al. in this domain and then we will detail their work presenting also our contributions and differences.

Stotts et al. [16] did not develop a new model checking technique, but they proved that it is possible to apply it to the domain of hypermedia. They have a new view of hyperdocuments, viewing them as an abstract process instead of a static data structure links-automaton. They modeled Trellis and Hyperties hyperdocuments as a Petri-net structure to build the corresponding links-automaton. Stotts et al. expressed the

browsing properties using Hypertext Temporal Logic (HTL and HTL*), a branching temporal logic based on CTL*, CTL, and POTL temporal notation. Properties expressed with HTL and HTL* can be efficiently verified with Clarke's model checking technique.

Model checking technique was borrowed from concurrent–system verification and Stotts proved that it can be adapted to be used to increase the utility of hypertext documents and specifically document's browsing properties, as for example: what sequence of links a reader may be allowed to follow during browsing.

Stotts et al. shows how to verify in an automated fashion whether the linked structure of a document satisfies the required property specifications. The focus of the authors is concentrated on the behaviour that is allowed by links alone, independent of any navigation aids that a browser or navigation programs might provide as "Back", "Forward" or "History" button.

We extend this idea and we will prove that we can successfully apply model checking for other kinds of hyperdocuments and web sites, using a more straightforward model, the finite state machine.

4.1.1 Links-Only Document Behaviour

Researchers have successfully modelled the Web as a graph in different representations. It is beyond the scope of the thesis to make a synthesis about the web represented as a graph, but we will discuss how Stotts represented Trellis and Hyperties and how we see the Web model.

Stotts approach is not exclusively applicable only on Trellis and Hyperties, it is applicable to any hypertext system. According to the authors, what is required in particular about a hypertext document is that it must be viewed as an abstract automaton that specifies the process of the browsing within it. One of Stotts et al.

novel contributions is the idea of links-automaton, in which they view hyperdocuments as an abstract process instead of a static data structure [16].

Stotts et al. represents Trellis and Hyperties as a Petri net or PT-net [URL15] which is a formal and graphical appealing language which is appropriate for modelling systems. Also, they are using the model checking algorithm and software from Clarke which uses a finite state model. In order to use this model checking technique they are transforming PT-net representation of the documents into a finite state machine [16].

Figure 7 shows the traditional view of the hyperdocument: a browser program allowing navigation over a directed graph.

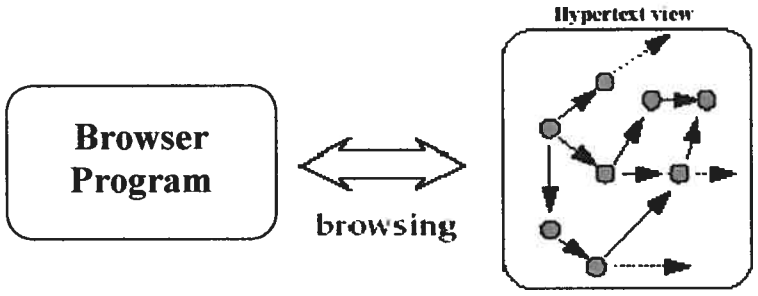


Figure 7 Traditional view of the hypertext document

Figure 8 represents the structure of a hypertext document view by Stotts et al. There exists a browsing path from the starting node (A), continuing through the nodes E and D, and ending with node C. The link-only behaviour of the document does not allow any further browsing from this point, because there is no transition out of node C. In order for browsing to continue, the author of this structure must be relying on some feature of the browser.

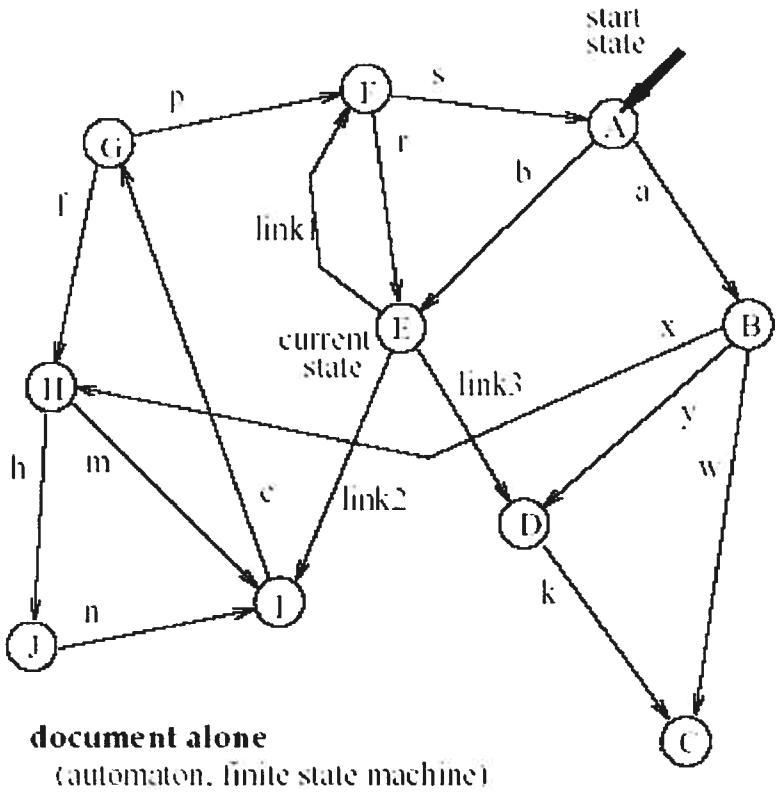


Figure 8 Automaton view of the hyperdocument

The structure of this document will be much more attractive for the user if the user can search for information in the document without going back to the old information that was already seen. A good design in our opinion will be one in which from every node we have at least one way out and the ability to navigate to an index, for example.

4.1.2 Temporal logic and dynamic properties of systems

A popular formalism to express properties of state-transition based concurrent systems is temporal logic. Temporal logic model checking algorithms introduced in the 1980s allowed the reasoning to be automated [14]. Besides being automatic, model checking has another important advantage over proof-checker based methods: if a formula is not true of a model, we can produce an execution trace that shows why the formula is not satisfied.

Stotts et al. are using temporal logic for expressing browsing properties of hyperdocuments, and the use of links-automaton of hyperdocuments as a basis for model checking of these properties [16]. CTL (Computation Tree Logic) was the first temporal logic language used by Clarke's original model checker to define the specifications [1].

The Stotts's approach introduces HTL*, or *hypertext temporal logic* for succinctly expressing hyperdocument browsing properties, and the HTL subset of HTL* that can be verified with the model checking algorithm and software from Clarke, which they use to implement their hyperdocument analysis tool.

Instead of HTL* we are using LTL or *linear temporal logic* which is a widely used logic for expressing properties of programs viewed as sets of executions, and is accepted by SPIN model checker [11]. An LTL formulae or formula is supposed to hold for all possible executions that the system might produce.

Stotts's approach wishes to be general, being applicable to a wide range of hyperdocuments types, but we believe that their methods can be improved. We are convinced that improvements can be achieved in the model representation where they are using the reachability graph of a Petri-net as the finite-state model for the verification of the properties, relating mainly to browsing aspects of hyperdocuments without taking in account the content of the state itself.

4.2 Stotts model

To understand better the differences between our model and the model created by Stotts et al. let us expose their model as they describe it in [16].

Trellis hypermedia was developed by Stotts and Furuta in 1989. In Trellis system, links can have multiple source nodes and multiple destination nodes. Moreover this system allows parallel browsing paths with multiple concurrently displayed content elements; meaning, for example, that when we click on a link leading to a new page, the target content popup on the screen and the source content remains visible. The source can have a supplementary action: the click on the "remove" button.

To model this kind of hypermedia Stotts et al. used PT-net. In order to use the model checking technique, Stotts et al. must transform the PT-net in a finite state machine and they are doing so by computing the coverability graph of the PT-net and simplify it for not having redundant states.
In Figure 9 we show the finite state machine obtained by Stotts et al. in their case study [16].

```
NAME = RefB.fsm;
INPUTS = ;
STATES = 8;
CUBES = 12;
MOORE-OUTPUTS = c.welcome, c.overview, c.shuttle, c.engine, c.allow, c.inhibit,
                b.begin, b.orbiter, b.propulsion, b.start, b.return, b.remove;

#0  100010100000
    1
#1  010010011000
    2
    6
#2  010101000011
    3
    4
```

```
#3  000110000010
    0
#4  110001100001
    0
    5
#5  010001000001
    1
#6  011001000101
    7
    4
#7  001010000100
    0
#END
```

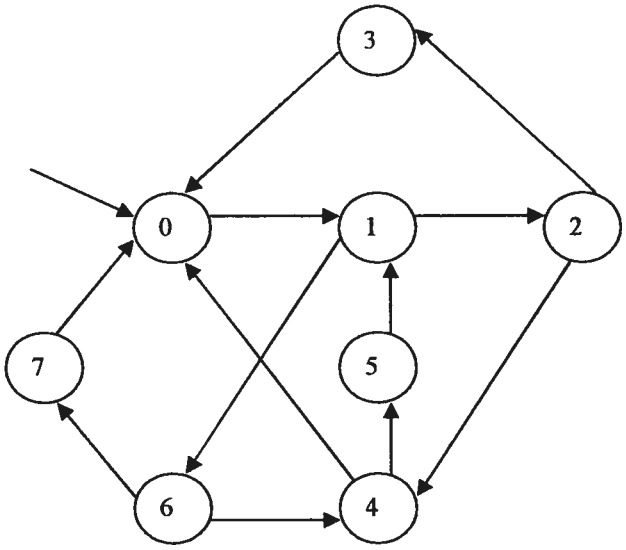


Figure 9 FSM encoding of the links-automaton for Trellis document

Variable STATES in the picture represents the number of states, variable MOORE OUTPUTS represents the name of the atomic predicates used and the bit vector following the state number indicates which atomic predicate(s) – respecting the order listed in MOORE OUTPUTS – are presented or not in that state. The transitions out of the state are present after each state number and they are represented as a list of destination state numbers.

In the presented example there are some concurrent elements. In state 2, in the bit vector on positions 2 and 4 the bits are set to 1, signifying the presence of two predicates from the MOORE OUTPUTS list (the 2nd and the 4th). This atomic predicates are concurrently visible while browsing the page represented by state 2.
At this point, after modeling the hyperdocument, using the model checking technique, Stotts can verify browsing properties expressed with HTL* [16].

We will show an example of one of the properties formulated in proper English and HTL*:
*"Does there exist a browsing path such that at some point both the "**shuttle**" text and "**engines**" text are concurrently visible?"* [16]

$$\vec{\exists} \Diamond (c.shuttle \wedge c.engine)$$

```
|=EF(c.shuttle & c.engine).
The formulae is FALSE.
```

4.3 Formal modeling in WWW multimedia

Chung-Ming Huang and Ming-Yuhe Jang [9] focus on the formal modeling of interactive temporal behaviour and hyperlink temporal behaviour in multimedia WWW application systems and they propose the extended final state machine approach for formal modeling of interactive temporal and link control architecture .

The response time and the synchronization of the elements involved in multimedia system are the main keys. In this paper [9] the authors study and model the behaviour of WWW multimedia components synchronization correlated to user interaction.

They define two types of temporal synchronization: (a) intramedium synchronization which deals with internal behaviour in medium stream and (b) intermedia synchronization which reduce asynchronous anomalies among media stream.

Huang and Jang theory includes the user interactive temporal behaviour with the web application, called "user interaction" for simplicity. User interaction in WWW is classified as: (i) interpage user interaction and (ii) intrapage user interaction.
A multimedia application is not affected only by the user interaction but also by the availability of its components which are accessible via hyperlinks. The hyperlinks could be valid during certain time periods and invalid during other time periods. Therefore the authors separate their theory in (i) interpage and intrapage user interactions and (ii) hyperlink temporal behaviour.

4.3.1 Interactive temporal behaviour

Huang and Jang like Stotts et al. represent the navigation of a set of web pages as a graph in which a circle denotes a webpage and an arc denotes a hyperlink. They give

us an example of a distance learning course for a mathematical theorem graphically represented in Figure 10.

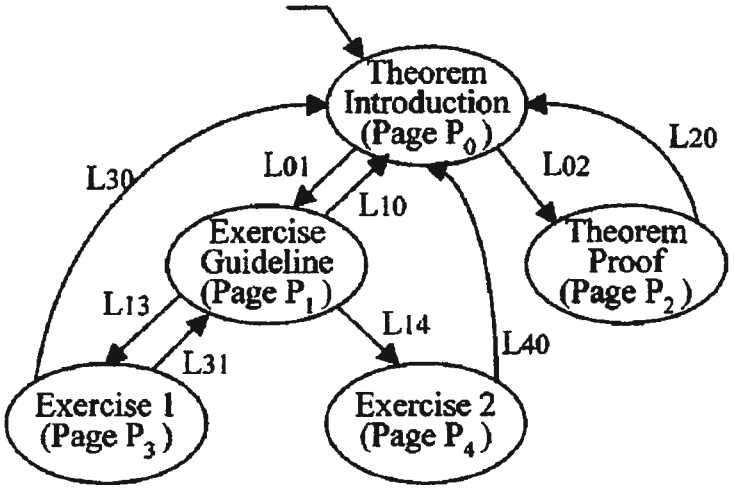


Figure 10 A distance course in several linked web pages

P_i are the web pages and L_{ij} is the hyperlink from page P_i to page P_j. In Figure 10, P_0 is *Theorem* introduction web page, P_1 is the *Exercise Guideline* page and L_{01} is the hyperlink which leads from the *Theorem* page to the *Exercise Guideline* page.

In a WWW multimedia system, multimedia presentations are carried out by the user's clicks on the available hyperlinks on presentation's web pages. Web pages may contain images, text, graphics, video objects and multimedia scenarios which consist in several presentation stages.

In the above example, Huang and Jang assume that P_0 has three image objects to illustrate the formula ${}_0I_1$, ${}_0I_2$, ${}_0I_3$; two text objects ${}_0T_1$ and ${}_0T_2$ for the legends used in ${}_0I_1$ and ${}_0I_3$ respectively; two audio objects ${}_0A_1$ and ${}_0A_2$ for teacher annotations and two video objects ${}_0V_1$ and ${}_0V_2$ associated with the teacher's annotations as in Figure 11 from Huang and Jang.

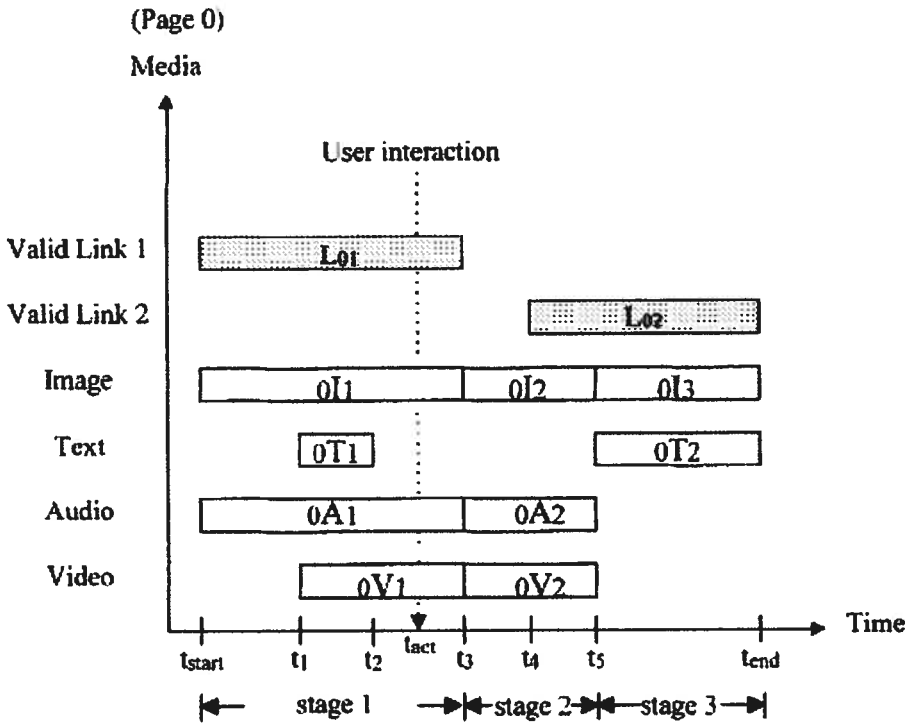


Figure 11 *Multimedia presentation schedules for page P_0*

Figure 11 presents page P_0 as follows: the P_0 presentation can be divided into three presentation stages, stage 1, 2 and 3, which associate their own temporally related media objects in three time durations $[t_{start}, t_3]$, $[t_3, t_5]$ and $[t_5, t_{end}]$ respectively.
We use few paragraphs from Huang and Jang [9] to be faithful to their examples and explanations and we use italic font to enhance them in this sub-chapter.

Stage 1 consists of the following:

(1) image object $_0I_1$ and audio object $_0A_1$ begin their presentations at time t_{start} simultaneously;

(2) at time t_1, text object $_0T_1$ and video object $_0V_1$ simultaneously begin their presentations;

(3) text object $_0T_1$ ends its presentation at time t_2;

(4) objects $_0I_1$, $_0A_1$ and $_0V_1$ end their presentations together at time t_3 and at that time (t3,) stage 2 starts up.

Stage 2 consists of the following:
(1) objects ${}_0I_2$, ${}_0A_2$ and ${}_0V_2$ simultaneously begin their presentations;
(2) at time t_5, objects ${}_0I_2$, ${}_0A_2$ and ${}_0V_2$ end their presentations at which time stage 3 starts up.

Stage 3 consists of the following: objects ${}_0I_3$ and ${}_0T_2$ start their presentation sequences until time t_{end} is attained. At time t_{end}, presentation of P_0 is completed and the user can switch to the other pages' presentations by clicking some other hyperlinks. Hyperlinks can be valid/invalid, i.e. the corresponding marks are/are not shown on the screen, so that users are/are not able to click on them. Hyperlink L_{01} is valid from time t_{start} to time t_3; hyperlink L_{02} is valid from time t_4 to time t_{end}.

We will not present the Huang and Jang theory that discusses user interactions, as we are doing offline testing, which means that we no longer deals with the user interactions but rather their actions that are stored in a log file, as we will describe later. We are more interested in showing the similarities of Huang and Jang theory with our work regarding hyperlink behaviour.

4.3.2 The abstract temporal synchronization control architecture

Huang and Jang named the processing of the hyperlink temporal behaviour, "link synchronization". Link synchronization is responsible for the continuity of the related pages' presentations. Users cannot avoid some waiting time for requested pages because some fetching time is required to load the requested pages into the local media buffer. Link synchronization in WWW multimedia presentation is necessary to shorten the waiting period.

Huang and Jang designed and architecture for multimedia WWW interactive temporal synchronization control, as depicted in Figure 12.

The hypermedia browser and several remote WWW servers are interconnected over the Internet. The remote WWW media servers can receive request messages from hypermedia browsers and respond with media objects.

The kernel of the hypermedia browser's interactive temporal synchronization control mechanism consists of three parts, i.e. *Pager*, *Synchronizer* and *Actor*.
The main function of *Pager* is exception handling and fetching media objects contained in requested pages from WWW servers. Each Web page *P* has a *Synchronizer* and a set of *Actors*, which cooperate with each other to control the associated multimedia presentation of *P*. Each medium stream is associated with an *Actor*.

The *Synchronizer* is responsible for intermedia synchronization; an *Actor* is responsible for the medium stream's intra-medium synchronization. The *Synchronizer* is also responsible for receiving user's interactions from the user interface and processing the corresponding reactions, for example, having *Actors* pause the ongoing presentation and asking *Pager* to fetch the next page.

A set of hyperlinked Web pages has a *Pager*. Interpage interactive synchronization and intrapage interactive synchronization are achieved through the cooperation of *Pager*, *Synchronizer* and *Actors*. With this approach the *Pager*, *Synchronizer* and *Actor* can be represented as *Extended Finite State Machines* (EFSM) respectively.

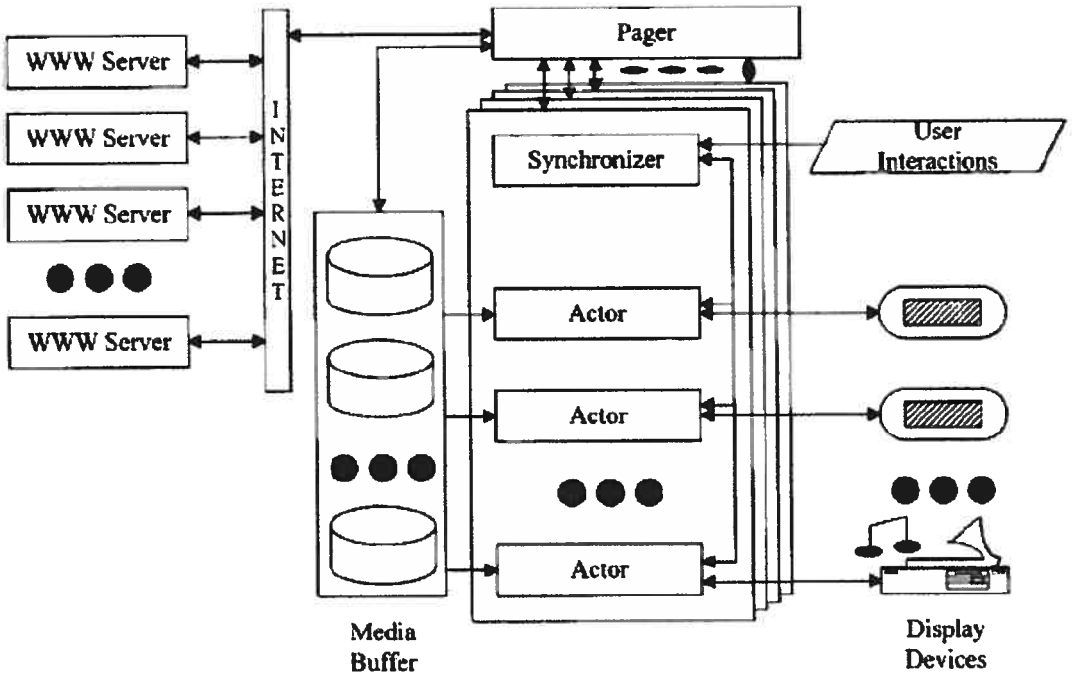


Figure 12 *The abstract temporal synchronization control architecture*

4.3.3 The EFSM model

Huang and Jang used EFSM model to formally model (i) interpage and intrapage user interaction and (ii) the hyperlink temporal behaviour.

An EFSM is formally represented as a nine-tuple $(\sum; S; s_o; s_f; V; B; A; P; \delta)$*, where (i)* $\sum$ *is the set of messages that can be sent or received; (ii) S is a set of states; (iii)* s_o *is the initial state; (iv)* s_f *is the final state; (v) V is the set of context variables; (vi) B is the set of predicates that operate on context variables; (vii) A is the set of actions that operate on context variables; (viii) P is the set of priority clauses; (ix)* δ *is the set of state transition functions in which each state transition function can be formally represented as follows:* $S \times \sum \times B(V) \times P \rightarrow S \times \sum \times A(V)$*. For convenience, each state transition is represented as* $S_1 \xrightarrow{T} S_2$*, where* S_1 (S_2) *is called the head (tail) state of transition T, T is called the incoming (outgoing) transition of state* S_2 (S_1)*.*

We will present the modeling of link synchronization formalized by Huang and Jang using EFSM model.

4.3.4 Link synchronization

An important part of WWW media presentation is the process of an interpage user interaction. The interpage user interaction in this context consists of two events. One is to terminate the fetching of the current page and the other is to fetch the target page that is requested by the user.

The progress of the current page pre-fetching in Pager EFSM is terminated when Pager EFSM changes from any state S_i to state Valid^i , $i = 0...3$, state transactions depicted in Figure 14. The behaviour of fetching the target page is divided into three phases which correspond to states Valid^i , FetchingPage^i and In-lineFetching^i (Figure 14).

(i) At state Valid^i , Pager EFSM checks whether the target page is on the local buffer or not. If the target page is unavailable on the local buffer transition Tnv^i is executed for fetching the page. However, if the target page is available on the local buffer transition Tv^i is executed to skip the page fetching.

In transition Tv^i, Pager EFSM reloads the most recently interrupted information of the target page. This is to inform Synchronizer EFSM of the target page to determine from when and which part of the target page's presentation is to be resumed. Pager EFSM changes from state Valid^i to state S^0 (FetchingPage^i) after executing transition Tv^i (Tnv^i).

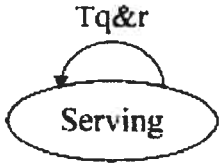
(ii) At state FetchingPage^i, *Pager EFSM* (a) receives the page document in the response message, i.e. the input event '*When Server(Port).RESPONSE(response)*'.

If the status of the response is *OK*, transition *Tparse^i* is executed. Otherwise, transition *Tnv^i* is executed to notify the corresponding *Synchronizer EFSM* with an

error message using the output event '*output Syn(P).ERROR(P, NextP, IT, IS)*'. *Pager EFSM* changes from state *FetchingPage*^{i} to state *In-lineFetching*^{j} (S^{4}) after executing transition *Tparse*^{i} (Trcv^{i}).

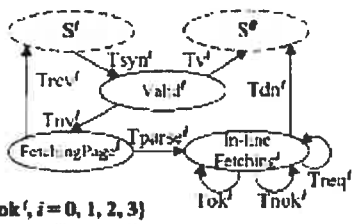
(iii) At state *In-lineFetching*^{j}, *Pager EFSM*: (a) continues requesting the rest of the in-line objects of the target page to be fetched. This is done by repeatedly executing transition *Treq*^{j}; and (b) by checking the response status of the fetched object during the fetching process. If the response is OK, transition *Tok*^{j} is executed to store the fetched object; otherwise, transition *Tnok*^{j} is executed to store default objects. When the fetching of all in-line objects in the target page is complete *Pager EFSM* executes transition *Tdn*^{i} to initiate the associated *Synchronizer* and *Actor EFSMs* for the target page's presentation. *Pager EFSM* stays at state *In-lineFetching*^{j} after executing transition *Treq*^{j}, *Tok*^{j} or *Tnok*^{j} and changes to state S^{0} after executing transition *Tdn*^{i}. The media server EFSM at the server site is depicted in Figure 13.

In transition *Tq&r*, the media server receives the request message sent from *Pager EFSM* and sends the corresponding response message to *Pager EFSM* in the action part. After the '*one-request and one-response*' communication session is complete, the corresponding HTTP connection is disconnected.




```
{Tq&r}
From  Serving
To    Serving
When  Pager.REQUEST(X);
Begin
  load(RES, X);
  output  Pager.RESPONSE(RES);
  disconnect(X);
End
```

Figure 13 *The media server EFSM*



{Tnok^i, i = 0, 1, 2, 3}
From In-lineFetching^i
To In-lineFetching^i
When Server(Port).RESPNSE(response)
Provided response.StatusCode @ ErrorCodes = True
Begin
store(DefaultObject, response.StatusCode);
index := index + 1;
End

{Tv^i, i = 0, 1, 2, 3}
From Valid^i
To S*
Provided valid = True
Begin
pre_status(NextP, Itype, Istatus);
update_status(P, IT, IS);
initialization(NextP);
output Syn(NextP).OK(NextP, Itype, Istatus);
End

{Tnv^i, i = 0, 1, 2, 3}
From Valid^i
To FetchingPage^i
Provided valid = False
Begin
getURL(NextP, url);
resolve(url, Method, Source, Port);
tcpconnect(Source, Port, handle);
prepare_http(handle, Method, Source, request);
output Server(Port).REQUEST(request);
End

{Trcv^i, i = 0, 1, 2, 3}
From FetchingPage^i
To S^i
When Server(Port).RESPNSE(response)
Provided response.StatusCode @ ErrorCodes = True
Begin
pre_status(P, Itype, Istatus);
output Syn(P).OK(P, Itype, Istatus);
End

{Tsyn^i, i = 0, 1, 2, 3}
From S^i
To Valid^i
When Syn(P).PAGE(P, NextP, IT, IS)
Begin
local_find(NextP, valid);
End

{Tok^i, i = 0, 1, 2, 3}
From In-lineFetching^i
To In-lineFetching^i
When Server(Port).RESPNSE(response)
Provided response.StatusCode = OK
Begin
store(response, OK);
index := index + 1;
End

{Treq^i, i = 0, 1, 2, 3}
From In-lineFetching^i
To In-lineFetching^i
Provided count > 0
Begin
resolve(url_list[count], Method, Source, Port);
tcpconnect(Source, Port, handle);
prepare_http(handle, Method, Source, request);
count := count - 1;
output Server(Port).REQUEST(request);
index := 0;
End

{Tdn^i, i = 0, 1, 2, 3}
From In-lineFetching^i
To S*
Provided index > url_count
Begin
initialization(NextP);
pre_status(NextP, Itype, Istatus);
update_status(P, IT, IS);
output Syn(NextP).OK(NextP, Itype, Istatus);
End

{Tparse^i, i = 0, 1, 2, 3}
From FetchingPage^i
To In-lineFetching^i
When Server(Port).RESPNSE(response)
Provided response.StatusCode = OK
Begin
parse(response, url_list[ ], url_count);
count := url_count;
index := 0;
End

Figure 14 *Added states and transitions in Pager EFSM for processing interpage user interactions*

Huang and Jang are using EFSM to model temporal WWW media behaviour into taking account the user interactions. They describe the architecture of a synchronization control for links in WWW multimedia presentations, necessary to shorten the waiting time and they create the EFSM model for a WWW media presentation using the synchronization control. The synchronization control works

like the caching principle for web-based applications; when the requested web-age exists in cache then the page is not requested from the web server. The difference is that the related objects of the pages are required from the server even if they were not requested for immediate presentation and they are stored in the buffer. Also if the page is not available they are fetched from the media server.

Absent from their work is a description of the important and difficult task of capturing the structure and timing as they are presented to the user in a relatively automatic fashion. Considerable more work, including arbitrary model structure selection, must be done from the server-side perspective, looking directly at the source of the multimedia, and developing a correlated model alongside the multimedia presentation. Such parallel work reduces the attractiveness of formal modelling and is likely to introduce user error in the process, although this approach can be useful as a planning tool before the multimedia is fully produced. In contrast, in this paper this information is captured from the log files and input to create the models directly. We use the output of the web server to the browser as a source for the formal model which increases the correctness of the model, compared to this approach where all of a user's browsing related behaviour is split into different EFSM. This makes it more difficult to test the application as a whole.

4.4 Other Related work

In "Testing Web Applications by Modeling with FSM" by Andrews et al. [2], the technique used is based on a black box system and most significantly attempts to address the problem with state space explosion when large web sites are analyzed using an FSM technique. State space explosion is a significant problem as each additional user input or web page in a heavily cross-linked web site can create exponentially more states and test cases.

This research takes the approach of reducing the possible number of states by clustering groups of logical functions together in a hierarchical format. The smallest units are made up of logical web pages – possibly a single page, but possibly multiple pages performing a common function. These logical web pages are grouped together as a cluster. Clusters may contain other clusters. To simplify the FSM, each logical web page is assumed to have only one starting state and one finishing state. The approach assumes continues input values, single-use or non-propagated inputs.

Continues input values are ones that have been selected for sample input and will continue to be used. For example, language selection may be an early logical web page – and its own node among the clustered finite state machines, and this choice will continue through the test as inputs are propagated among the aggregated finite state machines. Single use input values are ones that are not permitted to be re-used in a single test, for example the serial number of a new item in a shipped product list.

Importantly, the decision of clustering logical web pages together is a manual process, thus the process is likely repeatable from tester to tester, particularly as the size of the web application grows. Although not directly incorporated in this research, it is suggested that the logical web pages can be clustered together on the basis of the quantity of common other pages they link to, suggesting they should

likely be grouped together as a cluster. This approach does assist in reducing the test process to a more reasonable problem space even if the introduction of subjective clustering adds more uncertainty to this approach.

For example, a section of a web application may allow users to modify their personal profiles, while another permits them to modify their billing settings. Each application is likely to be an independent cluster. However at a higher level in the hierarchy they could be grouped together as a common cluster "user preferences"

The use of the clustering technique following a comprehensive modeling phase permits more straightforward test sequences to be defined based on possible inputs at each logical web page and then analyzed from the clustering perspective.

The tools used in this research were able to automate, to a certain extent, four tasks: identifying logical web pages and input selection constraints, identify connections between logical web pages, and partitioning the connectivity model between the logical web pages. These tasks assumed the pre-processing of the web application code. The tools could also build a test value database if one already exists with the web application.

The model is limited for further work due to the artificial imposition of the hierarchical clustering, technique. While it reduces the test problem space it does so manually and attempts to automate this element is likely to cause illogical clustering and interfere with defining constraints.

4.5 Our approach

Our approach is more straight-forward. We represent the browsing behaviour of a web site as a finite state machine (FSM) directly without going through another representation. Moreover, we are concerned not only about the linked structure, but also about the characterization of the states (web pages).

Another new element in our work is the representation of the finite state machine in an XML [URL4] file. The benefits of this representation are multiple. XML is a standard format easy to create, modify or update. It is expressive even for someone uninitiated in XML technology, and gives us a general format, which permits the reuse of models with any model-checking tool.

Unlike Stotts et al. our approach is flexible and modular; we can easily modify our tool to work with any model checker, not only with SPIN.

We are testing web applications offline. To build the model from the web site we extract the necessary information to create the model. This information extraction is achieved by intercepting the communication between a client and the server that holds the specified web site, and records this data for further processing and is programming language independent.

After collecting the response/request pairs we extract a model that conforms to the structure of the web site in terms of navigation properties. We organize the information necessary to create the model as a finite state machine. A state represents a web page with static information found in the page, such as textual information and links. The transitions between states represent the hyperlinks between pages. After the model extraction, we transform the model from a standard intermediary format model to the model accepted by the chosen model checker, in our case SPIN. The

next step is to formulate the properties in LTL. Finally, the model is verified against the properties.

With the aim to automate the process as much as possible, and starting from the specifications of the project, we will build an application to do the model extraction automatically.

Chapter 5

5 Formal Checking of Web Based Applications

The goal of the project is to create a prototype tool that automates the testing of web-based applications and to demonstrate the applicability of model checking techniques in the context of verification of web applications. We follow the process of model checking described by Edmund M. Clarke, Jr. et al., i.e., modeling, specification, and verification where we try to automate the modeling process.

5.1.1 Communication interception

In order to intercept the communication between a user of a web site and the web server we are using a proxy server. The proxy server must record in its log file the requests sent by the user (Internet Explorer, Netscape Navigator) to the web server and the responses of the last one back to the initiator. The navigation can be done manually for small models and with a crawler for large models.

The next step is to extract the useful information from the proxy server log file that can help us transform the surveyed web site in the desired model.

5.1.2 Java HTTP Proxy Server

The Java Proxy Server is an open source project developed by Stefan and Emily Reitshamer, available from [URL3]. The proxy reads a client's HTTP request, forwards the request to the origin server specified in the "start line" of the request,

reads the HTTP response from the origin server, and forwards it to the client. The implementation included here follows the HTTP protocol for sending and receiving messages among distributed objects.
All the code is written in Java, is well documented, and is "open source". HTTP requests of all content-types are handled correctly.

The Java Proxy Server is used to proxy HTTP requests. To use the proxy server, we simply have to set the web browser to use the proxy server at the host and port.
To do this in Internet Explorer, we select Tools/Internet Options.../Connections/LAN Settings and check Use a proxy for your LAN and for Netscape Navigator, we go to Edit/Preferences.../Advanced/Proxies and check Manual proxy configuration where we set the name of the machine where the proxy is installed and the port 8080 for HTTP proxy.

This proxy server creates the log file as output file, which provides the information needed for further processing. However, for the sake of accuracy, it should be mentioned that the proxy server does not classify the information that it receives. It is not aware of the web site and page visited. In this case, the user who wants to use our application and this proxy server should correctly operate within the chosen site for an accurate model.

To access a page from a certain server we need to generate a request for this page. The request could be initiated by the user clicking on a hypertext anchor pointing to the file, for example

```
<a href="http://www.crim.ca/">Home Page</a>
```

The actual data sent by a client to the server and registered in the proxy server's log file is shown in Figure 15.

```
GET http://www.crim.ca/index.epl?selec=2300&href=/rd/telecom.htm HTTP/1.0
Host: www.crim.ca
Accept: application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint,
image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 4.0)
Accept-Language: en-us

----------END OF HTTP REQUEST----------
```

Figure 15 HTTP Request

The request message consists of a *request header* containing several *request header fields.* Each field is a simple line of text, terminated by a carriage-return linefeed character pair (CRLF). The blank line (containing only CRLF pair) at the end of the collection of header fields indicates the end of the header and the beginning of the *data* being sent from the client to the server, (POST requests in the example). So the blank line is the end of the request [6].

The request message contains two parts. The first part, namely the first line of the request, is the *method* field, which specifies both, the HTTP method to be used and the location of the desired resource on the server. This is followed by the server HTTP *request* fields, which provide information to the server about the capabilities of the client, and about the nature of the data, if any, being sent by the client to the server. This request, as shown on the upper example, is registered in the proxy server's log file.

The information that we need to extract from the request is the method field: specifically the exact the location of the requested resource on the server.
After the request is registered, the corresponding response will follow as shown in Figure 16.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 18316
Server: Apache/1.3.9 (Unix) mod_perl/1.21 mod_ssl/2.4.9 OpenSSL/0.9.4
Date: Wed, 10 Apr 2002 19:40:02 GMT

<HTML>
<HEAD>
<LINK rel="stylesheet" href="/styles.css">
...
<TITLE>CRIM : Développement de réseaux de télécommunications</TITLE>
...
<A href="/index.epl?selec=2700&href=/rd/personnes/alexandre_petrenko.htm">
Alexandre Petrenko</A>
....
<AREA href="/index.epl?href=/visite/index.htm" coords="79, 11, 143, 33"
shape="rect">....
</HTML>
----------END OF HTTP RESPONSE----------
```

Figure 16 HTTP Response

When the server receives the request, it tries to apply the designated method to the specified object (file or program), and passes the results of this effort back to the client. The returned data is preceded by a *response header*, consisting of *response header fields,* which communicates information about the state of the transaction back to the client. As with the request header fields sent from the client to the server, those are single lines of text terminated by a CRLF, while the end of the response header is indicated by a single blank line containing only a CRLF. [5]

The data of the response follow the blank line, as we can see from the upper example extracted from the proxy server's log file. The header of the response and the data sent from the server to the client is completely registered in the log file.

In the response case, the data that interests us consists in the "href" found in the HTML page as in the example shown in Figure 16 and textual information that can characterize the state.

Each request has a response but the data may not be registered on the log file if the response is a GIF file for example or if the desired document was moved from the known location or deleted.

Figure 17 represents an example of an image request and its response extracted from the log file:

```
GET http://www.crim.ca/img/top_left5.gif HTTP/1.0
User-Agent: Mozilla/4.76 [en] (WinNT; U)
Referer: http://www.crim.ca/
Accept-Charset: iso-8859-1,*,utf-8
Accept-Language: en
Accept-Encoding: gzip
Cookie: SID=5cbed9537305f759
Host: www.crim.ca

Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png

-----------------------END OF HTTP REQUEST--------------------------
HTTP/1.1 200 OK
Content-Type: image/gif
Content-Length: 8243
Last-Modified: Fri, 25 Jan 2002 14:57:47 GMT
Server: Apache/1.3.9 (Unix) mod_perl/1.21 mod_ssl/2.4.9 OpenSSL/0.9.4
Date: Mon, 18 Feb 2002 18:40:58 GMT
Accept-Ranges: bytes
ETag: "c2b6d-2033-3c51726b"

(BINARY DATA – NOT SHOWN)
-----------------------END OF HTTP RESPONSE-------------------------
```

Figure 17 HTTP Request/Response for an Image

5.2 The Model

In this section we give more details about the model that we use.

5.2.1 FSM in XML

In this section we explain the representation of a web site as a finite state machine in order to understand the next steps and we get into details later on, when we discuss the design and implementation.

We want to represent the pages as states of finite state machine and the links as transitions of the finite state machine.

Since we created the first version of the tool, it was not clear which model checker best matched with our goals. As a result, we have created an application to be flexible at the later decisions and/or changes. So we needed an intermediary data structure which can be easily translated in another kind of representation.

We have decided to use XML file where we can stock the first format of the model extracted from the web site, the finite state machine (i.e., with explicitly defined states and edges).

As we already mentioned, we want our application to be flexible with respect to the choice of the model checking tool. As consequence before creating the model in the language accepted by the model checking tool of our choice, we first want to output the model representing the web site to be checked in a standard format easy to modify and transform.

In this section we explain how we represent a web site as a finite state machine. We are creating an XML intermediary data structure which reflects the finite state machine architecture of the web site (i.e. with explicitly defined states and edges).

We are focusing mainly on the navigational properties of the site, expressed by links between pages. Starting from the log file of the proxy server we can extract the information that we need to create a finite state machine. We refer to the content of pages as states, and to transition from page to page (which are possible without browser navigation facilities) as edges.

We described above the structure of the data in the log file. And as we said every request causes a response. Starting from the proxy server log file we extract from every HTTP response, whose body is an HTML page, all the links of page. We consider the requests that generated these pages as response, the transitions. There is a *transition* from a page to another (or there is an *edge* from a state to another) if there is a request for the second page and the first page has a link, that matches this request.

If the user decides to navigate, and we concentrate our attention in this direction, everything that user does is registered in the log file. We have to consider that the user can make mistakes and the pages requested from the other sites are not considered. On the other hand the user cannot try all possible navigation trajectories. However, some transitions, even those not performed by the user could be deduced, assuming that the content of the page does not change (i.e. URLs lead to the same pages). Such deduced transitions are discussed below.

Let us assume a page A that contains links b and c. From the page A, clicking on link b we can go to page B with success. Now we suppose that page B contains link c and clicking on link c we can go to page C with success. We can conclude that from page A, we can go to page C with success. In the site graph we denote such a transition with a dotted edge from A to C as labelled by c.

We can talk now about two kinds of transitions: *consumed transitions* and *deduced transitions*. *Consumed transitions* are the transitions, which conforms to the definition of the transition and the user performed them. The *deduced transitions* are the transitions that conform to the definition of the transition but the user never performed these transitions (as in the situation described in the previous paragraph).

This explanation is important to understand how the edges are constructed in the XML file. We will explain now the algorithm of XML file construction.

We are processing the proxy log file, page by page. If a link is present in a page and we have a request for a new page corresponding to this link, then between these two pages/nodes we have an edge labelled with the request.

First let us consider a simple example site graph. Suppose that with the help of a browser we generate a request for the home page of CRIM web site `GET http://www.crim.ca/` and we navigate from the obtained page to another one by clicking one of the encountered links.

The nodes represent the pages and edges represent the links (Figure 18). The dotted edges correspond to possible transitions, solid edges correspond to transitions performed by the user. The content of edges is not shown for clarity of the figure.

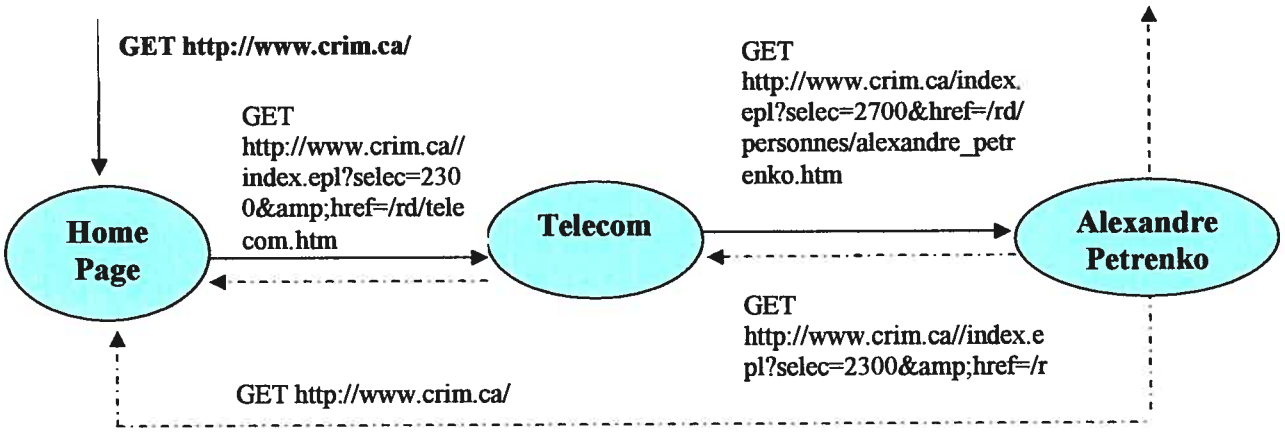


Figure 18 A Simple Site Graph

This graph represents a partial model of CRIM's web site. The intermediary XML file contains the information extracted from the proxy's log file, and represents only the pages and edges with the related information.

For this graph we can give some examples of browsing properties expressed with linear temporal logic (LTL).

"Each time we hit Home page, eventually we can hit Telecom page"

To express this using LTL the events "hit Home page" and "hit Telecom page" must be specified. If they are given the name 'p' and 'q' respectively, this is expressed as follows in LTL:

$$\Box(p \rightarrow \Diamond q),$$

where $\Box$ denotes always, $\rightarrow$ denotes implies and $\Diamond$ denotes eventually.

It is also possible to specify the behaviour that should never occur in any navigation. This can be defined as follows:

"Eventually it should not happen that when we hit the Home page eventually we can hit Alexandre Petrenko page"

This can be expressed in LTL using the same name for the events, where ! denotes *negation*:

$$\Diamond !(p \rightarrow \Diamond q)$$

We'll see more examples of properties in subchapter 5.2.4.1.

5.2.2 DTD

XML offers an adaptable standardized markup language for describing documents according to a given structure Document Type Definition (DTD). Therefore it will be sufficient to look at the currently implemented DTD file to understand the structure of our XML file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT site ( #PCDATA | vector | edge | request | page )* >
<!ELEMENT href ( #PCDATA ) >
<!ELEMENT form ( #PCDATA ) >
<!ELEMENT edge ( #PCDATA ) >
<!ATTLIST edge Start NMTOKEN #REQUIRED >
<!ATTLIST edge End NMTOKEN #REQUIRED >
<!ATTLIST edge Label CDATA #REQUIRED >
<!ATTLIST edge edgeId NMTOKEN #REQUIRED >
<!ELEMENT vector ( element* ) >
<!ATTLIST vector Id NMTOKEN #REQUIRED >
<!ELEMENT request ( #PCDATA ) >
<!ELEMENT element ( #PCDATA ) >
<!ELEMENT string ( #PCDATA ) >
<!ATTLIST string Occurence NMTOKEN #REQUIRED >
<!ELEMENT page ( href*, form*, string ) >
<!ATTLIST page Id NMTOKEN #REQUIRED >
<!ATTLIST page Code CDATA #FIXED "N/A" >
<!ATTLIST page Request CDATA #REQUIRED >
```

Figure 19 The DTD file for the implemented model in XML

In the following we describe the meaning of each sentence from the DTD file.

<!ELEMENT site (#PCDATA | vector | edge | request | page)* >: For every site (the root of the XML file) we have pages, requests, edges and the vectors which characterize the presence of the links in pages. We give more details below. The "*" sign in the example above declares that the child element message can occur zero or more times inside the state element.

<!ELEMENT page (href*, form*, string*) > Every page has zero or more links (href's), zero or more forms and zero or more strings.

<!ELEMENT vector (element*) >

<!ELEMENT href (#PCDATA) > !ELEMENT href defines the "href" element to be of the type "PCDATA".

<!ELEMENT edge (#PCDATA) > !ELEMENT edge defines the "href" element to be of the type "PCDATA".

<!ATTLIST page Request CDATA #REQUIRED >: For every page we chose five attributes: the Id of the page, a number, the request for this page, the URL which points to this page and the code which is the error code returned by the server.

<!ATTLIST vector Id NMTOKEN #REQUIRED >: The vector contains one or more elements and the label in common with the page that it characterizes. It is the page number in the order of apparition in the log file.

<!ATTLIST page Id NMTOKEN #REQUIRED >

<!ATTLIST page Code CDATA #FIXED "N/A" >

<!ATTLIST edge Start NMTOKEN #REQUIRED >

<!ATTLIST edge End NMTOKEN #REQUIRED >

<!ATTLIST edge Label CDATA #REQUIRED >

<!ATTLIST edge edgeId NMTOKEN #REQUIRED > : Edge element has four attributes: "edgeId" representing the ID of the edge, a number given to identify the edge in the order in which the edge was found; "Start" represents the Id of the page from where the request was generated; "End" represents the Id of the requested page

and "Label" is the URL of the requested page.

(#PCDATA) stands for parsed character data. It's the tag that is shown and also will be parsed (interpreted) by the program that reads the XML document.
(#CDATA) stands for character data. CDATA will not be parsed or shown
The first character of an NMTOKEN value must be a letter, digit, '.', '-', '_', or ':'.
#REQUIRED means that the attribute must always be included - validity constraint.
#IMPLIED: the attribute does not have to be included.
#FIXED or "Default_Value": the attribute must always have the default value that is specified by a validity constraint. If the attribute is not physically added to the element tag in the XML document, the XML processor will behave as though the default value does exist. In our case "N/A" means that the server did not send back to the client an error code.

Now we explain the role of the vectors in this structure. As we explained before, we extracted from every page the links (href's). With these links we create another XML file which contains all the links from all visited pages that belong to this site.
The DTD of this XML file is described below :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT href ( #PCDATA ) >

<!ELEMENT UniqueListOfHrefs ( #PCDATA | href )* >

<!ELEMENT processing ( UniqueListOfHrefs ) >
```

After this file is created, we can create the lists with the links number that characterizes the presence of the links in a particular page. We need these lists to keep the record of the links that we have in a page. The numbers represent the position of the link that we can find in the file, where the latter contains all the links of the.

The content of a state in the XML file is characterized by the links extracted from the page if there is any and the textual information that interests us if there is any, described as "string" in the DTD.

Following is a fragment of an XML file representing our model, shown to better understand the meaning of the described DTD.

```
…
<request>GET http://www.crim.ca HTTP/1.0</request>

<page Request="http://www.crim.ca" Code="N/A" Id="0">
…
   <href> http://www.crim.ca/index.epl?href=/visite/index.htm</href>
   <href> http://www.crim.ca//index.epl?selec=2300&amp;href=/rd/telecom.htm </href>
   <href>http://www.crim.ca/</href>
…
   <string Occurence="0">Alexandre</string>
</page>

<request>GET http://www.crim.ca//index.epl?selec=2300&amp;href=/rd/telecom.htm HTTP/1.0</request>

<page Request=" http://www.crim.ca//index.epl?selec=2300&amp;href=/rd/telecom.htm " Code="N/A"
Id="1">…
    <href>http://www.crim.ca/index.epl?href=/visite/index.htm</href>
    <href>http://www.crim.ca/index.epl?href=/coordonnees.htm</href>
    <href>http://www.crim.ca/index.epl?selec=2700&href=/rd/personnes/alexandre_petrenko.htm</href>
    <href>http://www.crim.ca/</href> …
    <string Occurence="3">Alexandre</string>
</page>

<request>GET http://www.crim.ca/index.epl?selec=2700&href=/rd/personnes/alexandre_petrenko.htm
HTTP/1.0</request>

<page    Request="http://www.crim.ca/index.epl?selec=2700&href=/rd/personnes/alexandre_petrenko.htm"
Code="N/A" Id="2">
…
   <href> http://www.crim.ca//index.epl?selec=2300&amp;href=/rd/telecom.htm </href>
   <href>http://www.crim.ca/index.epl?href=/coordonnees.htm</href>
   <href>http://www.crim.ca/</href>
…
   <string Occurence="6">Alexandre</string>
</page>

<edge edgeId="2" Label="http://www.crim.ca/index.epl?href=/menus/menu_all.inc" Start="2"
End="0">Start = source page Id =2 End = requested page Id 0</edge>

…

<edge edgeId="13" Label="http://www.crim.ca/index.epl?selec=2300&amp;href=/rd/telecom.htm"
Start="0" End="1">Start = source page Id =0 End = requested page Id 1</edge>
…
<edge                                                                    edgeId="25"
Label="http://www.crim.ca/index.epl?selec=2700&amp;href=/rd/personnes/alexandre_petrenko.htm"
Start="1" End="2">Start = source page Id =1 End = requested page Id 2</edge>…
```

Figure 20 The XML Partial Model

Figure 20 shows the code in the XML file for the model in Figure 18.

We can observe in Figure 18 and Figure 20 that the first page with the Id="0" and the attribute "Request = "http://www.crim.ca/" contains the link which leads to telecom.htm page. Clicking on the anchor text corresponding to this link on the original HTML document generates the request:
GET http://www.crim.ca//index.epl?selec=2300&href=/rd/telecom.htm

In this instance we can say that from the page with Id="0" to the page with Id="1", there is an edge generated by the new event, namely the click with the mouse on the anchor text corresponding to that link. Similarly, from the page with Id="1", we have a request for alexandre_petrenko.htm, the page with Id="2", so we create the new edge from page 1 to page 2 (<edge edgeId="25" Label= "http://www.crim.ca/index.epl?selec=2700&href=/rd/personnes/alexandre_petrenko.htm" Start="1" End="2">Start = source page Id =1 End = requested page Id 2</edge>). Those pages were sent successfully from the server to the client and the code is set to N/A. In case of a failure we could have one of the error codes (404, 304 etc).

As observed from the XML file we have in page "2" the link which can generate a request for page "0". This event never occurred in this testing session (it was not registered in the log file). Nevertheless, we can conclude that *we can have* an edge from page "2" to page "0" because there is a link in page 2 which can lead us to page 0. We represented the deduced edges in the graph with dotted lines (for clarity, not all deduced edges are shown in Figure 18 and Figure 20).

Thus, the *conditions* to have an *edge* from a HTML page to another HTML page are:

a) to have a link in the source HTML page which corresponds to the HTML destination page;
b) to have a HTTP request for the destination page (*real edge*) or the destination page was requested before and registered in the log file (*deduced edge*);

The method of creating the XML file can be outlined as follows:

1. If the source page and the destination page are encountered for the first time in the log file and a request is generated from the source page to the destination, page, then register the corresponding nodes (with their links) and the edge in the XML file.

2. If the source page and the destination page are already registered in the XML file and an edge between them is registered, no action is performed.

3. If both the destination and source pages were registered in the XML file but the source page contains a link to the destination page, then we just create a new edge tag which binds these pages in the XML file;

4. Lastly, when only one of them is encountered on the log file before a node was created in the XML file, and then we register the new one and create a corresponding edge.

With this method, we can construct the graph corresponding to a chosen web site.
We identify a page only by its URL. Thus, to check if a page is already in the XML file or not, we can compare the new request with all the other requests already registered in the graph. If the page has the same URL then with one already registered, then we can say that we have already seen this page and we are not creating another node in the XML file (thus, we don't have redundant information in the site graph). However, we can not say that we found the same page if we have the same content but a different request. So the pages with the same link list but with different requests are registered separately in the XML file, and as a consequence we will have a new node in the graph.

Defining a page not only by its URL but also by a linked list may be inadequate in case of sites which change their content rapidly, like news sites, etc. For such sites,

pages should be identified only by URLs, or may be by a URL with a list of essential links.

5.2.3 FSM in PROMELA

After describing the finite state machine in XML format we need to describe it in a specification language of the model checker of our choice.

5.2.4 Formal Properties

The last step is to specify the properties using the specification language of the model checker and then to run the model checker with the finite state machine and the property to check.

Figure 21 is a schema of the transformation from a state and a transition described in XML to a state and transition in PROMELA.

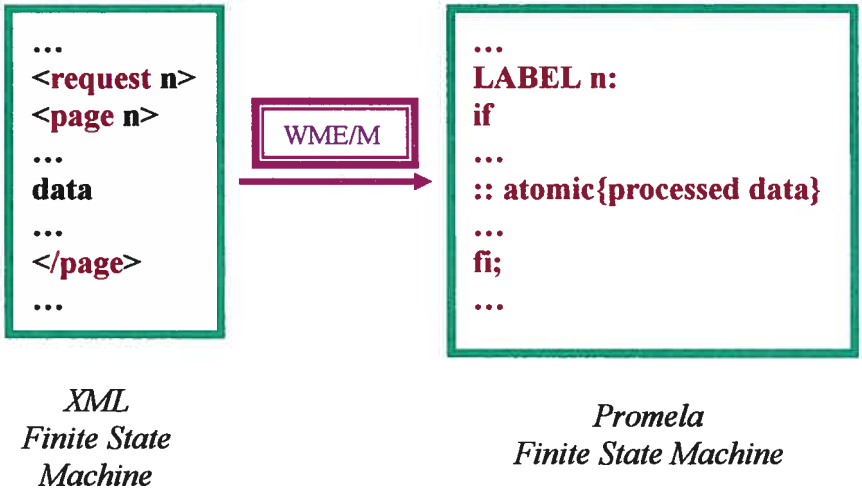


Figure 21 XML to PROMELA with Web Model Extractor/Manipulator

We define a set of logical properties to be checked against web applications. These properties are classified into four categories: browsing properties, connectivity

properties, frame properties, and quantitative properties. For each category, we present the possible properties to be checked.

5.2.4.1 Browsing Properties

These properties refer to the browsing of a web site and they are independent of the browser's navigational functions.We will present some browsing properties examples:

At start of your navigation on a web-site, you must encounter the home page.
LTL formulae: □p where p is "*home page*"

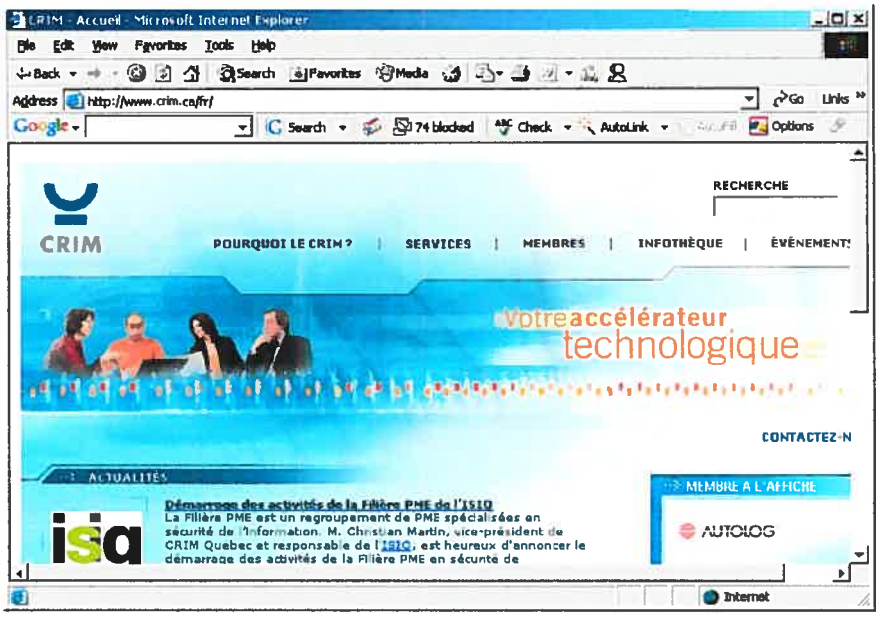


Figure 22 CRIM's home page

Every company's web site should start with a home page which presents minimum information about their company and menu and links which lead to more choices/information and even other web-sites.

From any page you have a link out.
LTL formulae: □(p→□q), where p is "*any page*" and q is a "*link out*" or □q

There is no commercial website created with one page which leads nowhere. From every web-page we should have at least one link to another page. A web site which presents a business should create their web-site with enough information to make it appealing to clients.

Example: from CRIM's home page we could go to Services as in the screenshot below.

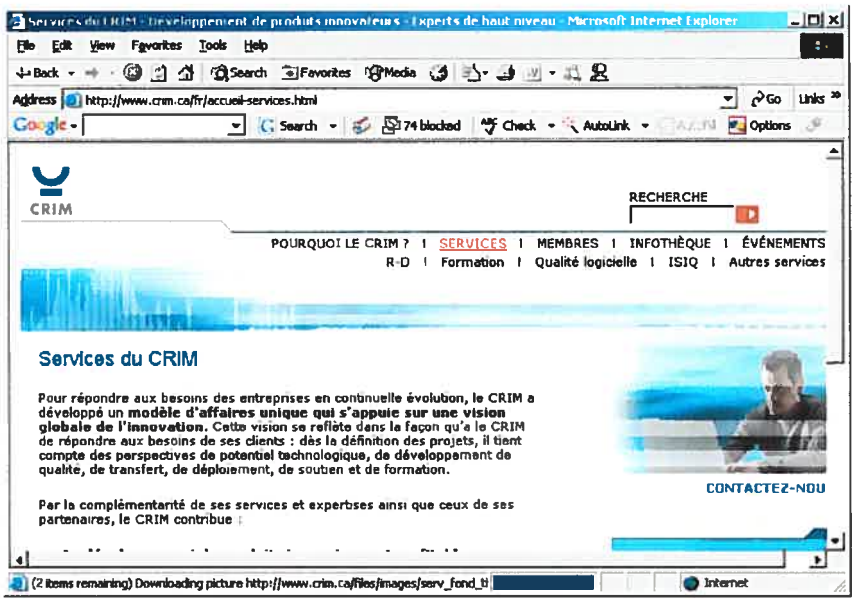


Figure 23 CRIM's Services page

A counter example is www.zombo.com who has one page on his web-site:

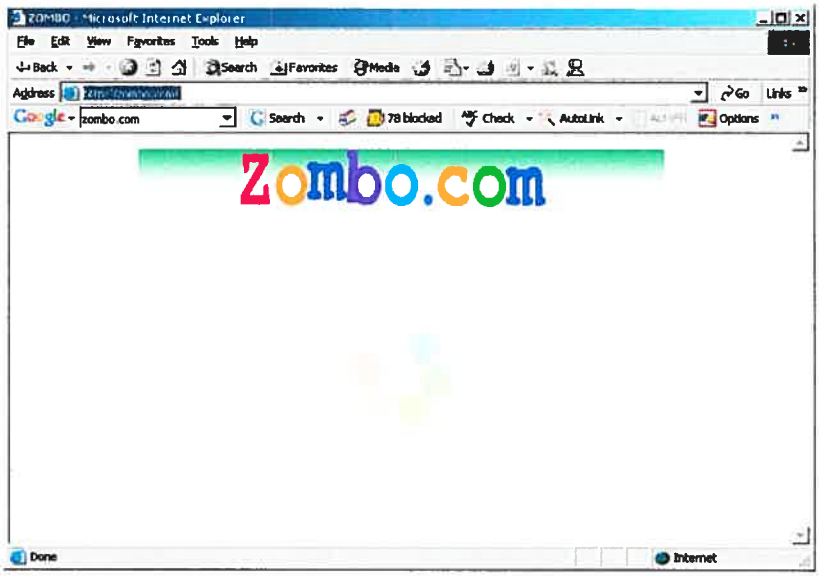


Figure 24 One page web site

From any page you can go to the home page.

LTL formulae: $\Box(p \rightarrow \Box q)$, where p is "*any page*" and q is "*home page*".

It is good practice to have a link to home page from any other page of the web-site. This browsing property gives the user the possibility to go back to the main menu and have more browsing choices, increasing the visibility of the company who presents its business.

In all situations, a certain button/anchor a can be eventually selectable (in the form's case).

LTL formulae: $\Box(p \rightarrow \Diamond q)$, where p is "*certain button/anchor a*" and q is "*selectable*"

It is possible for a certain button/anchor a to be eventually disabled (in the form's case).

LTL formulae: $\Diamond (p \rightarrow \Diamond q)$ where p is "*button/anchor a*" and q is "*disabled*"

Example:

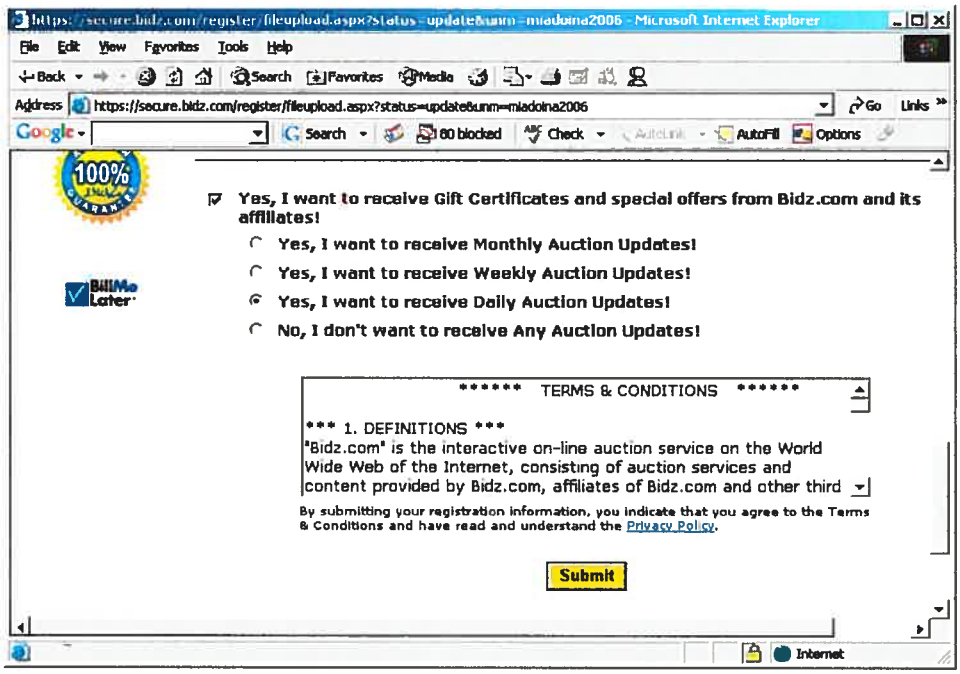


Figure 25 Enable/disable objects

In the previous screenshot we have the choice to receive information about the Gift Certificates from www.bidz.com web site or not. It adds business value to the company in the case of customers who want to give presents to their friends or family gift certificates to buy jewelleries.

In all forward browsing sessions, the menu/table of content is eventually visible.

LTL formulae: $\Diamond(p \rightarrow \Diamond q)$, where p represents "*forward browsing sessions*" and q the property to be "*visible*".

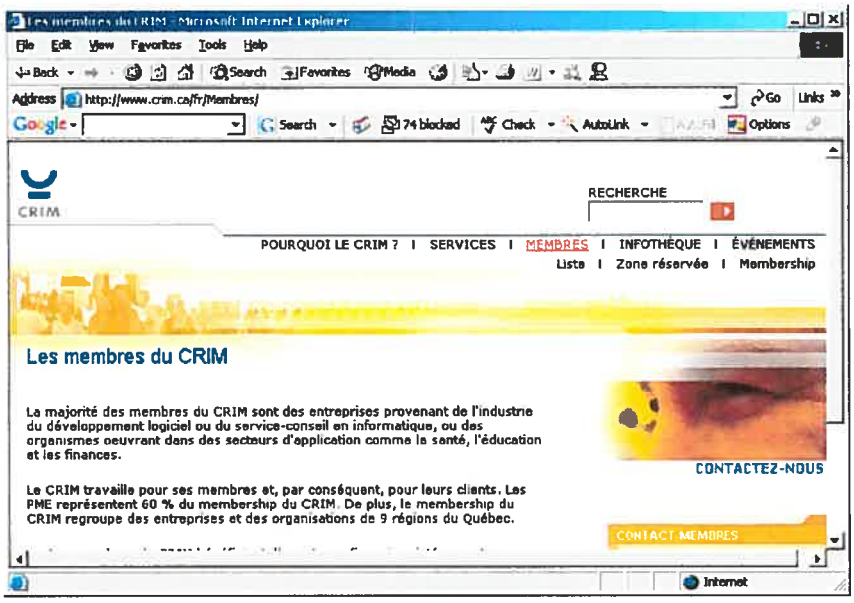


Figure 26 CRIM's Members page

We can notice in the previous web page example that we have the same menu as the one found in home page. This gives more browsing flexibility to the user who can go from the *Members* page to any other choice from the menu instead of going back to the home page.

Immediately after the first occurrence of a φ, ψ will happen.

LTL formulae: $\varphi \rightarrow O\psi$

For example, after you fill up a form you will encounter a Submit button.

5.2.4.2 Connectivity Properties

Connectivity properties correspond to the structure of the web site. Consider a domain Δ, where Δ can be a web site address; for example, www.crim.ca. The properties include:

- In a web site, there is a certain page such that the URL from which it is retrieved contains the substring Δ. For example, the research and development web page at CRIM is derived from the URL http://www.crim.ca/rd/index.htm. This page is within the domain of CRIM since its URL contains the substring www.crim.ca .
- Another property is to ensure that all the links within the web site point to pages in its domain. It can be formulated as follows. All URL pages are such that the URLs from which they are retrieved contain the substring Δ. This property forces the author to exclude from his web site any link to external web sites. For example, in CRIM web site, www.crim.ca , this property will not hold since there exist a link to FCAR funds web site, http://www.fcar.qc.ca , which is external to CRIM's domain. A web administrator can enforce the design of a web-site to remain within the domain in the situation when the employees of the company have no Internet connection just Intranet, so he/she doesn't want to have broken links on their company web-site.
- In a web site, while loading some URL page, the HTTP error numbered *k* occurs where a HTTP error number is a status code delivered in the header of the server's response. Every status code refers to a message describing the status of the server's response such as successful transactions, redirection transactions, and error messages. This is part of every development error management system. You want to have a user friendly message not a HTTP error code which is irrelevant for the user.
- Non-existing link targets: this property checks for dead-ends. In case the author uses URLs to point to a different web site or a different domain, then it is sufficient to check whether the targeted web site really exists or not and that the link to it is not broken.

Chapter 6

6 Web Model Extractor/Manipulator

In this chapter, we describe the functionalities of the Web Model Extractor/Manipulator. The subsequent sections contain the description of the architecture and the graphical user interface.

6.1 Functionalities

The prototype tool design contains nine classes as shown in Figure 27. Three of them: "MyFrame", "FrApplication" and "AboutBox" are built for the graphical user interface. The classes: "XMLTreatment", "CutLogFile", "OneList", "DoOro", "GetPostFilter" and "Html" are designated to implement the algorithm for the extraction of the necessary information from the proxy log file and to process this information in order to create the finite state machine using an XML format. The class "PROMELA" is used to translate the finite state machine from the XML format into PROMELA language used by SPIN model checking tool.

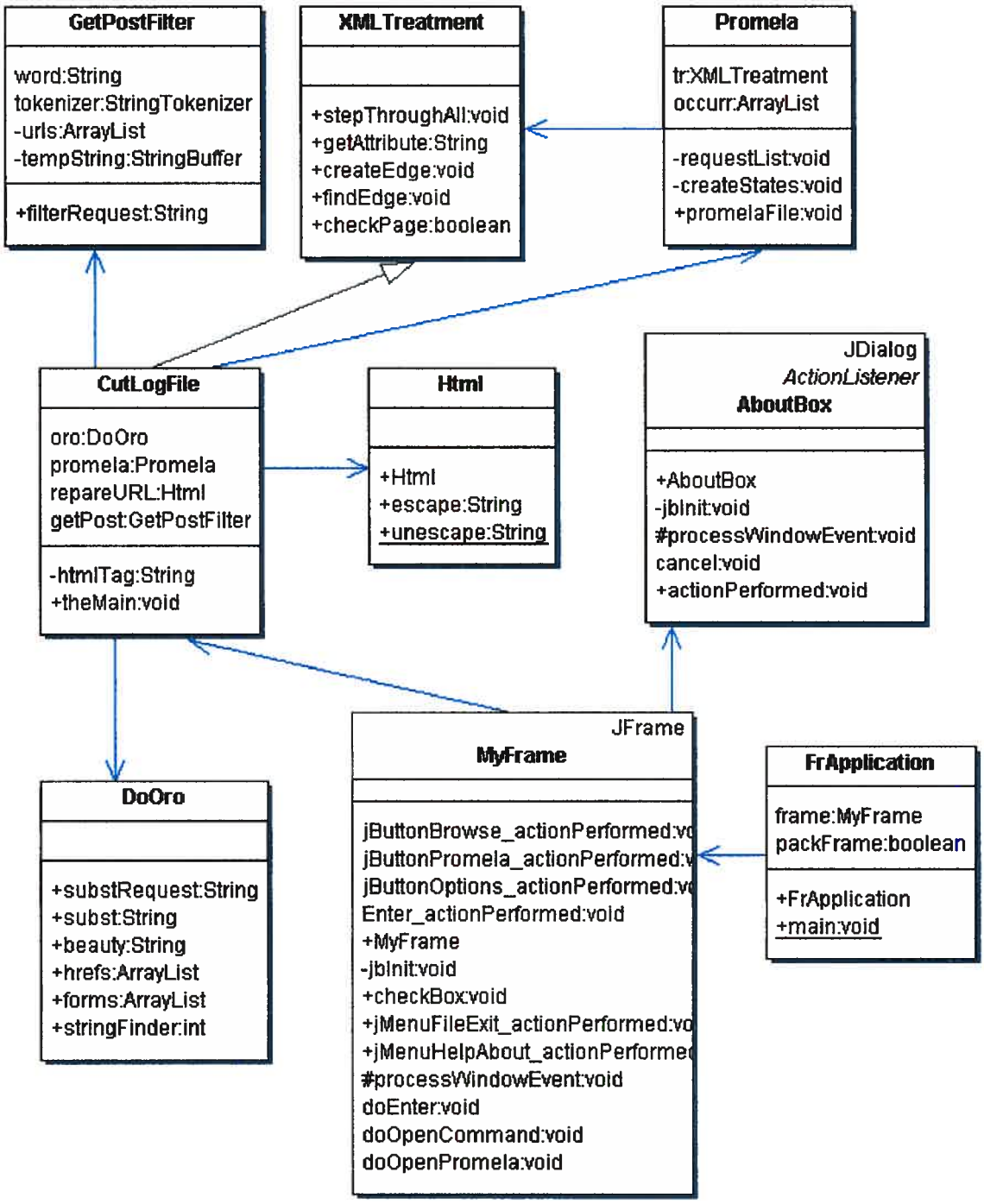


Figure 27 Class Diagram

The main function of the Web Model Extractor/Manipulator is processing proxy server's log file. The main steps of this function are as follows:

- The tool splits the proxy server log file into request/response pairs by finding the beginning and the end of the HTML pages contained in the log file (Figure 28).

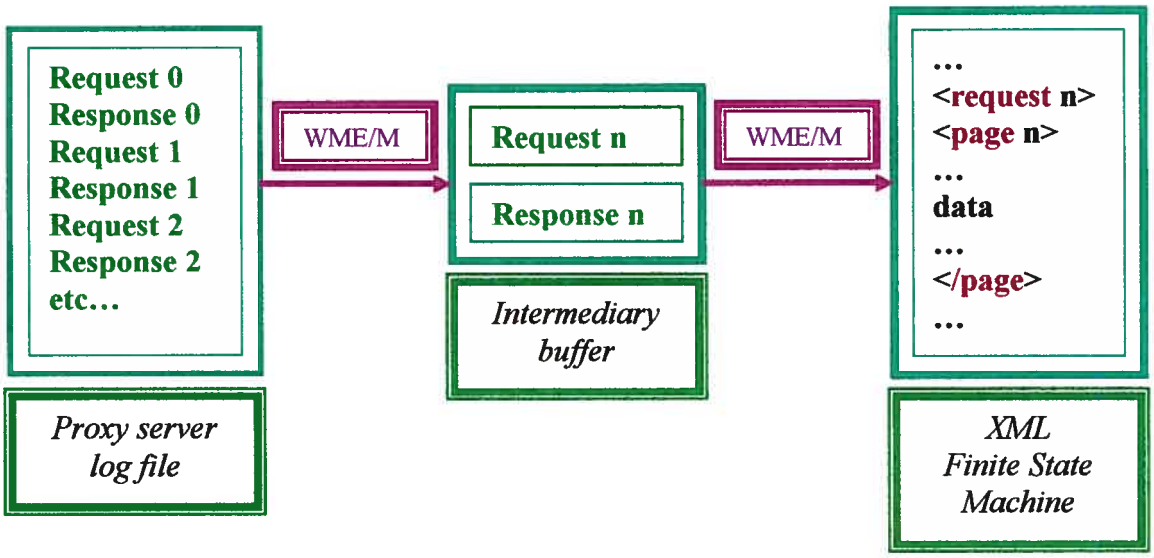


Figure 28 Processing of the proxy server log file

- The Web Model Extractor/Manipulator afterwards treats the HTML pages one by one in order to extract the necessary information that helps create the finite state machine. Currently, the focus is on extracting the hyperlinks and the number of occurrences of a selected string. We consider one single specimen from every hyperlink because we do not want to expand the model with duplicate transitions if there is no benefit.

- We also avoid creating duplicate states in the model for duplicate pages. Therefore, we check if the page already exists in the XML file. If the page is found, we skip its analysis. Otherwise, we extract the information needed to create a new state of the finite state machine.

To process the XML file we use a java parser from org.w3c.dom [URL7] that provides the interfaces for the Document Object Model (DOM), a component of the

Java API for XML Processing [URL8]. DOM is a tool for manipulating data, not a data structure itself. DOM is a memory representation of the data in a tree representation. It uses a pointer to the root node. Building the *Document* tree is an expensive operation, being a Java object equivalent of the whole XML tree, but it can iterate over the tree to look at the nodes and it also can edit the tree: add/remove nodes. The last step is to create the finite state machine in PROMELA. The states and transitions described in XML are mapped to the states and transitions in PROMELA language.

6.2 GUI

In this section, the graphical user interface (GUI) of the tool is described. The GUI is relatively simple as seen in Figure 29.

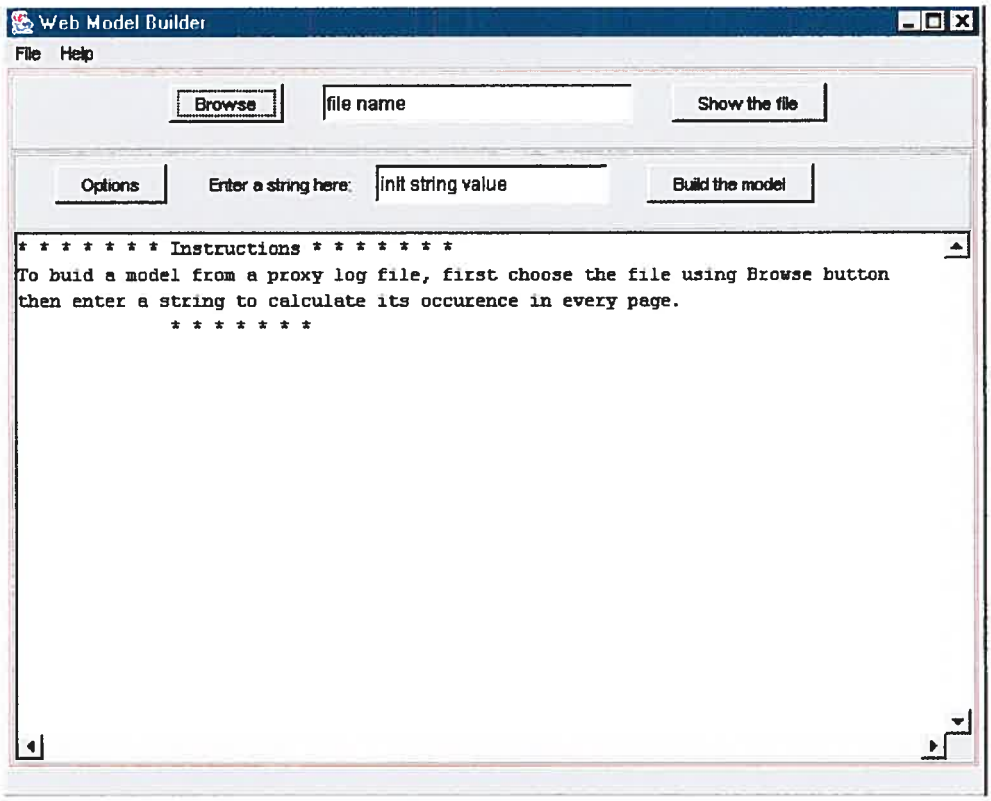
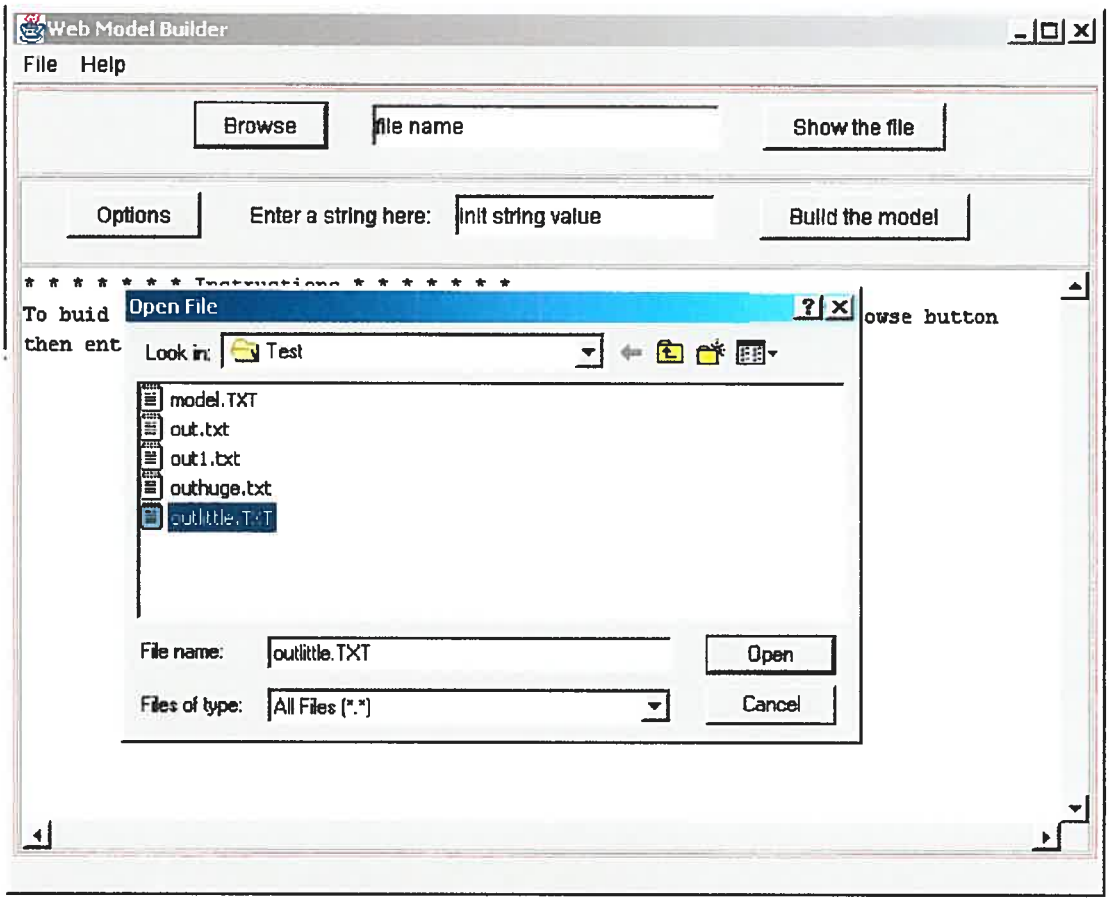


Figure 29 Web Model Extractor/Manipulator Window

It consists of one main window with four buttons: "*Browse*", "*Options*", "*Show the file*" and "*Build the model*".

With the *Browse* button, the user browses the files/directories structure and can choose the proxy log file to be processed.



Options button opens a new window, as seen in Figure 30 where the user can select one of two options: "*whole word*" and "*case sensitive*". These options are relative to the string to be searched that the user enters in the GUI.

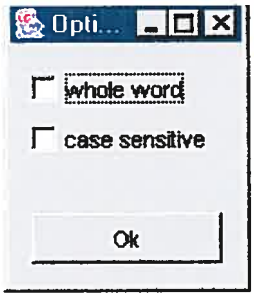


Figure 30 Option Window.

"*Show the file*" button displays the content of the chosen file on the text area from the interior part of the application's window, as in the following screenshot example:

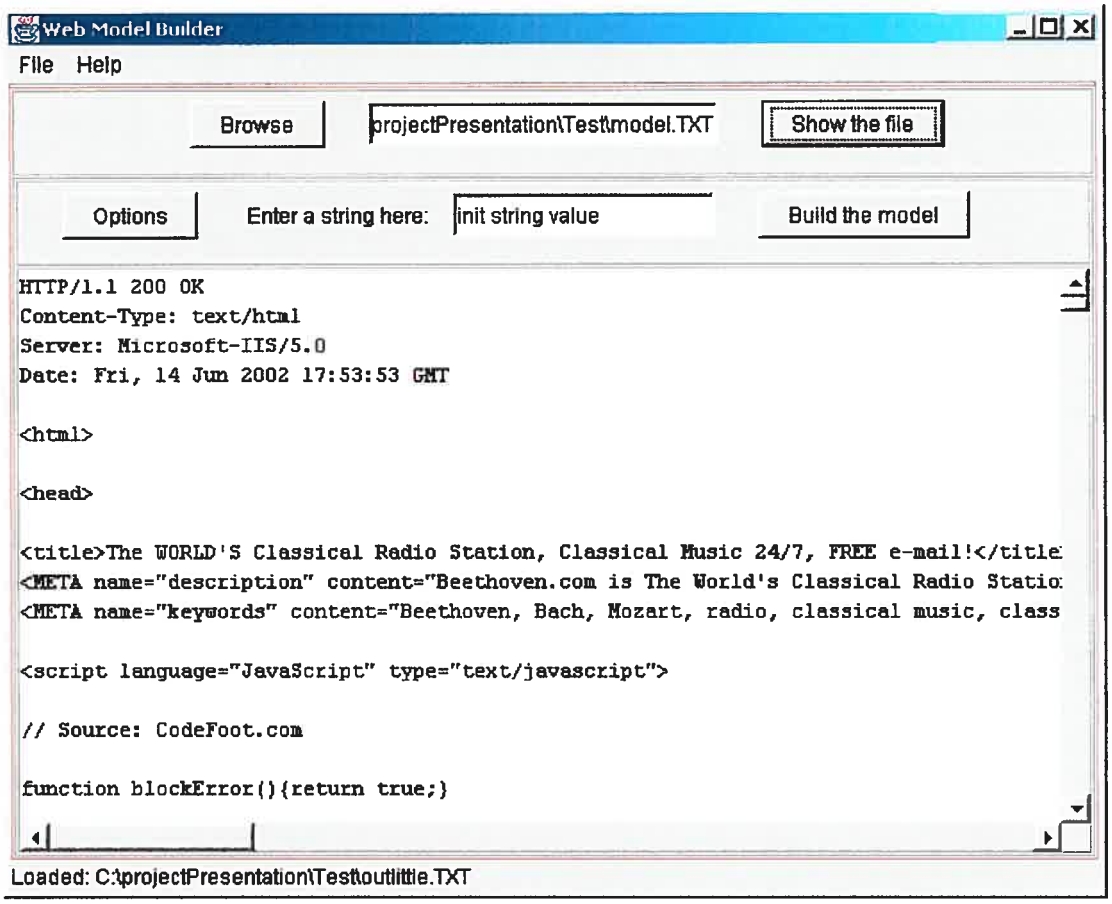


Figure 31 Show the file button

The last one, "*Build the model*" button, starts the main engine and extracts the final state machine from the proxy log file.

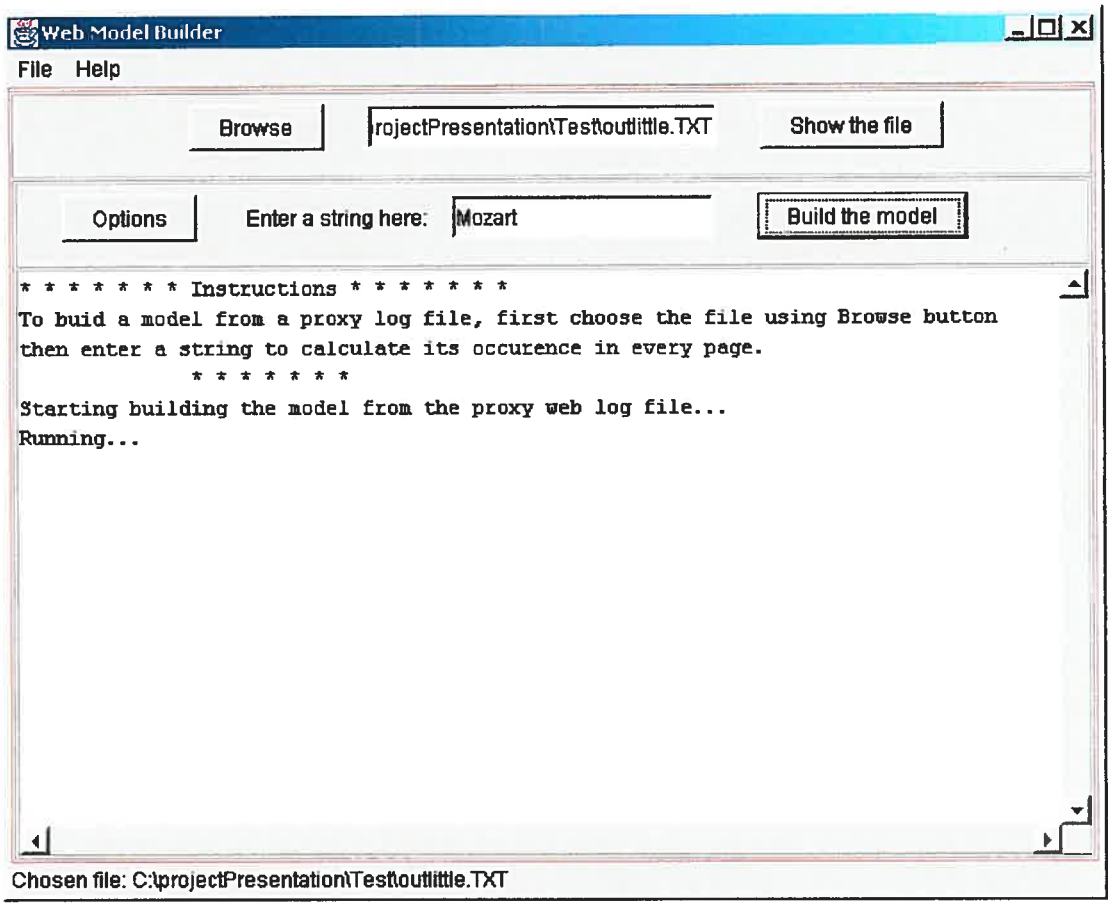


Figure 32 Build the model button

There are three fields: the one on the left of the Browse button displays the name of the chosen log file produced by the proxy server. The second text field is for the user to enter the string for which the number of occurrences is calculated. The third text field is the text area where we display the main steps executed by the application as it runs.

Chapter 7

7 Case Study

In this chapter we present a case study on which we applied our approach and framework. We select the web site of a classical music radio station "Beethoven" that has the following URL: www.beethoven.com.

The aim of this case study is to demonstrate applicability of formal methods for verification of Web based applications and the correctness of Web Model Extractor/Manipulator tool. We first present our browsing experiments with the web site. We then present the formal model extracted by the Extractor/Manipulator tool and represented as an automaton in SPIN. Then we formulate properties used for verification. We formulate the properties to be checked in LTL. Finally, we discuss the scalability of the tool to generate large models of web applications.

7.1 The "Beethoven" Radio Station Web Site

We surf the web site to observe the particularities of our test target, starting from the home page at www.beethoven.com. A snapshot of this page is shown in Figure 22.

Figure 33 Home Page from *www.beethoven.com* site

On the left side of the Home Page, we can observe that there is a menu which is supposed to help us navigate easily through the web site.

Looking at the left menu from the Home Page, Figure 33, we observe the item "Explore". Holding the "hand" cursor on this item, another menu is opened. This new menu holds another eleven items which can lead us to eleven pages, respectively: The Liszt, Culture Connection, Beethoven's Backyard, Photo Gallery, Beethoven University, Notes and Quotes, Ludwig Wallpaper, Link To Us, Start With Us, New Releases and Beethoven Movie. On the bottom of the page there is also a link called "Explore" that leads us to a page "Explore" shown in the next snapshot. This page has no relationship with the menu "Explorer" that was described before. Moreover the menu displayed in this page looks exactly like the menu encountered in the Home

Page, but is not the same. In the Explore page, the menu is composed of ten items: Now Playing, Music Log, E-Mail, Shopin Mall, Live! Webcam, Bulletin Board, Ludwig's Links, Contests, About Us and Help. Though the menus from previously described pages look similar, they do not lead to the same set of pages.

We also observe that a certain page named "Schedule" can be reached only starting from the Home Page. So, to navigate from "Explore" page to "Schedule" page, the user has to go through the Home Page.

7.2 Properties

We can formulate some browsing properties that could be checked against the model of this web site. For example:

- *"Is page Schedule reachable from Explore page without going through Home Page?" and*
- *"Is there at least a page from which page Schedule is reachable without going through the Home Page?"*

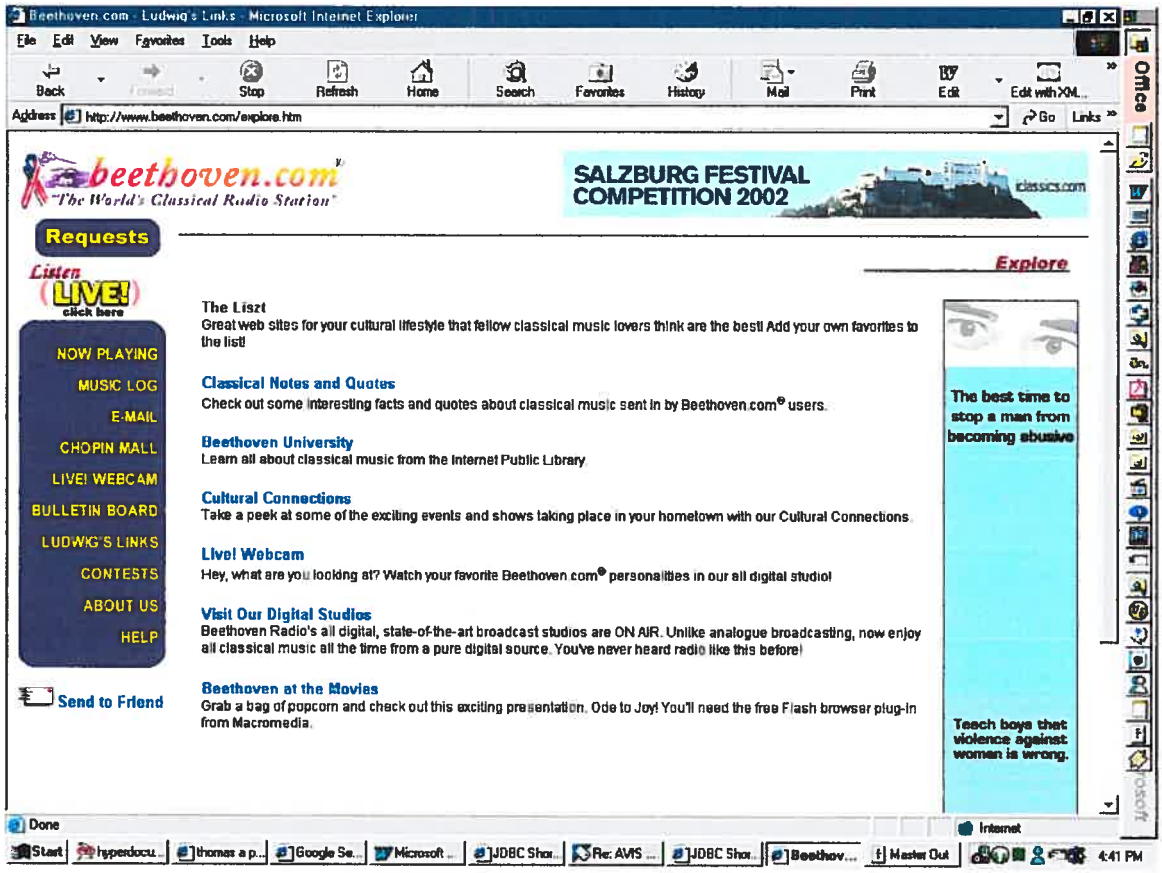


Figure 34 Explore page from *www.beethoven.com* site

We would like to check if a certain string, say "music", is not repeated more than a certain number of times. For instance, we want to check if "the number of occurrences of the string *music* in every page that we visited, is strictly less than a given threshold, let it be 6". If we combine this last property with the last formulated browsing property we can have a new one, now more complex: "There exists at least one page with occurrence of string *music* < 3 reachable from Explore page without going through the Home Page". After we establish what we want to verify in the web site, we extract the model and formalize the properties.

7.3 Formal Model

The next step is to extract a model for the web site case study. We use our prototype tool, the Web Model Extractor/Manipulator to generate a finite state machine that corresponds to the above mentioned web site.

During the navigation of the web site, the request/response pairs of the visited pages are collected in the log file of the proxy server - "out.txt".

The following is a snapshot of the GUI of the Web Model Extractor/Manipulator when executed using our case study.

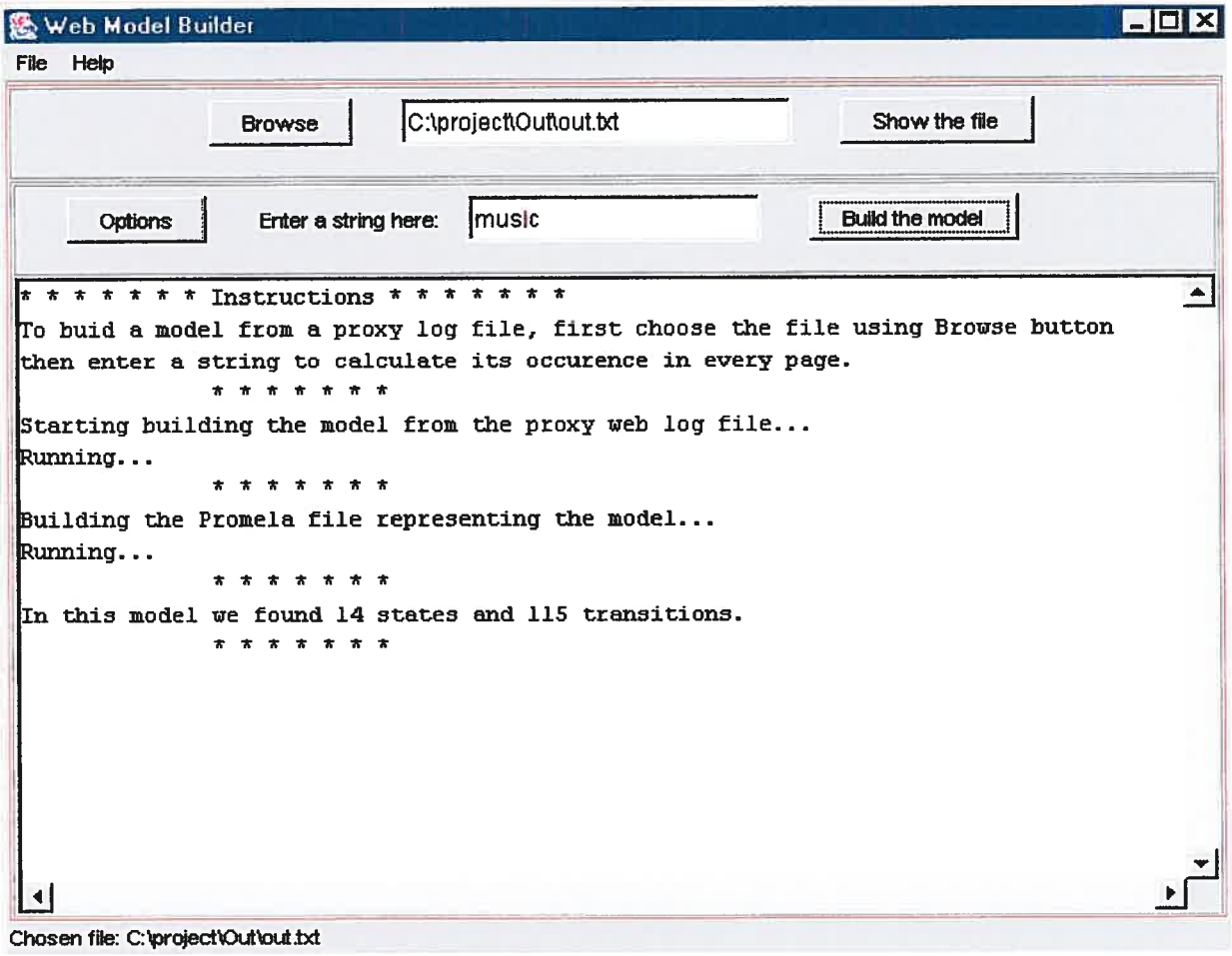


Figure 35 Web Model Extractor/Manipulator

We browse the files/directories structure and we have chosen the proxy log file to be processed (out.txt file).

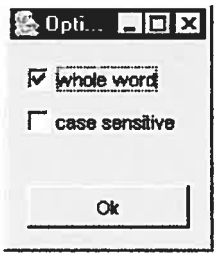


Figure 36 Option Window

We then select the "Option" button which opens a new window. Two options are available in this window: "whole word" and "case sensitive". We select "whole word" and we validate our selection with the "OK" button.

Then we write the string "music" for which we calculate the occurrence in every visited page and we click on the "Build the model" button. The main engine starts and extracts the finite state machine from the proxy log file. The main area of the GUI is a text area where the main steps executed by the application while running are shown.

The prototype tool first builds an XML FSM model from the proxy server log file. Since we explained the main steps of this module in the analysis and functionalities part, we only show the concrete results. Figure 37 shows a fragment of the finite state machine expressed in XML format.

```
<?xml version="1.0" encoding="UTF-8"?>
<site>
<request>GET http://www.beethoven.com:80/ HTTP/1.0</request>
<page Request="http://www.beethoven.com" Code="N/A" Id="0">
    …
    <href>http://www.beethoven.com</href>
    <href>http://www.beethoven.com/notes.htm</href>
    <href>http://www.beethoven.com/help.htm</href>
    <href>http://www.beethoven.com/schedule.htm</href>
    <href>http://www.beethoven.com/store.htm</href>
    <href>http://www.beethoven.com/contact.htm</href>
    …
    <string Occurence="2">music</string>
</page>

<request>GET http://www.beethoven.com:80/Assets/menu/images/aboutus_f2.gif HTTP/1.0</request>
<request>GET http://www.beethoven.com:80/contest.htm HTTP/1.0</request>

<page Request="http://www.beethoven.com/contest.htm" Code="N/A" Id="1">
    …
    <href>http://www.beethoven.com/contest.htm</href>
    <href>http://www.beethoven.com/about.htm</href>
    <href>http://www.beethoven.com/contest.htm</href>
    …
    <string Occurence="2">music</string>
</page>
…
</site>
```

Figure 37 The model – XML Format

The next step of the execution is to translate the XML model to the PROMELA language which is understood by the model checking tool that we have chosen, SPIN. In Figure 38 we show a fragment of the PROMELA model created from the XML file representing the classical music radio station site.

We can observe here a non-exhaustive list of the hit links and 2 of the 14 states.

```
/*t0 = http://www.beethoven.com,
…
t4 = http://www.beethoven.com/explore.htm,
…
t11 = http://www.beethoven.com/schedule.htm,
t12 = http://www.beethoven.com/newsletter/newsletter.htm,
t13 = http://www.beethoven.com/bbackyard.htm;*/
bit t0= 1, t1= 1, t2= 1, t3= 1, t4= 1, t5= 1, t6= 1, t7= 1, t8= 1, t9=
1, t10= 1, t11= 1, t12= 1, t13= 1;
byte state =0, strocc = 2 ;
active proctype website()
{
goto LABEL0;
LABEL0:
If
:: atomic{ t0 ->strocc = 2; state =0; goto LABEL0;}
:: atomic{ t1 ->strocc = 2; state =1; goto LABEL1;}
:: atomic{ t2 ->strocc = 2; state =2; goto LABEL2;}
:: atomic{ t3 ->strocc = 4; state =3; goto LABEL3;}
:: atomic{ t4 ->strocc = 4; state =4; goto LABEL4;}
:: atomic{ t5 ->strocc = 0; state =5; goto LABEL5;}
:: atomic{ t7 ->strocc = 4; state =7; goto LABEL7;}
:: atomic{ t8 ->strocc = 1; state =8; goto LABEL8;}
:: atomic{ t9 ->strocc = 3; state =9; goto LABEL9;}
:: atomic{ t10 ->strocc = 2; state =10; goto LABEL10;}
:: atomic{ t11 ->strocc = 6; state =11; goto LABEL11;}
:: atomic{ t12 ->strocc = 3; state =12; goto LABEL12;}
:: atomic{ t13 ->strocc = 8; state =13; goto LABEL13;}
fi;
LABEL1:
If
:: atomic{ t0 ->strocc = 2; state =0; goto LABEL0;}
:: atomic{ t1 ->strocc = 2; state =1; goto LABEL1;}
:: atomic{ t2 ->strocc = 2; state =2; goto LABEL2;}
:: atomic{ t3 ->strocc = 4; state =3; goto LABEL3;}
:: atomic{ t4 ->strocc = 4; state =4; goto LABEL4;}
:: atomic{ t7 ->strocc = 4; state =7; goto LABEL7;}
:: atomic{ t8 ->strocc = 1; state =8; goto LABEL8;}
:: atomic{ t9 ->strocc = 3; state =9; goto LABEL9;}
:: atomic{ t10 ->strocc = 2; state =10; goto LABEL10;}
fi;
…
```

Figure 38 Finite State Machine in PROMELA

The attributes Request of the states (pages) from the XML file here became the labels t0, t1, … t13 all initialized to 1; means that all transitions are enabled and any of them could be taken.

The labels from PROMELA file represent the states. For example, "Label 0" is representing the start state that is the home page. After the "if" statement we have the transitions. For example: "::atomic{ t10 ->strocc = 2; state =10; goto LABEL10;}" means that from the home page (Label 0) we can leave the state using the transition t10 (where t10 = http://www.beethoven.com/privacy.htm) to go to the state 10 (Label 10) with the attribute state=10, the state number and strocc = 2 meaning the occurrence of the string "music" in this page is 2.

The state description is finished with "fi".

In Figure 38 we show the first two states from the model extracted from www.beethoven.com web site in the XML format.

After the translation of the model from XML to PROMELA, the model can be visualized in a graphical representation as shown in Figure 39. This representation is generated by SPIN starting from the PROMELA model.

The circle labelled with 80 represents our Label 0 which is the home page. The edge labelled t0, …, t5, t7, …,t13 represent the transitions which leave the state 80 and target the corresponding states: Label 0 (80) – "Home Page", Label 1(136), Label 2 (192), Label 3 (242), Label 4 (298) – "Explore" page, Label 5 (354), Label 7 (404), Label 8(454), Label 9 (504), Label 10 (554), Label 11 (610) "Schedule" page, Label 12 (666), Label 13 (722). Label 6 is not in the model because it is an external link and the model checker did not find a transition from the other states to this state. We can search for the transitions which leave each state to see the potential browsing flow.

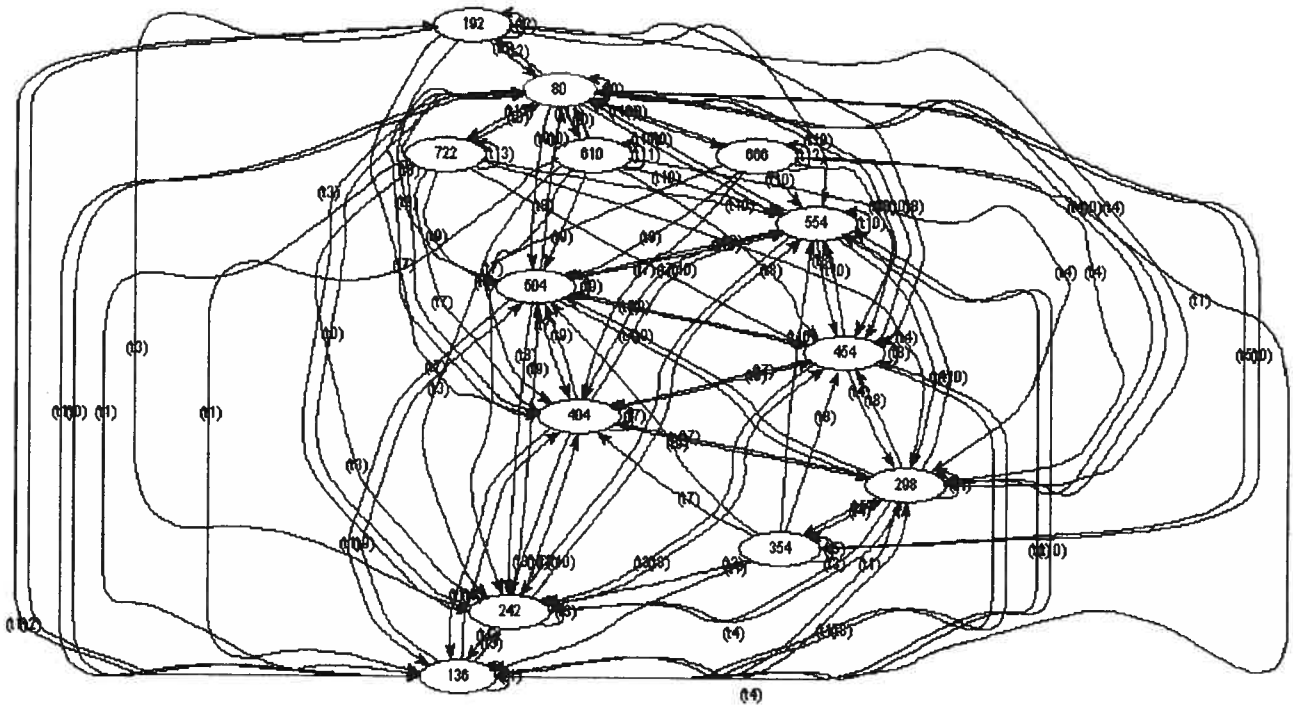


Figure 39 The Model – Graphical Representation

It can be observed that there is no edge between Label 4 (298) – "Explore" page and Label 11 (610) – "Scheduler" page.

7.4 Formal Properties (LTL)

In this section, we formally specify the properties described in section 3 in LTL (Linear Time Temporal Logic). We then present the results of checking these properties against the model extracted in section 4 using SPIN model checker.

7.4.1 Property 1

"Is the number of occurrences of the string music in every page strictly less than a given threshold 6?" **LTL:** the LTL formulae and the definition of the predicate "occur" are defined as follows:

[] occur

define occur (strocc < 6)

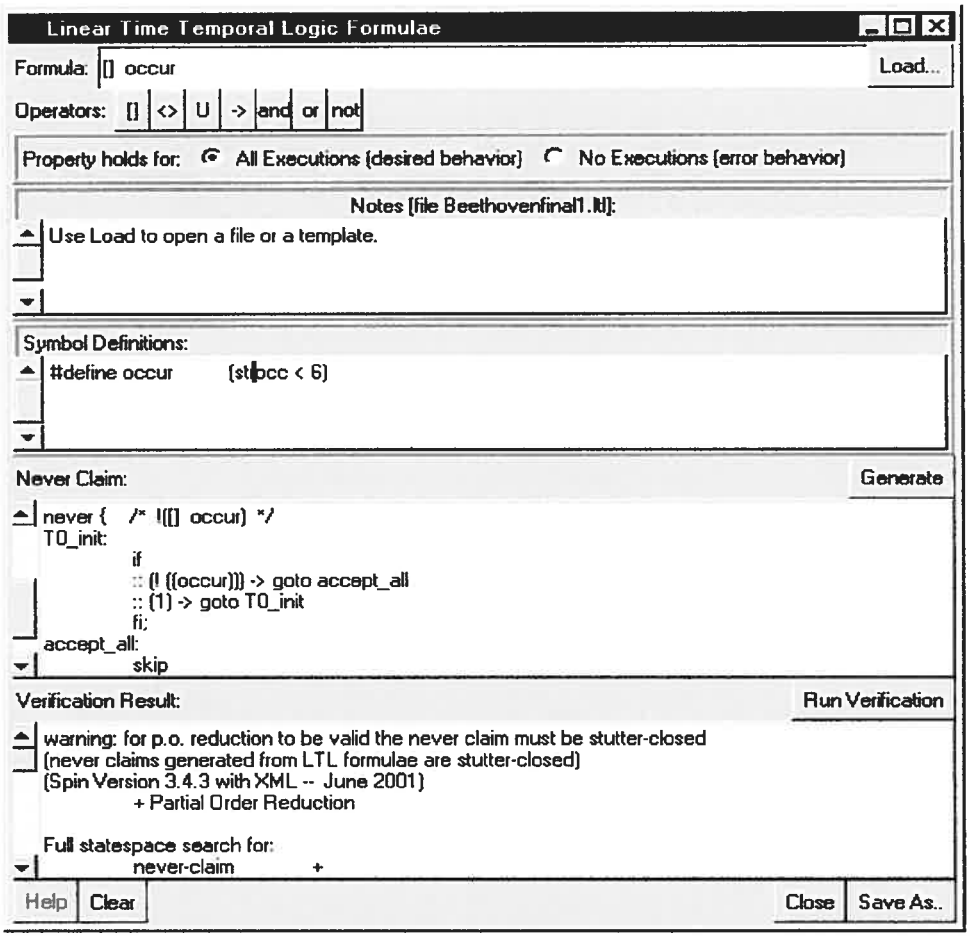


Figure 40 Linear Time Temporal Logic Formulae.

Figure 40 shows a snapshot of the LTL Property Manager window of SPIN.

The Formulae field introduces the LTL formulae "[] occur". In the Symbol Definition field, we give a definition of the predicate "occur" which states that the state attribute "strocc" is strictly less than a given threshold 6: "#define occur (strocc < 6)".

Never Claim:

The next step is to generate the never claim (negative) of the property by selecting the "Generate" button from the "Never Claim" section. Figure 41 shows the never claim of the property:

```
/*
 * Formula As Typed: [] occur
 * The Never Claim Below Corresponds
 * To The Negated Formula !([] occur)
 * (formalizing violations of the original)
 */

never {/* !([] occur) */
T0_init:
        If
        :: (! ((occur))) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

Figure 41 The Never Claim for LTL: [] occur.

Verification Result: The result of verifying the property is invalid. SPIN produces a counter example with an option to run a guided simulation of the counter example that shows a path to a page that violates the property. Figure 42 shows the simulation run where the "music" string occurrence is equal to 8.

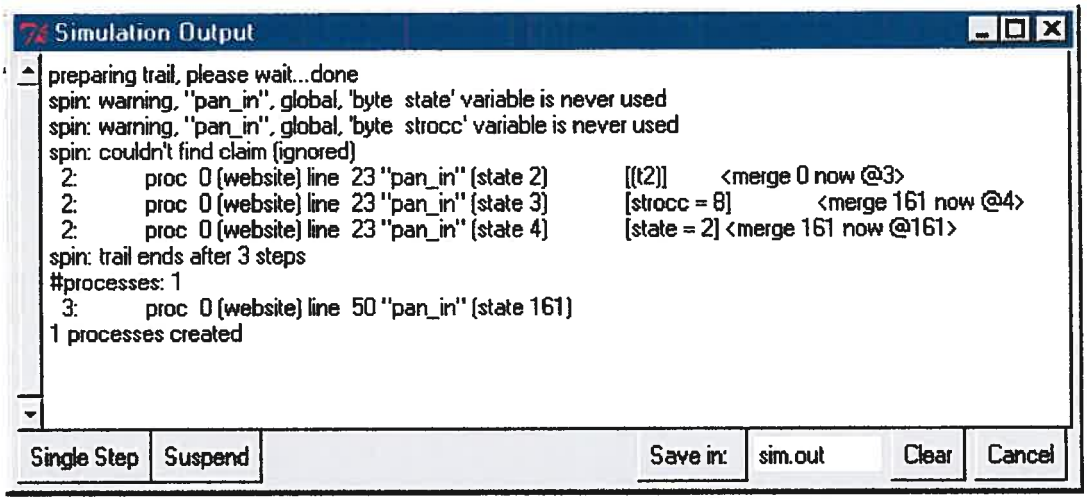


Figure 42 Guided Simulation Output – Property 1

7.4.2 Property 2

"Is page Schedule reachable from Explore page without going through Home Page?"

This property implies that there exists a path from Explore page to Schedule page without going through the home page. However, properties with existential operators cannot be specified in LTL. Therefore, the property is negated to check if in every path between the designated pages, the home page is present. So, if the negation of the property is satisfied, it means that the original property is violated and vice versa. Negation: *On all paths from Explore page to Schedule page Home Page is present.*

To specify this property in LTL, we make use of the property patterns found in [URL7]. Predicate P becomes true at a time t_p between t_q and t_r, which are the times when predicates Q and R become true, respectively. The property is specified as follows:

ExistBetween (home, explore, schedule) where home, explore, and schedule are predicates that designate the web pages in question.

LTL: As we did previously, the following LTL formulae and definitions of the predicates are inserted in the Formulae and Symbol Definitions fields respectively.

[] (explore && !schedule -> ((! schedule) U ((home && !schedule) || [] (!schedule))))# define home (state==0)
 # define explore (state==4)
 # define schedule (state==11)

Verification Result:

The verification result is valid which means that the original property is invalid: "page Schedule is not reachable from Explore page without going through Home Page". This result conforms to our observations, which means the model conforms to the web site navigation structure and the property was verified.

7.4.3 Property 3

"There exists at least a page from which page Schedule is reachable without going through the Home page."

For the same reasons explained for Property 2, this property has to be negated.

Negation:

"On all paths to Schedule page, the Home page is present." Using the Existence pattern: ExistBetween (home, (!(home) & !(schedule)), schedule)

- The predicate (home) is designated for the home page.
- The predicate (!(home) & !(schedule)) is designated for any page that is not the Home page nor the Schedule page.
- The predicate schedule is designated for the Schedule page.

LTL:

([] ((!home) && !schedule -> ((! schedule) U ((home && !schedule) || [] (!schedule)))))# define home (state==0)

 # define schedule (state==11)

As in the previous case the "Verification Result" is valid which means the original property is invalid which conforms to our observation.

7.4.4 Property 4

"There exists at least one page with occurrence of string music < 3 reachable from Explore page without going through the Home Page."

Negation:

"On all paths from Explore page to pages where occurrence of string music < 3 the Home Page is present." Using pattern: ExistBetween (home, explore, (minocc && !explore))

LTL: the following are the property in LTL and the definitions of the predicates used in the formulae:

[] (explore && !(minocc && !explore)-> ((! (minocc && !explore)) U (home || [] (!(minocc && !explore)))))#define home (state==0)

#define explore (state==4)

#define minocc (strocc<3)

Verification Result: The result of verifying the property is invalid. Figure 43 shows the output of the simulation run for the counter example. It shows the sequence of states that violates the property where the page explore (state = 4) is followed by the page where state = 8 and where strocc = 2. The latter page obviously is not the home page (state = 0). So, it can be concluded that there exists at least a path from page Eexplore to a page where *strocc* is less than 3 without going through the Home page which proves the validity of the original property.

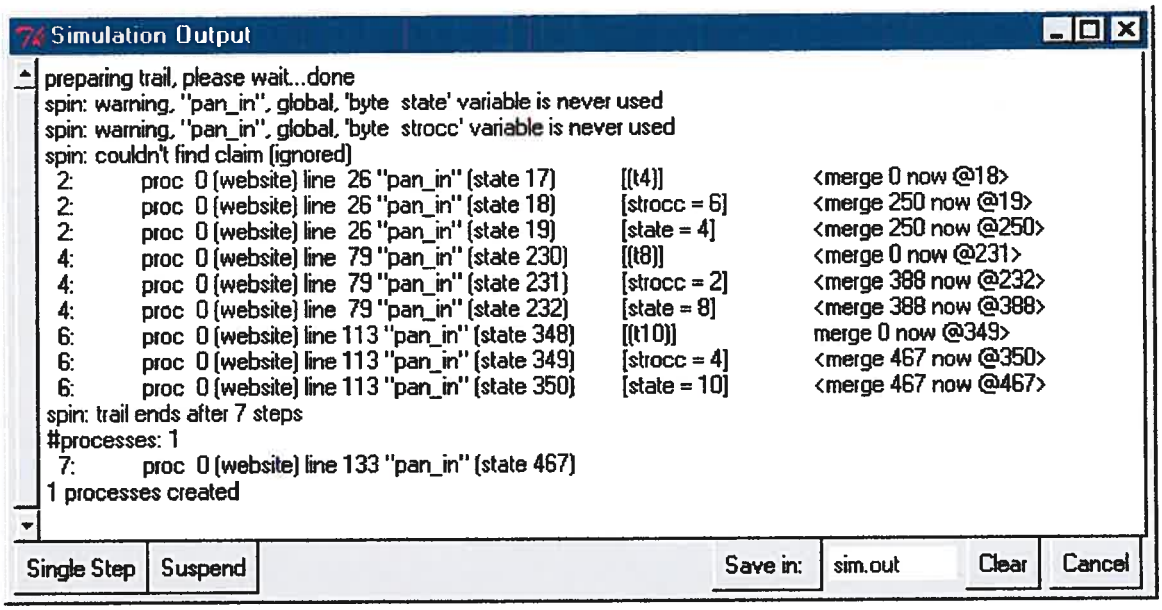


Figure 43 Guided Simulation Output – Property 4

7.5 Limitations of the Extractor/Manipulator Tool

The scalability of the application is limited by the processing power.

The following example taken from CRIM's web site includes 441 states and 8867 edges. The time for model extraction in this case was 2 to 3 minutes but the transformation of the XML finite state machine file in the PROMELA finite state machine file took a considerable amount of time of 2-3 hours. Thus, more work should be done in order to reduce the execution time. Since we are working with XML files, it is very expensive in terms of memory. The solution would be to store the information that we need in another data structure which can be accessed faster such as hash tables.

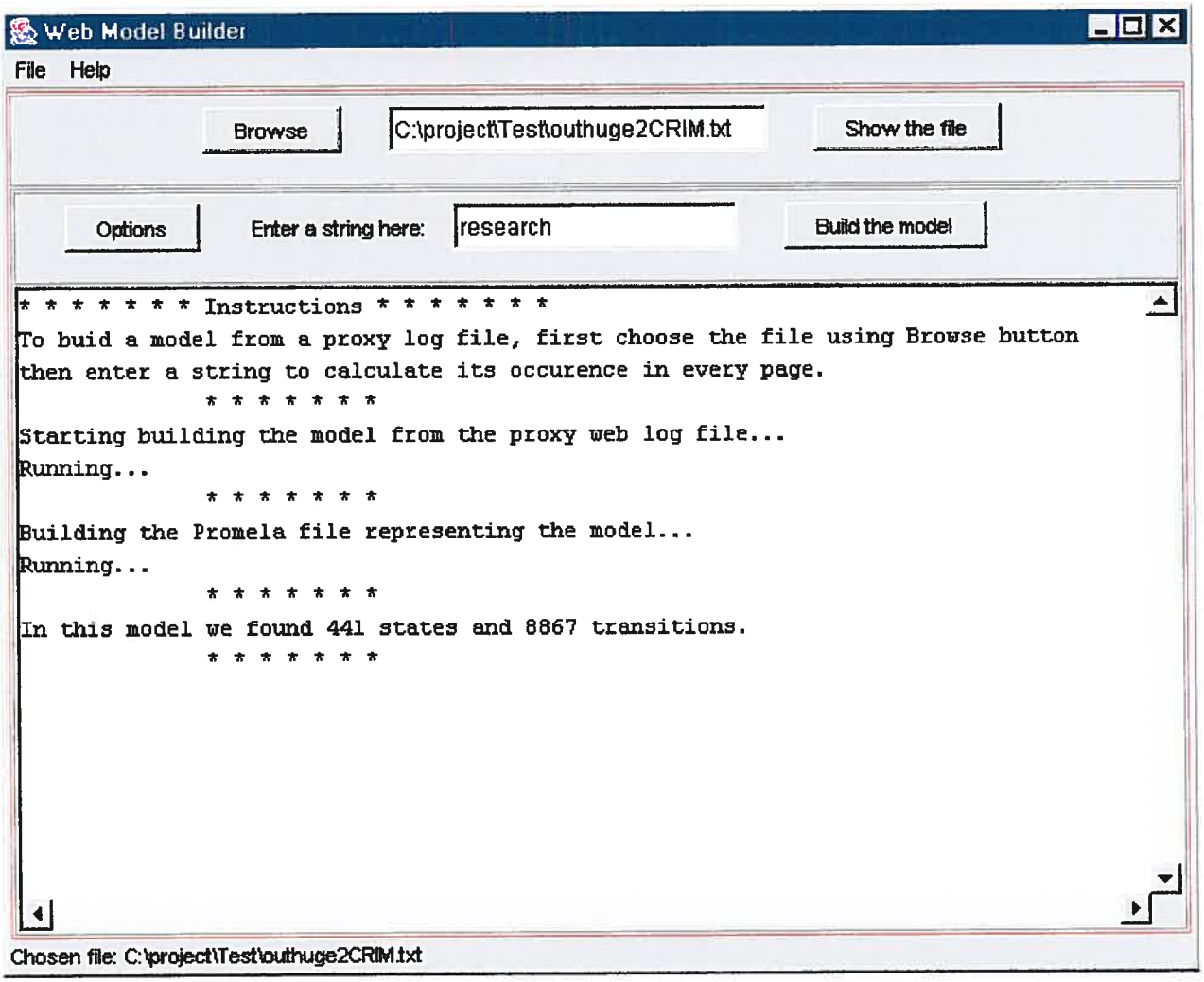


Figure 44 Scalability

Chapter 8

8 Conclusions

8.1 Summary of the Results

We demonstrated that the model checking technology can effectively be used for verification of web applications.

Our work was just the beginning of a research for defining a more general modeling framework and implementation of the toolkit. We described a framework for modeling simple web applications defined in LTL (Linear Temporal Logic). The properties expressing the web navigation can then be verified by the SPIN model checker on the model extracted from the browsing session. The process includes interception of the traffic between the client and server, its analysis and the construction of a formal model (FSM) out of it, using a standardized format (XML), and the translation to PROMELA which is the language of the SPIN model checker. We implemented a prototype tool that is based on this framework and demonstrated on case studies that the theory of model checking can be efficiently applied for web application verification.

8.2 Future Work

We built a prototype tool that can be used for formal verification of simple web applications. At the same time, more work remains to be done. In particular, the

modeling approach has to be expanded to applications with frames and windows, forms and other features. Accordingly, the developed tools have to be extended to handle such features. It would also be of interest to investigate the applicability of the developed framework for web services.

References

[1] L. de Alfaro, "Model Checking the World Wide Web", In Proc. of the 13th International Conference on Computer Aided Verification, Paris, France, July 2001.

[2] A. A. Andrews, J. Offutt and R. T. Alexander: "Testing Web applications by modeling with FSMs", Software and Systems Modeling, Publisher: Springer Berlin/Heidelberg, Volume 4, Number 3 D, 2005.

[3] S. Budkowski, "Estelle development toolset (EDT)". Comput. Network, 1992.

[4] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. "Model Checking", The MIT Press, 1999.

[5] M. G. Gouda, "Protocol verification made simple: a tutorial." Comput. Network., 1993.

[6] I. S. Graham, "HTML Sourcebook", Second Edition, John Wiley & Sons, Inc., New York, 1996.

[7] M. Haydar, "Formal Framework for Automated Analysis and Verification of Web-Based Applications", Proceedings of the 19th IEEE international conference on Automated software engineering, September 20-24, 2004.

[8] R. M. Hierons, "Extending test sequence overlap by invertibility." Comp. J., 1996.

[9] C. M. Huang, and J. M. Hsu, "An incremental protocol verification method." Comp., 1994.

[10] C.-M. Huang, M.-Y. Jang: "Interactive Temporal Behaviours and Modelling for Multimedia Presentations in the WWW Environment." Comput. J. vol. 42, 1999.

[11] G.J. Holzmann. "The Model Checker Spin", IEEE Transactions on Software Engineering, vol 23, no 5, May 1997.

[12] K. Naik, "Fault-tolerant UIO sequences in finite state machines." Proc. 8th Int. Workshop on Protocol Test Systems, Cavalli, A. and Budkowski, S. (eds), 1995.

[13] M. Nilsson, "Regular Model Checking", Printed by the Department of Information Technology, Uppsala University, Sweden, 2000.

[14] M.C.F. de Oliveira, P.C. Masiero, "A Statehart-Based Model for Hypermedia Applications", ACM Transactions on Information Systems, Vol. 19, No. 1, 28-52, January 2001.

[15] T.A. Powell, "Web Site Development, Beyond Web Page Design", Prentice Hall, 1998.

[16] P.D. Stotts, C.R. Cabarrus, "Hyperdocuments as Automata: Verification of Trace-Based Browsing Properties by Model Checking", ACM Transactions on Information Systems, Vol.16, No. 1, January 1998.

[17] D. Stotts, J. Navon, "Model Checking CobWeb Protocols for Verification of HTML Frames Behaviour", In Proc. of the 11th International World Wide Web Conference, Hawai, U.S.A., May 2002.

URLs

[URL1] http://www.isc.org/index.pl?/ops/ds/

[URL2] http://www.ascusc.org/jcmc/vol3/issue1/ho.html

[URL3] http://www.useit.com/alertbox/990502.html

[URL4] http://www.reitshamer.com/

[URL5] http://www.w3schools.com/xml/

[URL6] http://www.w3schools.com/dtd/

[URL7] http://netlib.bell-labs.com/netlib/spin/whatispin.html

[URL8] http://JAVA.sun.com/j2se/1.4/docs/api/index.html

[URL9] http://JAVA.sun.com/xml/

[URL10] http://www.cis.ksu.edu/santos/spec-patterns/

[URL11] http://cm.bell-labs.com/cm/cs/what/spin/Man/ltl.html

[URL12] http://www-2.cs.cmu.edu/~modelcheck/

[URL13] http://spinroot.com/spin/whatispin.html

[URL14] http://www.softwareqatest.com/qatweb1.html

[URL15] http://pdv.cs.tu-berlin.de/~azi/petri.html