2004

Model checking: Correct Web page navigations with browser behavior.

Xiaoshan Zhao
University of Windsor

Model Checking

Correct Web Page Navigations with Browser Behavior

by

Xiaoshan Zhao

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2004

Your file Votre référence
ISBN: 0-494-00166-6
Our file Notre référence
ISBN: 0-494-00166-6

1018316

Model Checking Correct Web Page Navigations with Browser Behavior

by

Xiaoshan Zhao

APPROVED BY:

H. Wu
Department of Electrical and Computer Engineering

L. Li
School of Computer Science

J. Chen, Advisor
School of Computer Science

A.K. Aggarwal, Chair of Defense
School of Computer Science

Sep. 10, 2004

Abstract

While providing better performance, transparency and expressiveness, the main features of the web technologies such as web caching, session and cookies, dynamically generated web pages etc. may also affect the correct understanding of the web applications running on top of them. From the viewpoint of formal verification and specification-based testing, this suggests that the formal model of the web application we use for static analysis or test case generation should contain the abstract behavior of the underlying web application environment. Here we consider the automated generation of such a model in terms of extended finite state machines from a given abstract description of a web application by incorporating the abstract behavioral model of the web browsers in the presence of session/cookies and dynamically generated web pages. The derived model can serve as the formal basis for both model checking and specification-based testing on the web applications where we take into account the effect of the internal caching mechanism to the correct accessibility of the web pages, which can be quite sensitive to the security of the information they carry. In order to check the correctness of the derived model against required properties, we provide the automated translation of the model into Promela. By applying SPIN on Promela models, we present experimental results on the evaluation of the proposed modeling in terms of scalability.

Keywords: Hypertext, Hypermedia, Web Navigation, Model Checking, Web caching, Verification, Extended Finite State Machine

Acknowledgement

First of all, I would like to thank my supervisor, Dr. Jessica Chen, for her invaluable guidance and advices, for her enthusiastic encouragement and her great patience to me. Without her help, the work presented here would not have been possible.

Next, I would like to thank my committee members, Dr Li, Dr. Aggarwal and Dr. Wu, for spending their precious time to read this thesis and putting on their comments, suggestions on the thesis work.

My special thanks go to Mr. Songtao Chen, Mrs. Hanmei Cui, Mr. Haitao Zheng and other members of our research group, for their help.

Finally, I also would like to thank my wife Chong Weng for her understanding, patience and support, and my daughter, Patricia, for giving me endless happiness.

Table of Contents

Abstract iii

Acknowledgement iv

List of Tables vii

List of Figures viii

1 Introduction and Problem Description 1

2 Related Works 7

2.1 Web Application Design Models 7
2.2 Web Application Verification Models 8
2.3 Testing Models on Web Applications 8
2.4 Summary 10

3 Overview of Web Application 11

3.1 Architecture of Web Applications 11
3.2 Web Browser 12
3.2.1 Architecture 13
3.2.2 History Stack 14
3.2.3 Web Cache 16
3.2.4 Cache Control 18
3.2.5 Session and Cookie 19
3.3 URL 20
3.4 Web Form 20
3.5 Web Server and Dynamic Web Page 21

4 Modeling Web Page Navigations with Browser Behavior 24

4.1 Overview 24
4.2 Web Page Navigation Design 25
4.2.1 Assumptions 25
4.2.2 Web Pages 25
4.2.3 Hyper Links 27
4.2.4 Session and Access Control 28
4.2.5 Navigation Design Model: An Example 29
4.3 Modeling Web Page Navigations with EFSM 29
4.3.1 Extended Finite State Machine (EFSM) 29
4.3.2 EFSM and Web Page Navigation 30
4.3.3 Adding Browser Behavior into the EFSM Model 31
4.3.4 Modeling History Stack 33
4.3.5 Modeling Browser Cache Repository Management 34
4.3.6 Constructing EFSM from Web Navigation Design 35
4.3.7 EFSM Conversion Rules 36
4.4 Summary 40

5 Implementation of Extended Finite State Machine 41

5.1 Navigation Design of Web Applications 41

5.1.1 Describing Navigation Design of Web Applications with XML..... 41
5.1.2 Object Model of Web Application Navigation Design..... 43
5.1.3 Constructing Web Application Design Object Model..... 44
5.2 Constructing EFSM Navigation Model from Web Navigation Design..... 44
5.2.1 Overview..... 44
5.2.2 EFSM Object Model..... 45
5.2.3 Algorithm..... 46
5.2.4 Rules..... 47
6 Spin and Promela..... 48
6.1 Overview..... 48
6.2 Spin..... 48
6.3 Promela Basics..... 50
6.3.1 Procsses, Channels and Variables..... 50
6.3.2 Executablility of Statements..... 52
6.3.3 Flow Control Statements..... 52
6.3.4 Atomic Sequences..... 54
6.4 Correctness Requirements..... 54
6.4.1 Assertions..... 54
6.4.2 State Labels..... 55
6.4.3 Never Claims..... 56
6.4.4 Linear Temporal Logic..... 57
7 Translating EFSM into Promela..... 59
7.1 Expressing EFSM with Promela..... 59
7.1.1 Data Type Defintion and Variable Declaration..... 59
7.1.2 Implementing EFSM with Promela..... 60
7.1.3 Conditions..... 62
7.1.4 Post Actions..... 63
7.2 Translation Algorithm..... 64
7.3 Expressing Correctness Requirements..... 65
7.3.1 Assertion..... 65
7.3.2 EFSM Output and LTL Expression..... 66
7.3.3 Error Tracing..... 67
8 Experiments and Evaluation..... 68
8.1 Stack Size..... 68
8.2 Number of States..... 70
8.3 Number of Transitions..... 71
9 Conclusion and Future Work..... 73
9.1 Conclusion..... 73
9.2 Future Work..... 74
Bibliography..... 75
Appendix A A Sample Web Design..... 78
Appendix B Translated Promela EFSM Model..... 79
Vita Auctoris..... 94

List of Tables

Table 4.1 Page attributes 27
Table 4.2 Attributes of the example of navigation design 29
Table 4.3 History stack operations 34
Table 4.4 Browser cache operations 35
Table 4.5 Transition rules I 37
Table 4.6 Transition rules II 38
Table 4.7 Transition rules III 39
Table 8.1 Stack size and verified states I 69
Table 8.2 Stack size and verified states II 70

List of Figures

Figure 1.1 Illustration of the resubmit phenomenon 2
Figure 1.2 Re-login phenomenon 3
Figure 3.1 Architecture of web application 11
Figure 3.2 A Browser's conceptual architecture 14
Figure 3.3 History stack operation 15
Figure 3.4 Internet Explorer 6.0 cache setting 17
Figure 3.5 Workflow of web page request in a browser 18
Figure 3.6 Conceptual architecture of web server 22
Figure 4.1 Pages of a web application 26
Figure 4.2 An example of a navigation design 29
Figure 4.3 An example of EFSM 30
Figure 4.4 Transitions for a hyperlink 32
Figure 4.5 Add Back/Forward transitions for a hyperlink 32
Figure 5.1 XML Schema for web navigation design 42
Figure 5.2 An example of navigation design described in XML 43
Figure 5.3 Object model of navigation design 43
Figure 5.4 EFSM based navigation model construction 44
Figure 5.5 EFSM object model 45
Figure 6.1 The structure of Spin 49
Figure 7.1 EFSM with Promela 61
Figure 7.2 A state with multiple incoming transitions 63
Figure 8.1 History stack size experiment schemes 69
Figure 8.2 Stack size and verified states relation diagram I 69
Figure 8.3 Stack size and verified states relation diagram II 70
Figure 8.4 Number of states and verified states 71
Figure 8.5 Number of transitions and verified states 71

1 Introduction and Problem Description

With the advance of networking and web technology, more and more information is being posted into and retrieved from the web. We have web systems for all major areas such as education systems, finance systems, heath systems, and transportation systems. In fact, nowadays, it is hard to find an area where web technology has not been applied: it has become the primary device for information sharing and retrieval.

A web system can be as simple as a set of static web pages for a course. It can also be as complicated as a world wide banking system that handles all sorts of transaction requests from different machines in different countries in multiple languages. The diversity and the intensive use of the web systems are enabled by the advance of the emerging technologies, such as web caching and dynamic web pages.

Web caching is a technique that stores cacheable web pages and sends back these pages by intercepting the web browsers' requested URLs. It can reduce the workload of the web servers. When the web caches are placed close to the web browsers, the use of web caching can significantly reduce the network traffic. From the user's viewpoint, the use of a caching mechanism greatly reduces the response time.

Dynamic web pages are generated on-the-fly by web servers according to the requested URLs. With the introduction of dynamic web pages, the use of web technology has moved from simple sets of hyperlinked web pages to complex web applications. A dynamic web page is the combination of a predefined page template and dynamic contents that are obtained after a web server receives a specific web page request. The interactions of a web application between its users and the application itself highly rely on the dynamic content generated with different request parameters.

Dynamic web pages are often used together with cookie/session techniques. A cookie is a small piece of data that is sent from a web server and stored in the local storage of a web browser. When a web browser sends out a web page request, a cookie may be included in the request message and the web server can retrieve the cookie from the message. The use of cookies gives a web server the ability to trace the status of its client browsers and

maintain the communication sessions with them. A web server could identify each browser's identity by issuing different cookies on the browsers that visit it.

While providing better performance, transparency and expressiveness, the above features of the web technology have also raised some important issues and posed additional difficulties on the validation of the correctness of the web applications built upon it. In the presence of browser cache, for example, the users can interact not only with the web pages but also with the web browser via the use of special buttons or menu-items such as the back, forward, refresh buttons. The use of these buttons or menu-items may affect the accessibility of the web pages, which can be quite sensitive to the security of the information they carry. We give a couple of examples to show this point.

Let us assume that the web browser can manage the cached pages and provide the users with the previously visited pages whenever enabled by the cookie.

The first example demonstrates the so-called *resubmit* phenomenon. Suppose in an on-line banking system, the user clicked the submit button on page A to confirm a transaction and reached page B.

The system does not provide the acknowledgement on page B, so the user is not sure whether the transaction is successful. As the back button is available, the user clicks on it to get back to page A. Since page A is a cached one, it is identical to the previous one and thus the user cannot notice any change of the information on it. Then the user may click on the submit button again. While the user needs only one transaction, the banking system will treat the two submit actions as different ones and process both of them.

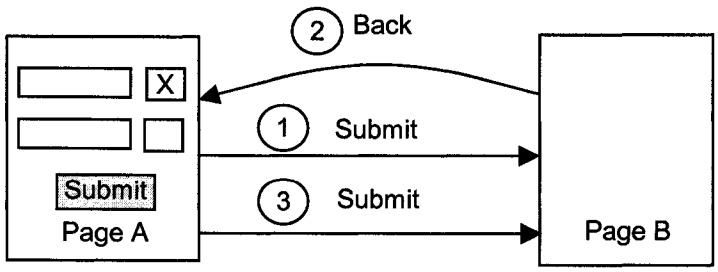


Figure 1.1 Illustration of the resubmit phenomenon

This phenomenon may not exhibit itself if there is no caching system involved: if the pages are not cached, the user will have to go through a sequence of given links to reach page A again, just like the first time. Along with the navigated pages, the user will normally be able to observe the updated information, e.g. the updated account balance, because the requests for the pages are always sent to the server side and re-generated. In the presence of caching, on the other hand, when a user requests a previously visited page, the information on the page will not be updated, and thus may cause confusion to the user. The resubmit phenomenon is quite common especially in web applications that involve on-line transactions, such as on-line shopping, on-line banking.

The second example demonstrates the so-called *re-login* phenomenon. Suppose the user is required to login to view certain secured pages.

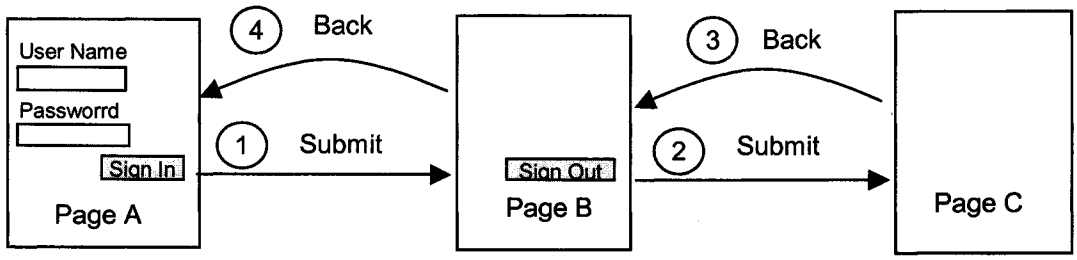


Figure 1.2 Re-login phenomenon

In page A, the user enters the user name and password, and clicks on the sign-in button. Upon this click, the user name and password are sent to the web server for authentication. When the authentication is passed, page B, a secure page, is loaded into the browser. Suppose in page B, the user clicks on the sign-out button and page C is shown. Page C is an insecure page and it also indicates that the user has signed out from the secured part of the system successfully.

With the caching mechanism, the back button may be enabled, and thus the user can actually click on it and can possibly view page B again, without re-entering the user name and password. If this happens in an area where the machines can be publicly accessed, e.g. public library, airport Internet zone, this will raise the issue that the information (such as credit card number) contained in a secured page B may be viewed by the wrong users.

Problem description: The above examples show that a web application providing all correct functionality by itself may however malfunction when it is put into its supporting environment: the behavior of the web browsers may have an impact on the correctness of the web applications.

Note that the behavior of the web browsers depends on how the web browser is implemented and configured, whether the cookie is enabled, etc., and the web developers have access to the configuration of the web browsers and cookies.

1. From the viewpoint of software design, this suggests that a web application should be carefully designed with correct configuration of the web browsers as well as some important properties of the web pages such as secured page, cacheable page, etc. In the first example, the synchronization phenomenon can be avoided if page A is defined as un-cacheable. In the second example, the re-login phenomenon can be avoided if page B is defined as un-cacheable. In fact, this re-login phenomenon appears only in some applications. For example, it appears in the current University of Windsor Web Mail System, but it does not exist in Microsoft's Hotmail.
2. From the viewpoint of validation, verification and specification-based testing, we need to obtain a design specification that contains the full details of the correct interactions between the users and the web system as a whole. This means, the user's possible interactions with the web browser should also be modeled and reflected in the design specification.

There exists a gap between a design specification in item 1 and the one in item 2: While it is reasonable to ask the web developers to provide the design specifications that contain the correct configuration and page properties, it is too demanding to ask them for the specification of the abstract behavior of the web application that subsumes the behavior of the web browser.

Proposed solution: we consider the automated generation of the latter from the former. In doing so, we provide a formal model of the behavior of web browsers in the presence of cookies and dynamic web pages, with the focus on browser caching.

Our model is abstracted from the implementations of existing commercial web browsers such as Microsoft's Internet Explorer, Netscape's Navigator. Such a model is incorporated into the design specification given by the web developers, so that it can be used for the verification and specification-based testing of the web application.

We assume the availability of the design specification of the web application in terms of *page navigation diagrams*. Such a page navigation diagram shows all the *desired* possible navigations among the web pages and web page templates.

We also assume that each page is associated with some properties to define whether it is a secured page, whether it is cacheable, whether it is an entry page, and whether its access requires an open session.

Based on the information on such a diagram, we provide automated construction of an extended finite state machine (EFSM)[17] that incorporates the behavior of the internal caching mechanism of web browser into the description of the web applications.

EFSM is extended from FSM (Finite State Machine) [17] by adding trigger condition onto each transition. It is a basic mechanism in modeling reactive system. An FSM consists of states, inputs, outputs, transitions, and initial state. The behavior of a system can be described with transitions of an FSM. A transition consists of start state, end state, input and output. At the beginning of each transition, the machine is in the start state. An input triggers the system from start state to end state and a value is output. An FSM can be described as a directive graph including nodes, and edges. Each node stands for a state of the FSM and a transition can be considered as a directive edge that connects two nodes.

The derived model can serve as the formal basis for both model checking and specification-based testing on the web applications where we take into account the affect of the internal caching mechanism to the correct accessibility of the web pages. To model-checking the correctness of the derived model against required properties, we provide the automated translation of the model into Promela[14]. Thus, the conflicts to the requirements in the design specification can be detected automatically by SPIN[13][29] and a report can be generated. By applying SPIN, we present experimental results on the evaluation of the proposed modeling in terms of scalability.

This thesis consists of 9 chapters. Chapter 2 introduces previous works on modeling web applications. Chapter 3 gives an outline of a web application including browser, web server, and communication protocol. Chapter 4 presents our work on constructing the verification model of web applications. Chapter 5 introduces the conversion from web application navigation design to EFSM based on a set of rules. Chapter 6 gives a brief introduction to model checking tool SPIN and modeling language Promela. Chapter 7 introduces the translation from EFSM to Promela. Chapter 8 presents our evaluation on the methodology we propose. Chapter 9 gives the conclusion and future work.

2 Related Works

Our work is divided into three steps: constructing the verification model of a web application, translating the verification model into PROMELA, and applying SPIN to perform model checking. Obviously, establishing the model for a web application forms the basis of our work. There are several approaches on modeling web applications, concentrating on design, testing and verification. As UML is largely involved in the design of web applications, several approaches adopted extended UML to model web application design. Automata, graph, labeled transition systems are used in establishing testing and verification models for web applications. We investigate the related modeling works in three aspects, web application design, web application verification and web application testing.

2.1 Web Application Design Models

The researches on the modeling methodologies for web application design are concentrated on developing the notations to describe the navigation relations. Since UML is widely adopted as modeling language during the development of an application, several approaches try to extend UML to model the web navigation behaviors. Conallen [8] proposed new stereotypes, <<client page>>, <<server page>>, to model dynamic web page generations. Gomez et al.[9] use NAD (Navigation Access Diagram) to model web applications. Each NAD modeling element is an extended UML stereostype. Hennicker and Koch [4] [12] [15] developed UWE (UML-based Web Engineering), a framework for web application development with UML. UWE includes navigational classes and these classes are inter-related with connection components, such as index, query, menu, etc. Ceri [6] proposed a modeling language WebML, an XML based modeling language for web application design. All WebML elements are notations described with XML. A tool is developed to support WebML and a design could be converted WebML format automatically. Besides the modeling methodologies above, statechart is used to model the navigation behaviors for web applications, especially for frame-based web pages. Zheng and Pong first introduced statechart to model hypertext user manual in [30]. Lieung et al. [18] used statechart to model dynamic server page and frame-based web pages.

2.2 Web Application Verification Models

Sciascio et al. [27] [28] presents how to verify a web application with NuSMV and CTL. The model of a web application is a web graph, which consists of nodes and arcs. In a web graph, nodes include pages, links, and windows, and the arcs only connect all nodes. The browsing from one page to another following a hyperlink includes at least three nodes and two arcs. The first arc connects the start page to one of its hyperlinks, and the second connects the hyperlink to the destination page. All the requirement properties are written in CTL formulas. A symbolic model verifier, NuSMV, is applied after the web application model and requirement properties are ready. The final results and counter examples are reported. In order to make it easy to use model checking tool, a series patters are developed.

Alfaro proposed a technique on model checking static web pages with μ-calculus in [1]. The main purpose of model checking in [1] is to verify the properties of a web site with pages that contain frames. A web site is considered as a web graph which consists of webnodes and edges. The edges of the webgraph are page links. Each webnode is a tree that contains URL pages as nodes and the edges of the tree are labeled by frame names. The requirements are described in constructive μ-calculus. A model checking tool, MCWeb was developed. After model checking, the tool can report errors automatically, such as broken links, duplicated frame names, non-hierarchical frame content, etc.

2.3 Testing Models on Web Applications

Ricca and Tonella [22] [23] proposed testing strategy on web site analysis through web browser. A tool called ReWeb is developed to gather all web pages and their relations within one web site, and a UML-based model for the web site is constructed by this ReWeb. A testing tool, TestWeb, is responsible for testing the UML web application model. Ricca and Tonella considered white-box testing on a web application. The testing is mainly concentrated on web forms of a web application. A test case generation engine inside TestWeb is used to generate test cases, and the generation is based on a reduced graph by removing static web pages without forms in a navigation paths.

Kung, Liu and Hsia [16] use an ORD (Object Relation Diagram) based web application testing model (WTM) for testing web applications. A WTM model of a web application is created by reverse engineering on the source documents of it. The testing work is

divided into three parts: object perspective, behavior perspective and structure perspective. Each part is tested separately. The object perspective of the WTM describes the class structures of a web application including request, response, navigation and redirection. The behavior perspective of the WTM focuses on page navigation, and page navigation diagram (PND) derived from ORD is employed. Finally a navigation tree started from the home page is constructed and the testing of the navigation behavior is based on this PND. The structure perspective of the WTM is related to control flow and data flow information of a web application. Thus block branch diagram (BBD) and function cluster diagrams (FCD) are used respectively for describing control flow and data flow.

Lucca and Penta [19] proposed a base-line testing strategy that creates a testing model by adding browser statechart to a series of pages with inter-related hyperlinks. Lucca and Penta considered the influence of web browser's behaviors on a web application. Each web browser contains buttons like back, forward and reload. A user's click on one of these buttons can force the browser to display the previously visited page or refresh the current web page with the same URL. In order to test a web application with browser's behavior, a statechart of back and forward buttons is constructed with four states, BDFD (Back Disable, Forward Disable), BEFD (Back Enable, Forward Disable), BEFE (Back Enable, Forward Enable) and BDFE (Back Disable, Forward Enable). For each navigation path, e.g. a base line, the testing model is a navigation tree generated by adding the statechart of browser's behavior. The root of the tree is the home page of the web application, and each path of the tree is tested separately.

Graunke et al. adopted λ-Calculus to model web form related web applications in [10]. Each web application in this model is divided into a single server and a single client. The server contains a table that maps the requested URL to a process program. A client consists of a current form web page, and all previously visited web pages. Each form contains variables and the URL that the form data will be sent to. A set of rules is defined that regulate the transitions from one page to another. This model is used to solve communication problem and observer problem.

Beek and Mauw [5] used labelled transitions systems to model web applications and the conformance testing is applied to this model. An MRRTS (Multi Request-Response Transition Systems) in which request is considered as input and response is output, is used. With this model, the navigation behaviors of a web application are modeled as URL label series.

2.4 Summary

Most of modeling methodologies for web application design only consider navigations among server pages. Although in [8] Connella models server page and client page, the client page does not reflect the attributes of browser cache and history stack. In [10], Graunke et al. introduce the formal model that integrates server page and client page, and the previously visited web pages are modeled. These previously visited web pages are URL series, and the browser behavior on controlling history stack and web cache is not modeled. Lucca and Penta [19] considered that the browser behavior has influences on the navigation behavior of a web application. According to our analysis on web browser, although Lucca and Penta proposed a testing method considering browser behavior, the browser model they used is not sufficient to describe the real browsers we use. It is necessary to consider the browser cache and history stack on testing a web application.

3 Overview of Web Application

In this chapter we introduce the architecture of web applications, web communication standards, web browser, and server side scripts for dynamic web generation.

3.1 Architecture of Web Applications

A web application is an interactive system that involves browsers, web servers, and possibly other servers that provide services for data processing. Figure 3.1 shows the architecture of a 3-tier web application

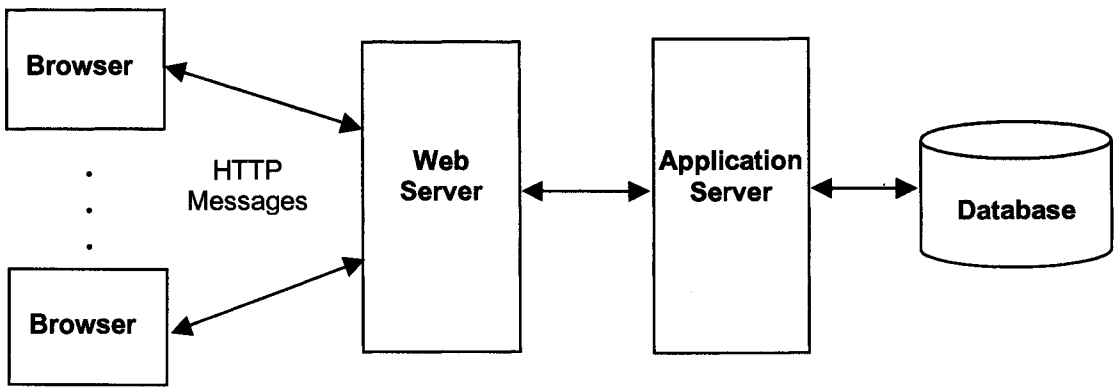


Figure 3.1 Architecture of web application

A web browser is a standard window application that displays web pages. Web browsers provide a graphical interface that lets users navigate web pages by clicking hyperlinks, toolbar buttons, or typing URLs (Universal Resource Locator)[25] in address text box. A web page is uniquely identified by a URL in the Internet and a browser obtains web pages by sending requests including URLs to web servers. After sending out a web page request to a web server, a browser waits for the response message from the web server. When the web browser receives a response message, it retrieves the HTML web page embedded in the message and presents it. At the same time, the browser also saves the URL and web page in its local directory.

A web server is responsible for monitoring the incoming request messages and replying response messages with web pages according to the requests. When a request reaches a web server, the web server retrieves the URL from the request message. This URL is used to identify a unique web page in a web server. If the requested web page is a static HTML web page stored in a directory of the web server, the server reads this HTML file,

encapsulates it in a response message and sends the message back to the requested web browser. If the web server cannot locate a static web page in its local directory with the URL, it passes the request URL to a dynamic web page generating module. Typically a dynamic web page generating module consists of a dispatcher and page generating procedures. The dispatcher receives the request URL and calls the corresponding procedure to generate a web page. This newly created web page is sent to the web server, and a response message that contains this new page is returned to the requested web browser.

The web server's ability on generating dynamic web pages forms the basis of modern web applications. The interactions between web browsers and web servers highly rely on dynamic web pages. The structure of a dynamic web page is the same as a static web page. A static web page is written and stored in a web server's local directory before it is requested. A dynamic web page is generated after it is requested. In order to generate dynamic web pages, a web server contains predefined procedures in a specific programming language, such as ASP (Active Server Page), JSP (Java Server Page), etc. Each procedure contains a page template that defines a web page's layout, format, and other static contents. While a procedure is called, a web page is assembled by adding dynamic content into the predefined page template.

The dynamic content used for dynamic web page generation may depend on database servers or distributed objects that encapsulate business logics. In Figure 3.1, the dynamic content is provided by an application server, and this server is responsible for data management and database operation.

3.2 Web Browser

A web browser is a client side application that works as user interface for a web application. Actually a browser is a standard window application including title bar, menu bar, tool bar, address bar and an area that presents web pages. A browser presents HTML web pages and allows a user to navigate web pages in a web site. Besides presenting web pages, a web browser maintains a local cache and a history stack. The use of cache can significantly reduce response time and network traffic. History stack stores

the recently visited URLs of the web pages, and when a user clicks "refresh", "back", or "forward" button, a corresponding page is requested and displayed.

3.2.1 Architecture

Figure 3.2 shows the conceptual architecture of a web browser. Conceptually, a web browser consists of a browser module, a network interface module, an HTML parser, a history stack and a local cache. The browser module is mainly responsible for presenting a web page, and processing user navigation events. It also maintains a history stack for reloading previously visited web pages. The reloading of a web page is activated by a user clicking "refresh", "back" or "forward" buttons in the browser's toolbar area. The HTML parser parses HTML web pages, and constructs an object model for each web page. The presentation of a web page and the user's operations on it all rely on this object model. The network interface module maintains communications between the browser and all web servers. It generates HTTP[24] request messages and sends them to the designated web servers. When response messages are received, the network interface module retrieves the web pages embedded in response messages and performs operations on its local cache according to the parameters associated with HTTP response messages.

A web browser uses HTTP to communicate with web servers through Internet. HTTP is a stateless communication protocol that consists of two kinds of messages, request message and response message. A request message is sent from the browser to a web server, and a response message is transmitted from a web server to the browser who sends out the request.

While a user clicks hyperlinks in a web page, "back", or "forward" buttons in the browser's tool bar, the browser module obtains a URL from current web page or history stack. This URL is passed onto the network interface module. The network interface module tries to search the web page related to this URL from the browser's local cache. If the page is stored in the local cache as fresh page, the page is returned to HTML parser. Otherwise, the network interface creates an HTTP request message including the URL, and sends this request message to the designated web server. After receiving the response message from the web server, the network interface retrieves the web page contained in it and returns the web page to HTML parser. At the same time, it stores the web page into

the browser's local cache with expire-time stamp. The expire-time is stored in the head part of the response message.

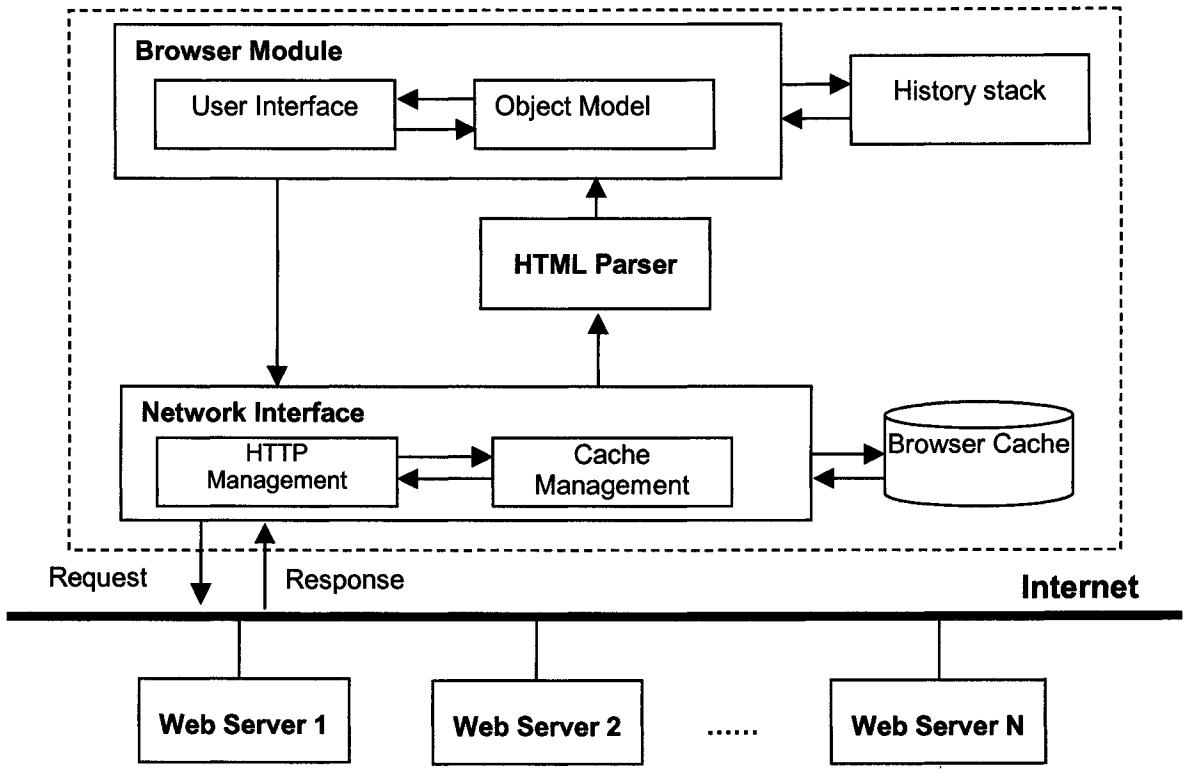


Figure 3.2 A Browser's conceptual architecture

3.2.2 History Stack

A browser's history stack[11] stores the previously visited URLs, and the stack is maintained by the browser module. Generally speaking, the history stack is a kind of stack with similar operations as normal stacks. A history stack maintains a stack pointer, top position and bottom position. The values of these variables determine whether a back or forward button can be enabled. Compared with normal stack, the stack pointer of browser's history stack can be moved back and forth if more than one item is stored in it.

When a browser starts up, its history stack is empty. The stack pointer points to nothing and the top position and bottom position are set to a null value. At this time, the back and forward button are disabled. After a URL is requested and a web page is received, the URL is pushed into the stack and the stack pointer points to the newly inserted URL. The top position and bottom position are set to their corresponding values.

The browser's back and forward buttons are enabled or disabled according to how many items are stored and the position of stack pointer. If the stack has more than 2 items and the stack pointer does not point to the first URL stored in the history stack, the back button is enabled. This means that there exists an item before the current item that the stack pointer points to. At this time, a user can click the back button, and the stack pointer will move from current position to its previous one. The URL stored in the position that the stack pointer points to is retrieved and sent to network interface for requesting its corresponding HTML web page. The page might be returned from local cache or the web server that sent the page before. The use of local cache depends on the cache policy set in the browser or HTTP cache controls that came with the received web page.

Similarly, the forward button is enabled when the stack has more than 2 items and the stack pointer does not point to the top item in the history stack. The clicks on the forward button can force the stack pointer moving from current position to its next one. The URL that the stack pointer points to is used to request its corresponding web page.

When a user clicks on a "back" or "forward" button, the stack pointer moves back or forth from current position. If a user clicks on a hyperlink or types a URL in the browser's address textbox, the URL of newly requested web page is pushed into the history stack after the browser receives the web page successfully. Before the push operation, all URLs above stack pointer's current position are popped up, and after the push operation, the stack pointer points to the top position of the stack.

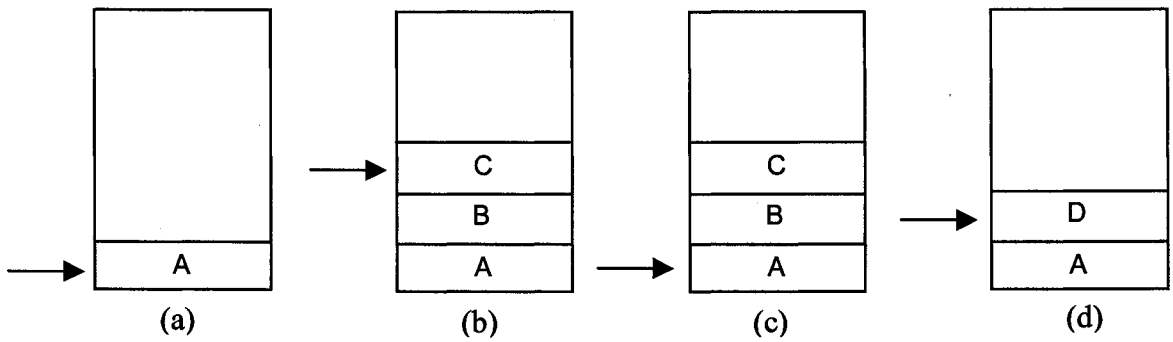


Figure 3.3 History stack operation

Figure 3.3 gives an example of operations on history stack. Figure 3.3(a) shows that page A's URL A is pushed into the stack after receiving page A. The stack pointer points to A.

In Figure 3.4(b), page B and C have been received and their URLs B and C are pushed into the stack. In Figure 3.4(c), the user clicks *back* button twice, and the stack pointer points to the first URL A. Because page A contains hyperlinks, after a user clicks a link to page D in page A, page D's URL is pushed into the stack on the position above A. Before D is pushed into the stack, previous URLs B and C are popped up.

3.2.3 Web Cache

Web cache[20] is a manageable storage place that stores previously visited pages. If one of these visited pages is requested again, the page can be retrieved from the web cache instead of the web server. Since web cache is physically close to a client, the use of web cache can reduce response time and network traffic for web page request.

There are two kinds of web caches: browser cache and proxy cache. A proxy cache is located between a web server and a web browser. It provides web pages for a large number of clients in its local network. All web page requests from these clients are intercepted by a proxy which looks up the request pages in the proxy cache. If a requested web page is stored in the cache and the page is considered as fresh according to its expire-time, the page is returned to the requested web browser. Otherwise the request is sent to the designated web server. After the proxy receives the cacheable web pages, it stores these pages into its proxy cache and sends them to the requested browsers at the same time. A browser cache has the same functions as a proxy cache and is maintained by a web browser in its local directory. A browser cache only stores cacheable web pages for a web browser.

A user can select a cache policy for the browser's cache setting. Figure 3.4 shows a browser cache setting in Microsoft's Internet Explorer 6.0.

There are 4 options for checking cacheable pages for each Internet Explorer 6.0 web browser. The first setting, "Every visit to the page", forces the browser to check if a previously visited web page has changed since last visit. If the page has changed, the browser displays the new page and stores it in its local cache. "Every time you start Internet Explorer" means in current session, all cacheable web pages are valid. Only the pages requested in other sessions need to be checked from the web server when it is requested. "Automatically" is the same as "Every time you start Internet Explorer", but it

is not limited to current session. If a previously visited page is stored in the local cache and is fresh, there is no need to check any change if the page is fresh. The last one, "Never", means the browser does not need to check the web server for new contents of the cacheable web pages.

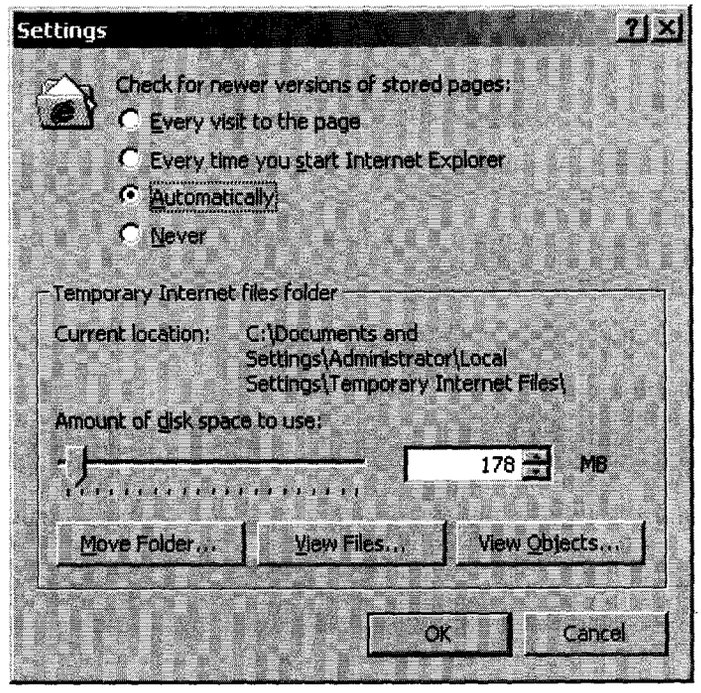


Figure 3.4 Internet Explorer 6.0 cache setting

The workflow of a page request within a browser is shown on Figure 3.5. At the beginning, a browser receives a user's requested URL for a web page. The requested URL comes from hyperlinks on current web page, address text box by typing in, or from history stack while a user clicks "back" or "forward" button.

If the requested page is already stored in the browser's cache, the cacheable page is fresh, and there is no need to check any change from the web server according to the browser's cache policy, the page is returned to the browser's HTML parser directly.

If the current browser's cache policy is set to check changes for this previously visited web page, a request is sent to the web server. After the web server returns a new web page for the requested URL, this page is stored in the browser's cache and the previous page with the same URL is deleted. At the same time, the new web page is transmitted to the HTML parser.

If the requested web page does not exist in the browser's cache or the page is not fresh in the cache, a request is sent to the web server directly. The received web page is transferred to both the local cache and the HTML parser.

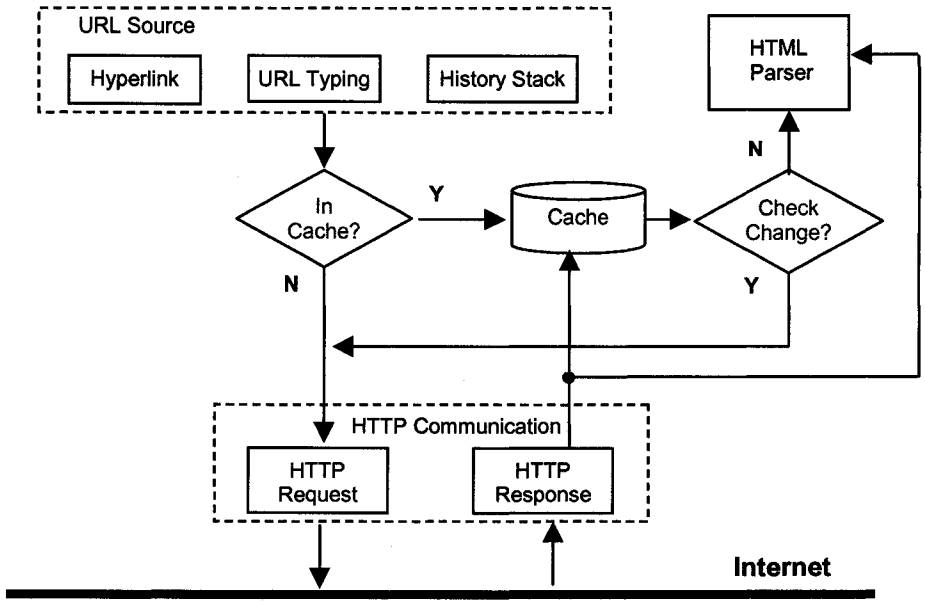


Figure 3.5 Workflow of web page request in a browser

3.2.4 Cache Control

A web browser maintains a local browser cache for previously visited web pages. The cacheable web pages should be the reflections of their original pages in the web servers. A cacheable web page is considered fresh if the original page does not change in a certain time period and this time period is used to control the freshness of a web page. When a browser receives a web page from a web server, it must determine if it is cacheable and valid period before saving the page into the browser cache. A web page's freshness is decided by expire time of the page. Expire time defines the validity time period of a web page.

Expire time of a web page can be set in the HTTP response message header part or in the web page itself with META tag.

```
<META http-equiv="Expires" content="Tue, 20 Aug 2003 14:25:27 GMT">
```

The example above shows the expire time setting in an HTML META tag. The setting is a name/value pair. The expire time is based on GMT (Greenwich Mean Time) and it

indicates the page is fresh before that time. A page can also be set uncacheable by defining its expire time as the same time it is created.

Except for expire time, HTTP 1.1 introduces new cache control elements, including "max-age", "s-maxage", "public", "no-cache", "must-revalidate", and "proxy-revalidate". These elements provide more options in the control of cacheable web pages. "max-age" defines the maximum amount of time that a web page can be considered fresh. "s-maxage" is similar to "max-age", except that it is only used for proxy cache. "public" indicates the response page is cacheable. "no-cache" means the web page received by a browser is not cacheable; if a browser requests the same page again, it must send the request to the web server directly. "must-revalidate" forces the cache to send validation request to the original server before returning a cacheable web page. "proxy-revalidate" is similar to "must-revalidate", except that it is used only by proxy caches.

3.2.5 Session and Cookie

HTTP is a stateless protocol and the communication between web browser and web server is based on the request/response model. After a response has been sent to a requested browser, the connection is closed. The web server does not maintain any information about the client. When the same browser sends another request to the same web server, the web server cannot recognize that the request is sent from the same client. In order to maintain a logical session between a web browser and a web server, the identification information of a web client should be included in each request/response communication cycle. This identification information stored in a web browser is called "cookie".

A cookie is a piece of information that is sent by a web server to a web browser. A cookie may contain any information a web server wants to store in a web browser. After a web browser receives a cookie, it saves the cookie in its local storage place. Whenever the browser sends requests to the web server, it retrieves the cookie and puts it into the request message. Thus the server can trace a client's communication by the unique cookie it issues.

3.3 URL

URL stands for Universal Resource Locator, and it is used to identify a unique resource in the Internet. The format of URL was defined in RFC 1738[25]. A URL consists of three parts, URL scheme, host address, and url-path. Each scheme is a category of resources, such as http, ftp, telnet, etc. A host's address consists of the host's IP address and the port number. A url-path is a relative path used inside a computer in the Internet. The communications between a web browser and a web server rely on the HTTP. Each HTTP request contains an HTTP URL for a specified web page in the web server. The format of an HTTP URL is: http://<host>:<port>/<path>?<searchpart>. Here "http:// " indicates the resource type is http, and the communication protocol that is used to send this request should be HTTP. The <host> and <port> identify a unique web server in the Internet. <host> is the web server's IP address and <port> is the port number that the web server uses for HTTP communication. The default port number for a web server is 80. If the port number is not included in an HTTP URL, it means 80 is used as destination port number. Because an IP address is related to a domain name that is stored in DNS (Domain Name Server), a client can obtain the IP address of a web server by requesting domain name service. <path> specifies the relative storage position of the request resource. <searchpart> consists of search parameters a client passes to the web server. These parameters could be used to generate a web page, or they are transaction data that is need to be processed by an application server.

http://www.google.ca/search?hl=en&lr=&ie=UTF-8&oe=UTF-8&q=related:www.rfc-editor.org/rfc.html

The example above shows the basic elements that consist of a URL. "http://" indicates the resource type is HTTP and the browser will use HTTP protocol to send this URL request. "www.google.ca" is a domain name and the corresponding IP address can be resolved by querying a DNS (Domain Name Service) server. The port number is implicit and it is the default 80. "search" is a relative directory and "hl=en&lr=&ie=UTF-8&oe=UTF-8&q=related:www.rfc-editor.org/rfc.html" is a search part.

3.4 Web Form

A form is an area in an HTML weg page containing normal content, markup tags, and controls including labels, checkboxes, radio buttons, combo boxes, etc. After a user fills

in the form and clicks a submit button in the form, all form data will be sent to the designated web server.

```
<FORM action="http://awebsite.com/process.jsp" method="post">
  <P>
  <LABEL for="User Name">First Name: </LABEL>
        <INPUT type="text" id="username"><BR>
  <LABEL for="Password">Password:     </LABEL>
        <INPUT type="text" id="password"><BR>
  <INPUT type="submit" value="Sign In"> <INPUT type="reset">
  </P>
</FORM>
```

The code above shows an example of a web form that sends user name and password to a web server. A form is defined in a web page with the form tag, "FORM" and "/FORM". "username" and "password" are text controls that allow a user inputs a user name and related password. When the submit button "Sign In" is clicked after filling in the data in user name and password textboxes, the data is sent to the place included in the form action attribute. A form data processing procedure will be called to process the data in the server side when the web server receives the form data. In the example above, a page named "process.jsp" in the server "awebsite.com", will process the data.

3.5 Web Server and Dynamic Web Page

A web server monitors the incoming requests on its HTTP port, and sends back a corresponding HTML web page according to the request. HTML pages are divided into two categories, static pages and dynamic pages. Static web pages are predefined web pages that are stored in a local directory of the web server. The web server can access these static web pages with their file name and file path. For a static web page request, the web server reads that HTML file, packs the file into the response message and sends the message back to the requested web browser. Dynamic web pages do not exist before a request comes and they are created dynamically according to the requested URL. The URL contains parameters that the web server uses to generate a HTML web page. After a dynamic web page has been generated, it is packed in a response message like a static web page and sent back to the requesting web browser.

Figure 3.6 shows a conceptual architecture of a web server. A typical web server consists of two parts: an HTTP communication module (HTTP Server) and a dynamic web page generation module. The architecture may be different according to different implementations of dynamic web page generation. The HTTP communication module

and dynamic web page generation module can be either implemented into one application, or executed in two different applications possibly on different machines.

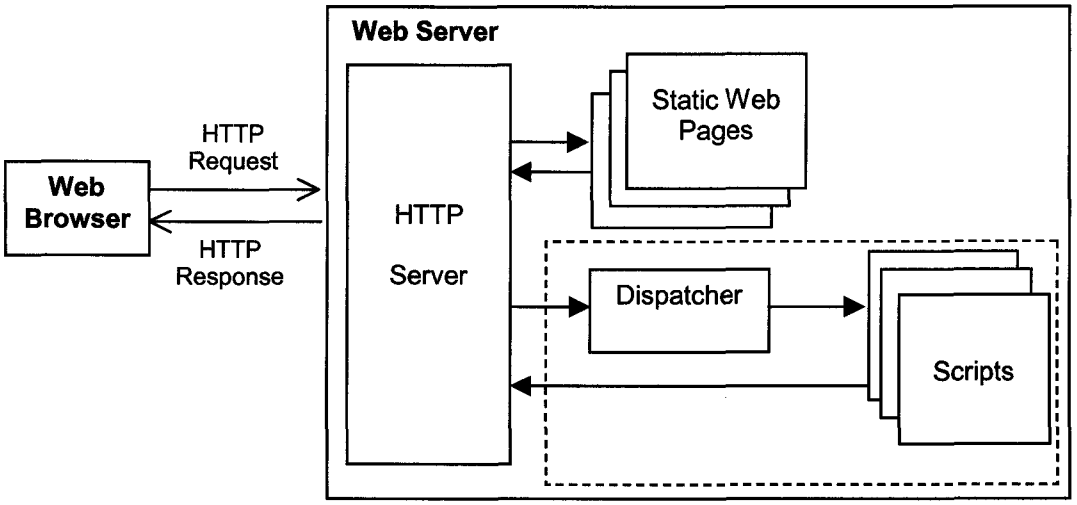


Figure 3.6 Conceptual architecture of web server

The HTTP server is a kind of communication module that is mainly responsible for data communications with web browsers. HTTP server retrieves the URL from an incoming HTTP message, analyzes the URL and passes the URL to different processing procedures. For a static web page request, the HTTP server searches the corresponding directory to read the HTML file, creates an HTTP response message encapsulating the file, and sends back to the requesting web browser. If a request is for a dynamic web page, the HTTP server passes the parameters to the dispatcher and the dispatcher calls a compiled procedure to generate a new web page. This newly generated web page is passed to HTTP server and the HTTP server sends it back to the requesting web browser.

The dynamic web page generation module in a web server consists of a dispatcher and a set of procedures written in some script languages. After these procedures are compiled, the dispatcher can call them to generate web pages. Each dynamic web page consists of predefined layout, text format, navigation links, and dynamic contents. These predefined layout, text format, and navigation links of a dynamic web page form a page template. Actually a dynamic web page is assembled by adding dynamic contents into a page template. The dynamic contents could be obtained by accessing databases, distributed objects, or other data services.

The procedures that generate dynamic web pages can be written in any programming langue. However, the programming languages, such as Basic, C or Java, are designed for programmers, and they are not straightforward to web designers. In order to make it easier in writing dynamic web page generating procedures, a series of HTML style scripts are designed. For example, JSP is a Java-based server side script language for design dynamic web page generating procedures. A JSP page is a text-based document that contains HTML tags, static contents and Java codes. These HTML tags and static contents consist of the static template and some Java codes to generate the dynamic content. A complete web page is assembled by adding this dynamic content into the static page template. For a designer, the format of a JSP page is straightforward, but actually after compiling, each JSP page is compiled as a Java method. The method is called by a dispatcher and the method generates output web pages according to the incoming parameters.

A dynamic page written in a script language for dynamic web page generation consists of static page template and codes for providing dynamic data. From the standpoint of a designer, this page can be considered as a server page in the design navigation model of a web application. We use server page in the following chapters of this proposal to stand for the dynamic web page written in a script language.

4 Modeling Web Page Navigations with Browser Behavior

4.1 Overview

A web application is an interactive system that involves web browsers, web servers and other data services. The interactions between the user and application itself are determined by the web pages of an application and hyperlinks of each page. When a user clicks a hyperlink of a web page in a web browser, a request message is created and sent to the web server of the application. After receiving the request, the web server processes the hyperlink, determines if the request page is a static page or dynamic page, and returns a corresponding page by searching a static page in its local directory or generating a dynamic web page according to the parameters of the hyperlink.

The static web pages, dynamic web pages and the hyperlinks are designed during the navigation design phase in the development of a web application. Since a dynamic web page is generated by adding dynamic contents into a predefined page template, a web application's navigation design consists of static web pages, page templates and hyperlinks that interconnect these static web pages and page templates.

As mentioned in Chapter 1, the navigation among the web pages in a web application is not only determined by the navigation design itself, the browser also has influence on it. Some inconsistencies are caused while a user clicks the "Back" or "Forward" button inside a web browser. A correct navigation design should also consider the influences of the browser's behavior.

In order to check the inconsistencies described in Chapter 1 for a web application, one possible solution is model checking. Model checking is a kind of formal verification that checks if a model satisfies the correctness requirements. In order to perform model checking for a web application navigation design, we need to establish a navigation model with browser behavior.

In the following sections, we present our methodology on establishing a navigation model with browser behavior. We use extended finite state machine (EFSM)[17] to model the navigation behavior of a web application. The EFSM-based navigation model is created by adding browser behavior into the navigation design of a web application. We assume the navigation design is provided by the designer with the format we define.

Since we are interested in the correct navigation of web applications, we only consider the attributes related to web page navigations, such as hyperlinks, authentication, etc.

4.2 Web Page Navigation Design

The navigation design defines presentations and interaction behavior of a web application. The presentation design focuses on the layout, text formats, static and dynamic contents, etc., and the design of the interaction behavior concentrates on hyperlinks, form data processing, and dynamic web page generation. Since we are concerned about the page accessibility of a navigation model, we only consider the interaction behavior of a web application. In other words, we are interested in the hyperlinks and dynamic web page generation.

4.2.1 Assumptions

As mentioned in Chapter 2, a web browser has a cache setting for the management of its browser cache. We assume the setting is always done "automatically". According to this setting, a web page is considered as a cacheable page if the cacheable setting is included in the header part of the HTTP message containing the web page. For simplicity, we assume that if a page is cacheable, the page is always fresh and there is no expire time for this cacheable page. In other words, if a web page is cacheable, it is always cacheable.

A dynamic web page is generated from a server page and each server page contains a page template that consists of static and dynamic contents. Such static or dynamic content may contain hyperlinks to other web pages within the same web application or other web sites outside the web server where the current web application is hosted. For the hyperlinks that connect to other web pages within the same web application, we assume the links are predefined and these links cannot be generated dynamically. According to this assumption, all hyperlinks contained in static HTML web pages or server pages are defined at design time. There is no new hyperlink generated during runtime of the application.

4.2.2 Web Pages

All web pages in a web application are divided into two categories: secure pages and insecure pages. Figure 4.1 illustrates the relations among these secure pages and insecure pages. Insecure pages are pages without protection, and every user can access an insecure

page anytime. Secure pages are protected pages and a user is required to pass authentication before accessing a secure page. After a user passes the authentication within a session, the user has the right to access all secure pages reachable from the authentication page.

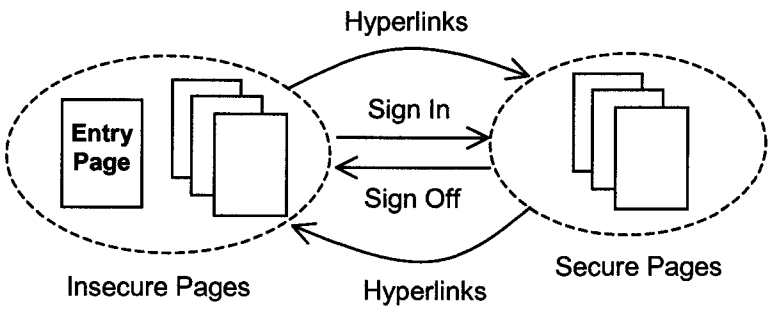


Figure 4.1 Pages of a web application

Typically, a user starts a web application by typing the home page's URL of the application in a browser's address bar. The user follows the hyperlinks defined in each page to load the subsequent pages. The address bar in a web browser gives a user the ability to visit any unprotected page by typing the page's URL. No matter whether the URL of a web page is defined in the current page or not, a user can type a page URL and corresponding parameters in the address bar to load it. Most of the URLs of a web application contain parameters that are not very easy to remember and type in the address bar of a web browser. In order to model the behavior of typed hyperlinks in the address bar of a browser, we assume there exists a special kind of insecure pages: entry pages. Each entry page is a normal insecure page, and a user can access an entry page without authentication. The main difference between an entry page and a normal insecure page is that the URL of an entry page does not contain parameters and it is easy to be remembered. Since a user can type the URL of any entry page in the address bar of a browser, each page of a web application actually contains hyperlinks to every entry page.

In the navigation design of a web application, we do not distinguish static pages and dynamic server pages. We use the term "page" to stand for both of them. Each page has a unique page ID, to identify a page in the web application. Actually a page is identified by its URL including web server address, relative directory and page name. For simplicity, we use a numeric page ID instead of the URL of the web page.

In Chapter 2, we introduced the mechanism of browser caching. If the browser cache is set to "automatically", the cache control in each page determines if a page is cacheable. According to our assumption, a web page is either a cacheable page or not, independent of the time. We use a Boolean variable to denote the cacheable property of a web page.

Table 4.1 lists all page attributes we use in the navigation design model of a web application.

Table 4.1 Page attributes

Name	Type	Note
PageID	integer	The identity of a page or page template. A page ID is an integer and pageID>=1
EntryPage	boolean	Specifies if this page can be accessed by typing URL without parameters in the address bar of a web browser. "EntryPage" lets the web application designer designates the pages that can be accessed directly in a web application. An entryPage is an insecure page.
SecurePage	boolean	Specify whether the page should be secured. For each session, a user must pass authentication before visiting secure pages.
EnableCache	boolean	Specify whether this page can be stored in the cache of a browser as cacheable page.

4.2.3 Hyper Links

All web pages of a web application are interconnected by hyperlinks. These links forms the navigation behavior of the web application. A hyperlink or URL identifies a unique web page that will be requested by a web browser. The format of a hyperlink includes web server address, relative directory that stores the page, page name, and request

parameters. We use a unique page ID to replace the whole URL except for the parameters.

A link in a web consists of a link name, a link-to page, and the link type. The link name is unique in a page and it is used to identify each link within the same page. Link-to page is the page that will be requested following the link. Since we use page ID to identify a web page or page template in a web application, this page ID can also be used to indicate the link-to page. Actually all hyperlinks in a web application have the same format and they are processed in the server side with the same technology. In order to simplify the navigation design, we separate authentication related links from normal hyperlinks. These link types include signInOK, signInFail and signOut. We name other normal hyperlinks as the type of "hyperlink".

Each authentication-related link type has specific meaning. If a user browses a secure page through "signInOK", the authentication is passed and an access control variable that indicates the user's right to visit secure pages is set to "true". When a user passes a "signInFail" link while browsing, the secure page access control variable is set to "false". "signOut" always set the access control variable to "false".

4.2.4 Session and Access Control

When a user starts a web browser and accesses a web application, a session is established between the user and the web application. The access control is related to the current session between the web browser and the web server. Because HTTP is a stateless communication protocol, a web browser and a web server cannot maintain a session actually: The logical session is established by using cookie. The cookie technology enables a web server to store unique identification information in a web browser's local directory, and when a browser sends requests to this specific web server, the identification information is included in the request message. So the web server can make use of this information to maintain a logical session with the browser.

For each session, we maintain a Boolean variable "session" for access control. The truth-value indicates that the current user has right to access all reachable secure pages, and if the value of the "session" is set to "false", the user who opens the session has no right to access secure pages. Note that "session=false" does not mean that the session is closed.

4.2.5 Navigation Design Model: An Example

Figure 4.2 shows an example of the navigation design of a web application. The example consists of 6 pages, the page ID are 1 to 6. Page 1 is an entry page, and a user can use page 1 to access the web application. Page 2, 3 and 6 are secure pages and others are insecure pages. The links in Figure 4.2 connect source pages to their destination pages.

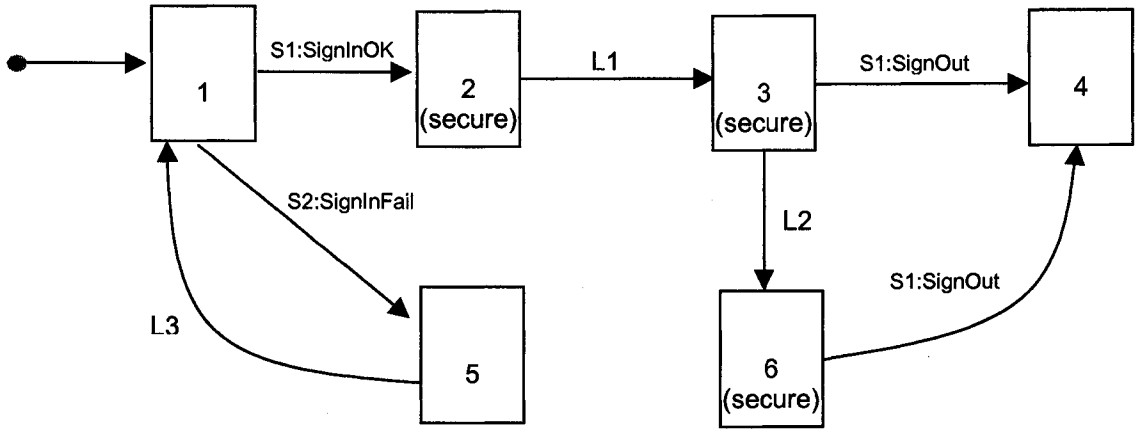


Figure 4.2 An example of a navigation design

Table 4.2 lists the attributes of each page.

Table 4.2 Attributes of the example of navigation design

Page ID	Attributes	Links
1	EntryPage=true SecurePage=false EnableCache=true	S1: signInOK,2 S2: signInFail,6
2	EntryPage=false SecurePage=true EnableCache=true	L1: hyperlink,3
3	EntryPage=false SecurePage=true EnableCache=false	L2: hyperlink,5 S1: signOut, 4
4	EntryPage=false SecurePage=false EnableCache=false	
5	EntryPage=false SecurePage=true EnableCache=false	S1: signOut, 4
6	EntryPage=false SecurePage=false EnableCache=true	L3: hyperlink, 1

4.3 Modeling Web Page Navigations with EFSM

4.3.1 Extended Finite State Machine (EFSM)

We follow the definition of EFSM in [17]. An extended finite state machine is a 5-tuple

$$M = (P, L, O, T, \upsilon)$$

Where P, L, T, υ are finite sets of states, input labels, outputs, transitions and variables respectively. A transition t of T is a 5-tuple

$$t = (s_t, e_t, l_t, P_t, A_t)$$

Where s_t, e_t, l_t are the start state, end state, and input label respectively. $P_t(\upsilon)$ is a predicate on the variable values and $A_t(\upsilon)$ are consequent post actions on the variables in the variable set υ. If the machine is in the state s_t, the input label is l_t, $P_t(\upsilon)$=True, following the transition t, the current state is changed to e_t, and the variables are changed by actions $A_t(\upsilon)$.

Figure 4.3 shows an example of EFSM. There are 3 states in the example: s_0, s_1, and s_2. Each state stands for a web page of a web application. At the beginning, the current state of the EFSM is in s_0. Each transition has three parts, and they are separated by ":" and "/". The first part is the input label. The conditions are quoted within a bracket. If the condition part is empty, it means the condition is always true. The last part contains all actions on the variables. A transition is activated if and only if the current input label is the same as the transition's input label and all conditions are true.

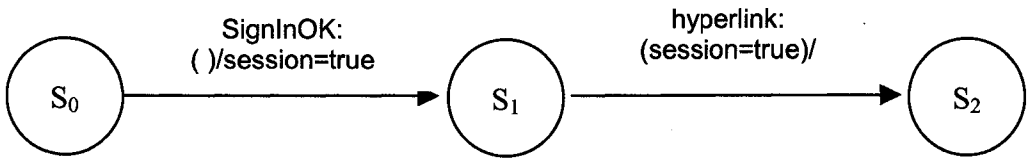


Figure 4.3 An example of EFSM

An input label "SignInOK" fires the transition from s_0 to s_1. Because there is no condition required in this transition, the transition is simply activated by "SignInOK" label. At the end of this transition, the variable session is set to true. The next transition is triggered when an input label "hyperlink" comes. The value of the session is already set to true, so the condition is true and the transition is triggered. There are no post-actions on the second transition.

4.3.2 EFSM and Web Page Navigation

The mapping from the web page navigation design of a web application to its corresponding EFSM model is straightforward. In an EFSM model, we define a state for

each web page in the navigation design of a web application, and each state's ID is the same as its corresponding page ID. A hyperlink that connects two pages in the navigation design is mapped as a transition in the EFSM model. As introduced above, a transition consists of a start state, an end state, an input label, a finite set of outputs, some conditions and some post actions. The start state and end state of the mapped transition correspond to the start page and the link-to page of the hyperlink respectively. The transition is triggered by a set of input labels, including *hyperlink*, *signInOK*, *signInFail* and *signOut*. The conditions and outputs of the transition are based on the attributes of the link-to page and the status of the variables in the start state. We define a set of rules for the transition conversion in Section 4.3.7. In addition, a variable session is declared to store the current user's session status in an EFSM model.

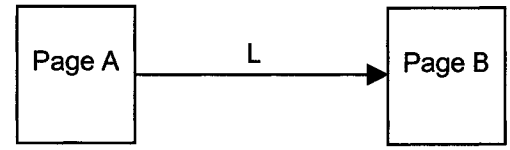
We assume the start state of an EFSM model is the state that is mapped from the home page of a web application navigation design. In our EFSM model, the reactions on a user's clicks on hyperlinks, back, or forward buttons are modeled as a transition triggered by an input label. An input label triggers a transition, and the end state of a transition becomes the current state of the system. Finally the post actions defined in this transition are performed.

4.3.3 Adding Browser Behavior into the EFSM Model

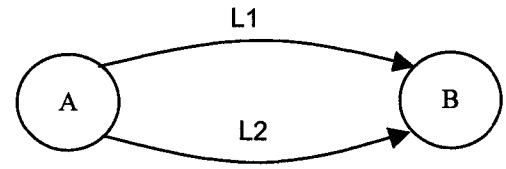
According to the introduction in Chapter 3, a browser has history stack and browser cache. While a user accesses a web application with a browser, the navigation behavior of the application is not only determined by the web page navigation design, but also determined by the browser's behavior.

Browser cache stores cacheable web pages for the future use. If a requested web page has been already stored in a browser cache, the page is loaded from the cache instead of getting from the web server of the application. In order to model the requested web page loaded from a browser's local cache or the web server, we add two transitions in the EFSM model for each hyperlink that links to a cacheable page (See Figure 4.3). These two transitions are triggered by the same input label. Because there is only one transition that will be selected during the system running, we add conditions for each of them. The condition indicates whether the page that the end state stands for has been already stored

in the browser cache. If the condition is true, the first transition is permitted to be triggered and the corresponding post-actions are applied. Otherwise, the second transition is enabled.



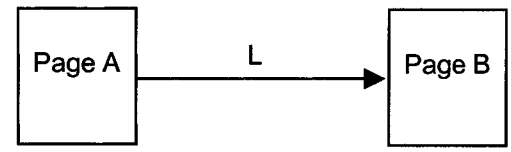
(a) A Link in Navigation Design (Page B is a cacheable page.)



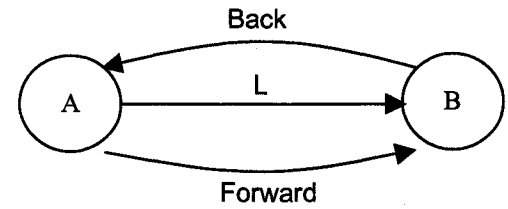
(b) Transitions in EFSM model

Figure 4.4 Transitions for a hyperlink

Other than the hyperlinks, signInOK, signInFail, and SignOut in the navigation of a web application, we should also consider the back and forward actions when a user clicks a web browser's "Back" or "Forward" button. Figure 4.4 shows the adding hyperlink, back and forward transitions for a hyperlink in the EFSM navigation model.



(a) A Link in Navigation Design



(b) Transitions in EFSM model

Figure 4.5 Add Back/Forward transitions for a hyperlink

For each hyperlink in the navigation design of a web application, a hyperlink transition is added into the EFSM model. At the same time, a back and a forward transition are added. These two transitions are used to model the actions while a user clicks the back or

forward button within a browser. As mentioned in Chapter 2, the user's operations on back and forward buttons are related to the operations on the history stack. The URL stored in the history stack will be used to retrieve the previously or subsequently visited web page.

A back or forward transition is triggered by an input label and a predicate condition which is determined by the status of the history stack. A back transition is enabled when the history stack has an item before the current item that the stack pointer points to. If the ID of a previous state stored in the history stack is the same as the end state ID of a back transition, the back transition is allowed to be triggered. A forward transition is enabled when the history stack has an item after the current one, and the ID stored after current item is the same as the end state ID of the forward transition.

4.3.4 Modeling History Stack

We define a variable history_stack as a stack that stores the previously visited state IDs (URLs). The operations on the history_stack include "push", "move_previous", "move_next", "get_previous" and "get_next". We use three pointers to indicate the status of the "history_stack". They are stack_pointer, bottom, and top. The stack pointer always points to an item that is operated on. "bottom" points to the first item that is pushed into the "history_stack", and "top" points to the last one. The definition of stack operations and conditions are listed in Table 4.3.

For push operation, history_stack stands for the history stack before push operation, and the status of the history_stack depends on the items stored in the history stack, stack pointer, top, and bottom. After the push operation, the status of history stack has been changed, and history_stack' denotes the new status of the history stack.

The "push" operation on the history stack is different from the normal stack operation. The detailed operation is divided into two steps. At first, all items stored above the stack pointer are deleted, and actually the item that the stack pointer points to becomes the last item in the history stack. Secondly, a new item, a state ID, is pushed into the history_stack, and the stack pointer points to the new item. At the same time, the top pointer is assigned the same value of the stack pointer.

Table 4.3 History stack operations

Operation Name	Operation	Description
Push	history_stack'=push(history_stack, ID)	Push the state ID into history stack
move_previous	history_stack'=move_previous(history_stack)	stack_pointer≠bottom and top-bottom>1
move_next	history_stack'=move_next(history_stack)	stack_pointer≠top and top-bottom>1
get_previous	get_previous(history_stack)	Return the state ID that is stored below the current item that the stack pointer points to.
get_next	get_next(history_stack)	Return the state ID that is stored above the current item that the stack pointer points to.

The "move_previous" and "move_next" operations move the stack pointer back and forth according to the current status of the history stack. If the history stack has more than two items and the stack pointer does not point to the bottom or top item respectively, the operations can be done.

"get_previous" and "get_next" are functions that return the value, e.g. the state ID, that is stored before or after the item the stack pointer points to. If the stack is empty, the return value is null. These two operations do not change the status of history stack, but return the corresponding values.

4.3.5 Modeling Browser Cache Repository Management

A browser cache stores previously visited web page in the browser's local storage place. Actually all previously visited pages are stored in a browser cache, but only the pages that do not exceed their expire_time are returned for their corresponding requests. A cached page is considered as fresh if the access time is no later than its expire_time. The synchronization between the pages stored in a browser cache and the web servers that provide the web pages is determined by the cache policy setting of a browser.

We define a variable "cache" to model a browser cache storage place. A new page could be added into the "cache" and a cached page could also be retrieved from the "cache" if it is in the cache. We assume all web pages stored in the "cache" are fresh, and they do not have expire time. There is no limitation on the size of the "cache" storage. The operations

on the "cache" are listed in Table 4.4. The operations are straightforward. "add_cache" adds a new state ID into the cache, and in_cache is a Boolean function that indicates if a state ID has been already stored in the cache.

Table 4.4 Browser cache operations

Operation Name	Operation	Condition
add_cache	cache'=add_cache(cache,ID)	add the page into local cache repository
in_cache	in_cache(ID)	returns true if a state ID is stored in local cache, and false otherwise

4.3.6 Constructing EFSM from Web Navigation Design

The construction of the EFSM model of a web application is based on the navigation design that the designer provides. As described above, the pages in a navigation design can be mapped into the states of its corresponding EFSM model. The transitions are added according to the hyperlinks of the navigation design. In order to describe the web navigation design of a web application and perform the construction of EFSM based navigation model, we define a set of XML tags in 4.2. These tags allow a designer to describe a navigation design with an XML file. Because XML has been supported by a wide range of programming languages, it is easy to process a navigation design with a specific programming language. Once a web navigation design is obtained, the constructing of EFSM model is based on the conversion rules.

The procedure of constructing EFSM based navigation model includes the following steps:

1. Create states for all pages defined in the navigation model, and each state ID is the same as the page ID in the navigation model.
2. Add transitions according to the hyperlinks and authentication-related links in the navigation model. Each link type is considered as an input label for a transition. The transitions are enabled by conditions based on the status of "history_stack" and "cache" of the model. The actions are operations on the EFSM variables, including "history_stack", "cache" and "session".
3. Add back/forward transitions for each transition generated following a link in the web page navigation design.

The adding of each transition in the EFSM model must follow one of transition construction rules listed in section 4.3.7.

4.3.7 EFSM Conversion Rules

As described in section 4.3.6, the conversion from a web navigation design to its EFSM model is divided into 2 steps, mapping states, and adding transitions. Obviously each page in a navigation design could be mapped into its corresponding state in the EFSM model. The adding of transitions is determined by the page attributes and EFSM variable status. For each hyperlink in a navigation design, there might exist more than one transitions. In order to cover all scenarios while, we define the conversion rules that regulate the conversion from a navigation design into its EFSM model.

We define two types of rules listed in Table 4.5-4.7. Rule 1.1-Rule 4.9 map the links of a navigation design into their corresponding EFSM transitions. Rule 5.1 –Rule 6.5 regulate the additional back/forward transitions. Each rule consists of two parts: one is related to the page attributes of a navigation design and another defines a corresponding transition of the converted EFSM model.

For example, in a navigation design, page A contains a hyperlink to page B. Page B is an insecure page and it could be stored in a web browser's cache as cacheable page. According to the attributes of page B and link type from A to B, two rules, rule 4.1 and rule 4.2, could be applied when transferring the navigation design into EFSM navigation model. For rule 4.1, a transition with condition "in_cache(B)=True" is added. The post action is "push(history_stack, B)", which pushes the state B's state ID into the history stack. This rule defines a transition when a user visits a cacheable page at first time. Rule 4.2 defines a transition that simulates a user's access to a page when the page has already been stored in a browser's local cache.

In addition to the states related to web pages in the navigation design of a web application, we also define an error state and this state is mapped to an error page that indicates some errors occurred during web page generating. The transitions to this error state indicate internal error while the application is executed. A typical error occurs when a user tries to access a secure page after a signOut is performed. Since all transitions of the error state contain conditions and each condition determines if an error transition

could be triggered. If a design does not contain any error, all transitions to the error state will be disabled. The actions on a transition with an error state include clearing current history stack, and pushing the error state into the history stack. These two post actions are abstractions of the observation of web mail systems, such as hotmail, yahoo mail, and the web mail system of University of Windsor.

Since the rules cover all the scenarios of the web page navigation design, when applying these rules, the EFSM based navigation model will be constructed completely from the navigation design of a web application.

Table 4.5 Transition rules I

No.	Navigation Design		EFSM Transitions			
	Navigation Link	Page Attribute	State & Input	Output	Condition	Post Actions
1.1	signInOK A → B	B.enableCache=True	Start state: A end state: B input label: signInOK	StateID=B Secure=True Source=Server		history_stack'= push(history_stack, B) cache'= add_cache(cache,B) session=true
1.2		B.enableCache=False		StateID=B Secure=True Source=Server		history_stack'= push(history_stack, B) session=true
2.1	signInFail A → B	B.enableCache=True	Start state: A end state: B input label: signInFail	StateID=B Secure=False Source=Server		history_stack'= push(history_stack, B) cache'= add_cache(cache,B) session=false
2.2		B.enableCache=False		StateID=B Secure=False Source=Cache		history_stack'= push(history_stack, B) session=false
3.1	signOut A → B	B.enableCache=True	start state: A end state: B input label: signOut	StateID=B Secure=False Source=Server		history_stack'= push(history_stack, B) cache'= add_cache(cache,B) session=false
3.2		B.enableCache=False		StateID=B Secure=False Source=Cache		history_stack'= push(history_stack, B) session=false

Table 4.6 Transition rules II

No.	Navigation Design		EFSM Transitions			
	Navigation Link	Page Attribute	State & Input	Output	Condition	Post Actions
4.1	hyperlink A → B	B.enableCache=True B.securePage=False	start state: A end state: B input label: hyperlink	StateID=B Secure=False Source=Cache	in_cache(B)=True	history_stack'= push(history_stack, B)
4.2				StateID=B Secure=False Source=Server	in_cache(B)=False	history_stack'= push(history_stack, B) cache'= add_cache(cache,B)
4.3		B.enableCache=True B.securePage=True	start state: A end state: B input label: hyperlink	StateID=B Secure=True Source=Server	in_cache(B)=False ^ session=on	history_stack'= push(history_stack, B) cache'= add_cache(cache,B)
4.4			start state: A end state: B input label: hyperlink	StateID=B Secure=True Source=Cache	in_cache(B)=True ^ session=on	history_stack'= push(history_stack, B)
4.5			start state: A end state: Err input label: hyperlink	StateID=Err Secure=True Source=Server	in_cache(B)=False^ session=off	history_stack'= clear(history_stack) history_stack'= push (err)
4.6			start state: A end state: B input label: hyperlink	StateID=B Secure=True Source=Cache	in_cache(B)=True^ session=off	history_stack'= push(history_stack, B)
4.7		B.enableCache=False B.securePage=False	start state: A end state: B input label: hyperlink	StateID=B Secure=False Source=Server		history_stack'= push(history_stack, B)
4.8		B.enableCache=False B.securePage=True	start state: A end state: B input label: hyperlink	StateID=B Secure=True Source=Server	session=on	history_stack'= push(history_stack, B)
4.9			start state: A end state: Err input label: hyperlink	StateID=Err Secure=True Source=Server	session=Off	history_stack'= clear(history_stack) history_stack'= push (err)

Table 4.7 Transition rules III

No.	Navigation Design		EFSM Transitions			
	Navigation Link	Page Attribute	State & Input	Output	Condition	Post Actions
5.1	back A→B	B.enableCache=True B.securePage=False	start state:A end state: B input label: back	StateID=B Secure=False Source=Cache	get_previous (history_stack)==B	history_stack'= move_previous (history_stack)
5.2		B.enableCache=True B.securePage=True	start state:A end state: B input label: back	StateID=B Secure=True Source=Cache	get_previous (history_stack)==B	history_stack'= move_previous (history_stack)
5.3		B.enableCache=False B.securePage=False	start state: A end state: B input label: back	StateID=B Secure=False Source=Server	get_previous (history_stack)==B	history_stack'= move_previous (history_stack)
5.4		B.enableCache=False B.securePage=True	start state: A end state: B input label: back	StateID=B Secure=True Source=Server	session=on^ get_previous (history_stack)=B	history_stack'= move_previous (history_stack)
5.5			start state: A end state: Err input label: back	StateID=Err Secure=False Source=Server	session=off^ get_previous (history_stack)==B	history_stack'= clear(history_stack) history_stack'=push (err)
6.1	forward A→B	B.enableCache=True B.securePage=False	start state: A end state: B input label: back	StateID=B Secure=False Source=Cache	get_next (history_stack)==B	history_stack'= move_next (history_stack)
6.2		B.enableCache=True B.securePage=True	start state: A end state: B input label: back	StateID=B Secure=True Source=Cache	get_next (history_stack)==B	history_stack'= move_next (history_stack)
6.3		B.enableCache=False B.securePage=False	start state: A end state: B input label: back	StateID=B Secure=False Source=Server	get_next_url(history_stack)==B	history_stack'= move_next (history_stack)
6.4		B.enableCache=False B.securePage=True	start state: A end state: B input label: back	StateID=B Secure=True Source=Server	session=on^ get_next (history_stack)==B	history_stack'= move_next (history_stack)
6.5			start state: A end state: Err input label: back	StateID=Err Secure=False Source=Server	session=off^ get_next (history_stack)==B	history_stack'= clear(history_stack) history_stack'= push (err)

4.4 Summary

This chapter introduces an EFSM based modeling method for web page navigations considering browser behavior on the history stack and browser cache of a web browser. With EFSM we model each page in a web page navigation design as states and the navigations among these pages are modeled as transitions. For each hyperlink in a web page navigation design, we add its corresponding back and forward transitions in the derived EFSM model. The behaviors of history stack, browser cache, and authentication are modeled as actions of these transitions.

Compared with other web navigation models that focus on the navigation design itself, the navigation model created in this chapter combines browser behavior into the navigation design. Because this model is created from the view of web browser, it models the running of a web page navigation design under a real browser environment.

The browser behavior used in this chapter is obtained from more than one resource and there is no publication that introduced them together. The history stack and browser cache are introduced in [11] and [20] respectively. We combine them and the experiences of using Internet Explorer and Netscape Navigator together and construct the conceptual model of a web browser. We also refer HTTP request, response and the setting of cache controls defined in HTTP1.1[24].

5 Implementation of Extended Finite State Machine

This chapter describes the implementation of constructing EFSM-based web application navigation model from the web application design by adding browser behavior. The implementation includes reading web application design written in XML, and EFSM construction following conversion rules. All implementation work has been done with Microsoft Visual Basic .Net.

5.1 Navigation Design of Web Applications

5.1.1 Describing Navigation Design of Web Applications with XML

As described in Chapter 3, the navigation design of a web application consists of pages and hyperlinks. For simplicity, we concentrated on the page attributes and links related to page navigation and authentication. All other web page properties, such as page layout, font, image, etc., are not considered in our web application navigation design.

We use XML to describe the navigation design of web applications. The XML schema of web navigation design is shown in Figure 5.1.

The root element of the navigation design of a web application is "web_design". This root tag is used to encapsulate all pages and links in the navigation design. The sub tags under "web_design" are "page" and "link".

A "page" is a composed element that contains other elements. The elements under "page" tag are "secure_page", "entry_page", "enable_cache" and "link". Each page has an attribute "id", which is defined by the attribute tag definition. The data type of page "id" is integer. "secure_page", "entry_page" and "enable_cache" are boolean values. We assume the page ID starts from integer 1 and is increased continuously in a web navigation design. This assumption will make the implementation of EFSM verification model easier.

A "link" is an empty tag and it has two attributes, "link_to" and "type". "link_to" indicates the page that current link points to, and "type" are the hyperlink type we defined in chapter 3. They are "hyperlink", "signInOK", "signInFail" and "SignOut".

```
<?xml version="1.0" encoding="utf-16" ?>
<xs:schema id="web_design" >
<xs:element name="web_design">
    <xs:complexType>
        <xs:choice maxOccurs="unbounded">
            <xs:element name="page">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="secure_page"
                                    type="xs:boolean"
                                    minOccurs="0" msdata:Ordinal="0" />
                        <xs:element name="entry_page"
                                    type="xs:boolean"
                                    minOccurs="0" msdata:Ordinal="1" />
                        <xs:element name="enable_cache"
                                    type="xs:boolean"
                                    minOccurs="0" msdata:Ordinal="2" />
                        <xs:element name="link"
                                    minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                                <xs:attribute   name="link_to"
                                                type="xs:integer" />
                                <xs:attribute   name="type"
                                                type="xs:string" />
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                    <xs:attribute name="id" type="xs:integer" />
                </xs:complexType>
            </xs:element>
        </xs:choice>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 5.1 XML schema for web navigation design

Since data type can be defined in XML schema, it is easy to convert the data within XML tags into the corresponding data type. XML reader can do all the conversion work while reading the XML file, and there is no need for additional codes.

Figure 5.2 shows a simple example of navigation design expressed in XML tags we defined. There are only two pages in this example: page 1 and page 2. Each page contains page attributes and links. According to the web design XML schema, a "web_design" can contains multiple pages and each page can also have more than one links or none. Once the navigation design of a web application is written in the XML format, the conversion

tool we developed will add back and forward transitions among the pages and construct the EFSM based navigation model.

```
<web_design>
        <page id="1">
                <secure_page>0</secure_page>
                <entry_page>1 </entry_page>
                <enable_cache>0</enable_cache>
                <link link_to="2" type="hyperlink" />
        </page>
        <page id="2">
                <secure_page>0</secure_page>
                <entry_page>0 </entry_page>
                <enable_cache>1</enable_cache>
        </page>
</web_design>
```

Figure 5.2 An example of navigation design described in XML

5.1.2 Object Model of Web Application Navigation Design

In order to process the navigation design of web applications, we define the object model of it. When an XML-based navigation design has been read, an object model that contains page objects and link objects are generated. The structure of the object model of navigation design consists of three kinds of elements: DesignPagesCollection is a collection object that only consists of page objects. Each page object is generatted from page class that contains page attributes and links defined in Chapter 4. Each page object is identified by its page_id, and this id has the same value of "id" attribute in the page tag of XML web navigation design. The link class consists properties of "link_to" and "link_type". The collection variable "links" is declared in the page class, and it contains all link objects of current page. The object model is shown in Figure 5.3.

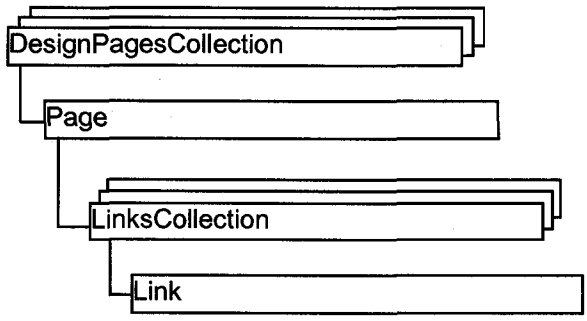


Figure 5.3 Object model of navigation design

5.1.3 Constructing Web Application Design Object Model

The reading of web navigation design XML file is based on dataset object of Microsoft .Net framework. DataSet is a class that contains data tables, and relations. It embeds XML operations that support XML reading and writing. When an XML file is processed by a dataset object, the structured XML data is mapped to tables with one-to-many relations. The XML reader in a dataset can assign primary keys for each table and establish their relations. The construction of tables is divided into two steps: 1) Reading "web_design" XML schema, and creating table schema. 2) Reading "web_design" XML file with the full file name.

Once the web navigation design XML file has been read by dataset object and the related tables are generated, the page object generating is performed by reading every record in a table and assigning the fields to the corresponding variables of page object. The link objects of each page are created by searching all links in the link table with the page id.

5.2 Constructing EFSM Navigation Model from Web Navigation Design

5.2.1 Overview

As described in Chapter 4, the EFSM-based navigation model is constructed by adding browser behavior into the navigation design of a web application. The navigation design is provided by the web application designer and it follows the XML format defined in section 5.1. We model a browser's behavior on history stack and browser cache in Chapter 4 and define a set of rules that map the hyperlinks from navigation design into back/forward and browser cache operations on EFSM model. The EFSM construction work is to convert the navigation design on pages into EFSM states and transitions with operations on history stack and browser cache. Figure 5.4 shows the work of EFSM model construction.

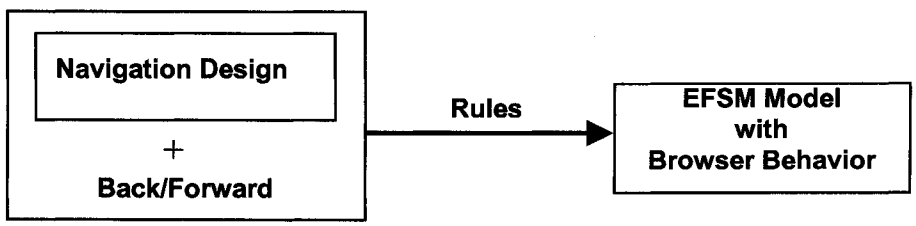


Figure 5.4 EFSM based navigation model construction

5.2.2 EFSM Object Model

The EFSM object model consists of an EFSM state collection that consists of state objects. Each state object has a state ID and a collection of all outgoing transitions. Each transition is an object that indicates the transition's from_state, to_state, conditions, input label and post-actions. "from_state", and "to_state" are variables that store state objects. Other members of a transition object are declared as strings. The input labels used in a transition class include "back", "forward" and all hyperlink types in web page navigation design.The EFSM object model is shown in Figure 5.5.

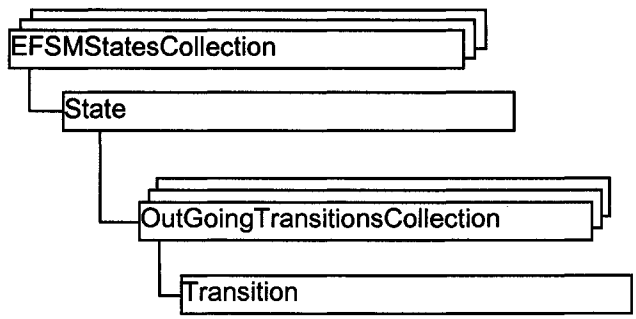


Figure 5.5 EFSM object model

The variables defined in an EFSM model include "history_stack", "cache", and "session". "history_stack" and "cache" are the same variable described in Chapter 4. "session" is a boolean variable that is set to true when a user login the web application by "signInOK" transition, and it can be set to false when "signInFail" and "signOut" transitions are triggered.

In an EFSM, a transition is fired when the transition's input label comes and all conditions of the transition become true. Conditions are the results of predicate functions upon the EFSM variables. In order to simplify the processing of the EFSM conditions, we define a set of enumeration values that stand for different predicate functions upon the EFSM variables. The "condition" in a transition is a string that consists of the enumeration values separated by "|". The enumeration values are InCacheTrue, InCacheFalse, SessionOn, SessionOff, PreviousStateID, NextStateID. InCacheTrue and InCacheFalse indicate whether the end state's ID is already stored in browser cache. Different boolean values lead to different operations on browser cache and history stack. SessionOn and SessionOff stand for the different value of session. SessionON means

session condition is "session=true" and SessionOff indicates the condition is "session=false". PreviousStateID and NextStateID stand for the conditions that the state ID stored in the previous or next position of stack pointer in history stack equals to the end state ID of current transition.

Post-actions of a transition include MovePrevious, MoveNext, AddCache, SetSessionOn, SetSessionOff and Err. MovePrevious, MoveNext and AddCache are described in Chapter 4. SetSessionOn, and SetSessionOff are actions that set the value of session true or false. Err actions include clear the history stack and push an error state ID into the history stack. We already add hyperlinks from a page to the entry page except the entry page itself. All post_actions in a transition are stored in the property "post_actions" in a transition object. The data type of "post_actions" is string and all actions are separated by the character "|". Since we implement the EFSM model with Promela, we use enumerated values to stand for the post actions.

5.2.3 Algorithm

The construction of EFSM model is based on web design object model introduced in section 5.1. The algorithm is:

1 Add an additional error state "Err" into EFSM states collection.
2 For each page of web design, create a corresponding state node, and the state ID of the node is assigned as the page ID. Add this node into EFSM states collection.
3 For each page except the entry page itself, add hyperlinks to all entry pages.
4 For each page of web design
  4.1 Search its corresponding state node in EFSM States collection, and assign this node to variable from_state;
  4.2 For each link in current page
    4.2.1 Search the state node with link_to page ID, and assign this node to variable to_state.
    4.2.2 Get the link-to page of current transition
    4.2.3 Add a transition into from_state's outgoing transitions collection and to_state's incoming transitions collection. The condition and post actions are added according to the rules defined in Chapter 4.
    4.2.4 Add "back" transition into from_state's incoming transitions collection and to_state's outgoing transitions collection, the condition and post actions are added following the rules defined in Chapter 4.
    4.2.5 Add "forward" transition into from_state's outgoing transitions collection and to_state's incoming transitions collection, the condition and post actions are added following the rules defined in Chapter 4.

5.2.4 Rules

The rules defined in Chapter 4 are used in 4.2.3, 4.2.4 and 4.2.5 of the construction algorithm. While adding transitions, the corresponding rules determine the transition conditions and post actions according to the attributes of the link-to page in navigation design, link type, and the values of EFSM variables.

In 4.2 of the EFSM construction algorithm, when a link object is processed, the page that owns the link and link-to page are obtained. The page IDs of these two pages can be used to obtain the corresponding state objects. When the hyperlink and forward transitions are added, the link-to page's attributes are used to select the corresponding rules. For back transition, current page's attributes are considered when selecting rules.

6 Spin and Promela

6.1 Overview

Model checking is one kind of formal verification, and it relies on building a finite model of a hardware or software system and checking that the model satisfies the desired properties. In order to perform model checking, a formal abstract model has been established in advanced. The desired correctness properties are expressed in a concise and unambiguous way. A series of model checking techniques will be applied to perform exhaustive state analysis in order to search the desired properties in verification models.

Spin is a state-based model-checking tool developed by Genalod Holzmann at Bell Labs. It is designed for the verification of distributed systems, especially for the verification of communication protocols. The native modeling language of Spin is Promela (Process/Protocol Meta Language), a modeling language that is used to describe verification model and correctness requirements. Actually Spin stands for Simple Promela Interpreter, and it is a model checker generator. It accepts a verification model and correctness requirement, and generates a C code model checker. After compiling and executing the model checker, the final results are reported. The correctness requirements can be expressed in 3 aspects: assertions, state labels and never claims. Never claims are used to describe the temporal properties of a Promela model and it can also be expressed in LTL (Linear Temporal Logic) expressions. Spin embeds a LTL converter, which translates LTL formula into never claims in Promela.

SPIN is a free, well-documented, and actively maintained model checking tool with a large and rapidly growing user-base. It won the 2001 ACM System Software Award. Other awards include TCP/IP, Unix, Java, and Tcl/Tk.

6.2 Spin

As mentioned above, Spin is a translation tool. It can be used separately as a command-line application or it can be used within Xspin, a graphical interface application. In addition, the execution of Spin needs a standard C compiler. Figure 6.1 shows the structure of Spin(Redraw from Fig.1 in [13]).

Xspin is a front-end graphical tool that is responsible for processing inputs and outputs. Developed in Tcl/Tk, Xspin is a standard GUI (Graphical User Interface) application that can be executed independently under Unix and Windows platforms. Xspin runs Spin in

the background with corresponding command parameters, and presents the results gathered from Spin. Promela verification models and LTL expressions could be written under Xspin.

Spin has two functions: simulation and verification. A user can select either simulation or verification in the menu of Xspin. For simulation, the parsed Promela model could be executed line by line, and a message sequence chart will show the message passing within communication channels. For verification, the Promela and LTL correctness claims are translated into a C model checker. After this model checker is compiled, an executable verifier is generated. When this verifier is executed, it performs on-the-fly modeling checking according to model checking algorithms provided with Spin. If the verification model does not satisfy the correctness requirements, the counter examples are created and these counter examples are returned to Xspin. Xpin will present these examples with simulation.

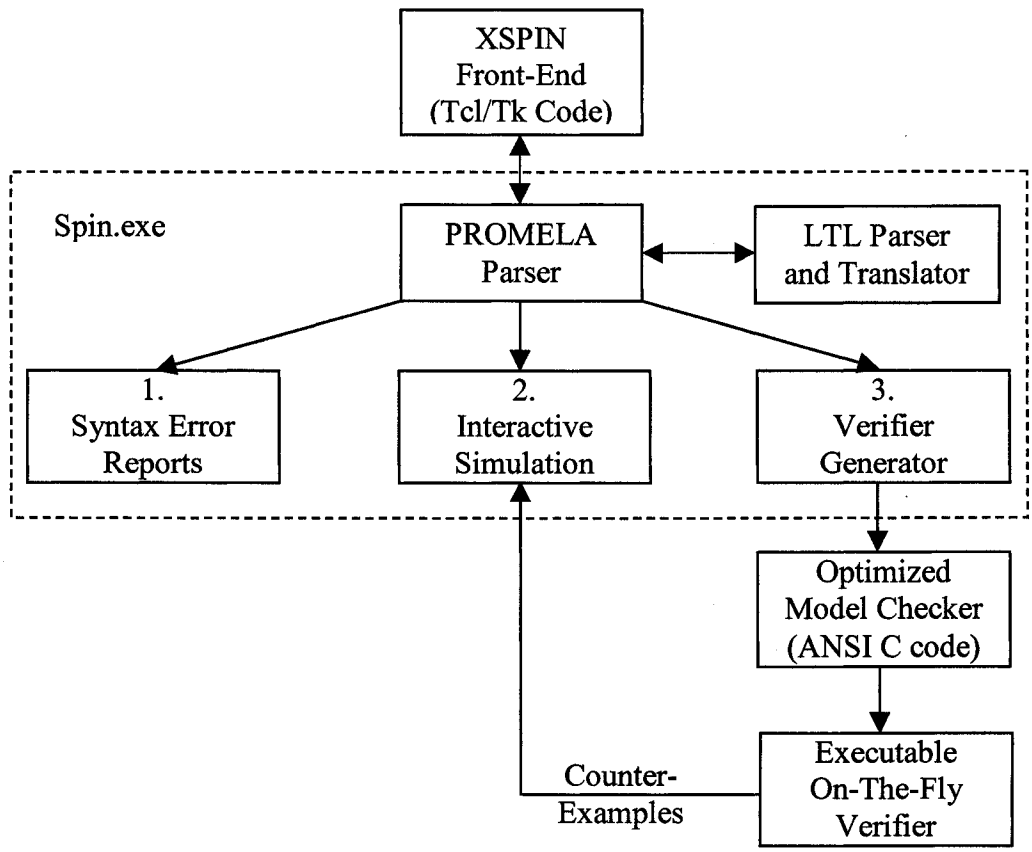


Figure 6.1 The structure of Spin

6.3 Promela Basics

Promela is a formal modeling language that is used to describe distributed systems. A model of distributed systems gives an abstract view over a system, focusing on certain important aspects and ignoring low-level details. As a modeling language, Promela has strong abilities in describing communications within processes in a distributed system. The communications by shared variables or message passing can be easily modeled with a simple statement. The non-deterministic execution of a distributed system could also be modeled with executable conditional conditions. Promela also provides the ways that the correctness requirements are described.

6.3.1 Procsses, Channels and Variables

In Promela, a basic distributed system contains three types of objects: processes, message channels and state variables.

- **Process**

A process must be declared before they are instantiated. A process can communicate with other processes synchronously or asynchronously. A sample process declaration is shown below.

```
proctype A(int x, int y) {
  /*Statements of the process*/
}
```

The process declaration starts with "proctype" and follows the process name and a list of parameters. The process body is marked with "{" and "}". A process declaration only defines a process type, and a process definition can have multiple instances after instantiated by "run" statement.

```
init {run A(1,2); run A(5,6)}
```

In the above example, two processes are initialized with the same process definition. "init" is the initial process of a system. It is similar as the *main* function in a C program. The initial process can initialize global variables, create message channels, and instantiate processes. "run" is a unary operator and it is used to instantiate processes from process definition. The values of parameters are initial values of the local variables used in the process.

- **Variable**

Variables in Promela include global variables storing system information or local variables that is only used in a process. There are six predefined data types: *bit*, *bool*, *byte*, *short*, *int*, and *chan*. The first five data types are basic data types and the variables of these types could only be of assigning a single value. The last one, chan, is used to define message channels that store more values depending the channel buffer size and the structure of each message.

The declarations are the same as C. A single variable or an array of the same data type could be declared. The *bit* defines a variable that holds one bit of the information. A *byte* variable is the same as C unsigned char. The length of *int* and *short* variables depends on the machine that the SPIN is executed. An example of variable declaration is shown below.

```
bool switch;
int x,y;
byte msg;
```

Each Promela model can have one enumeration data type which contains symbolic constants. The enumerated symbolic values are defined with *mtype={...}*, and a variable is also declared with *mtype*.

```
mtype = {green, red, yellow };
mytype traffic_light;
traffic_light=green;
```

The above example shows the declaration of the enumeration constants and a variable. The first statement declares mtype contains three traffic light colors, "green", "red", and "yellow". After an mtype variable *traffic_light* is declared, the color value could be assign to it.

- **Message Channel**

Message channels are used to model the data communication from one process to another by message passing. Each channel declaration defines the channel name, message type and maximum messages stored in the channel.

```
chan q=[16] of {int, mtype, bool}
q!5,green,1;
q?var1,var2,var3;
```

The above example shows a message channel declaration and operations. A message channel q is defined. *q* can store maximum 16 messages. Each message has three fields:

an integer value, an mtype value and a boolean value are stored in these three fields respectively. The operations on a channel include send and receive. The operator *!* and *?*stands for send and receive respectively. The second statement sends a message into the channel. The message contains three fields with the values 5, green and 1 respectively. The third statement shows a message receiving. Three variables, var1, var2 and var3 are assigned the value from the first message in the message channel q.

6.3.2 Executablility of Statements

In Promela, a statement can only be passed if it is executable, otherwise it is blocked. There is no difference between conditions and statements. A Boolean condition is also considered as a statement. If the condition is false, the execution is blocked until the condition becomes true. The executability of Promela statements forms the basis of synchronization in a verification model.

```
while (a!=b) skip; /*wait for a==b*/

(a==b)
```

The two statements listed above have the same meaning, and there is no difference between them.

Assignments to variables are the same as C assignments, and they are always executable.

6.3.3 Flow Control Statements

In Promela, there are three control flow statements. They are selection, repetition and unconditional jumps.

A selection statement begins with *if* and ends with *fi*. A statement block is separated with the flag ::. Following the separator, there is a condition or boolean variable that controls the execution of the statement block. When the condition becomes true, the subsequent statements could be executed. If more than one conditions become true at the same time, the executable block is selected randomly.

```
if
:: a>0 ->
   /* statement block 1 */
:: a=0 ->
   /* statement block 2 */
:: a<0->
   /* statement block 3 */
:: /* statement block 4 */
fi
```

In the example above, *a* is an integer. Among the first three conditions, only one can be true at one time. Suppose the current value of a is 0, then the second condition becomes true. The statements in block 2 could be executed. Because block 4 has no condition, it also could be executed. The statements in block 2 or block 4 will be selected randomly.

A repetition statement begins with the keyword *do* and ends with *od*. Like *if* statement, do statement also contains select conditions. When a condition becomes true, the subsequent statements could be executed. The main difference is after the execution of the statement block, the control is set back to the beginning of the *do* statement. A new statement block is selected and the statements in the selected block become executable. In order to exit the repetition statement, a break or unconditional jump statement is used. In the following example

```
do
:: a>0 -> break;

:: a=0 ->
   /* statement block 1 */
:: a<0->
   /* statement block 2 */
od
```

the *do* statement will execute until *a*'s value is greater than 0.

The unconditional jump statement is a *goto* statement. A label is put in the desired position before a statement. When a *goto* statement is executed, the next statement under the label will be executed.

```
do
:: a>0 -> goto done;
:: a=0 ->
   /* statement block 1 */
:: a<0->
   /* statement block 2 */
od
done:
   a=0;
```

The example above illustrates a *goto* statement. "*done*" is a label which is syntactically followed by a colon. When "*goto done*" is executed, the next executable statement is "*a=0*".

6.3.4 Atomic Sequences

When a sequence of statements is defined as atomic sequence by enclosing these statements in curly braces with the keyword *atomic*, the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. All statements except for the first statement must be executable if the atomic statement is executed. This means when the first statement in an atomic sequence becomes executable, there is no statement that can block the execution of all statements in the atomic sequence. Otherwise a run-time error is caused. The use of atomic sequences can significantly reduce the complexity of verification models.

6.4 Correctness Requirements

The behavior of a Promela verification model is determined by the set of all execution sequences it can perform. Each execution sequence is considered as a finite, ordered set of states. A state consists of all values for local and global variables, all control flow points of running processes, and the contents of all message channels. Once model checking is applied to a Promela model, all possible sequences are checked according to correctness requirements by the model checker that is generated with Spin. The model checker performs state reachable analysis with embedded state reduction algorithms and the violations will be reported.

There are 3 ways that the correctness requirements are expressed, assertions, state labels, and never claims.

6.4.1 Assertions

Assertions are statements that define invariant conditions for some specific process, or the whole system. A process's invariance is declared in a process, and the system invariance is declared in a standalone monitor process. The formats of assertions are shown below.

```
assert (invariant condition)

proctype monitor ( ) {
        assert(invariant condition)
}
```

The first one is an assertion statement, and the second one is an assertion defined in a separate monitor process as system invariance.

6.4.2 State Labels

State labels include end labels, progress labels and accept labels. The label name must be unique in a proctype definition. If more than one state labels with the same type are used, the labels have the same prefix, and different label names. For example, "end" is a state label that describes the proper end states. If a process contains two end labels, one label could be "end_first", and another is "end_second".

End State

An end state label is any label that starts with "end". Each end state label indicates that the current state can be considered as a proper end state in an execution sequence.

In a Promela model, all execution sequences either terminated after a finite number of state transitions, or they cycle back to a previously visited state. A final state in a termination sequence could be considered as a proper end state if it satisfies: 1) Every process that was instantiated has terminated; 2) All message channels are empty. All other end states are unexpected end states, including deadlocks. A model checker can check all end states and reports the unexpected end states after analysis. However, not all unexpected end states are necessarily bad, because some processes could be alive all the time. For example, server processes stay alive and wait to be activated after user process terminates. We must distinguish the expected or valid end states from the unexpected, or invalid ones. End state labels identify individual process states as expected end states even if the process is still alive.

Progress State and Accept State

Like unexpected end states, Spin also checks invalid cyclic execution sequences. There are two categories of invalid cyclic sequences: non-progress cycles and livelocks. Two common properties that specify the cyclic sequences are described in [8]: "There are no infinite behaviors of only unmarked states" and "There are no infinite behaviors that include marked states". The first one explains that the system cannot infinitely cycle through unmarked states, and the second one is the opposite of the first one. The marked states are progress-states, and the execution sequences that violate these two properties are called *"non-progress cycles"* or *"livelocks"*.

When SPIN is used to perform exhausted analysis of a Promela model, it will find all invalid cyclic execution sequences. We must define what cyclic sequences are invalid sequences. For progress cycles, we need to specify precisely which statements in the specification constitute progress. Progress state labels are used to specify progress cycles, thus violations are reported during model checking. For the non-livelock cycles, accept state labels are used to express properties that something cannot happen infinitely often. An accept-state label indicates a state that may not be part of a sequence of states that can be repeated infinitely often.

Progress and Accept state labels have "progress" and "accept" prefixes respectively, and they are unique in one process definition.

6.4.3 Never Claims

The never claim is used to express temporal order of propositions, and all temporal claims are claimed to be impossible. Therefore, if a temporal claim is matched, it means a violation is detected. In a Promela verification model, there is only one never claim statement.

The syntax of never claim is:

```
never {... body ...}
```

Where "never" is a keyword, the functionality of "never" is the same as "proctype". The body consists of statements that do not change the behaviors of the verification model. Every statement is interpreted as a proposition and the executability of a statement depends on the value of the proposition.

The checking of matched temporal ordered propositions starts from the first state of the model, i.e. the system state that is reached after first statement in the init process has been executed. For each system state along an execution path, the executability of the corresponding proposition of the never claim is evaluated. If the proposition becomes true, the statement of the never claim is executable and the state of the claim moves to the next statement. If the proposition is false, the behavior of the Promela model does not match the temporal claim, and there is no temporal claim violation so far. The model

checker continues the search by inspecting other reachable system states. After all statements in the body of a never claim become executable, a violation is reported.

```
never {
        do
        :: skip
        :: P->break
        od;
accpt:  do
        :: !Q
        od
}
```

The example above shows a temporal claim that expresses "every state in which property P is true is followed by a state in which property Q eventually becomes true". The first do statement indicates the start point that P becomes true, and second do statement is responsible for searching the condition that Q becomes true. If Q becomes true, then !Q blocks the further state property evaluation and indicates that Q is impossible to be false infinitely. If Q is false infinitely, the claim is matched and a violation is reported.

6.4.4 Linear Temporal Logic

In addition to never claims the temporal property of a system can also be expressed as linear temporal logic (LTL) formula. An LTL expression is a combination of predicate logical expressions and temporal operators. The temporal operators include "[]" (always), "<>" (eventually), and "U" (strong until). "[]" means a logical expression P has the truth value in a time series. "<>" stands for the logical expression P becomes true eventually in a time series. "U" involves 2 operands P and Q. "PUQ" is true when P becomes true until Q becomes true.

Spin provides the translation from LTL (Linear Temporal Logic) formula into never claim statement in Promela. The translation can be done either by typing "spin –f LTL_formula" under command line, or entering the formula in window of LTL property manager of Xspin.

For example, one property of a system is "any occurrence of p will eventually be followed by q". The property can be translated into LTL formula "p-> <>q". The translation from LTL formula into never claim will be done by typing *spin –f "p->(<>q)"*, and the never claim is:

```
never {    /* p->(<>q) */
T0_init:
        if
        :: (((! ((p))) || ((q)))) -> goto accept_all
        :: (1) -> goto T0_S2
        fi;
T0_S2:
        if
        :: ((q)) -> goto accept_all
        :: (1) -> goto T0_S2
        fi;
accept_all:
        skip
}
```

7 Translating EFSM into Promela

This chapter describes the translation from EFSM model generated in Chapter 5 into Promela model, and introduces the verification work on the translated Promela model with correctness requirements.

7.1 Expressing EFSM with Promela

A Promela model is mainly divided into three parts: data type definition and global variable declaration, process definition, and process instantiation. We implement these three parts separately. Mtype and Marco are defined first, and these definitions will be used in the following Promela statements. All global variables including EFSM variables are declared. The states and transitions are implemented in one process definition. At last, the process is instantiated with "init" statement. The example of EFSM Promela model is illustrated in Appendix D.

7.1.1 Data Type Defintion and Variable Declaration

In the first part of our verification model with Promela, we define all marcos, data type definitions and variable declarations.

In order to make the Promela model readable, we use named value, e.g. marco, to replace the actual value in Promela statements. "True" and "False" stand for the boolean value "1" or "0". "On" and "Off" are used to describe the session status, and the value is assigned to the boolean variable "session". "Local" and "Server" are used to indicate where the web page is loaded for a transition output. "Local" means that the web page is loaded from the local browser cache when a URL is used to request a web page. "Server" indicates the web page is generated by the web server. We use Err to stand for the error state, and its ID is 0. In our implementation, the history stack and browser cache are declared with array. In order to control the sizes of history stack and cache, we define two marcos, "STACK_SIZE" and "CACHE_SIZE". If we want to test the performance of the EFSM model with different stack size and cache size, we can simply change the value of these two marco settings, and we do not need to modify all statements that are related to these two values.

As introduced in Chapter 6, each Promela program can have one "mtype" statement that defines the constant message type. We define all link types as mtype, including

hyperlink, signInOK, signInFail, signOut, back and forward. A variable "input_lable" is declared as mtype variable.

The variables used in EFSM itself include history_stack, stack pointers, cache, and session. "history_stack" and "cache" are declared as array of integer. State ID or page ID is stored in these two arrays. "stack_pointer", "top", and "bottom" are integer variables, and they are all assigned with non negative values. Obviously, session is a boolean variable, and its value is either "On" or "Off".

In order to trace the current state of EFSM, we use a variable "state" to identify it. "state" is an integer variable, and the value of "state" always indicates the current state ID. We also use a boolean variable "stack_overflow" to indicate the status of the history stack. When the history stack is overflown, the back and forward transitions are disabled.

7.1.2 Implementing EFSM with Promela

An EFSM is defined in a Promela "proctype" definition. "proctype" defines a process type, and the instance of the process is executed concurrently with other process in the same verification model. The "proctype" definition consists of two parts, initialization and EFSM body.

All variables are set their initial values in the initialization part. We assume the EFSM's start state is the state 1. The initialization works include "session", "state", "top", "bottom", and "stack_overflow". Since the history_stack and cache are arrays of integer, the initial value of each element is already set to 0 by Spin automatically.

EFSM body consists of at least two if-selections. The outer "if" selection statement is used to select a designated state, and the inner one is responsible for triggering outgoing transitions of the current state. Figure 7.1 shows the structure of an EFSM implemented in Promela. The EFSM begins with a label "efsm_start:" and each transition ends with a statement "goto efsm_start" which forces the outer selection statement to reselect a state as current state according to the value stored in the variable "state". Because the first state's ID is already set into variable *state*, and the operations for the first state have been done in the initialization part, we can consider the EFSM starts from the first entry state. When the outer if-statement is executed, each condition "state= =state_id" is responsible for selecting a corresponding state as the current state. For example, at the very

beginning, the value of variable *state* is 1, and the current state should be 1. When the outer "if" statement is executed, it searches all conditions and find the value of expression "state= =1" becomes "True". That means "state= =1" is executable and the subsequent statements following "->" could also be executable. Since there is only one transition selection statement under "->", this "if" statement can be executed and an outgoing transition of the state 1 is triggered.

```
efsm_start:
   if
   ::(state==1)->
       if
       ::  (condition1)->
                   state=Transition1's end state ID
                   input_label=Transition1's input_label
                   Transition1's post actions
                   Transition1's output
                   goto efsm_start
       :: (condition2)->
                   state=Transition2's end state ID
                   input_label=Transition2's input_label
                   Transition2's post actions
                   Transition2's output
                   goto efsm_start
       fi
   ::(state==2)->
       if
       ::  (condition1)->
                   state=Transition1's end state ID
                   input_label=Transition1's input_label
                   Transition1's post actions
                   Transition1's output
                   goto efsm_start
       :: (condition2)->
                   state=Transition2's end state ID
                   input_label=Transition2's input_label
                   Transition2's post actions
                   Transition2's output
                   goto efsm_start
       fi
   fi
```

Figure 7.1 EFSM with Promela

When the current state is 1, the two conditions, "condition1" and "condition2" are responsible for selecting a subsequent transition. If "condition1" and "condition2" are not all true at one time, only the statements following the truth expression value will become executable. If "condition1" and "condition2" become true at the same time, the subsequent statements following "condition1" or "condition2" are selected randomly. We

assume only "condition2" is true at a moment, then the statements after "(condition2)->" will be executable. All the statements under a transition selection are responsible for changing the current state and doing the post actions. After the "condition2" becomes true, the state is assigned with the end state ID of Transition2, and input label is also assigned with the corresponding link type. The post actions are the operations on global variables declared in the definition and declaration part of the Promela model, including "history_stack", "cache", "session", etc. The outputs of a transition will be used to verify the correctness of the model. At last, "goto efsm_start" leads the execution restarting from the label "efsm_start:" and the state ID stored in "state" will become new current state of the EFSM.

7.1.3 Conditions

We define three kinds of conditions: cache status condition, session on/off condition and back/forward enable condition. Each condition is expressed as an enumeration value, and all conditions in a transition are stored in this transition object's member variable "condition" as a string. All conditions are extracted from the string and translated into the corresponding Promela condition respectively. At last, these Promela conditions are connected with boolean operator "&&" as one condition.

Since we use an array cache to express the cache status of each page, the InCache condition is translated into "cache[to_state]==True", where "to_state" is the variable that stores the end state ID of a transition. The translation of SessionOn and SessionOff is straightforward. We only use "session==On" or "session==Off" to express then.

For back/forward condition, we need to identify which state is the previous state or next state of current state. Figure 7.2 shows a simple example. A state s has three incoming transitions: t0, t1, and t2. For each incoming transition there is a corresponding back transition. We use b0, b1, and b2 to stand for the corresponding back transitions of t0, t1, and t2 respectively.

When the current state of an EFSM is s, there are three back transitions starting from current state. According to the introduction in Chapter 3, only the page stored before the stack pointer in a browser's history stack can be retrieved when a back button is clicked. Among the three back transitions, only one transition can be triggered at one time. In

order to identify which transition can be triggered, we need to check if the end state of a back transition matches the state that is stored in the previous position of the stack pointer points to. If these two state IDs are the same, it indicates that back transition can be triggered. The comparison work is done by the expression "history_stack[stack_pointer-1]==to_state", where to_state is the state ID of the end state in the back transition. For forward transition, we need to consider if "(history_stack[stack_pointer+1]== to_state) && (stack_pointer!=top)". This is a forward transition which is enabled when the stack pointer does not point to the top of the stack.

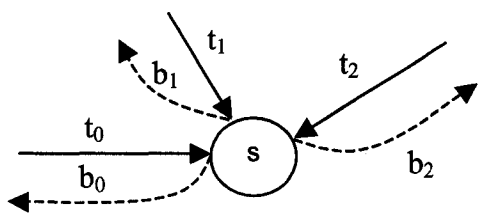


Figure 7.2 A state with multiple incoming transitions

7.1.4 Post Actions

Post actions are operations on EFSM variables. We declare *history_stack*, *cache*, and *session* as EFSM global variables. At the same time, we use *stack_pointer*, and *top* to indicate the status of the history stack. We assume the home page of the application has been loaded when a user accesses a web application, thus the first member of variable *history_stack* must be the state ID which is mapped from the home page of the web application. Since we do not need to model initial transition that leads the current state to an entry state, all post operations are easily implemented.

A push operation adds the state ID of current state into history stack and moves the stack pointer and top to new position that the newly pushed member is stored. Typically a push operation is:

```
if
:: (stack_pointer<STACK_SIZE-1)->
      stack_pointer=stack_pointer+1;
      top=stack_pointer;
      history_stack[stack_pointer]=1;
:: else->
      stack_overflow=True;
fi;
```

We define the stack size of the history stack. When the stack is full, the push operation aborts and generates stack overflow. This stack overflow also disables the use of back and forward. In real life, a stack is impossible to overflow, because the storage is large enough. Due to the capability of the verification tool, we must limit the stack size in a reasonable scope.

The operations of adding cache, set session on, set session off are straightforward and they are:

```
cache[to_state]=True;
session=On;
session=Off;
```

When a back or forward transition is triggered, it means the state ID stored in the previous or next position of stack pointer is available. The only work is to move the stack pointer back and forth.

7.2 Translation Algorithm

According to the introduction above, we summarize the translation algorithm below:

1 Add marco and data type definition.
2 Declare all variables, including *state*, *history_stack*, *cache*, etc.
3 Set the corresponding state of the home page as the EFSM's initial state, and add the state ID into history stack and browser cache if applicable.
4 Set *statck_pointer*, *top* and *bottom*.
5 Add state and transitions
  - 5.1 Add a start label: EFSM_Start and "if" statement. Each state will be selected according to the ID stored in variable state.
  - 5.2 For each state, add an if-statement condition (::(state=='state_id'))
  - 5.3 Add assertion if the assertion variable of current state is not empty
  - 5.4 Add "if" statement to start the processing of post actions for all incoming transitions.
  - 5.5 Put all outgoing transitions of current state into three groups: back, forward, and normal.
    - 5.5.1 For each transition in normal group, write "::" and the following conditions and post actions.
    - 5.5.2 Add "::((!stack_overflow) && (stack_pointer>bottom))->" for the transitions on back group and create a new "if" statement.
    - 5.5.3 For each transition in back group, write "::"and the following conditions and post actions.
    - 5.5.4 Add "::((!stack_overflow) &&(stack_pointer<top))->" for the transitions on forward group and create a new "if" statement.
    - 5.5.5 For each transition in forward group, write "::"and the following conditions and post actions.

5.5.6 Add "fi" to finish the post action operations.
5.6 Add "fi" as the end of the "if" statement.
6 Add "init" statement to start the EFSM process.

7.3 Expressing Correctness Requirements

As mentioned in Chapter 6, there are 3 ways that the correctness requirements are expressed in Promela. For verifying the correctness if a web navigation design has been involved in re-login, and re-submit phenomenon, we will use assertions and LTL formula to express the correctness requirements.

7.3.1 Assertion

In order to investigate the re-login related errors, we need to identify if the session is on when a secured page is accessed. We use an assertion statement to express this kind of correctness:

```
assert(session= =On);
```

This assertion statement should be put into each state mapped from secure page in the navigation design. In order to implement the insertion of the assertion statement, we add a new attribute "assertion" in the state class. If the variable assertion is not empty, the assertion statement will be added into the corresponding state implemented with Promela. The following sample code shows the position where we put the assertion statement.

```
efsm_start:
  if
  ::(state==1)->
      /*assertion statement for state 1*/
      assert(session==On);
      if
      ::  (condition1)->
              state=Transition1's end state ID
              input_label=Transition1's input_label
              Transition1's post actions
              Transition1's output
              goto efsm_start
      :: (condition2)->
              state=Transition2's end state ID
              input_label=Transition2's input_label
              Transition2's post actions
              Transition2's output
              goto efsm_start
      fi
  fi
```

When the state is created according to the corresponding web page in the navigation design, we can add the assertion into the assertion variable if the page is a secure page. We also put this assertion in the place shown above during the EFSM -> Promela translation.

7.3.2 EFSM Output and LTL Expression

The correctness of page navigation sequences can be expressed in LTL. For the expression that describes the resubmit phenomenon, we need to investigate where a page is from when a request is sent. In the EFSM introduced above, a state stands for a web page that can be presented in a browser's windows, but there is no information that describes the source of the web page. In order to provide additional information in an EFSM, we consider putting this kind of information into the output of a transition.

We use a data type "efsm_output" to describe the output of a transition. Each transition's output includes state ID, secure, and source. State ID is a transition's end state ID, which stands for the page to be displayed in a browser's window, when the transition finishes. "secure" identifies if the page that the end state stands for in a transition is a secure page or not. "source" indicates the page returned to a web browser is from browser cache or the application's web server. The definition of "efsm_output" is shown below:

```
typedef efsm_output{
  int state_id;
  bool secure;
  bool source
}

efsm_output output;
```

With this data type definition, we declare a new variable "output". When a transition is finished and the current state is set to the end state of a transition, the members of this "output" are also assigned to the corresponding values. The outputs of each transition are listed in the rules defined in Table 4.5-4.7.

According to the LTL grammar supported by Spin, the logical comparison operations, such as "a>=b", "a= =b", etc., cannot be accepted for the translation from LTL formula to never claims. These kinds of logical expressions should be replaced by marco definitions in order to perform LTL-never claim translation.

For the re-submit phenomenon, let p1 be the page that contains a form, and p2 is one of the subsequent pages after the form in p1 is submitted. P1 and p2 may not be linked directly. When p2 is visited, the actual information displayed in p1 is changed. At this time, when p1 is revisited again, it should be regenerated with the updated information.

In order to create the LTL formula that describe the properties above, we define p1_visited, p2_visited and p1_from_server as marcos.

```
define  p1_visited (output.state_id= =1)
define p2_visited (output.state_id = =2 )
define p1_visited_again (output.stat_id= =1)  && ((output.source= =Cache))
```

The LTL expression is:

[]! ((p1_visited && [] !(p2_visited&& []!(p1_visited_again)))

7.3.3 Error Tracing

The verification work with Spin is divided into four steps: 1) Describe the model with Promela; 2) Express correctness requirements by state labels, assertions or LTL expressions; 3) Generate a verification program by running Spin with the model and correctness requirements as input; 4) Compile and execute the verification program. After the execution of the verification program, the result that indicates if the model is correct according to the correctness requirements will be reported. If the model is incorrect, the execution sequences that lead to the errors are reported. By analyzing the execution sequences, we can conclude where an error is located and what causes the error in the Promela model.

8 Experiments and Evaluation

Model checking is based on the state analysis of a system model and the ability of a model-checking tool is highly related to the maximum states that the tool can handle. Although each model checking tool embeds state reduction algorithms, the maximum processing states of a model checking tool is restricted within a certain state scope, due to the limitation of computer hardware. After generating EFSM based web page navigation models with Promela, we have to face the problem that the model to be verified by Spin can only be of reasonable size. The evaluation works of these EFSM models are mainly focused on investigating the number of actual verified states of each model while they are verified. We evaluate the influence on the number of verified states by experiments with different sizes of history stack, and different numbers of states and transitions.

8.1 Stack Size

During the state analysis of an EFSM model, the state explosion might happen if the stack size is set too big. Another problem is the cycled links. We assume there are two pages in a navigation design, A and B. Page A contains a hyperlink to page B and page B also has a link to page A. These two hyperlinks construct a link cycle. If a web design includes cycled links or it contains entry pages, the history stack of EFSM model will be full very quickly during the executing of the model. No matter how large the stack size we set, the sequence "ABABAB..." is increased continuously and the stack is overflow finally. Each sequence and the position of stack pointer lead to different states. The state explosion is inevitable for the EFSM model with cycled links.

We select two experiment schemes to evaluate the size of history stack on verified states of EFSM models (See Figure 8.1). Most of experiments of Experiment Scheme I are the cases that the number of visited web pages along a hyperlink chain exceeds the stack size. On the contrary, all experiments followed Experiment Scheme II do not contain hyperlink chain that the number of pages in each chain exceeds the stack size. For each experiment, the web navigation design contains one entry page, and this page is also the home page. Other pages are insecure pages. Every page has only one hyperlink that links the page to a subsequent one. All test pages consist of a link chain.

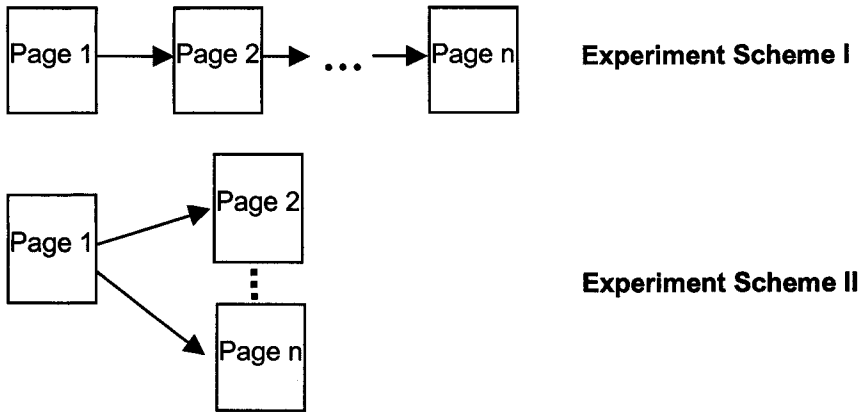


Figure 8.1 History stack size experiment schemes

Table 8.1 and Table 8.2 illustrate the experiment results of verified states on navigation design followed experiment schemes I, II respectively. The maximum number of pages used in these 2 kinds of experiment is 40.

Table 8.1 Stack size and verified states I

Stack Size	Verified States							
	Page=5	Page=10	Page=15	Page=20	Page=25	Page=30	Page=35	Page=40
5	4998	5351	5701	6051	6401	6751	7101	7451
6	20728	25195	25755	26315	26875	27436	27995	28555
7	100674	144569	145426	146357	147299	148209	149119	150012
8	501152	967987	972615	973798	957331	975160	973948	981302
9	2242410	4735700	4553550	4776650	4793140	4717800	4348710	4803820
10	5869130	7600520	7417420	7382000	7287720	7407000	7325710	7593480

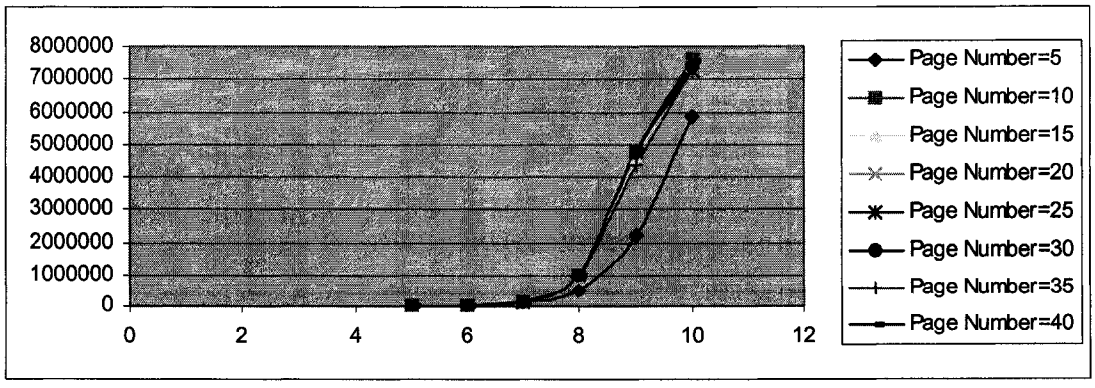


Figure 8.2 Stack size and verified states relation diagram I

Table 8.2 Stack size and verified states II

Stack Size	Verified States							
	Page=5	Page=10	Page=15	Page=20	Page=25	Page=30	Page=35	Page=40
5	10526	80056	260449	602436	1153230	1949850	2995240	4234080
6	44798	691126	3205260	7573170	11600100	14651800	16713200	17891100
7	87486	1387340	5664730	10476900	13714700	15196200	17303280	18142800
8	321321	7221950	14706600	17540100	18221200	18582800	18946900	19279900
9	600926	9874580	14381200	16847900	18236100	18558800	18948700	19119300
10	1967740	14261000	16419300	17934200	18305900	18731800	18878800	19232500

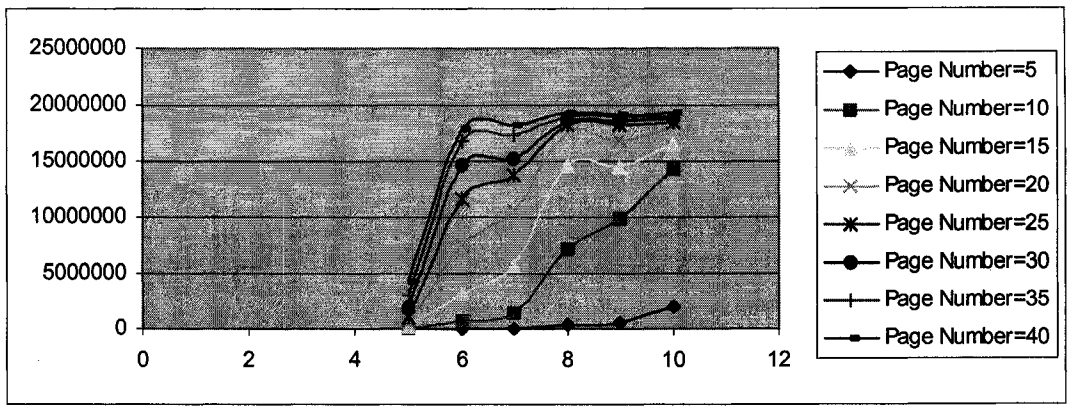


Figure 8.3 Stack size and verified states relation diagram II

Figure 8.2 and 7.3 illustrate the trend of verified states with different stack size. The results of the two kinds of experiments indicate that the EFSM models we generate can be verified very quickly if the stack size is less than 6. If the stack size is greater than 6, the number verified states is increased significantly.

8.2 Number of States

Once the stack size is set to a const value, we can evaluate the relations between verified states and the number of state on the generated EFSM models with browser behavior. Because a web page in a web application must be connect with others by any kind of link in a web application, we can not used an experiment scheme that the number of states can be changed, but the number of transitions is kept in a certain value. In order to evaluate the influence of the number of states on our EFSM model, we choose the experiment scheme II described in section 8.1. According to this experiment scheme, the change of number of states will also cause the change of number of transitions. Since the number of transitions is not increased significantly, the experiment results are acceptable.

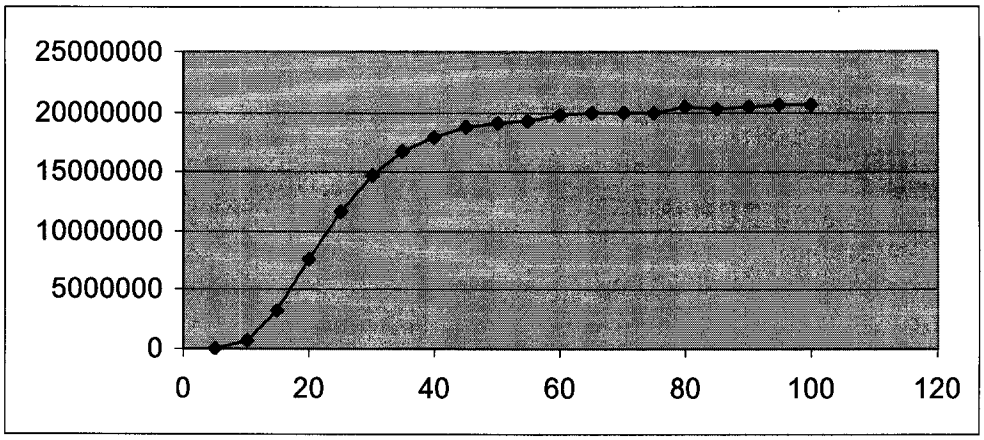


Figure 8.4 Number of states and verified states

Figure 8.4 shows the relationship between the number of states and verified states when the stack size is set to 6. The change on number of states will cause the verified states increased significantly if the states of EFSM models less than 40. When the number of states increases continuously, the increasing of verified states will be a little bit difference.

8.3 Number of Transitions

The evaluation of the relations between verified states and the number of transitions on the generated EFSM models with browser behavior is based on a navigation design with 20 pages. Initially all the web pages in this navigation design are linked by at least one hyperlink. We add more hyperlinks randomly while doing the experiment. The stack size is also set to 6. Figure 8.5 shows the experiment results.

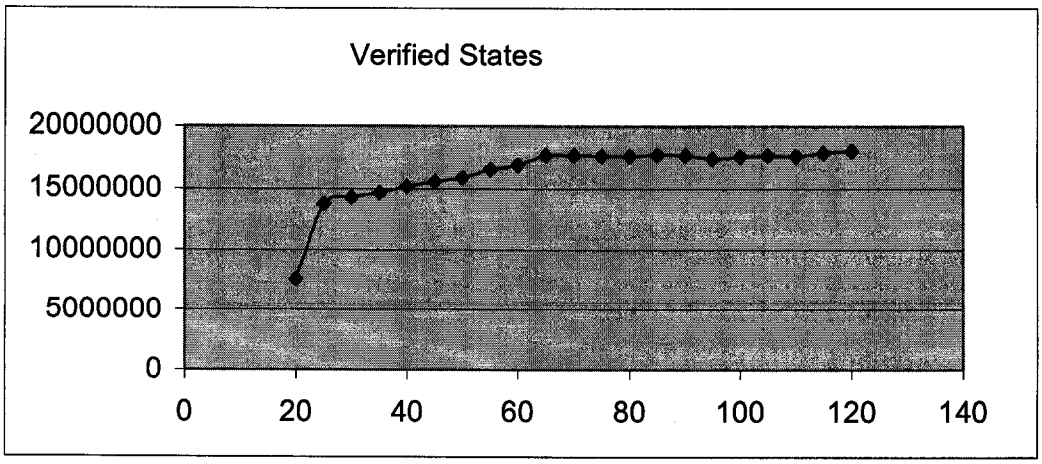


Figure 8.5 Number of transitions and verified states

Figure 8.4 shows no matter how many transitions are added into a web navigation design with certain number of pages, the verified states of its corresponding EFSM model does not change significantly.

9 Conclusion and Future Work

9.1 Conclusion

The research work presented in this thesis includes: 1) applying EFSM to model web page navigations with browser behavior, 2) translating EFSM navigation model into Promela, 3) using Spin to perform model checking on the navigation model described in Promela.

Focusing on the browser's influence on web page navigation design of a web application, we establish an EFSM-based navigation model considering static and dynamic web pages. The browser's behavior on history stack and browser cache is integrated into this EFSM model. Because the EFSM model is generated from the navigation design of a web application, the web designer does not need to know much details of the modeling methodology. When the correctness of the navigation behavior needs to be checked, the designer's work is to provide the navigation design with the format we defined. The following work will be done with the tool we developed.

According to our investigation so far, some researchers proposed techniques to verify the correctness of web applications with model checking. For model checking tool, Spin, no other researchers have been applied it on checking the correctness of web page navigations. The evaluations in Chapter 8 indicate it is possible to perform model checking on the EFSM based navigation model and it is also practical and realistic to used model checking tool to check the navigation correctness of web applications. For the quality assurance on web page navigation, model checking can be considered as an effective solution.

Due to the limitation of model checking techniques, no model-checking tool can handle a model with large amount of states. The pages that can be handled by Spin are limited. According to the evaluation works in Chapter 8, the verification work on the EFSM model we created has good performance if the size of history stack is less than 6. The verified states do not change significantly when the number of states, or the number of transitions change.

9.2 Future Work

The future work can be considered in two aspects: improving the model itself and generating web navigation design from other tools (such as UML) or from source code.

In this thesis, we only considered normal web page without frames. For a page with multi-frames, the communication between one frame and another should be considered and more than one EFSMs are needed. These EFSMs needs to communicate via communication channel.

In this thesis, we defined a simple XML schema that allows the designers to provide the navigation design with this format. Since UML is widely used in application design, our web navigation model could be derived from the UML directly. Another important source of web application design is the source code. If an application's navigation design can be obtained from the source code by reverse engineering, the created verification model can be used to check the correctness of the implementation of a web application.

Bibliography

[1] L. D. Alfaro, "Model Checking the World Wide Web", In *Proc. of the 13th Conference on Computer Aided Verification,* 2001.

[2] L. Baresi, G. Denaro, L. Mainetti, and P. Paolini, "Assertions to better specify the amazon bug", In *14th Software Engineering and Knowledge Engineering*, 2002.

[3] L. Baresi, F. Garzotto, and P. Paolini, "Extending UML for Modeling Web Applications". In *Proc. of the 34th Annual Hawaii International Conference on System Sciences*, 2001.

[4] H. Baumeister, N. Koch, and L. Mandel, "Towards a UML Extension for Hypermedia Design", In *UML-99, Lecture Notes in Computer Science, Vol. 1723*, pages 614 – 629, 1999.

[5] H. V. Beek and S. Mauw, "Automatic conformance testing of internet applications", In *Proc. of the 3rd International Workshop on Formal Approaches to Testing of Software, Lecture Notes in Computer Science, vol. 2931*, pages 205-222, 2004

[6] S. Ceri, P. Fraternali and A. Bongi, "Web Modeling Language (WebML): a modeling language for designing Web sites", In *Computer Networks, Vol. 33, Issue 1-6*, pages137-157, 2000.

[7] J. Chen and S. Chovanec, "Towards Specification-Based Web Testing", In *NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing, Lecture Notes in Computer Science, Vol. 2376*, pages 165-171, 2002.

[8] J. Conallen, "Modeling Web Application Architectures with UML", *Communications of the ACM, Vol. 42, No. 10*, 1999.

[9] J. Gómez and C. Cachero, "OO-H: Extending UML to Model Web Interfaces", In *Information Modeling for Internet Applications, Idea Group Publishing,* 2002.

[10] P. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. "Modeling Web Interactions". In *European Symposium on Programming*, 2003.

[11] S. Greenberg, and A. Cockburn, "Getting Back to Back: Alternate Behaviors for a Web Browser's Back Button", In *Proc. of the 5th Annual Human Factors and the Web Conference*, 1999.

[12] R. Hennicker, N. Koch, "A UML-Based Methodology for Hypermedia Design", In *Proc. of the 3rd International Conference on the Unified Modeling Language (UML 2000), Lecture Notes in Computer Science, Vol. 1939*, Pages 410-424, 2000.

[13] G. Holzman, "The Model Checker Spin", In *IEEE Trans. on Software Engineering, Vol. 23, No. 5*, pages 279-295, 1997.

[14] G. Holzman, "Design and Validation of Computer Protocols", *Prentice Hall, New Jersey*, ISBN 0-13-539925-4, 1991.

[15] N. Koch, H. Baumeister, R. Hennicker, and L. Mandel, "Extending UML for Modeling Navigation and Presentation in Web Applications", In *Modeling Web Applications in the UML Workshop, UML2000*, 2000.

[16] D. C. Kung, C. Liu and P. Hsia, "An Object-Oriented Web Test Model for Testing Web Applications", In *The 1st Asia-Pacific Conference on Quality Software (APAQS 2000)*, pages 111-120, 2000.

[17] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey", In *Proc. of The IEEE, vol. 84, No. 8, pages 1090-1123*, August 1996.

[18] K. R. Leung, L. C. Hui, S. M. Yiu and R. W. Tang, "Modeling Web Navigation by Statechart", In *the 24th Annual International Computer Software and Applications Conference*, 2000.

[19] G. D. Lucca and M. D. Penta, "Considering Browser Interaction in Web Application Testing", In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution(WSE'03)*, 2003.

[20] M. Nottingham, "Caching Tutorial for Web Authors and Webmasters", http://www.web-caching.com/mnot_tutorial/, last access: August 2004.

[21] J. Offutt, Y. Wu, X. Du and H. Huang, "Bypass Tesing of Web Applications", In *the 15th IEEE International Symposium on Software Reliability Engineering*, 2004.

[22] F. Ricca and P. Tonella, "Testing Processes of Web Applications", *Annals of Software Engineering, vol. 14*, pages 93-114, 2002

[23] F. Ricca and P. Tonella, "Building a Tool for the Analysis and Testing of Web Applications: Problems and Solutions", In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), Lecture Notes in Computer Science, vol. 2031*, pages 373-388, 2001.

[24] RFC2616, "Hypertext Transfer Protocol -- HTTP/1.1", 1999.

[25] RFC1738, "Uniform Resource Locators (URL)", 1994.

[26] D. Schwabe, R. de Almeida Pontes, and I. Moura, "OOHDM-Web: An environment for implementation of hypermedia applications in the WWW", In *ACM SIGWEB Newsletter, vol. 8, issue 2*, June, 1999.

[27] E. D. Sciascio, F. M. Donini, M. Mongiello, G. Piscitelli, "Web Applications Design and Maintenance Using Symbolic Model Checking", In *Proc. of IEEE the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, 2003

[28] E. D. Sciascio, F. M. Donini, M. Mongiello, G. Piscitelli, "AnWeb: a System for Automatic Support to Web Application Verification", In *Proc. of the 14th International Conference on Software Engineering and Knowledge Engineering*, 2002.

[29] SPIN, http://spinroot.com/spin/whatispin.html, last access: Autust 2004

[30] Y. Zheng, and M. Pong, "Using statecharts to model hypertext", In *Proc. ACM EHCT, European Conference on Hypertext Technology*, 1992.

Appendix A A Sample Web Design

```
<web_design>
        <page id="1">
                <secure_page>0</secure_page>
                <entry_page>1 </entry_page>
                <enable_cache>1</enable_cache>
                <link link_to="2" type="signInOK" />
                <link link_to="5" type="signInFail" />
        </page>
        <page id="2">
                <secure_page>1</secure_page>
                <entry_page>0</entry_page>
                <enable_cache>1</enable_cache>
                <link link_to="3" type="hyperlink" />
        </page>
        <page id="3">
                <secure_page>1</secure_page>
                <entry_page>0</entry_page>
                <enable_cache>0</enable_cache>
                <link link_to="6" type="hyperlink" />
                <link link_to="4" type="signOut" />
        </page>
        <page id="4">
                <secure_page>0</secure_page>
                <entry_page>0</entry_page>
                <enable_cache>0</enable_cache>
        </page>
        <page id="5">
                <secure_page>0</secure_page>
                <entry_page>0</entry_page>
                <enable_cache>0</enable_cache>
                <link link_to="1" type="hyperlink" />
        </page>
        <page id="6">
                <secure_page>1</secure_page>
                <entry_page>0</entry_page>
                <enable_cache>1</enable_cache>
                <link link_to="4" type="signOut" />
        </page>
</web_design>
```

Appendix B Translated Promela EFSM Model

```
#define On 1
#define Off 0
#define True 1
#define False 0
#define Err 0
#define STACK_SIZE 5
#define CACHE_SIZE 100
#define Local 0
#define Server 1

typedef efsm_output{
  int state_id;
  bool secure;
  bool source
}

int state;
int history_stack[STACK_SIZE];
bool stack_overflow;
int top,  stack_pointer, bottom;
bit cache[CACHE_SIZE];
bit session;
mtype={hyperlink, back, forward, signInOK, signInFail, signOut};
mtype input_label;
efsm_output output;

/*EFSM Proces Definition   */
proctype efsm(){

/*Initialize start state */
session=False;
top=0;
bottom=0;
stack_pointer=0;
stack_overflow=False;
state=1;
history_stack[0]=state;
cache[state]=True;

efsm_start:
    if
    ::(state==0)->
        if
         /*Transitions for state 0  */
        ::((cache[1]==True))->
            state= 1;
            input_label=hyperlink;
            /* Post actions for transition 0 ==> 1*/
            /* input label=hyperlink  */
            if
            :: (stack_pointer<STACK_SIZE-1)->
               stack_pointer=stack_pointer+1;
               top=stack_pointer;
```

```
            history_stack[stack_pointer]=1;
        :: else->
             stack_overflow=True;
        fi;
        /* Outputs for transition 0 ==> 1  */
        output.state_id=1;
        output.secure=False;
        output.source=Local;
        goto efsm_start
    ::((cache[1] ==False))->
        state= 1;
        input_label=hyperlink;
        /* Post actions for transition 0 ==> 1*/
        /* input label=hyperlink  */
        if
        :: (stack_pointer<STACK_SIZE-1)->
           stack_pointer=stack_pointer+1;
           top=stack_pointer;
           history_stack[stack_pointer]=1;
        :: else->
             stack_overflow=True;
        fi;
        cache[1]=True;
        /* Outputs for transition 0 ==> 1  */
        output.state_id=1;
        output.secure=False;
        output.source=Server;
        goto efsm_start

      /*Forward transitions for state 0  */
    ::((!stack_overflow) &&(stack_pointer<top))->
        if
        ::((history_stack[stack_pointer+1] ==1))->
            state= 1;
            input_label=forward;
            stack_pointer=stack_pointer+1;
            /* Outputs for transition 0 ==> 1  */
            output.state_id=1;
            output.secure=False;
            output.source=Local;
            goto efsm_start
        fi;
    fi;
::(state==1)->
    if
      /*Transitions for state 1  */
    ::
        state= 2;
        input_label=signInOK;
        /* Post actions for transition 1 ==> 2*/
        /* input label=signInOK  */
        if
        :: (stack_pointer<STACK_SIZE-1)->
           stack_pointer=stack_pointer+1;
           top=stack_pointer;
           history_stack[stack_pointer]=2;
```

```
    :: else->
         stack_overflow=True;
    fi;
    cache[2]=True;
    session=On;
    /* Outputs for transition 1 ==> 2  */
    output.state_id=2;
    output.secure=True;
    output.source=Server;
    goto efsm_start
::
    state= 5;
    input_label=signInFail;
    /* Post actions for transition 1 ==> 5*/
    /* input label=signInFail  */
    if
    :: (stack_pointer<STACK_SIZE-1)->
       stack_pointer=stack_pointer+1;
       top=stack_pointer;
       history_stack[stack_pointer]=5;
    :: else->
         stack_overflow=True;
    fi;
    cache[5]=True;
    session=Off;
    /* Outputs for transition 1 ==> 5  */
    output.state_id=5;
    output.secure=False;
    output.source=Local;
    goto efsm_start

  /*Back transitions for state 1  */
::((!stack_overflow) && (stack_pointer>bottom))->
    if
    ::((history_stack[stack_pointer-1] ==0))->
        state= 0;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 1 ==> 0  */
        output.state_id=0;
        output.secure=False;
        output.source=Server;
        goto efsm_start
    ::((history_stack[stack_pointer-1] ==2))->
        state= 2;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 1 ==> 2  */
        output.state_id=2;
        output.secure=True;
        output.source=Local;
        goto efsm_start
    ::((session ==On) && (history_stack[stack_pointer-1] ==3))->
        state= 3;
        input_label=back;
        stack_pointer=stack_pointer-1;
```

```
        /* Outputs for transition 1 ==> 3  */
        output.state_id=3;
        output.secure=True;
        output.source=Server;
        goto efsm_start
    ::((session ==Off) && (history_stack[stack_pointer-1] ==3))->
        state=Err;
        input_label=back;
        stack_pointer=0;
        top=0;
        history_stack[stack_pointer]=Err;
        /* Outputs for transition 1 ==> 3  */
        output.state_id=Err;
        output.secure=False;
        output.source=Server;
        goto efsm_start
    ::((history_stack[stack_pointer-1] ==4))->
        state= 4;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 1 ==> 4  */
        output.state_id=4;
        output.secure=False;
        output.source=Server;
        goto efsm_start
    ::((history_stack[stack_pointer-1] ==5))->
        state= 5;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 1 ==> 5  */
        output.state_id=5;
        output.secure=False;
        output.source=Server;
        goto efsm_start
    ::((history_stack[stack_pointer-1] ==6))->
        state= 6;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 1 ==> 6  */
        output.state_id=6;
        output.secure=True;
        output.source=Local;
        goto efsm_start
    fi;

  /*Forward transitions for state 1  */
::((!stack_overflow) &&(stack_pointer<top))->
    if
    ::((history_stack[stack_pointer+1] ==2))->
        state= 2;
        input_label=forward;
        stack_pointer=stack_pointer+1;
        /* Outputs for transition 1 ==> 2  */
        output.state_id=2;
        output.secure=True;
        output.source=Local;
```

```
            goto efsm_start
        ::((history_stack[stack_pointer+1] ==5))->
            state= 5;
            input_label=forward;
            stack_pointer=stack_pointer+1;
            /* Outputs for transition 1 ==> 5  */
            output.state_id=5;
            output.secure=False;
            output.source=Server;
            goto efsm_start
        fi;
    fi;
::(state==2)->
     assert(session==On);
    if
     /*Transitions for state 2  */
    ::((session ==On))->
        state= 3;
        input_label=hyperlink;
        /* Post actions for transition 2 ==> 3*/
        /* input label=hyperlink  */
        if
        :: (stack_pointer<STACK_SIZE-1)->
           stack_pointer=stack_pointer+1;
           top=stack_pointer;
           history_stack[stack_pointer]=3;
        :: else->
              stack_overflow=True;
        fi;
        /* Outputs for transition 2 ==> 3  */
        output.state_id=3;
        output.secure=True;
        output.source=Server;
        goto efsm_start
    ::((session ==Off))->
        state= 0;
        input_label=hyperlink;
        /* Post actions for transition 2 ==> 0*/
        /* input label=hyperlink  */
        stack_pointer=0;
        top=0;
        history_stack[0]=Err;
        /* Outputs for transition 2 ==> 0  */
        output.state_id=Err;
        output.secure=False;
        output.source=Server;
        goto efsm_start
    ::((cache[1]==True))->
        state= 1;
        input_label=hyperlink;
        /* Post actions for transition 2 ==> 1*/
        /* input label=hyperlink  */
        if
        :: (stack_pointer<STACK_SIZE-1)->
           stack_pointer=stack_pointer+1;
           top=stack_pointer;
```

```
        history_stack[stack_pointer]=1;
    :: else->
         stack_overflow=True;
    fi;
    /* Outputs for transition 2 ==> 1  */
    output.state_id=1;
    output.secure=False;
    output.source=Local;
    goto efsm_start
::((cache[1] ==False))->
    state= 1;
    input_label=hyperlink;
    /* Post actions for transition 2 ==> 1*/
    /* input label=hyperlink  */
    if
    :: (stack_pointer<STACK_SIZE-1)->
       stack_pointer=stack_pointer+1;
       top=stack_pointer;
       history_stack[stack_pointer]=1;
    :: else->
         stack_overflow=True;
    fi;
    cache[1]=True;
    /* Outputs for transition 2 ==> 1  */
    output.state_id=1;
    output.secure=False;
    output.source=Server;
    goto efsm_start

  /*Back transitions for state 2  */
::((!stack_overflow) && (stack_pointer>bottom))->
    if
    ::((history_stack[stack_pointer-1] ==1))->
        state= 1;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 2 ==> 1  */
        output.state_id=1;
        output.secure=False;
        output.source=Local;
        goto efsm_start
    fi;

  /*Forward transitions for state 2  */
::((!stack_overflow) &&(stack_pointer<top))->
    if
    ::((session ==On) && (history_stack[stack_pointer+1] ==3))->
        state= 3;
        input_label=forward;
        stack_pointer=stack_pointer+1;
        /* Outputs for transition 2 ==> 3  */
        output.state_id=3;
        output.secure=True;
        output.source=Server;
        goto efsm_start
    ::((session ==Off) && (history_stack[stack_pointer+1] ==3))->
```

```
                state=Err;
                input_label=forward;
                stack_pointer=0;
                top=0;
                history_stack[stack_pointer]=Err;
                /* Outputs for transition 2 ==> 3  */
                output.state_id=Err;
                output.secure=False;
                output.source=Server;
                goto efsm_start
            ::((history_stack[stack_pointer+1] ==1))->
                state= 1;
                input_label=forward;
                stack_pointer=stack_pointer+1;
                /* Outputs for transition 2 ==> 1  */
                output.state_id=1;
                output.secure=False;
                output.source=Local;
                goto efsm_start
            fi;
        fi;
    ::(state==3)->
         assert(session==On);
        if
         /*Transitions for state 3  */
        ::((cache[6] ==False)&& (session==On))->
            state= 6;
            input_label=hyperlink;
            /* Post actions for transition 3 ==> 6*/
            /* input label=hyperlink  */
            if
            :: (stack_pointer<STACK_SIZE-1)->
               stack_pointer=stack_pointer+1;
               top=stack_pointer;
               history_stack[stack_pointer]=6;
            :: else->
                  stack_overflow=True;
            fi;
            cache[6]=True;
            /* Outputs for transition 3 ==> 6  */
            output.state_id=6;
            output.secure=True;
            output.source=Server;
            goto efsm_start
        ::((cache[6]==True)&& (session==On))->
            state= 6;
            input_label=hyperlink;
            /* Post actions for transition 3 ==> 6*/
            /* input label=hyperlink  */
            if
            :: (stack_pointer<STACK_SIZE-1)->
               stack_pointer=stack_pointer+1;
               top=stack_pointer;
               history_stack[stack_pointer]=6;
            :: else->
                  stack_overflow=True;
```

```
    fi;
    /* Outputs for transition 3 ==> 6  */
    output.state_id=6;
    output.secure=True;
    output.source=Local;
    goto efsm_start
::((cache[0] ==False)&& (session==Off))->
    state= 0;
    input_label=hyperlink;
    /* Post actions for transition 3 ==> 0*/
    /* input label=hyperlink  */
    stack_pointer=0;
    top=0;
    history_stack[0]=Err;
    /* Outputs for transition 3 ==> 0  */
    output.state_id=Err;
    output.secure=True;
    output.source=Server;
    goto efsm_start
::((cache[6]==True)&& (session==Off))->
    state= 6;
    input_label=hyperlink;
    /* Post actions for transition 3 ==> 6*/
    /* input label=hyperlink  */
    if
    :: (stack_pointer<STACK_SIZE-1)->
       stack_pointer=stack_pointer+1;
       top=stack_pointer;
       history_stack[stack_pointer]=6;
    :: else->
          stack_overflow=True;
    fi;
    /* Outputs for transition 3 ==> 6  */
    output.state_id=6;
    output.secure=True;
    output.source=Server;
    goto efsm_start
::
    state= 4;
    input_label=signOut;
    /* Post actions for transition 3 ==> 4*/
    /* input label=signOut  */
    if
    :: (stack_pointer<STACK_SIZE-1)->
       stack_pointer=stack_pointer+1;
       top=stack_pointer;
       history_stack[stack_pointer]=4;
    :: else->
          stack_overflow=True;
    fi;
    session=Off;
    /* Outputs for transition 3 ==> 4  */
    output.state_id=4;
    output.secure=False;
    output.source=Local;
    goto efsm_start
```

```
::((cache[1]==True))->
    state= 1;
    input_label=hyperlink;
    /* Post actions for transition 3 ==> 1*/
    /* input label=hyperlink  */
    if
    :: (stack_pointer<STACK_SIZE-1)->
       stack_pointer=stack_pointer+1;
       top=stack_pointer;
       history_stack[stack_pointer]=1;
    :: else->
          stack_overflow=True;
    fi;
    /* Outputs for transition 3 ==> 1  */
    output.state_id=1;
    output.secure=False;
    output.source=Local;
    goto efsm_start
::((cache[1] ==False))->
    state= 1;
    input_label=hyperlink;
    /* Post actions for transition 3 ==> 1*/
    /* input label=hyperlink  */
    if
    :: (stack_pointer<STACK_SIZE-1)->
       stack_pointer=stack_pointer+1;
       top=stack_pointer;
       history_stack[stack_pointer]=1;
    :: else->
          stack_overflow=True;
    fi;
    cache[1]=True;
    /* Outputs for transition 3 ==> 1  */
    output.state_id=1;
    output.secure=False;
    output.source=Server;
    goto efsm_start

  /*Back transitions for state 3  */
::((!stack_overflow) && (stack_pointer>bottom))->
    if
    ::((history_stack[stack_pointer-1] ==2))->
        state= 2;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 3 ==> 2  */
        output.state_id=2;
        output.secure=True;
        output.source=Local;
        goto efsm_start
    fi;

  /*Forward transitions for state 3  */
::((!stack_overflow) &&(stack_pointer<top))->
    if
    ::((history_stack[stack_pointer+1] ==6))->
```

```
                state= 6;
                input_label=forward;
                stack_pointer=stack_pointer+1;
                /* Outputs for transition 3 ==> 6  */
                output.state_id=6;
                output.secure=True;
                output.source=Local;
                goto efsm_start
            ::((history_stack[stack_pointer+1] ==4))->
                state= 4;
                input_label=forward;
                stack_pointer=stack_pointer+1;
                /* Outputs for transition 3 ==> 4  */
                output.state_id=4;
                output.secure=False;
                output.source=Server;
                goto efsm_start
            ::((history_stack[stack_pointer+1] ==1))->
                state= 1;
                input_label=forward;
                stack_pointer=stack_pointer+1;
                /* Outputs for transition 3 ==> 1  */
                output.state_id=1;
                output.secure=False;
                output.source=Local;
                goto efsm_start
            fi;
        fi;
    ::(state==4)->
        if
         /*Transitions for state 4  */
        ::((cache[1]==True))->
            state= 1;
            input_label=hyperlink;
            /* Post actions for transition 4 ==> 1*/
            /* input label=hyperlink  */
            if
            :: (stack_pointer<STACK_SIZE-1)->
               stack_pointer=stack_pointer+1;
               top=stack_pointer;
               history_stack[stack_pointer]=1;
            :: else->
                  stack_overflow=True;
            fi;
            /* Outputs for transition 4 ==> 1  */
            output.state_id=1;
            output.secure=False;
            output.source=Local;
            goto efsm_start
        ::((cache[1] ==False))->
            state= 1;
            input_label=hyperlink;
            /* Post actions for transition 4 ==> 1*/
            /* input label=hyperlink  */
            if
            :: (stack_pointer<STACK_SIZE-1)->
```

```
        stack_pointer=stack_pointer+1;
        top=stack_pointer;
        history_stack[stack_pointer]=1;
    :: else->
         stack_overflow=True;
    fi;
    cache[1]=True;
    /* Outputs for transition 4 ==> 1  */
    output.state_id=1;
    output.secure=False;
    output.source=Server;
    goto efsm_start

  /*Back transitions for state 4  */
::((!stack_overflow) && (stack_pointer>bottom))->
    if
    ::((session ==On) && (history_stack[stack_pointer-1] ==3))->
        state= 3;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 4 ==> 3  */
        output.state_id=3;
        output.secure=True;
        output.source=Server;
        goto efsm_start
    ::((session ==Off) && (history_stack[stack_pointer-1] ==3))->
        state=Err;
        input_label=back;
        stack_pointer=0;
        top=0;
        history_stack[stack_pointer]=Err;
        /* Outputs for transition 4 ==> 3  */
        output.state_id=Err;
        output.secure=False;
        output.source=Server;
        goto efsm_start
    ::((history_stack[stack_pointer-1] ==6))->
        state= 6;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 4 ==> 6  */
        output.state_id=6;
        output.secure=True;
        output.source=Local;
        goto efsm_start
    fi;

  /*Forward transitions for state 4  */
::((!stack_overflow) &&(stack_pointer<top))->
    if
    ::((history_stack[stack_pointer+1] ==1))->
        state= 1;
        input_label=forward;
        stack_pointer=stack_pointer+1;
        /* Outputs for transition 4 ==> 1  */
        output.state_id=1;
```

```
                output.secure=False;
                output.source=Local;
                goto efsm_start
            fi;
        fi;
    ::(state==5)->
        if
         /*Transitions for state 5  */
        ::((cache[1]==True))->
            state= 1;
            input_label=hyperlink;
            /* Post actions for transition 5 ==> 1*/
            /* input label=hyperlink  */
            if
            :: (stack_pointer<STACK_SIZE-1)->
               stack_pointer=stack_pointer+1;
               top=stack_pointer;
               history_stack[stack_pointer]=1;
            :: else->
                 stack_overflow=True;
            fi;
            /* Outputs for transition 5 ==> 1  */
            output.state_id=1;
            output.secure=False;
            output.source=Local;
            goto efsm_start
        ::((cache[1] ==False))->
            state= 1;
            input_label=hyperlink;
            /* Post actions for transition 5 ==> 1*/
            /* input label=hyperlink  */
            if
            :: (stack_pointer<STACK_SIZE-1)->
               stack_pointer=stack_pointer+1;
               top=stack_pointer;
               history_stack[stack_pointer]=1;
            :: else->
                 stack_overflow=True;
            fi;
            cache[1]=True;
            /* Outputs for transition 5 ==> 1  */
            output.state_id=1;
            output.secure=False;
            output.source=Server;
            goto efsm_start

         /*Back transitions for state 5  */
        ::((!stack_overflow) && (stack_pointer>bottom))->
            if
            ::((history_stack[stack_pointer-1] ==1))->
                state= 1;
                input_label=back;
                stack_pointer=stack_pointer-1;
                /* Outputs for transition 5 ==> 1  */
                output.state_id=1;
                output.secure=False;
```

```
            output.source=Local;
            goto efsm_start
        fi;

      /*Forward transitions for state 5  */
    ::((!stack_overflow) &&(stack_pointer<top))->
        if
        ::((history_stack[stack_pointer+1] ==1))->
            state= 1;
            input_label=forward;
            stack_pointer=stack_pointer+1;
            /* Outputs for transition 5 ==> 1  */
            output.state_id=1;
            output.secure=False;
            output.source=Local;
            goto efsm_start
        fi;
    fi;
::(state==6)->
     assert(session==On);
    if
     /*Transitions for state 6  */
    ::
        state= 4;
        input_label=signOut;
        /* Post actions for transition 6 ==> 4*/
        /* input label=signOut  */
        if
        :: (stack_pointer<STACK_SIZE-1)->
           stack_pointer=stack_pointer+1;
           top=stack_pointer;
           history_stack[stack_pointer]=4;
        :: else->
             stack_overflow=True;
        fi;
        session=Off;
        /* Outputs for transition 6 ==> 4  */
        output.state_id=4;
        output.secure=False;
        output.source=Local;
        goto efsm_start
    ::((cache[1]==True))->
        state= 1;
        input_label=hyperlink;
        /* Post actions for transition 6 ==> 1*/
        /* input label=hyperlink  */
        if
        :: (stack_pointer<STACK_SIZE-1)->
           stack_pointer=stack_pointer+1;
           top=stack_pointer;
           history_stack[stack_pointer]=1;
        :: else->
             stack_overflow=True;
        fi;
        /* Outputs for transition 6 ==> 1  */
        output.state_id=1;
```

```
    output.secure=False;
    output.source=Local;
    goto efsm_start
::((cache[1] ==False))->
    state= 1;
    input_label=hyperlink;
    /* Post actions for transition 6 ==> 1*/
    /* input label=hyperlink  */
    if
    :: (stack_pointer<STACK_SIZE-1)->
       stack_pointer=stack_pointer+1;
       top=stack_pointer;
       history_stack[stack_pointer]=1;
    :: else->
         stack_overflow=True;
    fi;
    cache[1]=True;
    /* Outputs for transition 6 ==> 1  */
    output.state_id=1;
    output.secure=False;
    output.source=Server;
    goto efsm_start

  /*Back transitions for state 6  */
::((!stack_overflow) && (stack_pointer>bottom))->
    if
    ::((session ==On) && (history_stack[stack_pointer-1] ==3))->
        state= 3;
        input_label=back;
        stack_pointer=stack_pointer-1;
        /* Outputs for transition 6 ==> 3  */
        output.state_id=3;
        output.secure=True;
        output.source=Server;
        goto efsm_start
    ::((session ==Off) && (history_stack[stack_pointer-1] ==3))->
        state=Err;
        input_label=back;
        stack_pointer=0;
        top=0;
        history_stack[stack_pointer]=Err;
        /* Outputs for transition 6 ==> 3  */
        output.state_id=Err;
        output.secure=False;
        output.source=Server;
        goto efsm_start
    fi;

  /*Forward transitions for state 6  */
::((!stack_overflow) &&(stack_pointer<top))->
    if
    ::((history_stack[stack_pointer+1] ==4))->
        state= 4;
        input_label=forward;
        stack_pointer=stack_pointer+1;
        /* Outputs for transition 6 ==> 4  */
```

```
                output.state_id=4;
                output.secure=False;
                output.source=Server;
                goto efsm_start
            ::((history_stack[stack_pointer+1] ==1))->
                state= 1;
                input_label=forward;
                stack_pointer=stack_pointer+1;
                /* Outputs for transition 6 ==> 1  */
                output.state_id=1;
                output.secure=False;
                output.source=Local;
                goto efsm_start
            fi;
        fi;
    fi;
}

init{run efsm()}
```

Vita Auctoris

Name: Xiaoshan Zhao

Place of Birth: Beijing, China

Year of Birth: 1968

Education:

University of Windsor, Windsor, Ontario, Canada
2001-2004 M.Sc. in Computer Science

Zhejiang University, Hangzhou, China
1986-1990 B.Eng. in Computer Science and Engineering

Working Experience:

Programmer/Analyst
Walter Products Inc., Windsor, Ontario, Canada
2001-2004

Network Administrator
Beijing Petrol Chemical Engineering Co., Beijing, China
1990-2000