Formal Verification of Web Applications Modeled by Communicating Automata

May Haydar[1,2], Alexandre Petrenko[1], and Houari Sahraoui[2]

[1] CRIM, Centre de recherche informatique de Montréal
550 Sherbrooke West, Suite 100, Montreal, Quebec, H3A 1B9, Canada
{mhaydar,petrenko}@crim.ca
[2] Département d'informatique et de recherche opérationnelle, Université de Montréal
CP 6128 succ. Centre-Ville, Montreal, Quebec, H3C 3J7, Canada
sahraouh@iro.umontreal.ca

Abstract. In this paper, we present an approach for modeling an existing web application using communicating finite automata model based on the user-defined properties to be validated. We elaborate a method for automatic generation of such a model from a recorded browsing session. The obtained model could then be used to verify properties with a model checker, as well as for regression testing and documentation. Unlike previous attempts, our approach is oriented towards complex multi-window/frame applications. We present an implementation of the approach that uses the model checker Spin and provide an example.

1 Introduction

The Internet has reshaped the way people deal with information. In particular, web applications have affected the daily life in many ways, where they are used in information management/gathering, e-commerce, software development, learning, education, entertainment, etc. With such pervasive and radical growth of web applications, correctness is a primary concern, especially that Web Applications (WA) interact with many components such as scripts (CGI, ASP, JSP, PHP, etc.), browsers, proxy servers, backend databases, etc. Unlike traditional software, WA have an extremely short development and evolution life cycle and often have a large number of untrained users that could experiment with the WA unpredictably. Therefore, thorough analysis and verification of WA is indispensable to assure the release of high quality applications. In recent years, software community started to acquire formal methods as a practical and reliable solution to analyze various applications. In this paper, we present a formal approach for modeling web applications using a communicating automata model. We observe the external behavior of an explored part of a web application using a monitoring tool. The observed behavior is then converted into communicating automata representing all windows, frames, and framesets of the application under test. The obtained model could then be used to verify user-defined properties of the application with a model checker. Our implementation of the approach uses the model checker Spin. In

D. de Frutos-Escrig and M. Núñez (Eds.): FORTE 2004, LNCS 3235, pp. 115-132, 2004.

Section 2, we present an overview of the main notions of the web. In Section 3, we discuss the related work on formal modeling and analysis of web applications. Section 4 introduces our approach. In Section 5, we suggest a method to model a browsing session of a single window application by a single automaton. Section 6 describes a method to partition a single browsing session into local sessions and to convert the local sessions into communicating automata. In Section 7, we present the implementation of the approach using Spin and provide a case study in Section 8. We conclude in Section 9.

2 Preliminaries

We present the main terminology encountered in studying web applications. Further information can be found in [1,2,3]. A web application is defined in [3] as "a software application that is accessible using a web browser or HTTP user agent. It typically consists of a thin-client tier (the web browser), a presentation tier (web servers), an application tier (application servers) and a database tier". We see a *web application* as an application providing interactive services by rendering web resources in the form of web pages (containing text and images, forms and etc.). A page can be static, residing on the server, or dynamic, resulting from the execution of a script at the server or the client side. A page is rendered by a browser to the user in windows. A *form* is a section of a web page that includes textual content, controls (buttons, checkboxes, etc.), optional labels, an action and a method. A frame element is an HTML tag that defines a frame. It includes a source *src* attribute specifying the URI of the source (initial) page loaded in the frame and an optional *name* attribute that assigns a name to the frame. A *frameset* element is an HTML tag that groups frame elements and possibly other frameset elements. The HTML document that describes the layout of frames is called the Frameset document having a frameset element that can be nested at any level. A Frameset document can be viewed as a *frame tree* whose leaves are frame elements and internal nodes are frameset elements. Detailed information on forms, frames, and HTTP protocol can be found in [1,3,20,21]. Note that we distinguish between two classes of WA: applications whose behavior is independent of its history and does not rely on the client's or the server's state. The second class represents WA whose behavior is determined by its history and thus affected by previous information kept at the client/server side (such as cookies). In this work, we consider the first class of WA where the same HTTP request always has the same response independently of past information in previous request/response pairs.

3 Related Work

Formal modeling of web applications is a relatively new research direction. Previous work on the topic includes modeling approaches that target the verification of such applications [25,26,27,28], testing [29,30,32,32], design and implementation [10,11,12,13,14]. In [25,26] an approach is presented where a web site is modeled as a directed graph. A node in the graph represents a web page and the edges represent

links clicked. If the page contains frames the graph node is then a tree whose tree nodes are pages loaded in frames and tree edges are labeled by frame names. This model is used to verify properties of web sites with frames. However, only static pages are considered in this work, concurrent behavior of multiple windows is not modeled, and all the links whose targets could create new independent windows are treated as broken links. Besides, any frameset that could be present in the application is completely ignored. Also, in the model, a page loaded in an unnamed window (as a result of the predefined target "_blank" associated with a link) is represented as a graph node that replaces the existing node as if the page is loaded on top of page where the link was clicked; this incorrectness is due to the inadequacy of the proposed model to represent concurrent behavior of multiple windows. In [27,28] the authors present a model based on Petri nets to model check static hyperdocuments [27] and framed pages [28]. While Petri nets can express parallel and concurrent behavior, the authors build the overall state space as input of the model checker, which is a tedious and erroneous approach especially with large applications with several frames and windows. [25,26,27,28] do not tackle the modeling and verification of form-based pages that are dynamically generated by a server program, neither concurrent behavior of applications with multiple windows. The work in [29,30] focuses on inferring a UML model of web applications. This model, merely a class diagram, is mainly used for the static analysis of web applications: HTML code inspection and scanning, data flow analysis, and semi automatic test case generation. In [31], the above mentioned modeling technique is extended such that a web application is executed to extract models for dynamic web pages using server's access logs. These logs present limited information on the requests since only the request headers are logged. In case dynamic pages are generated based on POST method requests, the form data submitted is usually stored in the message body of the request; thus, making those pages requests undistinguishable and introduce unnecessary non-determinism into the resulting model. Besides, the approach is inadequate for modeling concurrent behavior of frames and multiple windows. In [32], a modeling technique for web applications is presented based on regular expressions. The focus is on modeling the behavior of web applications, consisting of merely dynamically generated pages, for the purpose of functional testing. Other approaches for modeling web applications are oriented towards the design rather than analysis of WA. These include object oriented based models [10] and statechart based models [11,12,13,14], that are tailored to forward engineering, logical and hierarchical representation of web applications. Such models are not available for analyzing existing WA developed without formal models. Each of the existing related work concentrates on some aspects of web applications leaving out other aspects that remain untouched, or unfeasible to model using the corresponding proposed approach. These attempts indicate that formal modeling of WA is still an open complex problem especially when it comes down to modeling multiple frames and windows, and properties which have to reflect various concerns of different stakeholders of WA.

In this paper, we attempt to develop a modeling approach that could produce a finite automaton model tuned to features of WA that have to be validated, while delegating the task of property verification to an existing model checker. We elaborate a black-box (dynamic) approach by executing the web application under

test (WAUT) and analyzing only its external behavior without any access to server programs or databases. The observations are provided by a monitoring tool, a proxy server [5] or an off-the-shelf network monitoring tool [18], where HTTP requests and responses are logged. Our model is a system of communicating automata representing all windows, frames, and framesets of the application under test. The existence of frames, framesets, and windows reflects concurrent behavior of the WAUT, where these objects affect each other behaviors via links and forms with specified targets. Therefore, a suitable and natural modeling technique is communicating automata, where the burden of building a global state graph of the model is left to a model checker. As opposed to the existing approaches, we model not only static pages, but also dynamic pages with form filling (with GET and POST methods), frames and frameset behavior, multiple windows, and their concurrent behavior. Generally speaking, one could build a special web-oriented model checker, as in [26], to verify specific properties. This task requires the building of all the necessary algorithms from scratch. We opt to the use of an existing model checker, Spin, used in several industrial applications [22], such that we only have to describe our model in the model checker's input language.

4 Observing Web Application

To define a formal model of a web application in case when the code of the application is available, one may apply abstraction techniques developed in software reverse engineering following a *static* (white-box based) approach [7,8,9]. To build a formal model according to a *dynamic* (black-box based) approach, one executes a given application and uses only the observations of an external behavior of the application [15,16,17,31]. In case of web applications that rely on the HTTP protocol considered in this work, an “external” behavior consists of requests and responses. In our framework, we follow the dynamic approach and assume that the request/response traffic between a client side and a server in the WA under test is observable. One possible way of achieving this is to use a proxy server [5]. A proxy server monitors the traffic between the client and the server and records it in proxy logs. The proxy logs contain the requests for the pages and the responses to these requests.

With this approach, a behavior of a WAUT, we call it a *browsing session*, is interpreted as a possible sequence of web pages that have the same domain name intermittent with the corresponding requests. Note that a behavior of a WA is independent of the navigation aids provided by the browser (back button, forward button, etc.). In other words, we build a model that is independent of a browser. We assume that a next request is not submitted before the browser delivers a response to a previous request. If the user clicks a link, and that leads to a page with k frames then k+1 request/response pairs are observed. The first request/response pair corresponds to the link clicked and thus to the frameset document; and k requests, initiated by the browser, along with their responses, correspond to the URIs defined in the frameset document. Exhaustive exploration could hardly be achieved for non-trivial web applications with a database tier. This is why we have to build a model just for a part of the WAUT, which is explored in a browsing session. To generate sequences of

requests, instead of the user, one may consider a crawler that automatically explores links in the WAUT [6], though in case of pages with forms to fill, the user actions would still be required. In the next section, we explain our approach for building a finite automaton that models a browsing session.

5 Modeling Single Window Web Applications

We first present our modeling approach for web applications whose web pages do not have frames and assume that the WAUT is browsed in a single browser window, in other words, that all the links have undefined target attributes. Later we provide extensions to more complex applications.

The purpose of building a formal model for a WAUT is to validate whether the application exhibits certain predefined properties. We assume that the properties to be specified in a temporal logic of a chosen model checker are composed of atomic propositions, and for each visited page the value of each proposition is uniquely determined by the content of the page, be it dynamic or static. These propositions refer to the page attributes that have to be checked (and reflected in a model). These attributes can be of various types, for instance: a numerical type to count the occurrence of a certain entity, a string type to denote the domain name of a page, or features of a page link, such as a hypertext associated with the link. However, there are cases when an attribute representing a certain feature of the visited page cannot be defined for another page. For instance, a Boolean attribute that indicates whether the menu is framed in a page that does not contain menus, or an attribute representing the percentage of the number of occurrences of a certain string with respect to the number of all the strings in a page that contains no text. In such cases, we assign to these attributes the value "not available". The atomic propositions that refer to such attributes are then false in the corresponding pages. In the following, we describe how to determine automata that model an observed behavior of a WAUT based on the information available in the corresponding browsing session. The session includes requests initiated by the user, namely links clicked and filled form submissions, as well as requests initiated by the browser, namely requests for URIs present in an HTTP-EQUIV tag [3,4]; for simplicity, we call those URIs *implicit* links.

5.1 Definitions

Each request is represented by a string l. In case the request method is Get or Head, l is the URI sent in the request. If the request is for a filled form then we represent it in the form $a?d$, where a is the form action and d is the form data set that corresponds to the data fields filled in the form; in case of the Get method, data set is a part of the URI sent in the request, while in case of Post method, data set is included in the message body as a data stream.

Each response corresponding to a visited page is abstracted by a tuple $<u, c, I, L, V>$, where u denotes the request l identifying the page; $c \in C$ represents the status code of the page, C is the set of valid status codes defined as integers ranging between 100 and 599 [20]; I is the set of URIs specified by the action attribute of

each form in the page; L is the set of URIs associated with links, including the implicit links if any, in the page (L does not include links that cause the scrolling to sections in the same page); and V is a vector $<v_1, \ldots, v_k>$, where v_i is the valuation of the page attribute i and k is the number of all the page attributes over which the atomic propositions are defined. Pages with status code 3xx have their URL u different from the request l that triggered the response due to a redirection to another location of the pages. Pages with status code 4xx or 5xx may or may not have links leading back to the application.

A browsing session is a Request/Response Sequence RRS = $<u_0, c_0, I_0, L_0, V_0> l_1 <u_1, c_1, I_1, L_1, V_1> \ldots l_n <u_n, c_n, I_n, L_n, V_n>$, where $<u_0, c_0, I_0, L_0, V_0>$ is the default page displayed in the browser window from which the first request was triggered; this page is not observed in the browsing session, therefore, u_0 and c_0 are null, and I_0, L_0, and V_0 are empty sets; l_i is a request that is followed by the response page $<u_i, c_i, I_i, L_i, V_i>$; for all $i > 1$, $l_i \in L_{i-1}$ if l_i is a request corresponding to a clicked or implicit link, or if l_i is of the form $a_i?d_i$, then $a_i \in I_{i-1}$; and for all $i > 0$ $l_i = u_i$ if $c_i \neq$ 3xx; (otherwise, $l_i \neq u_i$); and n is the total number of requests in the browsing session, starting from the first request l_1 for the initial (home) page of the application. Page attributes or atomic propositions, along with u and c, are considered as state attributes and used for model checking in a way similar to Kripke structure [19].

We say that a link of the application under test is *explored* in a browsing session if it's URI is one of the requests in the browsing session; otherwise, we say that the link is *unexplored*. Similarly, we say that a form is explored if its action a appears in one of the requests $a?d$ in the browsing session; otherwise we say the form is unexplored.

Two pages $<u_i, c_i, I_i, L_i, V_i>$ and $<u_j, c_j, I_j, L_j, V_j>$ have a repeated (common) link if $L_i \cap L_j \neq \emptyset$; similarly, a repeated form exists if $I_i \cap I_j \neq \emptyset$.

5.2 Converting a Browsing Session into an Automaton

In this section, we provide a high-level description of our algorithm to convert *RRS* into an automaton, called a *session automaton*.

Algorithm 1. Given a browsing session $RRS = <u_0, c_0, I_0, L_0, V_0> l_1 <u_1, c_1, I_1, L_1, V_1> \ldots l_n <u_n, c_n, I_n, L_n, V_n>$, where n is the total number of observed responses:

1. The tuple $<u_0, c_0, I_0, L_0, V_0>$ is mapped into a designated state called *inactive*, denoted s_0, where u_0 and c_0 are null, and I_0, L_0, and V_0 are empty sets.
2. For all $i > 0$, a tuple $<u_i, c_i, I_i, L_i, V_i>$ corresponds to a state of the session automaton. Tuples $<u_i, c_i, I_i, L_i, V_i>$ and $<u_j, c_j, I_j, L_j, V_j>$, where $j > i$, are mapped into the same state if $c_i = c_j$, $I_i = I_j$, $L_i = L_j$, and $V_i = V_j$. Let S denote the set of thus defined states.
3. The set of events of the automaton is defined by the union of the sets Γ, Δ, *Req*. $\Gamma = \{l \mid l \in L_i, 1 \leq i \leq n\}$ is the set of all the URIs associated with links in the observed responses, $\Delta \subseteq \{a \mid a \in I_i, 1 \leq i \leq n\}$ is the set of all form actions that correspond to the unexplored forms in the observed responses, *Req* is the set of all the observed requests. Thus, $\Gamma \cup \Delta \cup Req$ is the alphabet of the automaton, denoted Σ.

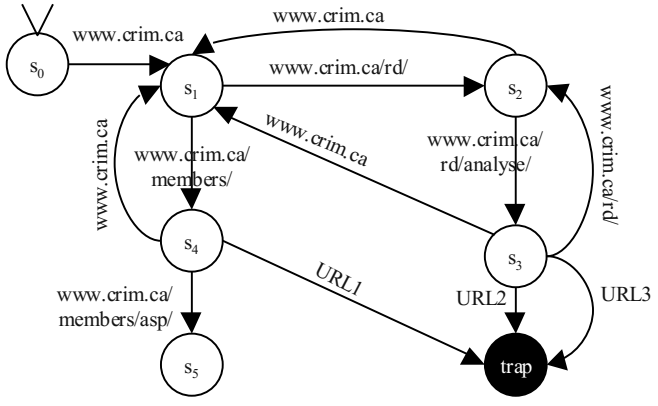

Fig.1. Example of a Session Automaton

4. Each triple (<u_i, c_i, I_i, L_i, V_i>, l_{i+1}, <u_{i+1}, c_{i+1}, I_{i+1}, L_{i+1}, V_{i+1}>) defines a transition (s_i, l_i, s_{i+1}), where s_i, s_{i+1} correspond to the pages <u_i, c_i, I_i, L_i, V_i>, <u_{i+1}, c_{i+1}, I_{i+1}, L_{i+1}, V_{i+1}> respectively, and $l_{i+1} \in L_i$ if l_{i+1} is a request corresponding to a clicked or implicit link, or if l_{i+1} is of the form $a_{i+1}?d_{i+1}$, then $a_{i+1} \in I_i$; and $l_{i+1} = u_{i+1}$ if $c_{i+1} \neq 3xx$; (otherwise, $l_{i+1} \neq u_{i+1}$);
5. Each request corresponding to an explored repeated form or link defines a transition from the state where it occurs to the state that corresponds to the response of the submitted filled form or clicked link.
6. Each request corresponding to an unexplored link $l \in L_i$ or unexplored form $a \in I_i$ defines a transition from the state representing the page <u_i, c_i, I_i, L_i, V_i> to a designated state, called a *trap* state that represents the unexplored part of the WAUT and whose attributes are undefined. Let T denote the set of thus defined transitions.
7. The session automaton is $A_{RRS} = < S \cup \{trap\}, s_0, \Sigma, T>$.

The automaton that models the whole WAUT could be built from an exhaustive browsing session obtained by exploring each link, and filling in every possible way and submitting each form, on every page of the application (which is usually unfeasible).

The following is a fragment of a browsing session representing five web pages, and Figure 1 shows the automaton that represents the browsing session, where state s_5 is a deadlock state representing an error page whose status code is 404. URL1, URL2, and URL3 (named as such for simplicity) represent few unexplored links that label transitions to the trap state.

GET http://www.crim.ca HTTP/1.0
Host: www.crim.ca
Accept: application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, image/gif,
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 4.0)
Accept-Language: en-us
----------END OF HTTP REQUEST----------

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 18316
Server: Apache/1.3.9 (Unix) mod_perl/1.21 mod_ssl/2.4.9 OpenSSL/0.9.4
Date: Wed, 10 Apr 2003 19:40:02 GMT
<HTML>
<HEAD> <LINK rel="stylesheet" href="/styles.css">
<TITLE> CRIM</TITLE></HEAD> …
…<a href="/rd/"> recherche-développement </a> …
</HTML>
-------------------------END OF HTTP RESPONSE----------------------------------
```

6 Web Applications with Frames and Multiple Windows

In the previous section, we presented an automata model for single window web applications. However, web applications often use frames and multiple windows. These options allow rendering several pages at the same time, thus introducing concurrency in the behaviors of such web applications. Therefore, using a single automaton is insufficient to adequately model a concurrent behavior of web applications with several frames/windows. In this section, we extend our approach, using communicating finite automata, to model such web applications, which we call *multi-display WA* for simplicity. Before we introduce our extended approach, we define the elements of a browsing session of a multi-display WA.

6.1 Definitions

A response in a multi-display WA is defined as a tuple $<u, c, I, F, L, V>$, where u, c, and V are the same as in Section 5. I and L are extended to include for each action and link the corresponding target. Therefore, an element of L is a tuple $<l, t>$, where l is a URI associated with a link and t is the corresponding target or the empty string ε when no target is defined. Similarly, an element of I becomes a tuple $<a, t>$, where a denotes a form action and t its corresponding target. F is a frame tree defined in the page and whose leaves are frames and internal nodes are framesets. A frame is a tuple of the form $<f, b>$ where f is the URI defined by the value of the *src* attribute of the HTML frame element and b is the frame name. We denote by *leaves*(F) a function that returns the set of leaf nodes (frames) of the tree F.

We define a browsing session of a multi-display WA as a sequence of requests (along with their corresponding targets) and responses. For simplicity, we keep using the term Request/Response Sequence (RRS) to represent a browsing session.

A RRS = $<u_0, c_0, I_0, F_0, L_0, V_0> <r_1, l_1, t_1> <u_1, c_1, I_1, F_1, L_1, V_1> \ldots <r_n, l_n, t_n> <u_n, c_n, I_n, F_n, L_n, V_n>$, where n is the total number of requests in the browsing session starting from $<r_1, l_1, t_1>$. $<r_i, l_i, t_i>$ represents a request such that r_i is a string denoting the request header field, "referer", which is the URI of the page where the request was triggered; and $<l_i, t_i>$ is such that

- if the request is for a filled form then l_i is of the form $a_i?d_i$, where a_i forms with the target t_i a tuple $<a_i, t_i> \in I_j$ of the page $<u_j, c_j, I_j, F_j, L_j, V_j>$, where $u_j = r_i$,
- if the request is for a frame source page then $<l_i, t_i> \in$ *leaves*(F_j) of the page $<u_j, c_j, I_j, F_j, L_j, V_j>$, where $u_j = r_i$, or

- otherwise (if the request is for a link, clicked or implicit), then $<l_i, t_i> \in L_j$ of the page $<u_j, c_j, I_j, F_j, L_j, V_j>$, where $u_j = r_i$,.

Notice that, similar to the case of a single window WA, $<u_0, c_0, I_0, F_0, L_0, V_0>$ corresponds to the initial default page displayed in the browser window such that u_0, c_0 are null, and I_0, F_0, L_0, V_0 are empty sets; $<r_1, l_1, t_1>$ includes the URI l_1 of the starting page, and r_1 and t_1 are the empty string ε. In addition, $l_i = u_i$ if $c_i \neq 3xx$; otherwise, $l_i \neq u_i$ and $<u_i, c_i, I_i, F_i, L_i, V_i>$ immediately follows r_i in the RRS.

6.2 Basic Assumptions

Before we elaborate the model of a multi-display WA, we state basic assumptions about the observed browsing session of the application under test. As in Section 4, we assume that a request is not submitted before the browser delivers the responses to the requests for all frames source pages or for pages displayed in different windows. Also, the following assumptions are essential due to a limitation to directly determine from a request the window/frame from which it was triggered. An observed request/response pair does not include the name of the window/frame targeted by the corresponding URI. To determine the window/frame, we track the "referer" header field in the request which is the URI of the page, where the request is triggered. Thus the following assumptions must hold in the observed browsing session:

1. At each moment, different pages are displayed in frames/windows. If two pages have links to the same page, then only one request corresponding to one of the links is present in the session.
2. If a link is repeated in the same page with different targets and a request for that link is in the session, then this request corresponds to the first instance of that link appearing on the page.

These assumptions are not difficult to satisfy when the browsing session is created by the tester.

6.3 Communicating Finite Automata Model of Multi-display Web Applications

Here we describe how an observed browsing session can be modeled by a system of communicating automata. Given the browsing session, we first determine local browsing sessions that correspond to the behaviors of the entities in the browsed part of the WAUT, such as windows, frames, and framesets, each of which is modeled by an automaton. Then we explain how to convert the local browsing sessions into communicating automata and present the corresponding algorithm which is an extension of Algorithm 1 presented in Section 5.2.

Finite state automata communicate synchronously by rendezvous, executing common actions. Such communication is formalized by the parallel composition operator on automata. Formally, two communicating automata $A_1 = < S_1, s_{01}, \Sigma_1, T_1 >$ and $A_2 = < S_2, s_{02}, \Sigma_2, T_2 >$ are composed using the $\|$ operator. The resulting automaton, denoted $A_1 \| A_2$, is a tuple $< S, s_0, \Sigma, T >$, where $s_0 = (s_{01}, s_{02})$ and $s_0 \in S$; Σ

$= \Sigma_1 \cup \Sigma_2$; and $S \subseteq S_1 \times S_2$ and T are the smallest sets obtained by applying the following rules:

- If $(s_1, e, s'_1) \in T_1$, $e \notin \Sigma_2$, and $(s_1, s_2) \in S$, then $(s'_1, s_2) \in S$, and $((s_1, s_2), e, (s'_1, s_2)) \in T$.
- If $(s_2, e, s'_2) \in T_2$, $e \notin \Sigma_1$, and $(s_1, s_2) \in S$, then $(s_1, s'_2) \in S$, and $((s_1, s_2), e, (s_1, s'_2)) \in T$.
- If $(s_1, e, s'_1) \in T_1$, $(s_2, e, s'_2) \in T_2$, and $(s_1, s_2) \in S$, then $(s'_1, s'_2) \in S$, and $((s_1, s_2), e, (s'_1, s'_2)) \in T$.

The composition is associative and can be applied to finitely many automata.

6.3.1 Local Browsing Sessions

A browsing session represents the behavior of k communicating entities, namely, browser's main and independent windows, frames and framesets, denoted o_1, o_2, …, o_k, where o_1 corresponds to the browser's main window. The entities corresponding to independent windows are determined by analyzing the targets present in the requests; if the target in a request is not an existing frame name, it corresponds to an independent window; for each request whose target is "_blank", a new entity is defined corresponding to a new unnamed independent window. The entities that correspond to frames are determined by the frame names indicated in the frame trees of the response pages; where each frame entity is uniquely identified by $<f, b>$ and the URI u of the frameset document where the corresponding frame tree is defined. The entities corresponding to framesets are identified by analyzing the internal nodes of the frame trees. The number of communicating entities k is then defined as follows. Given a browsing session, $<u_0, c_0, I_0, F_0, L_0, V_0> <r_1, l_1, t_1> <u_1, c_1, I_1, F_1, L_1, V_1> \ldots <r_n, l_n, t_n> <u_n, c_n, I_n, F_n, L_n, V_n>$, let $\{t_1, \ldots, t_q\}$, such that $q \leq n$, be the set of all the distinct targets observed in the requests including window names, frame names, and predefined targets ("_parent", "_top", "_self", "_blank"). Let $\{b_1, \ldots, b_p\}$ be the set of all the frame names defined in all the responses, and m the number of all the framesets defined as well in all the responses. Then, $k = 1 + |\{t_1, \ldots, t_q\} \cup \{b_1, \ldots, b_p\} - \{t_i \mid t_i =$ "_top" or $t_i =$ "_parent" or $t_i =$ "_self" or $t_i =$ "_blank" or $t_i = \varepsilon\}| + |\{<r_j, l_j, t_j> \mid t_i =$ "_blank"$\}| + m$. We further analyze the hierarchical relationship among the different entities of the application. We consider each window entity as a *window tree* whose root node represents the window itself. The first frame tree occurring in (frameset document loaded into) the window is appended to the root of the window tree. If a request's target is a frame name, such that the response is another frameset document (having a frame tree), in the window tree, the response's frame tree is appended to the node of the targeted frame. Similarly, if the target is a frame name, frameset, or the window itself, any subsequent children are removed from the node of the targeted entity and replaced by the response's frame tree if any.

The local browsing sessions (RRS_1, …, RRS_k) corresponding to the observed behavior of k entities of the application are determined as follows. A request/response pair $<r_j, l_j, t_j> <u_j, c_j, I_j, F_j, L_j, V_j>$ belongs to a RRS_i if the target t_j refers to the entity o_i. Also, the *RRS* of each frame/frameset that could be a child of o_i contains the same

request $<r_j, l_j, t_j>$ whose response is the inactive page. At the same time, the *RRS* of the (targeting) entity from which $<r_j, l_j, t_j>$ is triggered must contain $<r_j, l_j, t_j>$ itself with its response being the page where the request is initiated. This is explained by the fact that the targeting entity does not change its displayed page. However, if the target t_j is "_parent", "_top", or a parent entity name, then the response in the *RRS* of the targeting entity is the inactive page. Similarly, the *RRS* of each frame/frameset that is a child of the targeted entity contains the same request $<r_j, l_j, t_j>$ whose response is the inactive page. This means that those frames and framesets are deactivated and erased from the window. If the target attribute is absent or "_self" then $<r_j, l_j, t_j> <u_j, c_j, I_j, F_j, L_j, V_j>$ belongs to a RRS_i provided that the request is triggered from the last page displayed in the corresponding entity o_i. Following is a high-level description of the algorithm that determines the local sessions.

Algorithm 2. Sessions RRS_i, $i = 1, \ldots, k$, are formed using the following algorithm:

1. $RRS_1 := <u_0, c_0, I_0, F_0, L_0, V_0>$ corresponds to the inactive page of the *RRS* of the main window similar to the inactive page defined in Section 4. For $i > 1$, $RRS_i := <u_\Theta, c_\Theta, I_\Theta, F_\Theta, L_\Theta, V_\Theta>$, is defined similarly to $<u_0, c_0, I_0, F_0, L_0, V_0>$ which corresponds to the inactive page from which the local session starts.
2. The first request response pair $<r_1, l_1, t_1> <u_1, c_1, I_1, F_1, L_1, V_1>$ is appended to the session of the browser's main window, i.e., $RRS_1 := RRS_1 <r_1, l_1, t_1> <u_1, c_1, I_1, F_1, L_1, V_1>$.
3. For each request/response pair $<r_j, l_j, t_j> <u_j, c_j, I_j, F_j, L_j, V_j>$, $j > 1$,
   a. if the target t_j refers to entity o_i, $<r_j, l_j, t_j> <u_j, c_j, I_j, F_j, L_j, V_j>$ is appended to RRS_i i.e., $RRS_i := RRS_i <r_j, l_j, t_j> <u_j, c_j, I_j, F_j, L_j, V_j>$. At the same time, $<r_j, l_j, t_j> <u_\Theta, c_\Theta, I_\Theta, F_\Theta, L_\Theta, V_\Theta>$ is appended to the sessions of all the frames and framesets (if any) that are children of o_i.
   b. If the "referer" r_j is equal to the URI of the last response in RRS_i then,
      i. If the target t_j corresponds to a parent entity, the response corresponding to $<r_j, l_j, t_j>$ in RRS_i is the inactive page $<u_\Theta, c_\Theta, I_\Theta, F_\Theta, L_\Theta, V_\Theta>$. Thus, $RRS_i := RRS_i <r_j, l_j, t_j> <u_\Theta, c_\Theta, I_\Theta, F_\Theta, L_\Theta, V_\Theta>$. At the same time, $<r_j, l_j, t_j> <u_\Theta, c_\Theta, I_\Theta, F_\Theta, L_\Theta, V_\Theta>$ is also appended to the sessions of all the frames and framesets that are children of the targeted parent; otherwise,
      ii. the response to $<r_j, l_j, t_j>$ is a tuple $<u, c, I, F, L, V>$ such that $r_j = u$. Thus, $RRS_i := RRS_i <r_j, l_j, t_j> <u, c, I, F, L, V>$.
   c. If the target t_j = "_self" or $t_j = \varepsilon$ and r_j is the URI of the last page displayed in RRS_i, then $RRS_i := RRS_i <r_j, l_j, t_j> <u_j, c_j, I_j, F_j, L_j, V_j>$.

6.3.2 Communicating Finite Automata Model

To build an automata model of a browsing session of a multi-display WA, we convert each local browsing session $RRS_i = <u_{i\Theta}, c_{i\Theta}, I_{i\Theta}, F_{i\Theta}, L_{i\Theta}, V_{i\Theta}> <r_{i1}, l_{i1}, t_{i1}> <u_{i1}, c_{i1}, I_{i1}, F_{i1}, L_{i1}, V_{i1}> \ldots <r_{im}, l_{im}, t_{im}> <u_{im}, c_{im}, I_{im}, F_{im}, L_{im}, V_{im}>$ into an automaton A_i, called the *local session automaton*, by extending Algorithm 1 of Section 5.2.

The set of events Σ_i of the automaton A_i is defined by the union of the following four sets Γ_i , Δ_i, Req_i, and Φ_i. Similar to what is previously defined, $\Gamma_i = \{<l_i, t_i> \mid <l_i,$

$t_i> \in L_{iw}$, $1 \le w \le m\}$ is the set of all the URIs associated with links in the observed responses, $\Delta_i \subseteq \{<a_i, t_i> \mid <a_i, t_i> \in I_{iw}, 1 \le w \le m\}$ is the set of all form actions that correspond to the unexplored forms in the observed responses, and Req_i is the set of all the observed requests. $\Phi_i = \{<f_i, b_i> \mid <f_i, b_i> \in leaves(F_{iw}), 1 \le w \le m\}$ is the set of URIs corresponding to the source pages loaded in the frames.

Algorithm 3. Given an entity o_i and its local browsing session RRS_i, we extend Algorithm 1 to convert RRS_i into a local session automaton A_i as follows.

1. Algorithm 1 is used to convert RRS_i into A_i.
2. The set of events Σ_i is extended to include the set Φ_i of URIs corresponding to the source pages loaded in the frames; thus, $\Sigma_i := \Sigma_i \cup \Phi_i$.
3. Each triple ($<u_{ij}, c_{ij}, I_{ij}, F_{ij}, L_{ij}, V_{ij}> <r_{ij}, l_{ij}, t_{ij}> <u_{i\Theta}, c_{i\Theta}, I_{i\Theta}, F_{i\Theta}, L_{i\Theta}, V_{i\Theta}>$) defines a transition ($s_{ij}$, $<r_{ij}, l_{ij}, t_{ij}>$, s_{i0}), where s_{ij}, s_{i0} correspond to the pages $<u_{ij}, c_{ij}, I_{ij}, F_{ij}, L_{ij}, V_{ij}>$, $<u_{i\Theta}, c_{i\Theta}, I_{i\Theta}, F_{i\Theta}, L_{i\Theta}, V_{i\Theta}>$, respectively;
4. Each triple ($<u_{ij}, c_{ij}, I_{ij}, F_{ij}, L_{ij}, V_{ij}> <r_{ij}, l_{ij}, t_{ij}> <u_{ij+1}, c_{ij+1}, I_{ij+1}, F_{ij+1}, L_{ij+1}, V_{ij+1}>$) such that $<u_{ij}, c_{ij}, I_{ij}, F_{ij}, L_{ij}, V_{ij}> = <u_{ij+1}, c_{ij+1}, I_{ij+1}, F_{ij+1}, L_{ij+1}, V_{ij+1}>$, defines a transition ($s_{ij}$, $<r_{ij}, l_{ij}, t_{ij}>$, s_{ij}), where s_{ij} correspond to $<u_{ij}, c_{ij}, I_{ij}, F_{ij}, L_{ij}, V_{ij}>$;
5. Every event corresponding to a request targeting o_i itself labels a transition from every state of the automaton to the state of the corresponding response page.

The last three steps of the algorithm define the transitions labeled by the events shared by different automata. Step 3 of the algorithm defines transitions labeled by a request initiated by o_i or one of its siblings/children, and whose target is a parent entity. Then, o_i is deactivated and A_i is in the inactive state s_{i0}. Step 4 defines transitions labeled by a request initiated by o_i targeting another entity which is not a parent of o_i. In this case, o_i does not change its displayed page and A_i remains in the current state. The last step of the algorithm states that a shared event targeting o_i is not under the control of A_i and thus should label transitions from every state of A_i to the corresponding state. Thus, in case of an ill-designed application or unreasonable user behavior, where multiple instances of a same window created using the predefined target "_blank", are all treated as a single entity, avoiding state explosion.

Note that there are cases where a frameset in a web application is merely used to group nested frames/framesets within a certain layout without having any behavior itself (it is not the target of any of its children's links). As a result, the corresponding automaton has a single state s_0 (inactive). Therefore, to simplify the model, we discard every automaton that models a frameset entity without any behavior. An automaton for a frameset has more than one state in the case when a request, initiated from a child frame of the frameset and whose target is "_parent", exists in the observed browsing session. As described in Section 6.3.1, in the frameset automaton (initially in state s_0), a transition labeled by the event that corresponds to the request exists from s_0 to the state corresponding to the page displayed in the frameset. At the same time, this event labels transitions in the automaton of each child entity of the frameset from every state to its inactive state. This behavior of framesets in a WA is not modeled in any previous work that we know about.

Let $A_1, \ldots, A_z$ (k-$m \leq z \leq k$, m is the total number of framesets, k is the total number of existing entities in the application) be the automata that model z windows, frames, and possibly framesets. The composition automata A is $A_1 \parallel \ldots \parallel A_z$, such that $A = <S \cup \{trap\}, s_0, \Sigma, T>$. The initial state of A is $s_0 = (s_{01}, \ldots, s_{0z})$; the set of events Σ of A is the union of all Σ_i; the set of states S and the transition relation T of A are defined according to the semantics of the composition operator $\parallel$. The trap state of A is $trap = (trap_1, \ldots, trap_z)$.

7 Implementing the Approach

In this section we describe the framework and the tool that implement our approach for modeling a browsing session recorded when a WAUT is navigated.

7.1 Framework

Our approach is implemented following the framework illustrated in Figure 2:

- The user/tester starts by selecting the web application to test and defining some desired attributes. These attributes, which are defined prior to the analysis process, are used in formulating the properties to verify on the application.
- A monitoring tool intercepts HTTP requests and responses during the navigation of the WAUT performed by the user.
- The intercepted data is fed to an analysis tool that continuously analyzes the data in real time (online mode), incrementally builds an internal data structure of the automata model of the browsing session, and translates it into XML-Promela. The XML-Promela file is then imported to Promela using a functionality of aSpin [23], an extension of Spin model checker [22] that includes the feature of importing a XML-Promela file to Promela language and exporting a Promela file to XML-Promela. The specification of XML-Promela syntax is defined in the Document Type Definition (DTD) file provided with aSpin.

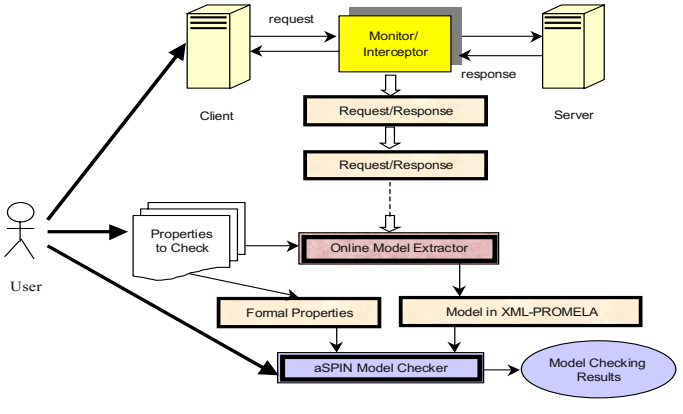

Fig.2. Framework

- aSpin verifies the properties against the model and generates a counter example if a property is not satisfied.

7.2 Online Model Extractor

The Online Model Extractor is implemented in Java as an experimental multithreaded tool that has the following components:

1. A graphical user interface where a range of web related attributes that characterize web applications is provided, and a window showing the progress of the analysis performed during the browsing session.
2. An HTTP Reader that continuously reads intercepted data in an online mode by a monitoring tool, HTTP proxy [5] in our case.
3. A Web Modeler that parses and analyzes the request/response pairs. This module incrementally builds an internal data structure representing the automata model of the WAUT.
4. An XML-Creator that reads the internal data structure and translates it into an XML-Promela based tree which is continuously updated.

8 Case Study

In this section, we illustrate the applicability of our approach using a browsing session of the web application of the Eclipse Consortium, www.eclipse.org. The corresponding web site uses framed pages and multiple windows. The first step in modeling the WAUT is to specify the desired attributes. This is done using the interface of the Model Extractor.

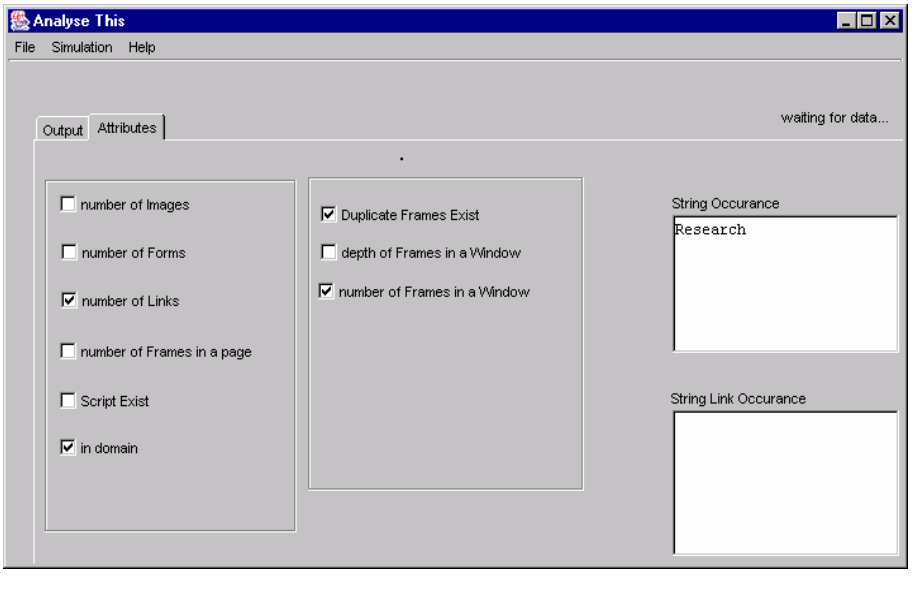

Fig.3. Attribute input window

Fig. 3 shows the attribute input window in the tool. Next, we navigate the application while the request/response pairs are intercepted by the proxy server. The intercepted pairs are fed into the Model Extractor/Manipulator, which produces the model of the application in XML Promela. The resulting XML file is imported into aSpin. The extracted model consists of ten processes reflecting the fact that the application includes seven frames and two windows in which 26 distinct web pages were visited. The frames are within the main browser's window and the second independent window has no frames within it. The global state graph corresponding to our model consists of 847 states and 9652 transitions (stored + matched). In order to prove the validity of our modeling approach, we verified various properties on the model of the application. These properties include reachability properties, and the checking for frame errors such as depth of frames does not exceed a user defined threshold, frames having same name are not active simultaneously, and pages displayed in frames are within the domain name of the application. As an example, we explain the verification of three properties. The first property requires that in the window *mainW*, and thus the frames within it, and the window *blank0*, the number of links in the displayed pages should be balanced, i.e., the difference between the number of links in the displayed pages in the two windows should not exceed a certain number which we fix to 15. This global property requires the exploration of all possible executions of the transitions of the automata of the main window *mainW*, the frames displayed with it, and the independent window *blank0*. The second property requires the absence of a frames error where frames having same names are not active simultaneously. The third property is a reachability property that requires that given three web pages, *program*, *conference*, and *home_main*, there exists at least a path where page *program* is reachable from page *home_main* without going through page *conference*. Note that pages *program* and *conference* are loaded in the independent window *blank0* and page *home_main* is loaded in the frame *main_0*. The first property is formulated in LTL as follows: [] (p || q), where p and q are predicates such that p = *nLinks2* - (*nLinks1* + *nLinks_banner0* + *nLinks_nav0* + *nLinks_main0*) <= 15, and q = *nLinks2* - (*nLinks1* + *nLinks_banner5* + *nLinks_home_nav5* + *nLinks_nav5* + *nLinks_main5*) <= 15. Each variable in these predicates is associated to a process and represents a page attribute that counts the number of links in the page. *nLinks2* is associated to the process of *blank0*, *nLinks1* to the process of *mainW*, and the rest of the variables are associated to the processes of the frames. This property is not satisfied in the model and the verification result produces a counter example simulating a trace that violates the property. The second property is formulated in LTL as follows: [] p, where p = *duplicateFrames_mainW* = = 0 such that *duplicateFrames_mainW* is a Boolean variable that is set to True if two frames having same name are active simultaneously. This property holds in our model. To verify the third property, we negate it and check if it holds in the model. The negation of the property becomes: on all paths from page *home_main* to page *program*, page *conference* is present. We use the LTL property pattern, *Exist Between*, from the repository in [24] to formulate this property as follows: [] (*home_main* && ! *program* → ((! *program*) U ((*conference* && ! *program*) || [] (! *program*)))). This property holds in the model, thus there is no path from page *home*_main to page *program* where page *conference* is absent. Thus, the original property does not hold in model.

9 Conclusion

In this paper, we presented an approach to formally model web applications for the purpose of verification and validation using model checking. We used the dynamic (black-box based) approach by executing the application under test (navigation and form filling), and observing the external behavior of the application by intercepting HTTP requests and responses using a proxy server. We devised algorithms to convert the observed behavior, which we call a browsing session, into an automata based model. In case of applications with frames and multiple windows that exhibit concurrent behavior, the browsing session is partitioned into local browsing sessions, each corresponding to the frame/window/frameset entities in the application under test. These local sessions are then converted into communicating automata. We also presented the framework and tools that implemented the proposed approach, and demonstrated the approach by applying it to a real web application. The constructed models can also be used for other purposes such as documenting, testing, and maintenance of web applications. Currently, we are experimenting with the tool using several types of web applications that reflect both good and bad practices in the development of WA. Our approach is based on the assumption that we observe behavior of WA which is independent of its history. As a future extension, we intend to treat WA behavior that is based on the observation of cookies in requests and responses.

Acknowledgements

We would like to thank Serge Boroday and Andreas Ulrich for the fruitful discussions and their feedback and insights on this work. We also acknowledge students support in the implementation process.

References

[1] "Online Dictionary and Search Engine for Computer and Internet Technology", http://www.pcwebopedia.com/.

[2] "A glossary of World Wide Web Terminology", http://www-personal.umich.edu/~zoe/Glossary.html.

[3] "W3C World Wide Web Consortium", http://www.w3.org.

[4] S. Graham, "HTML Sourcebook, A Complete Guide to HTML 3.0", John Wiley & Sons, Inc., 1996.

[5] "HTTP Proxy Server 1.0", http://www.reitshamer.com/source/httpproxy.html.

[6] "HTTrack Website Copier", http://www.httrack.com/index.php.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code", *In Proc. International Conference on Software Engineering*, 2000.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby, "A, Language Framework For Expressing Checkable Properties of Dynamic Software", *In Proc. of the SPIN Software Model Checking Workshop*, LNCS, Springer-Verlag, Aug, 2000.

[9] S. A. Becker, A. R. Hevner, "A White Box Analysis of Concurrent System Designs", *In Proc. of the 10th Annual International Phoenix Conference on Computers and Communications*, 1991, Scottsdale, AZ, USA, p. 332-338.

[10] J. Conallen, "Modeling Web Application Architectures, with UML", *In Proc. of Communications of the ACM*, October 1999, vol. 2, No. 10.

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

[12] F.B. Paulo, P.C. Masiero, M.C.F. de Olivieira, "Hypercharts: Extended Statecharts to Support Hypermedia Specification", *IEEE Transactions on Software Engineering*, Vol. 25, No. 1, Jan. 1999.

[13] F.B. Paulo, M.A.S. Turine, M.C.F. de Olivieira, "XHMBS: a Formal Model to Support Hypermedia Specification", *In Proc. of the 9th ACM Conference on Hypertext*, United Kingdom, June 1998.

[14] K.R.P.H. Leung, L.C.K. Hui, S.M. Yui, R.W.M. Tang, "Modeling Web Navigation by Statechart", *In Proc. of the 24th IEEE Annual International Computer Software and Applications Conference*, Taipei, Taiwan, October 2000.

[15] IEEE Computer Society, "Software Reengineering Bibliography", http://www.informatik.uni-stuttgart.de/ifi/ps/reengineering, October 28, 2002.

[16] T. Systä, "Static and Dynamic Reverse Engineering Techniques for Java Software Systems", Ph.D. dissertation, Dept. of Computer and Information Sciences, University of Tampere, 2000.

[17] Mansurov N., Probert R., "Dynamic Scenario-Based Approach to Re-Engineering of Legacy Telecommunication Systems", *In Proc. of the 9th SDL Forum (SDL1999)*, pp. 325–341, Montreal, 21–25 June 1999.

[18] "Ethereal, Network Protocol Analyzer", http://www.ethereal.com/.

[19] E. M. Clarke, O. Grumberg, D. A. Peled, "Model Checking", MIT Press, 2000.

[20] Luotonen, "Web Proxy Servers", Prentice Hall PTR, 1998.

[21] Krishnamurthy, J. Rexford, "Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement", Addison-Wesley, 2001.

[22] G.J. Holzmann, "The Spin Model Checker, Primer and Reference Manual", Addison-Wesley, 2003.

[23] "aSpin Model Checker", http://polaris.lcc.uma.es/~gisum/fmse/tools/mainframe.html.

[24] "Repository of Property Specification Patterns",http://patterns.projects.cis.ksu.edu/.

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

[26] L. de Alfaro, T.A. Henziger, F.Y.C. Mang, "MCWEB: A Model-Checking Tool for Web Site Debugging", *Poster, 10th WWW Conference*, Hong Kong, 2001.

[27] P.D. Stotts, C.R. Cabarrus, "Hyperdocuments as Automata: Verification of Trace-Based Browsing Properties by Model Checking", *ACM Transactions on Information Systems*, Vol.16, No. 1, January 1998, 1-30.

[28] P.D. Stotts, J. Navon, "Model Checking CobWeb Protocols for Verification of HTML Frames Behavior", *In Proc. of the 11th WWW Conference*, Hawai, U.S.A., May 2002.

[29] F. Ricca, P. Tonella, "Web Site Analysis: Structure and Evolution", *In Proc. of International Conference on Software Maintenance (ICSM'2000)*, pp. 76-86, San Jose, California, USA, October 11-14, 2000.

[30] F. Ricca and P. Tonella, "Analysis and Testing of Web Applications", *In Proc. of the International Conference on Software Engineering (ICSE'2001)*, pp. 25-34, Toronto, Ontario, Canada, May 12-19, 2001.

[31] P. Tonella and F. Ricca, "Dynamic Model Extraction and Statistical Analysis of Web Applications", *In Proc. of International Workshop on Web Site Evolution (WSE 2002)*, pp. 43-52, Montreal, Canada, October 2, 2002.
[32] Y. Wu, J. Offutt, "Modeling and Testing Web-based Applications", GMU ISE Technical ISE-TR-02-08, November 2002.