Specification and Verification of Web Applications in Rewriting Logic[*]

María Alpuente[1], Demis Ballis[2], and Daniel Romero[1]

[1] Universidad Politécnica de Valencia,
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
{alpuente,dromero}@dsic.upv.es

[2] Dipartimento di Matematica e Informatica,
Via delle Scienze 206, 33100 Udine, Italy
demis@dimi.uniud.it

Abstract. This paper presents a Rewriting Logic framework that formalizes the interactions between Web servers and Web browsers through a communicating protocol abstracting HTTP. The proposed framework includes a scripting language that is powerful enough to model the dynamics of complex Web applications by encompassing the main features of the most popular Web scripting languages (e.g. PHP, ASP, Java Servlets). We also provide a detailed characterization of browser actions (e.g. forward/backward navigation, page refresh, and new window/tab openings) via rewrite rules, and show how our models can be naturally model-checked by using the Linear Temporal Logic of Rewriting (LTLR), which is a Linear Temporal Logic specifically designed for model-checking rewrite theories. Our formalization is particularly suitable for verification purposes, since it allows one to perform in-depth analyses of many subtle aspects related to Web interaction. Finally, the framework has been completely implemented in Maude, and we report on some successful experiments that we conducted by using the Maude LTLR model-checker.

1 Introduction

Over the past decades, the Web has evolved from being a static medium to a highly interactive one. Currently, a number of corporations (including book retailers, auction sites, travel reservation services, etc.) interact primarily through the Web by means of complex interfaces which combine static content with dynamic data produced "on-the-fly" by the execution of server-side scripts (e.g. Java servlets, Microsoft ASP.NET and PHP code). Typically, a Web application consists of a series of Web scripts whose execution may involve several

[*] This work has been partially supported by the EU (FEDER) and the Spanish MEC TIN2007-68093-C02-02 project, by the Universidad Politécnica de Valencia program PAID-06-07 (TACPAS), Generalitat Valenciana, under grant Emergentes GV/2009/024, and by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004. Daniel Romero is also supported by FPI–MEC grant BES–2008–004860.

interactions between a Web browser and a Web server. In a typical scenario, the browser/server interact by means of a particular "client-server" protocol in which the browser requests the execution of a script to the server, then the server executes the script, and it finally turns its output into a response that the browser can display. This execution model -albeit very simple- hides some subtle intricacies which may yield erroneous behaviors.

Actually, Web browsers support backward and forward navigation through Web application stages, and allow the user to open distinct (instances of) Web scripts in distinct windows/tabs which are run in parallel. Such browser actions may be potentially dangerous, since they can change the browser state without notifying the server, and may easily lead to errors or undesired responses. For instance, [1] reports on a frequent error, called the *multiple windows problem*, which typically happens when a user opens the windows for two items in an online store, and after clicking to buy on the one that was opened first, he frequently gets the second one being bought. Moreover, clicking refresh/forward/backward browser buttons may sometimes produce error messages, since such buttons were designed for navigating stateless Web pages, while navigation through Web applications may require multiple state changes. These problems have occurred frequently in many popular Web sites (e.g. Orbitz, Apple, Continental Airlines, Hertz car rentals, Microsoft, and Register.com) [2]. Finally, naïvely written Web scripts may allow security holes (e.g. unvalidated input errors, access control flaws, *etc.* [3]) producing undesired results that are difficult to debug.

Although the problems mentioned above are well known in the Web community, there is a limited number of tools supporting the automated analysis and verification of Web applications. The aim of this paper is to explore the application of formal methods to formal modeling and automatic verification of complex, real-size Web applications.

Our contribution. This paper presents the following original contributions.
- We define a fine-grained, operational semantics of Web applications based on a formal navigational model which is suitable for the verification of real, dynamic Web sites. Our model is formalized within the Rewriting Logic (RWL) framework [4], a rule-based, logical formalism particularly appropriate to modeling concurrent systems [5]. Specifically, we provide a rigorous rewrite theory which i) completely formalizes the interactions between multiple browsers and a Web server through a request/response protocol that supports the main features of the HyperText Transfer Protocol (HTTP); ii) models browsers actions such as refresh, forward/backward navigation, and window/tab openings; iii) supports a scripting language which abstracts the main common features (e.g. session data manipulation, data base interactions) of the most popular Web scripting languages. iv) formalizes *adaptive navigation* [6], that is, a navigational model in which page transitions may depend on user's data or previous computation states of the Web application.
- We also show how rewrite theories specifying Web application models can be model-checked using the *Linear Temporal Logic of Rewriting* (LTLR) [7,8]. The

LTLR allows us to specify properties at a very high level using RWL rules and hence can be smoothly integrated into our RWL framework.
- Finally, the verification framework has been implemented in Maude [9], a high-performance RWL language, which is equipped with a built-in model-checker for LTLR. By using our prototype, we conducted an experimental evaluation which demonstrates the usefulness of our approach.

To the best of our knowledge, this work represents the first attempt to provide a formal RWL verification environment for Web applications which allows one to verify several important classes of properties (e.g. reachability, security, authentication constraints, mutual exclusion, liveness, *etc.*) w.r.t. a *realistic* model of a Web application which includes detailed browser-server protocol interactions, browser navigation capabilities, and Web script evaluation.

Plan of the paper. The rest of the paper is organized as follows. Section 2 briefly recalls some essential notions about Rewriting Logic. Section 3 illustrates a general model for Web interactions which informally describes the navigation through Web applications using HTTP. In Section 4, we specify a rewrite theory formalizing the navigation model of Section 3. This model captures the interaction of the server with multiple browsers and fully supports the most common browser navigation features. In Section 5, we introduce LTLR, and we show how we can use it to formally verify Web applications. In Section 6, we discuss some related work and then we conclude.

2 Preliminaries

We assume some basic knowledge of term rewriting [10] and Rewriting Logic [5]. Let us first recall some fundamental notions which are relevant to this work. The static state structure as well as the dynamic behavior of a concurrent system can be formalized by a RWL specification encoding a *rewrite theory*. More specifically, a *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$, where:
(**i**) (Σ, E) is an order-sorted equational theory equipped with a partial order $<$ modeling the usual subsort relation. Σ, which is called the *signature*, specifies the operators and sorts defining the type structure of $\mathcal{R}$, while E is a set of (possibly conditional) equational axioms which may include commutativity (comm), associativity (assoc) and unity (id). Intuitively, the sorts and operators contained in the signature Σ allow one to formalize system states as ground terms of a term algebra $\tau_{\Sigma,E}$ which is built upon Σ and E.
(**ii**) R defines a set of (possibly conditional) labeled rules of the form $(l : t \Rightarrow t' \text{ if } c)$ such that l is a label, t, t' are terms, and c is an optional boolean term representing the rule condition. Basically, rules in R specify general patterns modeling state transitions. In other words, R formalizes the dynamics of the considered system.

Variables may appear in both equational axioms and rules. By notation $x : S$, we denote that variable x has sort S. A *context* C is a term with a single hole, denoted by $[\,]$, which is used to indicate the location where a reduction occurs.

$C[t]$ is the result of placing t in the hole of C. A *substitution* σ is a finite mapping from variables to terms, and $t\sigma$ is the result of applying σ to term t.

The system evolves by applying the rules of the rewrite theory to the system states by means of *rewriting modulo* E, where E is the set of equational axioms. This is accomplished by means of *pattern matching modulo* E. More precisely, given an equational theory (Σ, E), a term t and a term t', we say that t *matches* t' *modulo* E (or that t *E-matches* t') via substitution σ if there exists a context C such that $C[t\sigma] =_E t'$, where $=_E$ is the congruence relation induced by the equational theory (Σ, E). Hence, given a rule $r = (l : t \Rightarrow t' \text{ if } c)$, and two ground terms s_1 and s_2 denoting two system states, we say that s_1 *rewrites* to s_2 modulo E via r (in symbols $s_1 \xrightarrow{r} s_2$), if there exists a substitution σ such that s_1 E-matches t via σ, $s_2 = C[t'\sigma]$ and $c\sigma$ holds (i.e. it is equal to *true* modulo E). A computation over $\mathcal{R}$ is a sequence of rewrites of the form $s_0 \xrightarrow{r_1} s_1 \dots \xrightarrow{r_k} s_k$, with $r_1, \dots, r_k \in R$, $s_0, \dots, s_k \in \tau_{\Sigma,E}$.

3 A Navigation Model for Web Applications

A Web *application* is a collection of related Web pages, hosted by a Web server, containing Web scripts and links to other Web pages. A Web application is accessed using a Web browser which allows one to navigate through Web pages by clicking and following links.

Communication between the browser and the server is given through the HTTP protocol, which works following a *request-response* scheme. Basically, in the *request* phase, the browser submits a URL to the server containing the Web page P to be accessed, along with a string of input parameters (called the *query* string). Then, the server retrieves P and, if P contains a Web script α, it executes α w.r.t. the input data specified by the query string. According to the execution of α, the server defines the Web application *continuation* (that is, the next page P' to be sent to the browser), and *enables* the links in P' dynamically (*adaptive navigation*). Finally, in the *response* phase, the server delivers P' to the browser.

Since HTTP is a stateless protocol, the Web servers are coupled with a session management technique, which allows one to define Web application states via the notion of *session*, that is, global stores that can be accessed and updated by Web scripts during an established connection between a browser and the server.

The *navigation model* of a Web application can be graphically depicted at a very abstract level by using a graph-like structure as follows. Web pages are represented by nodes which may contain a Web script to be executed (α). Solid arrows connecting Web pages model navigation links which are labeled by a condition and a query string. Conditions provide a simple mechanism to implement a general form of adaptive navigation: specifically, a navigation link will be enabled (i.e. clickable) whenever the associated condition holds. The query string represents the input parameters which are sent to the Web server. Finally, dashed arrows model Web application continuations, that is, arcs pointing to Web pages which are automatically computed by Web script executions. Conditions labeling continuations allow us to model any possible evolution of the

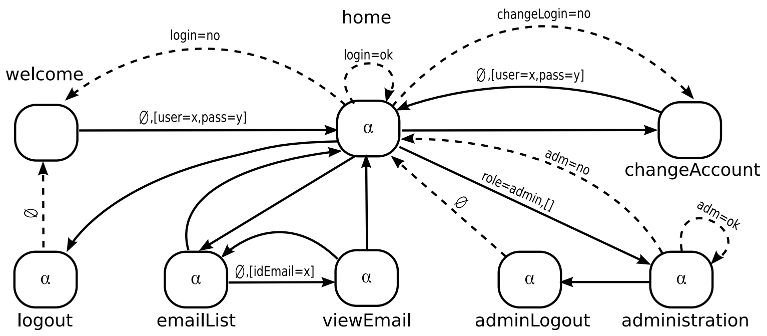

Fig. 1. The navigation model of a Webmail application

Web application of interest. Web application continuations as well as adaptive navigations are dynamically computed w.r.t. the current session (i.e. the current application state).

Example 1. Consider the navigation model given in Figure 1 representing a generic Webmail application which provides some typical functions such as login/logout features, email management, system administration capabilities, *etc.* The solid arrow between the welcome and the home page whose label is $\emptyset$,{user=x,pass=y} defines a navigation link which is always enabled and requires two input parameters. The home page has got two possible continuations (dashed arrows) login=ok and login=no. According to the user and pass values provided in the previous transition, only one of them is chosen. In the former case, the login succeeds and the home page is delivered to the browser, while in the latter case the login fails and the welcome page is sent back to the browser.

An example of adaptive navigation is provided by the navigation link connecting the home page to the administration page. In fact, navigation through that link is enabled only when the condition role=admin holds, that is, the role of the logged user is admin.

4 Formalizing the Navigation Model as a Rewrite Theory

In this section, we define a rewrite theory specifying a navigation model, which allows us to formalize the navigation through a Web application via a communicating protocol abstracting HTTP. The communication protocol includes the interaction of the server with multiple browsers as well as the browser navigation features. Our formalization of a Web application consists of the specifications of the following three components: the Web scripting language, the Web application structure, and the communication protocol.

The Web scripting language. We consider a scripting language which includes the main features of the most popular Web programming languages.

Basically, it extends an imperative programming language with some built-in primitives for reading/writing session data ($\mathsf{getSession}$, $\mathsf{setSession}$), accessing and updating a data base ($\mathsf{selectDB}$, $\mathsf{updateDB}$), capturing values contained in a query string sent by a browser ($\mathsf{getQuery}$). The language is defined by means of an equational theory (Σ_s, E_s), whose signature Σ_s specifies the syntax as well as the type structure of the language, while E_s is a set of equations modeling the operational semantics of the language through the definition of an *evaluation* operator $[\![\_]\!] : \mathsf{ScriptState} \rightarrow \mathsf{ScriptState}$, where $\mathsf{ScriptState}$ is defined by the operator

$$(\_,\_,\_,\_,\_) \colon (\mathsf{Script} \times \mathsf{PrivateMemory} \times \mathsf{Session} \times \mathsf{Query} \times \mathsf{DB}) \rightarrow \mathsf{ScriptState}$$

Roughly speaking, the operator $[\![\_]\!]$ takes as input a tuple $(\alpha, \mathsf{m}, \mathsf{s}, \mathsf{q}, \mathsf{db})$ representing a script α, a private memory m, a session s, a query string q and a data base db, and returns a new script state $(\mathsf{skip}, \mathsf{m}', \mathsf{s}', \mathsf{q}, \mathsf{db}')$ in which the script has been completely evaluated (i.e. it has been reduced to the skip statement) and the private memory, the session and the data base might have been changed because of the script evaluation. In our framework, sessions, private memories, query strings and data bases are modeled by sets of pairs $\mathsf{id} = \mathsf{val}$, where id is an identifier whose value is represented by val. The full formalization of the operational semantics of our scripting language can be found in the technical report [11].

The Web application structure. The Web application structure is modeled by an equational theory (Σ_w, E_w) such that $(\Sigma_w, E_w) \supseteq (\Sigma_s, E_s)$. (Σ_w, E_w) contains a specific sort Soup [9] for modeling multisets (i.e. *soup* of elements whose operators are defined using commutativity, associativity and unity axioms) as follows:

$$\emptyset : \rightarrow \mathsf{Soup} \quad (\mathsf{empty\ soup})$$
$$\_,\_ : \mathsf{Soup} \times \mathsf{Soup} \rightarrow \mathsf{Soup}\ [\mathsf{comm\ assoc\ Id} : \emptyset] \quad (\mathsf{soup\ concatenation}).$$

The structure of a Web page is defined with the following operators of (Σ_w, E_w)

$$(\_,\_,\{\_\},\{\_\}) : (\mathsf{PageName} \times \mathsf{Script} \times \mathsf{Continuation} \times \mathsf{Navigation}) \rightarrow \mathsf{Page}$$
$$(\_,\_) : (\mathsf{Condition} \times \mathsf{PageName}) \rightarrow \mathsf{Continuation}$$
$$\_,[\_] : (\mathsf{PageName} \times \mathsf{Query}) \rightarrow \mathsf{Url}$$
$$(\_,\_) : (\mathsf{Condition} \times \mathsf{Url}) \rightarrow \mathsf{Navigation}$$

where we enforce the following subsort relations $\mathsf{Page} < \mathsf{Soup}$, $\mathsf{Query} < \mathsf{Soup}$, $\mathsf{Continuation} < \mathsf{Soup}$, $\mathsf{Navigation} < \mathsf{Soup}$, $\mathsf{Condition} < \mathsf{Soup}$. Each subsort relation $\mathsf{S} < \mathsf{Soup}$ allows us to automatically define soups of sort S.

Basically a Web page is a tuple $(\mathsf{n}, \alpha, \{\mathsf{cs}\}, \{\mathsf{ns}\}) \in \mathsf{Page}$ such that n is a name identifying the Web page, α is the Web script included in the page, cs represents a soup of possible continuations, and ns defines the navigation links occurring in the page. Each continuation appearing in $\{\mathsf{cs}\}$ is a term of the form $(\mathsf{cond}, \mathsf{n}')$, while each navigation link in ns is a term of the form $(\mathsf{cond}, \mathsf{n}', [\mathsf{q}_1, \ldots, \mathsf{q}_\mathsf{n}])$. A condition is a term of the form $\{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{k} = \mathsf{val}_\mathsf{k}\}$. Given a session s, we say that a continuation $(\mathsf{cond}, \mathsf{n}')$ is *enabled* in s, iff $\mathsf{cond} \subseteq \mathsf{s}$, and a navigation link $(\mathsf{cond}, \mathsf{n}', [\mathsf{q}_1, \ldots, \mathsf{q}_\mathsf{n}])$ is *enabled* in s iff $\mathsf{cond} \subseteq \mathsf{s}$. A Web application is defined as a soup of Page elements defined by the operator $\langle\_\rangle \colon \mathsf{Page} \rightarrow \mathsf{WebApplication}$.

Example 2. Consider again the Web application of Example 1. Its Web application structure can be defined as a soup of Web pages $\mathsf{wapp} = \langle \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_4, \mathsf{p}_5, \mathsf{p}_6, \mathsf{p}_7, \mathsf{p}_8 \rangle$ as follows:

$$\begin{aligned}
\mathsf{p}_1 =& (\mathsf{welcome}, \mathsf{skip}, \{\emptyset\}, \{(\emptyset, \mathsf{home}, [\mathsf{user}, \mathsf{pass}])\})\\
\mathsf{p}_2 =& (\mathsf{home}, \alpha_{\mathsf{home}}, \{(\mathsf{login} = \mathsf{no}, \mathsf{welcome}), (\mathsf{changeLogin} = \mathsf{no}, \mathsf{changeAccount}),\\
& \qquad (\mathsf{login} = \mathsf{ok}, \mathsf{home})\},\\
& \qquad \{(\emptyset, \mathsf{changeAccount}, [\emptyset]), (\mathsf{role} = \mathsf{admin}, \mathsf{administration}, [\emptyset])\\
& \qquad (\emptyset, \mathsf{emailList}, [\emptyset]), (\emptyset, \mathsf{logout}, [\emptyset])\})\\
\mathsf{p}_3 =& (\mathsf{emailList}, \alpha_{\mathsf{emailList}}, \{\emptyset\}, \{(\emptyset, \mathsf{viewEmail}, [\mathsf{emailId}]), (\emptyset, \mathsf{home}, [\emptyset])\})\\
\mathsf{p}_4 =& (\mathsf{viewEmail}, \alpha_{\mathsf{viewEmail}}, \{\emptyset\}, \{(\emptyset, \mathsf{emailList}, [\emptyset]), (\emptyset, \mathsf{home}, [\emptyset])\})\\
\mathsf{p}_5 =& (\mathsf{changeAccount}, \mathsf{skip}, \{\emptyset\}, \{(\emptyset, \mathsf{home}, [\mathsf{user}, \mathsf{pass}])\})\\
\mathsf{p}_6 =& (\mathsf{administration}, \alpha_{\mathsf{admin}}, \{(\mathsf{adm} = \mathsf{no}, \mathsf{home}), (\mathsf{adm} = \mathsf{ok}, \mathsf{administration})\},\\
& \qquad \{\emptyset, \mathsf{adminLogout}, [\emptyset]\})\\
\mathsf{p}_7 =& (\mathsf{adminLogout}, \alpha_{\mathsf{adminLogout}}, \{(\emptyset, \mathsf{home})\}, \{\emptyset\})\\
\mathsf{p}_8 =& (\mathsf{logout}, \alpha_{\mathsf{logout}}, \{(\emptyset, \mathsf{welcome})\}, \{\emptyset\})
\end{aligned}$$

where the application Web scripts might be defined in the following way

$\alpha_{\mathsf{home}} =$

```
login := getSession("login") ;
if (login = null) then
   u := getQuery(user) ;
   p := getQuery(pass) ;
   p1 := selectDB(u) ;
   if (p = p1) then
      r := selectDB(u."-role") ;
      setSession("user", u) ;
      setSession("role", r) ;
      setSession("login", "ok")
   else
      setSession("login", "no") ;
      f := getSession("failed") ;
      if (f = 3) then
          setSession(forbid,"true") fi ;
      setSession("failed", f+1) ;
   fi   fi
```

$\alpha_{\mathsf{admin}} =$

```
u := getSession("user") ;
adm := selectDB("admPage") ;
if (adm = "free")∨(adm = u) then
   updateDB("admPage", u) ;
   setSession("adm", "ok")
else
   setSession("adm", "no")
fi
```

$\alpha_{\mathsf{emailList}} =$

```
u := getSession("user") ;
es := selectDB(u . "-email) ;
setSession("email-found", es)
```

$\alpha_{\mathsf{viewEmail}} =$

```
u := getSession("user") ;
id := getQuery(idEmail) ;
e := selectDB(id) ;
setSession("text-email", e)
```

$\alpha_{\mathsf{adminLogout}} =$ `updateDB("admPage", "free")` $\alpha_{\mathsf{logout}} =$ `clearSession`

The communication protocol. We define the communication protocol by means of a rewrite theory (Σ_p, E_p, R_p), where (Σ_p, E_p) is an equational theory formalizing the Web application states, and R_p is a set of rewrite rules specifying Web script evaluations and request/response protocol actions.

The equational theory (Σ_p, E_p). The theory is built on top of the equational theory (Σ_w, E_w) (i.e. $(\Sigma_p, E_p) \supseteq (\Sigma_w, E_w)$) and models, on the one hand, the entities into play (i.e. the Web server, the Web browser and the protocol messages); on the other hand, it provides a formal mechanism to evaluate enabled continuations as well as enabled adaptive navigations which may be generated "on-the-fly" by executing Web scripts. More formally, (Σ_p, E_p) includes the following operators.

$$\begin{aligned}
\mathsf{B}(\_,\_,\_,\{\_\},\{\_\},\_,\_,\_) &: (\mathsf{Id} \times \mathsf{Id} \times \mathsf{PageName} \times \mathsf{URL} \times \mathsf{Session} \times \mathsf{Message} \\
&\qquad \times \mathsf{History} \times \mathsf{Nat}) \to \mathsf{Browser} \\
\mathsf{S}(\_,\{\_\},\{\_\},\_,\_) &: (\mathsf{WebApplication} \times \mathsf{BrowserSession} \times \mathsf{DB} \times \mathsf{Message} \\
&\qquad \times \mathsf{Message}) \to \mathsf{Server} \\
\mathsf{H}(\_,\{\_\},\_) &: (\mathsf{PageName} \times \mathsf{URL} \times \mathsf{Message}) \to \mathsf{History} \\
\mathsf{B2S}(\_,\_,\_,[\_],\_) &: (\mathsf{Id} \times \mathsf{Id} \times \mathsf{PageName} \times \mathsf{Query} \times \mathsf{Nat}) \to \mathsf{Message} \\
\mathsf{S2B}(\_,\_,\_,\{\_\},\{\_\},\_) &: (\mathsf{Id} \times \mathsf{Id} \times \mathsf{PageName} \times \mathsf{URL} \times \mathsf{Session} \times \mathsf{Nat}) \to \mathsf{Message} \\
\mathsf{BS}(\_,\{\_\}) &: (\mathsf{Id} \times \mathsf{Session}) \to \mathsf{BrowserSession} \\
\_\|\_\|\_ &: (\mathsf{Browser} \times \mathsf{Message} \times \mathsf{Server}) \to \mathsf{WebState}
\end{aligned}$$

where we enforce the following subsort relations $\mathsf{History} < \mathsf{List}$, $\mathsf{URL} < \mathsf{Soup}$, $\mathsf{BrowserSession} < \mathsf{Soup}$, $\mathsf{Browser} < \mathsf{Soup}$, and $\mathsf{Message} < \mathsf{Queue}$[1].

We model a browser as a term of the form

$$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{n}, \{\mathsf{url_1}, \ldots, \mathsf{url_l}\}, \{\mathsf{id_1} = \mathsf{val_1}, \ldots, \mathsf{id_m} = \mathsf{val_m}\}, \mathsf{m}, \mathsf{h}, \mathsf{i})$$

where $\mathsf{id_b}$ is an identifier representing the browser; $\mathsf{id_t}$ is an identifier modeling an open tab of browser $\mathsf{id_b}$; n is the name of the Web page which is currently displayed on the Web browser; $\mathsf{url_1}, \ldots, \mathsf{url_l}$ represent the navigation links which appear in the Web page n; $\{\mathsf{id_1} = \mathsf{val_1}, \ldots, \mathsf{id_m} = \mathsf{val_m}\}$ is the last session that the server has sent to the browser; m is the last message sent to the server; h is a bidirectional list recording the history of the visited Web pages; and i is an internal counter used to distinguish among several response messages generated by repeated refresh actions (e.g. if a user pressed twice the refresh button, only the second refresh is displayed in the browser window).

The server is formalized by using a term of the form

$$\mathsf{S}(\langle \mathsf{p_1}, \ldots, \mathsf{p_l} \rangle, \{\mathsf{BS}(\mathsf{id_{b1}}, \{\mathsf{s_1}\}), .., \mathsf{BS}(\mathsf{id_{bn}}, \{\mathsf{s_n}\})\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}})$$

where $\langle \mathsf{p_1}, \ldots, \mathsf{p_l} \rangle$ defines the Web application currently in execution; $\mathsf{s_i} = \{\mathsf{id_1} = \mathsf{val_1}, \ldots, \mathsf{id_m} = \mathsf{val_m}\}$ is the session that belongs to browser $\mathsf{id_{bi}}$, which is needed to keep track of the Web application state of each user; $\mathsf{db} = \{\mathsf{id_1} = \mathsf{val_1}, \ldots, \mathsf{id_k} = \mathsf{val_k}\}$ specifies the data base hosted by the Web server; and $\mathsf{fifo_{req}}$,$\mathsf{fifo_{res}}$ are two queues of messages, which respectively model the request messages which still have to be processed by the server and the pending response messages that the server has still to send to the browsers.

We assume the existence of a bidirectional channel through which server and browser can communicate by message passing. In this context, terms of the form

$$\mathsf{B2S}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{n}, [\mathsf{id_1} = \mathsf{val_1}, \ldots, \mathsf{id_m} = \mathsf{val_m}], \mathsf{i})$$

model *request* messages, that is, messages sent from the browser $\mathsf{id_b}$ (and tab $\mathsf{id_t}$) to the server asking for the Web page n with query parameters $[\mathsf{id_1} = \mathsf{val_1}, \ldots, \mathsf{id_m} = \mathsf{val_m}]$. Instead, terms of the form

$$\mathsf{S2B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{n}, \{\mathsf{url_1}, \ldots, \mathsf{url_l}\}, \{\mathsf{id'_1} = \mathsf{val'_1}, \ldots, \mathsf{id'_m} = \mathsf{val'_m}\}, \mathsf{i})$$

[1] We represent a queue with elements $e_1, \ldots, e_n$ by $(e_1, \ldots, e_n)$, where e_1 is the first element of the queue.

model *response* messages, that is, messages sent from the server to the browser $\mathsf{id_b}$ (and tab $\mathsf{id_t}$) including the computed Web page n along with the navigation links $\{\mathsf{url_1}, \ldots, \mathsf{url_l}\}$ occurring in n, and the current session information[2].

Using the operators described so far, we can precisely formalize the notion of Web application state as a term of the form

$$\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv}$$

where br is a soup of browsers, m is a channel modeled as a queue of messages, and sv is a server. Intuitively, a Web application state can be interpreted as a snapshot of the system capturing the current configurations of the browsers, the server and the channel.

The equational theory (Σ_p, E_p) also defines the operator

$$\mathsf{eval(\_,\_,\_,\_) : WebApplication \times Session \times DB \times Message \rightarrow Session \times DB \times Message}$$

whose semantics is specified by means of E_p (see [11] for the precise formalization of eval). Given a Web application w, a session s, a data base db, and a request message $\mathsf{b2s} = \mathsf{B2S(id_b, id_t, n, [q], k)}$, $\mathsf{eval(w, s, db, b2s)}$ generates a triple $\mathsf{(s', db', s2b)}$ containing an updated session $\mathsf{s'}$, an updated data base $\mathsf{db'}$, and a response message $\mathsf{s2b} = \mathsf{S2B(id_b, id_t, n', \{url_1, \ldots, url_m\}, s', k)}$. Intuitively, the operator eval allows us to execute a Web script and dynamically determine (i) which Web page $\mathsf{n'}$ is generated by computing an enabled continuation, and (ii) which links of $\mathsf{n'}$ are enabled w.r.t. the current session.

The rewrite rule set R_p. It defines a collection of rewrite rules of the form $\mathsf{label : WebState \Rightarrow WebState}$ abstracting the standard request-response behavior of the HTTP protocol and the browser navigation features. First, we give the rules that formalize the HTTP protocol, and then the rules that correspond to the browser navigation features. More specifically, the HTTP protocol specifies the requests of multiple browsers, the script evaluations, and the server responses by means of the following rules.

$$\begin{aligned}\mathsf{ReqIni}: \ & \mathsf{B(\underline{id_b, id_t, p_c, \{(np,[q])}, urls\}, \{s\}, lm, h, i), br \parallel m \parallel sv \Rightarrow} \\ & \qquad \mathsf{B(id_b, id_t, \underline{emptyPage}, \emptyset, \{s\}, \underline{m_{idb,idt}}, \underline{h_c}, i), br \parallel (m, \underline{m_{idb,idt}}) \parallel sv}\end{aligned}$$
where $\mathsf{m_{idb,idt}} = \mathsf{B2S(id_b, id_t, np, [q], i)}$ and $\mathsf{h_c} = \mathsf{push((p_c, \{(np,[q]), urls\}, m_{idb,idt}), h)}$

$$\begin{aligned}\mathsf{ReqFin}: \ & \mathsf{br \parallel (\underline{m_{idb,idt}}, m) \parallel S(w, \{bs\}, \{db\}, fifo_{req}, fifo_{res}) \Rightarrow} \\ & \qquad \mathsf{br \parallel m \parallel S(w, \{bs\}, \{db\}, (fifo_{req}, \underline{m_{idb,idt}}), fifo_{res})}\end{aligned}$$
where $\mathsf{m_{idb,idt}} = \mathsf{B2S(id_b, id_t, np, [q], i)}$

$$\begin{aligned}\mathsf{Evl}: \ & \mathsf{br \parallel m \parallel S(w, \{BS\underline{(id_b, \{s\})}, bs\}, \underline{\{db\}, (m_{idb,idt}}, fifo_{req}), fifo_{res}) \Rightarrow} \\ & \qquad \mathsf{br \parallel m \parallel S(w, \{BS(id_b, \underline{\{s'\}}), bs\}, \underline{\{db'\}}, fifo_{req}, (fifo_{res}, \underline{m'}))}\end{aligned}$$
where $\mathsf{(s', db', m')} = \mathsf{eval(w, s, db, m_{idb,idt})}$

[2] Session information is typically represented by HTTP *cookies*, which are textual data sent from the server to the browser to let the browser know the current application state.

$$\mathsf{ResIni}\colon \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo}_{\mathsf{req}}, (\underline{\mathsf{m}_{\mathsf{idb,idt}}}, \mathsf{fifo}_{\mathsf{res}})) \Rightarrow$$
$$\mathsf{br} \parallel (\mathsf{m}, \underline{\mathsf{m}_{\mathsf{idb,idt}}}) \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo}_{\mathsf{req}}, \mathsf{fifo}_{\mathsf{res}})$$

$$\mathsf{ResFin}\colon \mathsf{B}(\underline{\mathsf{id_b}, \mathsf{id_t}}, \mathsf{emptyPage}, \emptyset, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \underline{\mathsf{i}}), \mathsf{br} \parallel (\underline{\mathsf{S2B}((\mathsf{id_b}, \mathsf{id_t}, \mathsf{p}', \mathsf{urls}, \{\mathsf{s}'\}), \mathsf{i})}, \mathsf{m}) \parallel \mathsf{sv}$$
$$\Rightarrow \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \underline{\mathsf{p}', \mathsf{urls}, \{\mathsf{s}'\}}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv}$$

where $\mathsf{id_b}, \mathsf{id_t} : \mathsf{Id}$, $\mathsf{br} : \mathsf{Browser}$, $\mathsf{sv} : \mathsf{Server}$, $\mathsf{urls} : \mathsf{URL}$, $\mathsf{i} : \mathsf{Nat}$, $\mathsf{q} : \mathsf{Query}$, $\mathsf{h} : \mathsf{History}$, $\mathsf{w} : \mathsf{WebApplication}$, $\mathsf{m}, \mathsf{m}', \mathsf{m}_{\mathsf{idb,idt}}, \mathsf{fifo}_{\mathsf{req}}, \mathsf{fifo}_{\mathsf{res}} : \mathsf{Message}$, $\mathsf{p_c}, \mathsf{p}', \mathsf{np} : \mathsf{PageName}$, $\mathsf{s}, \mathsf{s}' : \mathsf{Session}$, and $\mathsf{bs} : \mathsf{BrowserSession}$ are variables.

Roughly speaking, the request phase is split into two parts, which are respectively formalized by rules ReqIni and ReqFin. Initially, when a browser with identifier $\mathsf{id_b}$ requests the navigation link $(\mathsf{np}, [\mathsf{q}])$ appearing in a Web page $\mathsf{p_c}$ of the tab $\mathsf{id_t}$, rule ReqIni is fired. The execution of ReqIni generates a request message $\mathsf{m}_{\mathsf{idb,idt}}$ which is enqueued in the channel and saved in the browser as the last message sent. The history list is updated as well. Rule ReqFin simply dequeues the first request message $\mathsf{m}_{\mathsf{idb,idt}}$ of the channel and inserts it into $\mathsf{fifo}_{\mathsf{req}}$, which is the server queue containing pending requests. Rule Evl consumes the first request message $\mathsf{m}_{\mathsf{idb,idt}}$ of the queue $\mathsf{fifo}_{\mathsf{req}}$, evaluates the message w.r.t. the corresponding browser session $(\mathsf{id_b}, \{\mathsf{s}\})$, and generates the response message which is enqueued in $\mathsf{fifo}_{\mathsf{res}}$, that is, the server queue containing the responses to be sent to the browsers. Finally, rules ResIni and ResFin implement the response phase. First, rule ResIni dequeues a response message from $\mathsf{fifo}_{\mathsf{res}}$ and sends it to the channel m. Then, rule ResFin takes the first response message from the channel queue and sends it to the corresponding browser tab.

It is worth noting that the whole protocol semantics is elegantly defined by means of only five, high-level rewrite rules without making any implementation detail explicit. Implementation details are automatically managed by the rewriting logic engine (i.e. rewrite modulo equational theories). For instance, in the rule ReqIni, no tricky function is needed to select an arbitrary navigation link $(\mathsf{np}, [\mathsf{q}])$ from the URLs available in a Web page, since they are modeled as associative and commutative soups of elements (i.e. $\mathsf{URL} < \mathsf{Soup}$) and hence a single URL can be extracted from the soup by simply applying pattern matching modulo associativity and commutativity.

We formalize browser navigation features as follows.

$$\mathsf{Refresh}\colon \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \underline{\mathsf{lm}}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \Rightarrow$$
$$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{emptyPage}, \emptyset, \{\mathsf{s}\}, \underline{\mathsf{m}_{\mathsf{idb,idt}}}, \mathsf{h}, \underline{\mathsf{i}+1}), \mathsf{br} \parallel (\mathsf{m}, \underline{\mathsf{m}_{\mathsf{idb,idt}}}) \parallel \mathsf{sv}$$

where $\mathsf{lm} = \mathsf{B2S}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{np}, \mathsf{q}, \mathsf{i})$ and $\mathsf{m}_{\mathsf{idb,idt}} = \mathsf{B2S}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{np}, \mathsf{q}, \mathsf{i}+1)$

$$\mathsf{OldMsg}\colon \mathsf{B}(\underline{\mathsf{id_b}, \mathsf{id_t}}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \underline{\mathsf{i}}), \mathsf{br} \parallel (\underline{\mathsf{S2B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p}', \mathsf{urls}', \{\mathsf{s}'\}, \mathsf{k})}, \mathsf{m}) \parallel \mathsf{sv} \Rightarrow$$
$$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \quad \text{if } \underline{\mathsf{i} \neq \mathsf{k}}$$

$$\mathsf{NewTab}\colon \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \Rightarrow$$
$$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \underline{\mathsf{B}(\mathsf{id_b}, \mathsf{id_{nt}}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \emptyset, \emptyset, 0)}, \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv}$$

where $\mathsf{id_{nt}}$ is a new fresh value of the sort Id.

$$\mathsf{Backward}: \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \Rightarrow \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \underline{\mathsf{p_h}, \{\mathsf{urls_h}\}}, \{\mathsf{s}\}, \underline{\mathsf{lm_h}}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv}$$
where $(\mathsf{p_h}, \{\mathsf{url_h}\}, \mathsf{lm_h}) = \mathsf{prev}(\mathsf{h})$

$$\mathsf{Forward}: \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \Rightarrow \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \underline{\mathsf{p_h}, \{\mathsf{urls_h}\}}, \{\mathsf{s}\}, \underline{\mathsf{lm_h}}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv}$$
where $(\mathsf{p_h}, \{\mathsf{urls_h}\}, \mathsf{lm_h}) = \mathsf{next}(\mathsf{h})$

where $\mathsf{id_b}, \mathsf{id_t}, \mathsf{id_{nt}} : \mathsf{Id}$, $\mathsf{br} : \mathsf{Browser}$, $\mathsf{sv} : \mathsf{Server}$, $\mathsf{urls}, \mathsf{url}', \mathsf{urls_h} : \mathsf{URL}$, $\mathsf{q} : \mathsf{Query}$, $\mathsf{h} : \mathsf{History}$ $\mathsf{m}, \mathsf{lm}, \mathsf{lm_h}, \mathsf{m_{idb,idt}} : \mathsf{Message}$, $\mathsf{i}, \mathsf{k} : \mathsf{Nat}$, $\mathsf{p_c}, \mathsf{p}', \mathsf{np}, \mathsf{p_h} : \mathsf{PageName}$, and $\mathsf{s}, \mathsf{s}' : \mathsf{Session}$ are variables.

Rules Refresh and OldMsg model the behavior of the refresh button of a browser. Rule Refresh applies when a page refresh is invoked. Basically, it first increments the browser internal counter, and then a new version of the last request message lm, containing the updated counter, is inserted into the channel queue. Note that the browser internal counter keeps track of the number of repeated refresh button clicks. Rule OldMsg is used to consume all the response messages in the channel, which might have been generated by repeated clicks of the refresh button, with the exception of the last one ($i = k$).

Finally, rules NewTab, Backward and Forward are quite intuitive: an application of NewTab simply generates a new Web application state containing a new fresh tab in the soup of browsers, while Backward (resp. Forward)) extracts from the history list the previous (resp. next) Web page and sets it as the current browser Web page.

It is worth noting that applications of rules in R_p might produce an infinite number of (reachable) Web application states. For instance, an infinite applications of rule newTab would produce an infinite number of Web application states, each of which represents a finite number of open tabs. Therefore, to make analysis and verification feasible, we set some restrictions in our prototypical implementation of the model to limit the number of reachable states (e.g. we fixed upper bounds on the length of the history list, on the number of tabs the user can open, *etc.*). An alternative approach that we plan to pursue in the future is to define a state abstraction by means of a suitable equational theory in the style of [12]. This would allow us to produce finite (and hence effective) descriptions of infinite state systems.

5 Model Checking Web Applications Using LTLR

The formal specification framework presented so far allows us to specify a number of subtle aspects of the Web application semantics which can be verified using model-checking techniques. To this respect, the Linear Temporal Logic of Rewriting (LTLR)[8] can be fruitfully employed to formalize properties which are either not expressible or difficult to express using other verification frameworks.

The Linear Temporal Logic of Rewriting. LTLR is a sublogic of the family of the Temporal Logics of Rewriting TLR* [8], which allows one to specify properties of a given rewrite theory in a simple and natural way. In particular, we chosen the "tandem" $LTLR/(\Sigma_p, E_p, R_p)$. In the following, we provide an intuitive explanation of the main features of LTLR; for a thorough discussion, please refer to [8].

LTLR extends the traditional Linear Temporal Logic (LTL) with *state predicates* (SP) and *spatial action patterns* (Π). A LTLR formulae w.r.t. SP and Π can be defined by means of the following BNF-like syntax:

$$\varphi ::= \delta \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \varphi \mid \Diamond\varphi \mid \Box\varphi$$
$$\text{where } \delta \in SP, p \in \Pi, \text{ and } \varphi \in LTLR(SP, \Pi)$$

Since LTLR generalizes LTL, the modalities and semantic definitions are entirely similar to those for LTL (see, e.g., [13]). The key new addition is the semantics of spatial actions; the relation $R, (\pi, \gamma) \models \delta$ holds if and only if the proof term $\gamma(0)$ of a current computation is an instance of a spatial action pattern δ.

Let us illustrate the state predicates and the spatial action patterns by means of a rather intuitive example. Let (Σ_p, E_p, R_p) be the rewrite theory specified in Section 4, modeling the Web application states as terms $\mathsf{b}\|\mathsf{m}\|\mathsf{s}$ of sort $\mathsf{WebState}$. Then, the state predicate $\mathsf{B}(\mathsf{ib_b}, \mathsf{id_t}, \mathsf{page}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{m}, \mathsf{h}, \mathsf{i})\|\mathsf{m}\|\mathsf{sv} \models \mathsf{curPage}(\mathsf{page}) = \mathsf{true}$ holds (i.e. evaluates to true) for each state such that page is the current Web page displayed in the browser. Besides, the spatial action pattern $\mathsf{ReqIni}(\mathsf{id_b}\backslash\mathsf{A})$ allows us to localize all the applications of the rule ReqIni where the rule's variable $\mathsf{id_b}$ are instantiated with A, that is, all ReqIni applications which refer to the browser with identifier A.

LTLR properties for Web Applications. This section shows the main advantages of coupling LTLR with Web applications specified via the rewrite theory (Σ_p, E_p, R_p).

Concise and parametric properties. As LTLR is a highly parametric logic, it allows one to define complex properties in a concise way by means of state predicates and spatial action patters. As an example, consider the Webmail application given in Example 1, along with the property *"Incorrect login info is allowed only 3 times, and then login is forbidden"*. This property might be formalized as the following standard LTL formula:

$$\Diamond(\mathsf{welcomeA}) \rightarrow \Diamond(\mathsf{welcomeA} \wedge \bigcirc(\neg(\mathsf{forbiddenA}) \vee (\mathsf{welcomeA} \wedge \bigcirc(\neg(\mathsf{forbiddenA}) \vee$$
$$(\mathsf{welcomeA} \wedge \bigcirc(\neg(\mathsf{forbiddenA}) \vee \bigcirc(\mathsf{forbiddenA} \wedge \Box(\neg\mathsf{welcomeA})))))))$$

where $\mathsf{welcomeA}$ and $\mathsf{forbiddenA}$ are atomic propositions respectively describing (i) user A is displaying the $\mathsf{welcome}$ page, and (ii) login is forbidden for user A. Although the property to be modeled is rather simple, the resulting LTL formula is textually large and demands a hard effort to be specified and verified.

Moreover, the complexity of the formula would rapidly grow when a higher number of login attempts was considered[3].

By using LTLR we can simply define a login property which is parametric w.r.t. the number of login attempts as follows. First of all, we define the state predicates: (*i*) $\mathsf{curPage(id,pn)}$, which holds when user id[4] is displaying Web page pn; (*ii*) $\mathsf{failedAttempt(id,n)}$, which holds when user id has performed n failed login attempts; (*iii*) $\mathsf{userForbidden(id)}$, which holds when a user is forbidden from logging on to the system. Formally,

$$\begin{aligned}
\mathsf{B}(\underline{\mathsf{id}}, \mathsf{id_t}, \underline{\mathsf{pn}}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} &\models \mathsf{curPage}(\mathsf{id}, \mathsf{pn}) = \mathsf{true} \\
\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\underline{(\mathsf{id}, \{\mathsf{failed} = \mathsf{n}\})}, \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{f_{req}}, \mathsf{f_{res}}) &\models \mathsf{failedAttempt}(\mathsf{id}, \mathsf{n}) = \mathsf{true} \\
\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\underline{(\mathsf{id}, \{\mathsf{forbid} = \mathsf{true}\})}, \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{f_{req}}, \mathsf{f_{res}}) &\models \mathsf{userForbidden}(\mathsf{id}) = \mathsf{true}
\end{aligned}$$

Then, the security property mentioned above is elegantly formalized by means of the following LTLR formula

$$\Diamond(\mathsf{curPage}(\mathsf{A}, \mathsf{welcome}) \wedge \bigcirc(\Diamond\mathsf{failedAttempt}(\mathsf{A}, 3))) \rightarrow \Box\mathsf{userForbidden}(\mathsf{A})$$

Observe that the previous formula can be easily modified to deal with a distinct number of login attempts —it is indeed sufficient to change the parameter counting the login attempts in the state predicate $\mathsf{failedAttempt}(\mathsf{A}, 3)$. Besides, note that we can define state predicates (and more in general LTLR formulae) which depend on Web script evaluations. For instance, the predicate $\mathsf{failedAttempt}$ depends on the execution of the login script α_{home} which may or may not set the forbid value to true in the user's browser session.

Web script evaluation witnesses the "on-the-fly" capability of our framework which allows us to specify, in a natural way, suitable properties to check the behavior of the scripts.

Unreachability properties. Unreachability properties can be specified as LTLR formulae of the form $\Box \neg \langle \mathsf{State} \rangle$, where State is an unwanted state the system has not to reach. By using unreachability properties over the rewrite theory (Σ_p, E_p, R_p), we can detect very subtle instances of the *multiple windows problem* mentioned in Section 1.

Example 3. Consider again the Webmail application of Example 1. Assume that a user may interact with the application using two email accounts MA and MB. Let us consider a Web application state in which the user is logged in the home page with her account MA. Now, assume that the following sequence of actions is executed: (1) the user opens a new browser tab; (2) the user changes the account in one of the two open tabs and logs in using MB credentials; (3) the user accesses the $\mathsf{emailList}$ page from both tabs.

After applying the previous sequence of actions, one expects to see in the two open tabs the emails corresponding to the accounts MA and MB. However, the

[3] Try thinking of how to specify an LTL formula for a more flexible security policy permitting 10 login attempts.

[4] We assume that the browser identifier univocally identifies the user.

Webmail application of Example 1 displays the emails of MB in both tabs. This is basically caused by action (2) which makes the server override the browser session with MB data without notifying the state change to the tab associated with the MA account.

This unexpected behavior can be recognized by using the following LTLR unreachability formula $\Box\neg\,\mathsf{inconsistentState}$ where $\mathsf{inconsistentState}$ is a state predicate defined as:

$$\begin{array}{l}\mathsf{B}(\underline{\mathsf{id}}, \underline{\mathsf{idA}}, \mathsf{p_A}, \{\mathsf{urls_A}\}, \{\underline{(\mathsf{user} = \mathsf{MA})}, \mathsf{s_A}\}, \mathsf{lm_A}, \mathsf{h_A}, \mathsf{i_A}),\\ \mathsf{B}(\underline{\mathsf{id}}, \underline{\mathsf{idB}}, \mathsf{p_B}, \{\mathsf{urls_B}\}, \{\underline{(\mathsf{user} = \mathsf{MB})}, \mathsf{s_B}\}, \mathsf{lm_B}, \mathsf{h_B}, \mathsf{i_B}), \mathsf{br}\ \|\mathsf{m}\|\ \mathsf{sv}\\ \qquad\qquad\qquad\qquad\qquad\models \mathsf{inconsistentState} = \mathsf{true} \qquad \mathsf{if}(\mathsf{MA} \neq \mathsf{MB})\end{array}$$

Roughly speaking, the property $\Box\neg\,\mathsf{inconsistentState}$ states that we do not want to reach a Web application state in which two browser tabs refer to distinct user sessions. If this happens, one of the two session is out-of-date and hence inconsitent.

Finally, it is worth noting that by means of LTLR formulae expressing unreachability statements, we can formalize an entire family of interesting properties such as:

- *mutual exclusion* (e.g. $\Box\neg\,(\mathsf{curPage}(\mathsf{A}, \mathsf{administration}) \wedge \mathsf{curPage}(\mathsf{B}, \mathsf{administration}))$),
- *link accessibility* (e.g. $\Box\neg\,\mathsf{curPage}(\mathsf{A}, \mathsf{PageNotFound})$),
- *security properties*, (e.g. $\Box\neg\,(\mathsf{curPage}(\mathsf{A}, \mathsf{home}) \wedge \mathsf{userForbidden}(\mathsf{A}))$).

Liveness through spatial actions. Liveness properties state that something good keeps happening in the system. In our framework, we can employ spatial actions to detect *good* rule applications. For example, consider the following property "*user* A *always succeeds to access her* home *page from the* $\mathsf{welcome}$ *page*". This amount to saying that, whenever the protocol rule ReqIni is applied to request the home page of user A, the browser will eventually display the home page of user A. This property can be specified by the following LTLR formula:

$$\Box([\mathsf{ReqIni}(\mathsf{Id_b}\backslash\mathsf{A}, \mathsf{p_c}\backslash\mathsf{welcome}, \mathsf{np}\backslash\mathsf{home})] \rightarrow \Diamond\mathsf{curPage}(\mathsf{A}, \mathsf{home}))$$

Implementation. The verification framework we presented has been implemented in a prototypical system, written in Maude [9], which is publicly available along with several examples at
`http://www.dsic.upv.es/~dromero/web-tlr.html`.

The prototype allows one to define Web application models as extended rewrite theories specified by means of Maude specifications. Then, specifications can be automatically verified using the Maude built-in operator `tlr check`[7] which supports model checking of rewrite theories w.r.t. LTLR. We tested the tool on several examples including all the examples presented in the paper. Preliminary experiments have demonstrated that the experimental system work very satisfactorily on several Web applications models. We are currently developing a Web interface for the tool. Moreover, as future work, we want to define a translator from Web applications written in a commercial language to our Web models.

6 Related Work and Conclusions

Web applications are complex software systems playing a primary role of primary importance nowadays. Not surprisingly, several works have previously addressed the modeling and verification of such systems. A variant of the μ-calculus (called constructive μ-calculus) is proposed in [14] which allows one to model-check connectivity properties over the *static* graph-structure of a Web system. However, this methodology does not support the verification of dynamic properties— e.g. reachability over Web pages generated by means of Web script execution.

Both Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) have been used for the verification of dynamic Web applications. For instance, [15] and [16] support model-checking of LTL properties w.r.t. Web application models represented as Kripke structures. Similar methodologies have been developed in [17] and [18] to verify Web applications by using CTL formulae. All these model-checking approaches are based on coarse Web application models which are concerned neither with the communication protocol underlying the Web interactions nor the browser navigation features. Moreover, as shown in Section 5, CTL and LTL property specifications are very often textually large and hence difficult to formulate and understand. [6] presents a modeling and verification methodology that uses CTL and considers some basic adaptive navigation features. In constrast, our framework provides a complete formalization which supports more advanced adaptive navigation capabilities.

Finally, both [2] and [19] do provide accurate analyses of Web interactions which point out typical unexpected application behaviors which are essentially caused by the uncontrolled use of the browser navigation buttons as well as the shortcomings of HTTP. Their approach however is different from ours since it is based on defining a novel Web programming language which allows one to write safe Web applications: [2] exploits type checking techniques to ensure application correctness, whereas [19] adopts a semantic approach which is based on program continuations.

In this paper, we presented a detailed navigation model which accurately formalizes the behavior of Web applications by means of rewriting logic. The proposed model allows one to specify several critical aspects of Web applications such as concurrent Web interactions, browser navigation features and Web scripts evaluations in an elegant, high-level rewrite theory. We also coupled our formal specification with LTLR, which is a linear temporal logic designed to model-check rewrite theories. The verification framework we obtained allows us to specify and verify even sophisticated properties (e.g. the multiple windows problem) which are either not expressible or difficult to express within other verification frameworks. Finally, we developed a prototypical implementation and we conducted several experiments which demonstrated the practicality of our approach.

Model checking as a tool is widely used in both academia and industry. In order to improve the scalability of our technique, as future work we plan to look at the approach of encoding the model-checking problem into SAT, and the resulting question of determining the efficiency in Maude of different encodings.

References

1. Message, R., Mycroft, A.: Controlling Control Flow in Web Applications. Electronic Notes in Theoretical Computer Science 200(3), 119–131 (2008)
2. Graunke, P., Findler, R., Krishnamurthi, S., Felleisen, M.: Modeling Web Interactions. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 238–252. Springer, Heidelberg (2003)
3. Open Web Application Security Project: Top ten security flaws, `http://www.owasp.org/index.php/OWASP_Top_Ten_Project`
4. Martí-Oliet, N., Meseguer, J.: Rewriting Logic: Roadmap and Bibliography. Theoretical Computer Science 285(2), 121–154 (2002)
5. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
6. Han, M., Hofmeister, C.: Modeling and Verification of Adaptive Navigation in Web Applications. In: 6th International Conference on Web Engineering, pp. 329–336. ACM, New York (2006)
7. Bae, K., Meseguer, J.: A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. ENTCS. Elsevier, Amsterdam (to appear)
8. Meseguer, J.: The Temporal Logic of Rewriting: A Gentle Introduction. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 354–382. Springer, Heidelberg (2008)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
10. TeReSe (ed.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
11. Alpuente, M., Ballis, D., Romero, D.: A Rewriting Logic Framework for the Specification and the Analysis of Web Applications. Technical Report DSIC-II/01/09, Technical University of Valencia (2009), `http://www.dsic.upv.es/~dromero/web-tlr.html`
12. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational Abstractions. Theoretical Computer Science 403(2-3), 239–264 (2008)
13. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
14. Alfaro, L.: Model Checking the World Wide Web. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 337–349. Springer, Heidelberg (2001)
15. Flores, S., Lucas, S., Villanueva, A.: Formal Verification of Websites. Electronic notes in Theoretical Computer Science 200(3), 103–118 (2008)
16. Haydar, M., Sahraoui, H., Petrenko, A.: Specification Patterns for Formal Web Verification. In: 8th International Conference on Web Engineering, pp. 240–246. IEEE Computer Society, Los Alamitos (2008)
17. Miao, H., Zeng, H.: Model Checking-based Verification of Web Application. In: 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 47–55. IEEE Computer Society, Los Alamitos (2007)
18. Donini, F.M., Mongiello, M., Ruta, M., Totaro, R.: A Model Checking-based Method for Verifying Web Application Design. Electronic Notes in Theoretical Computer Science 151(2), 19–32 (2006)
19. Queinnec, C.: Continuations and Web Servers. Higher-Order and Symbolic Computation 17(4), 277–295 (2004)