Alpuente Frasnedo, M.; Ballis, D.; Romero, DO. (2014). A rewriting logic approach to the formal specification and verification of web applications. Science of Computer Programming. 81:79-107. doi:10.1016/j.scico.2013.07.014.

A Rewriting Logic Approach to the Formal Specification and Verification of Web applications[☆]

María Alpuente[a], Demis Ballis[b], Daniel Romero[a]

[a] *DSIC-ELP, Universitat Politècnica de València, Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain.*
[b] *Dipartimento di Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy.*

Abstract

This paper develops a Rewriting Logic framework for the automatic specification and verification of Web applications that considers the critical aspects of concurrent Web interactions, browser navigation features (e.g., forward/backward navigation, page refresh, and new window/tab opening), and Web script evaluation. By encompassing the main features of the most popular Web scripting languages (e.g., PHP, ASP, and Java Servlets), our scripting language is powerful enough to model the dynamics of complex Web applications, where the interactions among Web servers and Web browsers are formalized through a landmark communicating protocol that abstracts HTTP. We provide a detailed characterization of browser actions 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 that is specifically designed for model-checking rewrite theories. The framework has been completely implemented in Maude, and we report on some successful experiments that we conducted using the Maude LTLR model-checker.

[☆] This work has been partially supported by the EU (FEDER) and the Spanish MEC TIN2010-21062-C02-02 project, by Generalitat Valenciana, ref. PROMETEO2011/052, and by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004. Daniel Romero is also supported by FPI-MEC grant BES-2008-004860.

Email addresses: alpuente@dsic.upv.es (María Alpuente), demis@dimi.uniud.it (Demis Ballis), dromero@dsic.upv.es (Daniel Romero)

1. Introduction

Over the past few 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 with their users primarily through the web by means of complex interfaces that 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). A web application typically consists of a series of web scripts whose execution may involve several interactions between a web browser and a web server. In a typical scenario, browser/server interactions are channeled by the HyperText Transfer Protocol (HTTP). Specifically, the browser requests the execution of a script in the server. Then the server executes the script and finally returns a response that the browser can display. This execution model —albeit very simple— hides some subtle intricacies that can result in erroneous behavior.

First, web browsers commonly support backward and forward navigation through web application stages and allow users to invoke distinct (instances of) web scripts in distinct browser windows (or tabs) that run in parallel. These 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 that is often called the *multiple windows problem*, which typically happens when a user browses a web site by using two or more browser windows. A representative instance of this problem for an online selling application is one in which a user displays two items in two different windows in an online store. After clicking to buy the item on the window that was opened first, the user frequently finds that s/he has bought the second one.

Second, clicking refresh/forward/backward browser buttons may sometimes produce error messages since these buttons were designed for navigating static web sites, while navigation through web applications may require multiple dynamic state changes. In fact, HTTP provides a stateless communication infrastructure in which application states can only be simulated on top of HTTP by means of common practice subterfuges (e.g., HTTP cookies, server-side sessions, hidden variables) that do not offer robust and reliable state implementations. As a consequence, frequent problems have occurred

in popular web sites (e.g., Orbitz, Apple, Continental Airlines, Hertz car rentals, Microsoft, and Register.com) as reported in [2].

Third, naïvely written web scripts may involve security vulnerabilities (e.g., unvalidated input errors, access control flaws, and data leaks [3]) that might represent potential sources of hazard. Also, in many cases, they produce undesired results that are difficult to debug.

Although the problems mentioned above are well known in the web community, there are a limited number of tools that provide a guarantee that the web application design is free of specific flaws by supporting the automated analysis and verification of web applications.

Our Approach

Model Checking [4] is an automated verification technique that can analyze complicated properties that are expressed in temporal or other logics, which is done by exhaustively exploring the reachable states of a program. If the property does not hold, the model checking algorithm generates a counterexample that is an execution trace leading to a state in which the property is violated. Model checking techniques have relatively small complexity compared to heavier formal verification techniques, and they offer three important benefits: they provide counterexamples that can be used to debug the system, more precise models, and effective, sound and complete verification.

One can distinguish two main camps in temporal logic and model checking [5]: the state-based approach, in which all atoms in formulas are state predicates (e.g., LTL, CTL, and CTL* [6]); and the event-based approach, where the formulas' atoms are actions or events (e.g., A-CTL [7]). At the semantic level, state based formulas are evaluated on Kripke structures, whereas action-based formulas are evaluated on labeled transition systems. Some properties can be naturally expressed in state-based logics and are difficult to express in action-based logics, whereas the opposite is the case for other properties. This means that, when the property does not fit well a given logic, one has to "cook" in a possibly complex way both the system description (as a Kripke structure or a label transitions system depending on the logic's semantics) and the property in order to model check it in the given logic. The situation is even more challenging for mixed properties such as fairness properties, where both state-based predicates and actions are involved [5]. The *Linear Temporal Logic of Rewriting* (LTLR) [5, 8] is a state/event based extension of LTL with spatial action patterns which represent rewriting events.

When used in tandem with rewriting logic (RWL) [9, 10] —a very general logical and semantic framework that is particularly suitable for formalizing highly concurrent, complex systems— for system and property specification, it has substantially more expressive power than purely state-based logics, or purely event-based logics [5].

RWL has been efficiently implemented in the high-performance language Maude [9, 11] and a rewriting-based model-checker for the linear temporal logic of rewriting has been developed in [5, 12] and integrated in the Maude formal environment. In [13], we thoroughly explored the application of RWL and Maude's LTLR model-checker to the formal specification and automatic verification of complex, real-size web applications. Specifically, we defined a RWL-based verification methodology that has been successfully implemented in the WEB-TLR system [14, 15].

Our Contribution

In this article, we advocate a model-based approach to the formal verification of web applications. Roughly speaking, we propose to first specify an abstract —albeit accurate— model of the web application of interest in RWL, and then analyze the model by using LTLR model-checking techniques that allow us to analyze and prove its properties and perform in-depth, efficient analyses of many subtle aspects related to web interaction.

The advantages of this *modus operandi* are threefold:

- The inherent complexity of real web applications can be managed effectively both in terms of the number of lines of code and in terms of the number of interacting technologies that are required in a real implementation;
- Critical aspects, as well as design flaws of the considered web application, can be detected during the earlier stages of the software development, which simplifies the debugging phase and reduces the overall cost of the development;
- Model-driven engineering methodologies can be adopted to automatically derive correct-by-construction implementations from the models that have been formally verified, as advocated in [16, 17].

This article offers an up-to-date, comprehensive, and uniform presentation of the RWL methodology as developed in [13, 14, 15]. We summarize the main contributions of the article as follows.

- We define a formal, fine-grained web application model that accurately describes web application behavior and that is suitable for the verification of the core business logic of complex, dynamic web systems. Our model is formalized within the Rewriting Logic (RWL) framework by means of a rigorous rewrite theory that i) completely specifies the interactions among a web server and multiple web browsers through a request/response protocol that supports the main features of HTTP; ii) allows us to model browser actions such as refresh, forward/backward navigation, and window/tab openings; iii) supports a scripting language that abstracts the main common features (e.g., session data manipulation, and database interactions) of the most popular web scripting languages; iv) formalizes *adaptive navigation* [18], i.e., a navigational model in which web page transitions may depend on the user's data or previous computation states of the web application.

- We show how rewrite theories that specify web application models can be model-checked using LTLR. LTLR allows us to define mixed state/event properties at a very high description level using RWL rules and hence can be smoothly integrated into our RWL framework. Our methodology allows several important classes of properties (e.g., unreachability, safety, authentication constraints, mutual exclusion, liveness, and fairness conditions) to be verified w.r.t. a *realistic* model of a web application that includes detailed browser-server protocol interactions, browser navigation capabilities, and web script evaluations. Moreover, relying on Maude's LTLR model-checker, any web application property that can be expressed in LTLR can be effectively verified in Web-TLR, and this great expressiveness is achieved without compromising performance because the LTLR implementation minimizes the extra costs involved in handling system events.

- We present and experimentally evaluate the Web-TLR system, which implements our verification methodology in Maude. In Web-TLR, web applications are expressed as Maude programs that can be formally verified by using the LTLR model-checker included in the Maude formal environment. Web-TLR is equipped with a user-friendly, graphical web interface that shields users from unnecessary information and allows them to visualize, in a stepwise manner, the erroneous navigation traces (i.e., counter-examples) delivered by the model-checker.

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 that formalizes 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 it can be used to formally verify web applications. Section 6 describes WEB-TLR, which is a software tool designed for model-checking web applications that implements our theoretical framework. In Section 7, we present a thorough experimental evaluation of WEB-TLR using a webmail system application and virtual forum application that attests the usefulness of our verification technique. The related work is discussed in Section 8, and Section 9 concludes. Maude specifications that encode the operational semantics of the web scripting language and the protocol evaluation mechanism can be found in Appendix A and Appendix B, respectively. Appendix C describes the virtual forum web application that is used in Section 7.

2. Preliminaries

We assume some basic knowledge of term rewriting [19] and Rewriting Logic [9]. We also require some familiarity with the Maude language [11] since we adopt a Maude-like notation to express equational as well as rewrite theories. Let us first recall some fundamental notions that are relevant to this work.

The static state structure as well as the dynamic behavior of a concurrent system can be formalized as a RWL specification that encodes 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 that is equipped with a partial order $<$ that models the usual subsort relation [11]. Σ, which is called the *signature*, specifies the operators and sorts that define the type structure of $\mathcal{R}$, while E is a set of (possibly conditional) equational definitions that may include algebraic axioms such as commutativity (*comm*), associativity (*assoc*), and unity (*id*) axioms. Each operator *opname* in Σ is defined along with its sort and axiom declarations using

the syntax:

$$op\ opname : s_0 \times \ldots \times s_n \rightarrow s\ [axiom\ declaration]$$

where s_i, $i = 0, \ldots, n$, and s are sorts, while axiom declaration is a (possibly empty) list of *equational attributes* (e.g. *assoc*, *comm*) that denote the algebraic laws that the operator *opname* must obey. By default, declared operators adopt the prefix notation; however, we also let the user specify infix operators, this is done by using underscores as place fillers for the input arguments. So, for instance, the declaration $op\ \_ + \_ : Int \times Int \rightarrow Int\ [assoc\ comm\ id : 0]$ defines + as a binary, infix operator that takes two integer numbers and returns an integer number. The operator + is also declared associative, commutative and its unity is the constant 0.

ii R is a set of (possibly conditional) labeled rules of the form $[l] : t \Rightarrow t'\ if\ c$ such that l is a label that uniquely identifies the rule, t, t' are terms, and c is an optional boolean term that represents the rule condition. When a rule has no condition, we simply write $[l] : t \Rightarrow t'$.

Intuitively, the sorts and operators contained in the signature Σ allow system states to be formalized as ground terms (i.e. terms not including variables) of a term algebra $\tau_{\Sigma,E}$ that is built upon Σ and E. On the other hand, the rules in R specify general patterns that are used to model state transitions and allows the dynamics of the system to be specified. More specifically, the system evolves by applying the rules of the rewrite theory R to the system states by means of *rewriting modulo* E, where E is the set of equational axioms.

Let us illustrate the two components (Σ, E) and R of a rewrite theory by means of some examples.

Example 1. *Soups (i.e.,* multisets*) of elements of sort* S *can be easily defined by the equational theory* (Σ_m, E_m) *that contains two sorts* S *and* Soup*, such that* S *is a subsort of* Soup*, together with the following operators [11]:*

```
op empty :→ Soup .   *** empty soup
op _,_ : Soup × Soup → Soup [comm assoc Id : empty] .   *** soup concatenation
```

The operator empty *specifies the empty soup (denoted by* $\emptyset$ *in the sequel), while* _,_ *is an associative and commutative, binary operator with identity element* empty *that is used to build soups of the form* $t_1, t_2, \ldots, t_n$ *where each* t_i*, with* $i = 1, \ldots, n$*, has sort* S*.*

In the following, we denote by $=_E$ the least congruence relation induced on the term algebra $\tau_{\Sigma,E}$ by the equational theory (Σ, E).

Example 2. *Consider two multisets* a, b, c *and* b, c, a *defined according to the equational theory* (Σ_m, E_m) *of Example 1. Then,* a,b,c $=_{E_m}$ b,c,a*, since the operator* _,_ *obeys the associative and commutative laws declared in* E_m*.*

In order to define rewriting modulo equational theories, we need the following auxiliary definitions. 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. Variables may appear in both equational axioms and rules. By notation $x : S$, we denote that variable x has sort S.

Given a rule $[l] : t \Rightarrow t'$ 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 l (in symbols $s_1 \xrightarrow{r}_{R/E} s_2$), if there exists a context C and a substitution σ such that $s_1 =_E C[t\sigma]$, $s_2 =_E C[t'\sigma]$, and $c\sigma$ holds (i.e., it is equivalent to *true* modulo E). When no confusion can arise, we simply write $s_1 \xrightarrow{l} s_2$ instead of $s_1 \xrightarrow{l}_{R/E} s_2$.

Note that, since $\longrightarrow_{R/E}$-reducibility is undecidable in general, a decidable variant of $\longrightarrow_{R/E}$ is used instead [20]. This variant is equivalent to $\longrightarrow_{R/E}$ for a large class of RWL specifications [21] including those presented in this paper.

A computation in $\mathcal{R}$ is a sequence of rewrites of the form $s_0 \xrightarrow{l_1} s_1 \ldots \xrightarrow{l_k} s_k$, with $l_1, \ldots, l_k$ labels identifying rules in R, $s_0, \ldots, s_k \in \tau_{\Sigma,E}$.

Example 3. *Let us assume that the equational theory* (Σ_m, E_m) *of Example 1 includes the (constant) operators*

```
op a :→ S .
op b :→ S .
```

Let s *be a variable of sort* Soup*. Then, the rewrite rule*

$$[\text{Remove-a}]: \ \mathsf{a}, \mathsf{s} \ \Rightarrow \ \mathsf{s} \ . \tag{1}$$

removes an occurrence of a *from any soup containing at least one occurrence of such a constant. Given the rewrite theory* $\mathcal{R}_m = (\Sigma_m, E_m, \{\text{Remove-a}\})$ *and the soup* $\mathsf{b},\mathsf{a},\mathsf{b},\mathsf{a}$*, a possible computation in* $\mathcal{R}_m$ *is*

$$\mathsf{b}, \mathsf{a}, \mathsf{b}, \mathsf{a} \overset{\text{Remove-a}}{\longrightarrow} \mathsf{b}, \mathsf{b}, \mathsf{a} \overset{\text{Remove-a}}{\longrightarrow} \mathsf{b}, \mathsf{b}$$

3. A Navigation Model for Web Applications

A Web *application* is a collection of related Web pages hosted by a Web server that contains a mixture of (X)HTML code and executable code (Web scripts). A Web application is accessed using a web browser which allows web pages to be navigated by clicking and following (X)HTML links.

Communication between the browser and the server is provided by the HTTP protocol, which works following a *request-response* scheme. Basically, in the *request* phase, the browser submits to the server the URL of a web page P that it wants to access, which may include 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. Note that the web page P requested by the browser can be different from the web page P' delivered by the server. This is because, in the *request* phase, the control flow is passed from the browser to the server, and therefore, the server is capable of defining which the next web page is. This behavior is known as *continuation*. In other words, continuations support a form of URL server redirection which is encoded at the script level and depends on script evaluations.

Since HTTP is a stateless protocol, web servers are provided with a session management technique that allows web application states to be defined via the notion of *session*. Roughly speaking, the session is a global store 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 visually depicted at a very abstract level by using a graph-like structure as follows. web pages

are represented by nodes that may contain a web script α to be executed. Solid arrows that are connecting two web pages model navigation links that 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 parameter values which are sent to the web server once the link is clicked. Finally, dashed arrows model web application continuations, that is, arcs that point to web pages that are automatically computed through web script executions. Conditions labeling continuations allow us to model any possible evolution of the web application of interest. web application continuations as well as adaptive navigations are dynamically computed w.r.t. the values stored in the current session (i.e., the current application state). Let us illustrate this model by means of a representative example.

Example 4. *Consider the navigation model given in Figure 1, which represents a generic webmail application that provides some typical functions such as login/logout actions, email management, system administration capabilities,* etc. *The web pages of the application are connected by navigation links (i.e., solid arrows) or continuations (i.e., dashed arrows). For example, the solid arrow between the* welcome *and the* home *page whose label is "*$\emptyset$*,*[user=x,pass=y]*" records the values of two input parameters* user *and* pass*, and defines a navigation link which is always enabled, as its associated condition is the empty condition* $\emptyset$*. The* home *page has two possible continuations* "login=ok" *and* "login=no"*. Depending on the* user *and* pass *values provided in the previous transition, only one of them is chosen. In the first case, the login succeeds and the* home *page is delivered to the browser, while in the case when the login fails, the* welcome *page is sent back to the browser.*

An example of adaptive navigation is provided by the navigation link that connects 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 that specifies the navigation model which we intuitively described in Section 3. The formulation exploits

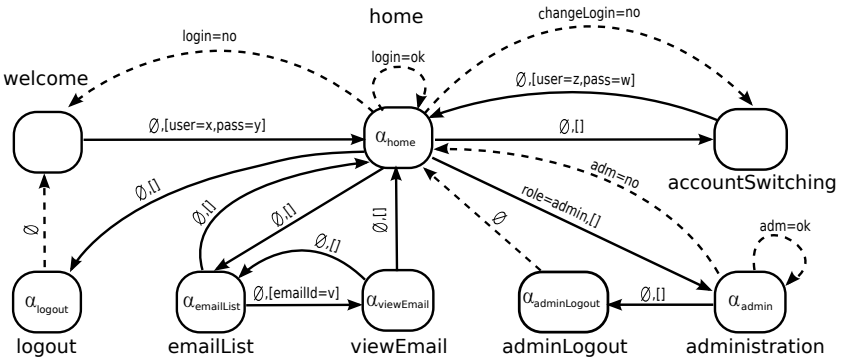

Figure 1: The navigation model of a webmail application.

the rewriting logic infrastructure to rigorously formalize the web application of interest along with its dynamic behavior. Our mechanization relies on a suitable web script evaluation engine and apropos communication protocol that abstracts HTTP. The communication protocol defines both the browser/server interaction and the most common browser navigation features. More specifically, our framework provides a complete, built-in specification of the following three components: the web scripting language, the basic structure of web applications, and the communication protocol.

4.1. 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 (getSession, setSession), accessing and updating a database (selectDB, updateDB), and capturing values contained in a query string sent by a browser (getQuery).

The scripting language is formalized by means of an equational theory (Σ_s, E_s) whose signature Σ_s specifies the language syntax and type structure. Figure 2 shows the operators of Σ_s that model the expressions and statements of the considered language. A script is a term of sort Script that models a semicolon-separated sequence of statements.

A state-based, operational semantics of the language is implemented by the equational definitions included in E_s, which allow us to specify script state changes due to script execution (e.g., changes to the memory and the database). In our framework, a script state is a tuple $(\alpha, \mathsf{m}, \mathsf{s}, \mathsf{q}, \mathsf{db})$ such that α is a script, m represents a private memory (i.e., a local memory that

```
*** Signature of the Expression operators

sorts Expression Test Qid .

op TRUE : -> Test .
op FALSE : -> Test .
op _=_  : Expression Expression -> Test .
op _!=_  : Expression Expression -> Test .
op _'+_ : Expression Expression -> Expression .
op _'*_ : Expression Expression -> Expression .
op _'._ : Expression Expression -> Expression .
op getSession : Expression -> Expression  .
op getQuery : Qid -> Expression .
op selectDB : Expression -> Expression .
op updateDB : Expression Expression -> Script .
```

```
*** Signature of the Statement operators

sorts Script .

op skip : -> Script .
op _;_ : Script Script -> Script
                    [assoc id: skip] .
op _:=_ : Qid Expression -> Script .
op if_then_else_fi : Test Script Script
                              -> Script .
op if_then_fi :  Test Script -> Script .
op while_do_od : Test Script -> Script .
op repeat_until_od : Script Test ->
                                 Script .
op setSession : Expression
                  Expression -> Script .
op clearSession : -> Script .
```

Figure 2: Syntax of the scripting language.

can only be accessed by the script α), s is a session (i.e., a global memory that can be accessed by any web script), q specifies a query string, and db models a database. We encode a script state by means of the operator

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

We represent sessions, private memories, query strings and databases by means of sets of pairs of the form id = val, where id is an identifier whose value is represented by val.

Furthermore, the set E_s contains the equational definition of the *evaluation* operator $[\![\_]\!] : \mathsf{ScriptState} \rightarrow \mathsf{ScriptState}$ that implements the operational semantics.

Intuitively, given a Web script a;as whose first statement is a and as is a sequence that represents the rest of the script, the operator $[\![\_]\!]$ takes a script state $(\mathsf{a}; \mathsf{as}, \mathsf{m}, \mathsf{s}, \mathsf{q}, \mathsf{db})$ as input and returns a new script state $(\mathsf{as}, \mathsf{m}', \mathsf{s}', \mathsf{q}, \mathsf{db}')$ in which the statement a of a;as has been completely evaluated and the private memory, the session and the database might have been changed. The operator $[\![\_]\!]$ is defined by cases over any legal statement of the language. For instance, Figure 3 shows the small-step equational definition of $[\![\_]\!]$ for the if _then _else _fi statement. This definition consists of two conditional equations. Given a statement $\mathsf{s_{if}} = \mathsf{if\ t\ then\ p1\ else\ p2\ fi}$, the first equation reduces the evaluation of $\mathsf{s_{if}}$ to the evaluation of the statement p1 whenever the guard t is fulfilled. Similarly, the second equation formalizes the case when the guard t is not fulfilled. The evaluation of the guard t is implemented

```
*** Statement evaluation: if _ then _ else _ fi

ceq [[((if t then p1 else p2 fi) ; ps , m, s, q , db)]]=
              [[(p1 ; ps , m, s, q, db )]] if (TRUE == evlEx(t, m, s, q, db) == true  .
ceq [[((if t then p1 else p2 fi) ; ps , m, s, q , db)]] =
              [[(p2 ; ps , m, s, q, db )]] if (TRUE ==  evlEx(t, m, s, q, db) =/= true  .
```

Figure 3: Equational definition of the if _ then _ else _ fi statement.

by the operator evlEx, which assigns a denotation to each legal expression.

For the sake of completeness, the full formalization of the syntax and operational semantics of our scripting language can be found in Appendix A.

4.2. The Web Application Structure

The web application structure is formalized by using an equational theory (Σ_w, E_w) such that $(\Sigma_w, E_w) \supseteq (\Sigma_s, E_s)$. The equational theory (Σ_w, E_w) includes the specific sort Soup, defined in Example 1, for representing multisets of elements.

A web application is modeled as a soup of web pages whose structure is specified by means of the following operators of (Σ_w, E_w):

$$\begin{array}{rl} \mathsf{op}\ (\_,\_,\{\_\},\{\_\}) & : (\mathsf{PageName} \times \mathsf{Script} \times \mathsf{Continuation} \times \mathsf{Navigation}) \rightarrow \mathsf{Page}\ . \\ \mathsf{op}\ (\_,\_) & : (\mathsf{Condition} \times \mathsf{PageName}) \rightarrow \mathsf{Continuation}\ . \\ \mathsf{op}\ \_,[\_] & : (\mathsf{PageName} \times \mathsf{Query}) \rightarrow \mathsf{Url}\ . \\ \mathsf{op}\ (\_,\_) & : (\mathsf{Condition} \times \mathsf{Url}) \rightarrow \mathsf{Navigation}\ . \end{array}$$

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 that identifies the web page, α is a term of sort Script that specifies the web script included in the page n, cs represents a soup of possible continuations, and ns defines a soup of navigation links that occur in the page.

Each continuation that appears in cs is a term of the form $(\mathsf{cond}, \mathsf{n}')$ stating that the computation must reach the web page n' whenever the condition cond holds. Each navigation link in ns is a term of the form $(\mathsf{cond}, \mathsf{n}', [\mathsf{q}_1, \ldots, \mathsf{q}_n])$ that specifies a link to the web page n' which can be followed using the input provided by the query string $[\mathsf{q}_1, \ldots, \mathsf{q}_n]$ if the condition

cond holds. A condition is either $\emptyset$ or a soup of the form $\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_k = \mathsf{val}_k$.

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 $(\mathsf{cond}, \mathsf{n}', [\mathsf{q}_1, \ldots, \mathsf{q}_n])$ is *enabled* in s iff $\mathsf{cond} \subseteq \mathsf{s}$. It is worth noting that any continuation/link with an empty condition is always enabled in any session s.

A web application is formally defined as a soup of terms of sort Page by means of the operator $\mathsf{op}\ \langle \_ \rangle : \mathsf{Page} \to \mathsf{WebApplication}$.

Example 5. *Consider again the Web application in Example 4. 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:

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

where we define the Web scripts that occur in wapp *in the following way:*

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

```
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

```
forbid := getSession("forbid");
if (forbid = "true") then
  setSession("login","no")
else
  login := getSession("logged") ;
  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") ;
      setSession("logged","yes")
    else
      setSession("login","no") ;
      f := getSession("failed") ;
      if (f = null) then f := 0 fi ;
      f := f + 1 ;
      setSession("failed",f) ;
      if (f = 3) then
        setSession("forbid","true")
  fi  fi  fi  fi
```

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

4.3. The Communication Protocol

We define the communication protocol by means of a rewrite theory (Σ_p, E_p, R_p), where (1) (Σ_p, E_p) is an equational theory that formalizes the web application states, and (2) R_p is a set of rewrite rules that specifies web script evaluations and request/response protocol actions.

4.3.1. The Equational Theory (Σ_p, E_p)

This theory extends the equational theory (Σ_w, E_w) (i.e., $(\Sigma_p, E_p) \supseteq (\Sigma_w, E_w)$) as follows. On the one hand, it models the entities in 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 that are generated at runtime.

More formally, (Σ_p, E_p) includes the following operators.

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

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]

A *web browser* is modeled by means of a rich data structure that supports *tabbed browsing* — that is, a browser feature that allows a user to load multiple web pages in separate tabs of a single browser window without the need to open a new session.

Specifically, a web browser is specified by a term of sort $\mathsf{Browser}$ 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 that represents the browser;
- $\mathsf{id_t}$ is an identifier that models an open browser tab of $\mathsf{id_b}$;
- n is the name of the web page that is currently displayed in the tab $\mathsf{id_t}$ of browser $\mathsf{id_b}$;
- $\mathsf{url}_1, \ldots, \mathsf{url_l}$ represent the navigation links that 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;

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

- m is the last message that the browser has sent to the server;
- h is a bidirectional list that records the history of the visited web pages;
- i is an internal counter that is used to distinguish among several response messages generated by repeated refresh actions (e.g., if a user pressed the refresh button twice, only the second refresh is displayed in the browser window).

The *web server* is formalized by using a term of the form

$$\mathsf{S}(\langle \mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{l} \rangle, \{\mathsf{BS}(\mathsf{id}_{\mathsf{b}1}, \{\mathsf{s}_1\}), .., \mathsf{BS}(\mathsf{id}_{\mathsf{bn}}, \{\mathsf{s}_\mathsf{n}\})\}, \{\mathsf{db}\}, \mathsf{fifo}_{\mathsf{req}}, \mathsf{fifo}_{\mathsf{res}})$$

where

- $\langle \mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{l} \rangle$ defines the web application that is currently in execution;
- $\mathsf{s}_\mathsf{i} = \{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_{\mathsf{m}_\mathsf{i}} = \mathsf{val}_{\mathsf{m}_\mathsf{i}}\}$ is the session of the browser $\mathsf{id}_{\mathsf{bi}}$, $\mathsf{i} = 1, \ldots, \mathsf{n}$, which is used to implement the web application state;
- $\mathsf{db} = \{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{k} = \mathsf{val}_\mathsf{k}\}$ specifies the database that is hosted by the web server and used by the application $\langle \mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{l} \rangle$;
- $\mathsf{fifo}_{\mathsf{req}}$ and $\mathsf{fifo}_{\mathsf{res}}$ are two queues of messages that both implement the FIFO queuing policy. Basically, they respectively model the request messages that still have to be processed by the server and the pending response messages that the server has not yet sent to the browsers.

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

$$\mathsf{B2S}(\mathsf{id}_\mathsf{b}, \mathsf{id}_\mathsf{t}, \mathsf{n}, [\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}], \mathsf{i})$$

represent *request* messages, that is, messages sent from the browser id_b (and tab id_t) to the server asking for the web page n with query parameters $[\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}]$. Instead, terms of the form

$$\mathsf{S2B}(\mathsf{id}_\mathsf{b}, \mathsf{id}_\mathsf{t}, \mathsf{n}, \{\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{l}\}, \{\mathsf{id}'_1 = \mathsf{val}'_1, \ldots, \mathsf{id}'_\mathsf{m} = \mathsf{val}'_\mathsf{m}\}, \mathsf{i})$$

model *response* messages, that is, messages sent from the server to the browser id_b (and tab id_t), including the computed web page n along with the navigation links $\{\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{l}\}$ that occur in n, and the current session

information[2]. The value i, which appears in request as well as in response messages, specifies a time stamp which is used to manage multiple responses that are originated by repeated refresh requests of a given web page.

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 that is modeled as a queue of messages, and sv is a server that interacts with browsers in br [3]. 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

$$\mathsf{op\ eval : WebApplication\ Session\ DB\ Message} \rightarrow \mathsf{Session\ DB\ Message}$$

whose semantics is specified by means of the equations in E_p (see Appendix B for the precise formalization of eval).

Given a web application w, a session s, a database db, and a request message $\mathsf{b2s} = \mathsf{B2S}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{n}, [\mathsf{q}], \mathsf{k})$, the operator $\mathsf{eval}(\mathsf{w}, \mathsf{s}, \mathsf{db}, \mathsf{b2s})$ generates a triple $(\mathsf{s'}, \mathsf{db'}, \mathsf{s2b})$ that contains an updated session s′, an updated database db′, and a response message $\mathsf{s2b} = \mathsf{S2B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{n'}, \{\mathsf{url_1}, \ldots, \mathsf{url_m}\}, \mathsf{s'}, \mathsf{k})$. Roughly speaking, the operator eval executes the web script, which is included in the web page n, and dynamically determines (i) which web page n′ is computed by the continuation enabled in the session s, and (ii) which links of n′ are enabled w.r.t. the current session s′.

4.3.2. The Rewrite Rule Set R_p

The term rewriting system R_p defines a collection of rewrite rules of the form $[\mathsf{label}] : \mathsf{WebState} \Rightarrow \mathsf{WebState}$ that formalize

- an abstraction of the standard request-response behavior of the HTTP protocol;
- some browser navigation features that are typically supported by web browsers (e.g., forward/backward navigation and page refresh).

[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.

[3]For the sake of simplicity, a web application state models interactions with a single web server. However, multiple web servers might be supported with little effort.

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

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

$$[\mathsf{Evl}]:\ \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\underline{\mathsf{id_b}, \{\mathsf{s}\}}), \mathsf{bs}\}, \underline{\{\mathsf{db}\}}, (\mathsf{m_{idb,idt}}, \mathsf{fifo_{req}}), \mathsf{fifo_{res}}) \Rightarrow \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\mathsf{id_b}, \underline{\{\mathsf{s}'\}}), \mathsf{bs}\}, \underline{\{\mathsf{db}'\}}, \mathsf{fifo_{req}}, (\mathsf{fifo_{res}}, \underline{\mathsf{m}'}))$$
$$\text{if } (\mathsf{s}', \mathsf{db}', \mathsf{m}') := \mathsf{eval}(\mathsf{w}, \mathsf{s}, \mathsf{db}, \mathsf{m_{idb,idt}})$$

$$[\mathsf{ResIni}]:\ \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, (\underline{\mathsf{m_{idb,idt}}}, \mathsf{fifo_{res}})) \Rightarrow \mathsf{br} \parallel (\mathsf{m}, \underline{\mathsf{m_{idb,idt}}}) \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}})$$

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

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

Figure 4: Protocol specification.

First, we give the rules that model the considered kernel version of the HTTP protocol, and then we present a rule-based formalization of the browser navigation features of interest.

Our protocol specification is shown in Figure 4 and includes five rewrite rules that specify web browser requests, web script evaluations, and web server responses.

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 $(\mathsf{np}, [\mathsf{q}])$ that appears 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 $\mathsf{m_{idb,idt}}$ that is enqueued in the channel and saved in the browser as the last message sent. The history list is updated by adding the requested link $(\mathsf{np}, [\mathsf{q}])$. Rule ReqFin simply dequeues the first request message $\mathsf{m_{idb,idt}}$ of the channel and inserts it into $\mathsf{fifo_{req}}$, which is the server queue that contains pending requests (i.e., requests not yet processed). Rule Evl consumes the first request message $\mathsf{m_{idb,idt}}$ of the queue of request messages, evaluates $\mathsf{m_{idb,idt}}$ w.r.t. the corresponding browser session $(\mathsf{id_b}, \{\mathsf{s}\})$,

and generates the response message that is enqueued in $\mathsf{fifo}_{\mathsf{res}}$, i.e., the server queue that contains the responses to be sent to the browsers. Finally, rules ResIni and ResFin implement the response phase. First, rule ResIni dequeues the response message $\mathsf{m}_{\mathsf{idb,idt}}$ from the queue $(\mathsf{m}_{\mathsf{idb,idt}}, \mathsf{fifo}_{\mathsf{res}})$ 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. Figure 5

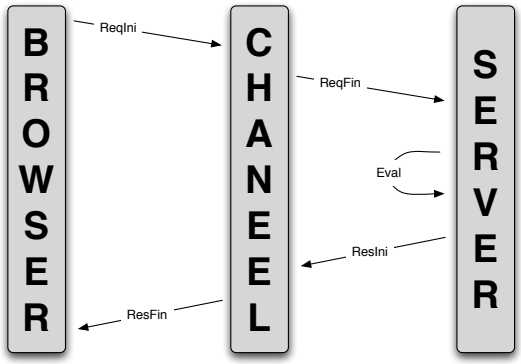

Figure 5: Request/response phase of the HTTP protocol.

provides a high-level picture of our rule-based protocol model that highlights the interactions among the three entities in play (i.e. the web browser, the channel and the web server) and their temporal sequencing during a typical request/response communication process.

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. In fact, implementation technicalities are automatically managed by the rewriting logic engine which is based on rewriting 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 that are available in a web page since they are modeled as associative and commutative soups of elements (i.e., $\mathsf{URL} < \mathsf{Soup}$), hence, a single URL can be extracted from the soup by simply applying pattern matching modulo associativity and commutativity.

Now, we are ready to formalize the browser navigation features by means of the rewrite rules shown in Figure 6.

Rules Refresh and OldMsg model the typical behavior of the refresh button of a web browser. Rule Refresh applies when a page refresh is invoked. Basically, it first increments the browser internal counter i, and then a new

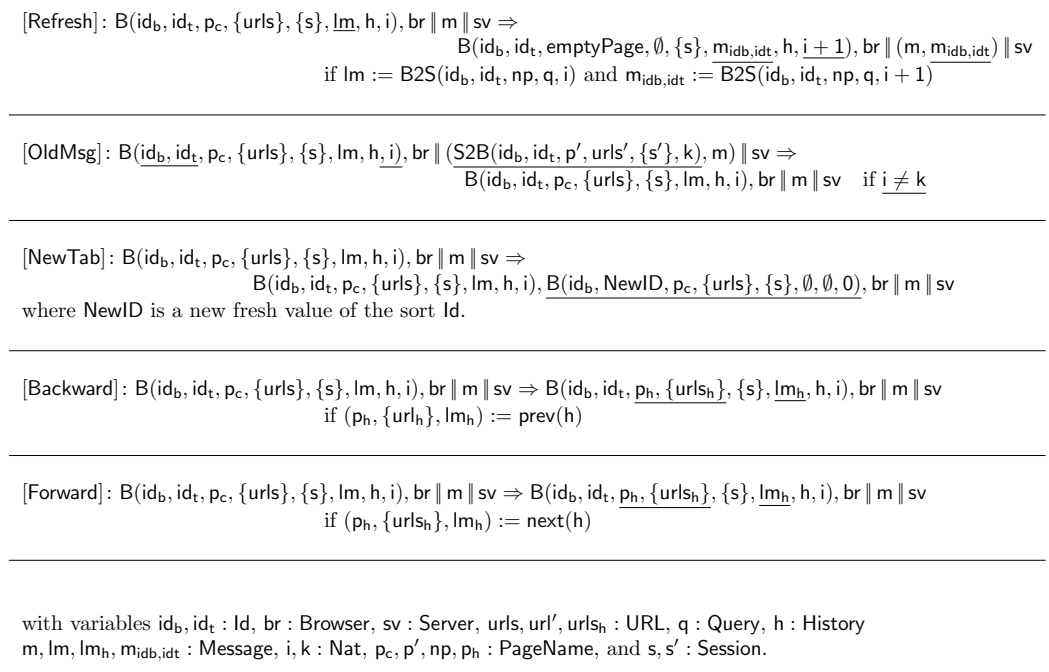

Figure 6: Specification of the browser navigation features.

version of the last request message lm, which contains the updated counter, is inserted into the channel queue. Intuitively, the internal counter of the browser keeps track of the number of repeated refresh button clicks. Rule OldMsg is used to consume all those response messages in the channel that correspond to repeated clicks of the refresh button, with the exception of the last one, which is identified by the timestamp i. Therefore, only the last page refresh will be executed.

Finally, rules NewTab, Backward and Forward are quite intuitive: an application of NewTab simply generates a new web application state that contains a new fresh tab in the soup of browsers with identifier NewID, while Backward (resp. Forward)) extracts the previous (resp. next) web page from the history list 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 number of applications of rule NewTab would produce an infinite number of web application states, each of which would represent a finite number of open tabs. Therefore, to make the analysis and verification feasible, we set

some restrictions in our prototypical implementation to limit the number of reachable states (e.g., we fixed upper bounds on the length of the history list and on the number of tabs that the user is permitted to open). An alternative approach that we plan to pursue in the future is to define *state abstractions* by means of a suitable equational theory in the style of [22]. This should 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 that can be verified by using model-checking techniques. Thus, the Linear Temporal Logic of Rewriting (LTLR)[8] can be fruitfully employed to formalize properties that are either not expressible or difficult to express by using other logical frameworks, and that we can easily verify by using an existing LTLR model-checker.

5.1. The Linear Temporal Logic of Rewriting

LTLR is a logic of the family of the Temporal Logics of Rewriting TLR* [8], which allows properties of a given rewrite theory to be specified in a simple and natural way. In the following, we provide an intuitive explanation of the main features of LTLR; for a thorough discussion, we refer to [8].

LTLR extends the traditional Linear Temporal Logic (LTL), originally introduced by Pnueli [23], with *state predicates* (SP) and *spatial action patterns* (Π). A LTLR formula w.r.t. SP and Π can be defined by means of the following BNF-like syntax:

$$\varphi ::= \delta \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \Diamond\varphi \mid \Box\varphi$$

where $\delta \in SP, p \in \Pi,$ and φ is an LTLR formula w.r.t. SP and Π

Since LTLR generalizes LTL, the modalities and semantic definitions of the temporal operators above are entirely similar to those for LTL. The symbols $\Box$ is the *always* operator, $\Diamond$ is the *eventually* operator, $\bigcirc$ is the *next* operator, and $\mathcal{U}$ is the *until* operator. LTL semantics is defined over non-terminating computations (e.g., see [23]). Informally, given an infinite computation $S = s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2, \ldots$, $\Box\varphi$ means that φ is true at any state of S. $\Diamond\varphi$ means that φ becomes true in a certain state of S. $\bigcirc\varphi$ means that

φ is true in the next state after the initial one, that is, at s_1. $\varphi_1 \,\mathcal{U}\, \varphi_2$ means that φ_1 remains true in S until φ_2 becomes true.

As mentioned above, the key new addition in LTLR w.r.t. the standard LTL framework is the semantics of state predicates and spatial actions.

State predicates. Given a system that is modeled as a rewrite theory $\mathcal{R}$, a state predicate is an expression of a specific sort *Prop* whose form is

$$\mathit{statePattern} \models \mathit{property}(a_1, \ldots, a_n) = \mathit{booleanValue}$$

where *statePattern* is a term (possibly with variables) that represents a class of system states, and $\mathit{property}(a_1, \ldots, a_n) = \mathit{booleanValue}$ is an equation for a boolean function that specifies a property of interest. Roughly speaking, a state predicate allows the condition $\mathit{property}(a_1, \ldots, a_n) = \mathit{booleanValue}$ to be checked over all the states specified by $\mathcal{R}$ that match *statePattern* (i.e., all the states that are instances of *statePattern*).

Example 6. *Let* (Σ_p, E_p, R_p) *be the rewrite theory specified in Section 4 that models the web application states as terms* $\mathsf{br}\|\mathsf{m}\|\mathsf{sv}$ *of sort* WebState. *Then, the state predicate*

$$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{page}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{m}, \mathsf{h}, \mathsf{i}), \mathsf{br}\|\mathsf{m}\|\mathsf{sv} \models \mathsf{curPage}(\mathsf{id_b}, \mathsf{page}) = \mathsf{true}$$

holds (i.e., evaluates to true*) for each state of* (Σ_p, E_p, R_p) *such that* page *is the current web page displayed in a tab of browser* $\mathsf{id_b}$.

Note that, in standard propositional LTL, state propositions are defined via atomic constants. On the contrary, LTLR supports the definition of parametric state propositions (i.e., state predicates) that allow us to formalize complex state propositions in a very concise and simple way.

Spatial action patterns. Spatial action patterns allow rewrite steps to be detected within a computation by using contexts and partial substitutions as search criteria. More precisely, given a rewrite rule $[l] : t \Rightarrow t'$ *if* c, a spatial action pattern has the general form

$$C[l(x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)]$$

where C is a context, and $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a substitution such that $x_1, \ldots, x_n$ are variables that appear in the rule l. The pattern $C[l(x_1 \mapsto$

$t_1, \ldots, x_n \mapsto t_n)]$ allows us to select the rewrite steps of the form $C[t\sigma] \xrightarrow{l} C[t'\sigma]$ that use the rule l and $\sigma \supseteq \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Formally, a spatial action pattern $C[l(x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)]$ is *true* in an infinite computation $s_0 \xrightarrow{l} s_1 \longrightarrow \ldots$ if and only if the rewrite step $s_0 \xrightarrow{l} s_1$ performs a reduction using the rule l in the given context C with a computed substitution σ that includes the bindings $x_1 \mapsto t_1, \ldots, x_n \mapsto t_n$. When the context is empty, the spatial action reduces to $[l(x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)]$ and specifies the applications of rule l where only the constraints given by the substitution have to be fulfilled.

Example 7. *Let (Σ_p, E_p, R_p) be the rewrite theory of Section 4 that specifies the proposed web application model. Then, given a computation S over (Σ_p, E_p, R_p), the spatial action pattern* $\mathsf{ReqIni}(\mathsf{id_b} \mapsto \mathsf{A})$ *is true in S, if the first rewrite step in S is an application of the rule* ReqIni *that refer to the browser with identifier* A*.*

5.2. LTLR properties for Web Applications

This section shows the main advantages of coupling LTLR with web application models that are specified via rewrite theories. First, we show the advantages of defining parametric properties in LTLR. Then, we describe how unreachability properties can be used to analyze an interesting group of properties that are critical in our setting. Finally, we discuss an example of a liveness property that successfully exploits the spatial action patterns provided by LTLR.

5.2.1. Parametric properties

LTLR is a highly parametric logic that allows complex properties to be defined in a concise way by means of state predicates and spatial action patters. As an example, consider the webmail application given in Example 4, together 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 [24]:

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

where welcomeA and forbiddenA are atomic propositions respectively asserting that (i) user A is displaying the welcome page, and (ii) login is forbidden for

user A. We assume that the welcome page includes the login script. Although the property to be modeled is conceptually rather simple, the resulting LTL formula is textually large and requires a great effort to be specified and verified. Also, the complexity of the formula would rapidly grow if a higher number of login attempts were considered[4].

By using LTLR, we can simply define a sign-in property that is parametric w.r.t. the number of login attempts as follows. We define the state predicates: (i) $\mathsf{failedAttempt}(\mathsf{id_b}, \mathsf{n})$, which holds when n failed login attempts have been tried from browser $\mathsf{id_b}$; (ii) $\mathsf{userForbidden}(\mathsf{id_b})$, which holds when the user of browser $\mathsf{id_b}$ has the login onto the system forbidden. The satisifiability of both state predicates depends on some $\mathsf{id_b}$'s session variables that are encoded in the Web state data structure. Specifically, $\mathsf{failAttempt}(\mathsf{id_b}, \mathsf{n})$ holds if the session variable failed, included in the $\mathsf{id_b}$'s browser session, is set to the value n; while $\mathsf{userForbidden}(\mathsf{id_b})$ holds when the value true is assigned to the session variable forbid.

Formally,

$$\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\underline{\mathsf{id_b}, \{\mathsf{failed{=}n}\}}), \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{f_{req}}, \mathsf{f_{res}}) \models \mathsf{failedAttempt}(\mathsf{id_b}, \mathsf{n}) = \mathsf{true}$$
$$\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\underline{\mathsf{id_b}, \{\mathsf{forbid{=}true}\}}), \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{f_{req}}, \mathsf{f_{res}}) \models \mathsf{userForbidden}(\mathsf{id_b}) = \mathsf{true}$$

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

$$\Box(\mathsf{failedAttempt}(\mathsf{A}, 3) \rightarrow \bigcirc(\Box\mathsf{userForbidden}(\mathsf{A})))$$

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 that counts the login attempts in the state predicate $\mathsf{failedAttempt}(\mathsf{A}, 3)$. Also, note that we can define state predicates (and more general LTLR formulae) that depend on web script evaluations. For instance, the predicate 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.

5.2.2. Unreachability properties using state predicates

Unreachability properties can be specified as LTLR formulae of the form $\Box \neg \langle \mathsf{State} \rangle$, where State is an unwanted state that the system should not

[4]An LTL formula for a looser authentication policy that permits 10 login attempts could be too large.

reach. By analyzing unreachability properties over the rewrite theory (Σ_p, E_p, R_p), we can detect very subtle errors such as intricate instances of the *multiple windows problem* mentioned in Section 1.

Example 8. *Consider again the webmail application of Example 4. Assume that a user may interact with the application by 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 the 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* emailList *page from both tabs.*

After applying the previous sequence of actions, you expect to see the emails that correspond to the accounts MA *and* MB *in the two open tabs. However, the webmail application of Example 4 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 testing the following LTLR unreachability formula $\Box\neg\,\mathsf{sessionClash}$ *where* sessionClash *is a state predicate defined as:*

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

Roughly speaking, the property $\Box\neg\,\mathsf{sessionClash}$ *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 sessions is out-of-date and hence inconsistent.*

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., $\Box\neg\,\mathsf{curPage}(\mathsf{A}, \mathsf{PageNotFound})$),
- *access control* (e.g., $\Box\neg\,(\mathsf{curPage}(\mathsf{A}, \mathsf{home}) \wedge \mathsf{userForbidden}(\mathsf{A}))$).

5.2.3. Liveness

Liveness properties state that something good keeps happening in the system. In our framework, we can employ mixed properties, that involve both spatial actions patterns and state predicates, to detect *good* rule applications. For example, consider the following property "*user* A *always succeeds in accessing her* home *page from the* welcome *page*"[5]. This amounts 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:

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

which can be satisfactorily verified by invoking the Maude LTLR model checker in our system, as we discuss in the following section.

6. The Web-TLR system

The verification framework described above has been implemented in the prototypical system Web-TLR, which is publicly available together with several examples in [25]. Web-TLR is the first verification engine that is based on the versatile and well-established Rewriting Logic/LTLR tandem for specifying web systems and properties. The implementation of Web-TLR consists of approximately 750 lines of Maude code and 1500 lines of Java code. The system architecture of our tool is depicted in Figure 7, and includes four main software components named *Web User Interface*, *Web Application Execution Environment*, *Verification and Trace Slicing Engine*, and *Output Module*.

6.1. Web User Interface

Web-TLR is equipped with a user-friendly, graphical web interface (GWI) written in JSP (Java Server Page) that shields the user from unnecessary information, and supports the online usage of the verification system. Through the GWI, the user can feed the system with a web application specification and a LTLR specification, and define an initial state that describes the system

[5] In this example, we assume that the browser identifier univocally identifies the user.

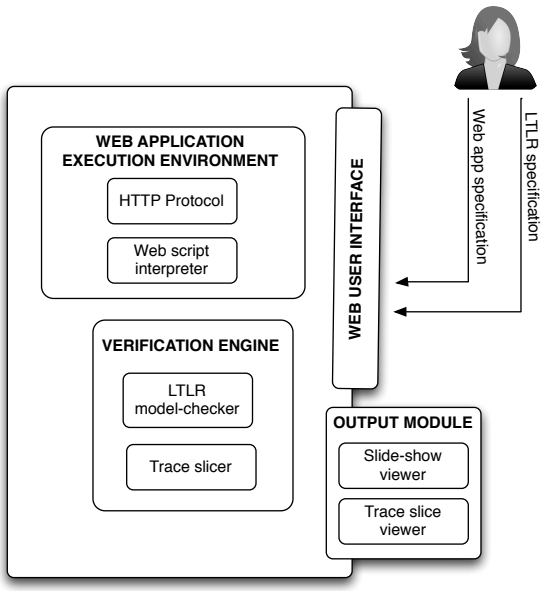

Figure 7: WEB-TLR system architecture

at the beginning of the model checking process[6]. The web application specification basically formalizes the underlying application navigation model and is defined as a rewrite theory that is expressed by means of a suitable Maude program. The LTLR specification consists of all the distinctive elements to be used in the formalization of the properties of interest to be checked. Specifically, it may include the specification of multiple state predicates, spatial action patterns, and LTRL formulae.

6.2. Web Application Execution Environment

This module encodes the Maude common infrastructure that is required to run the web application specifications. It contains the rule-based formalization of the kernel HTTP-like communication protocol and the browser actions described in Section 4.3. Also, it implements a dedicated interpreter for the evaluation of the web scripts that can be included in the web application specifications.

[6]Actually, in WEB-TLR, it is possible to define multiple initial states, which are univocally identified by their distinct labels.

6.3. Verification and Slicing Engine

In Web-TLR, a LTLR property can be automatically verified against the considered web application specification by using the Maude built-in operator `tlr check`[5] that supports model checking of rewrite theories w.r.t. a given LTLR specification. When a property is refuted, a counter-example trace is delivered that consists of the erroneous computation leading from the initial state to the state that violates the given property. This module provides a software layer that allows the web application specification and its execution environment to be interconnected with the LTLR model-checking functionality provided by the `tlr check` operator.

Furthermore, since counter-examples are often complex, textually-large objects to deal with, this module also endows Web-TLR with a novel slicing technique for rewrite sequences [26, 27, 28] that allows very large counter-examples to be debugged in an easy and effective way. Roughly speaking, the slicing facility allows the specific pieces of information that we are interested in (target data) to be traced back through a given counter-example, while irrelevant data are discarded. The slicing process drastically simplifies the counter-example trace by dropping useless data that do not influence the target data. By using this feature, the web engineer can focus on the relevant fragments of the failing application, which greatly reduces the manual debugging effort and also decreases the number of iterative verifications. A detailed description regarding the use of the slicing technique for debugging web applications can be found in [15, 29].

6.4. Output Module

This module makes available a graphical facility that favors a better inspection of the outcomes computed by Web-TLR. Specifically, when a LTRL property is refuted, the delivered counter-example is presented as an interactive slideshow (where each slide is the outcome of processing a chunk of the counter-example), which allows the erroneous trace to be visually navigated step by step. Each slide contains a graph that is generated on-the-fly and shows a part of the web application navigation model; namely, the nodes of the graph represent web pages and the edges specify web links or web script continuations. The graphical representation is combined with a detailed textual description, which is a portion of the provided counter-example that shows the current configurations of the web server and the active web browsers. Such a textual description can be simplified by invoking the built-in trace slicing facility which allows the user to automatically

extract all and only the information needed to produce the target data she intends to observe.

Example 9. *Consider the specification of the webmail application of Example 5. Let us consider two administrator users whose identifiers are* bidAdm1 *and* bidAdm2*, respectively, together with the following mutual exclusion property*

$$\Box \neg (\mathsf{curPage}(\mathsf{bidAdm1}, \mathsf{administration}) \wedge \mathsf{curPage}(\mathsf{bidAdm2}, \mathsf{administration}))$$

which states that "no two administrators can access the administration page simultaneously". In this example, note that the predicate state curPage(bidUser, administration) *holds when the user* bidUser *logs into the* administration *page. After checking the above property with* Web-TLR*, we get a outsize counterexample that proves that the property is not satisfied. Note that this is because the web scripts in the webmail example do not implement any mutual exclusion control.*

Figure 8 shows a snapshot of Web-TLR *for this example. The figure presents the considered LTLR formula and the label of the chosen initial state that was used together with a slide of the generated interactive slideshow. Also, note that the content of the graph node* administration *in the slide testifies that two users (*bidAdm1 *and* bidAdm2*) are logged into the* administration *web page, which proves that the considered mutual exclusion property has been violated.*

7. Experimental evaluation

In order to evaluate the usefulness of our approach, we have benchmarked our prototype w.r.t. two complex case studies: the webmail application model of Example 5, and the specification of a virtual forum application whose full formal specification, which includes the web scripts that encode the system behavior, is given in Appendix C. The former case study examines and formally checks several LTLR properties that aim at ascertaining the correct operation of the considered webmail system, while the latter provides an exhaustive analysis of the access control policies supported by the virtual forum. The described experiments are available at the Web-TLR website [25] and can be reproduced by accessing the system through its online web interface.

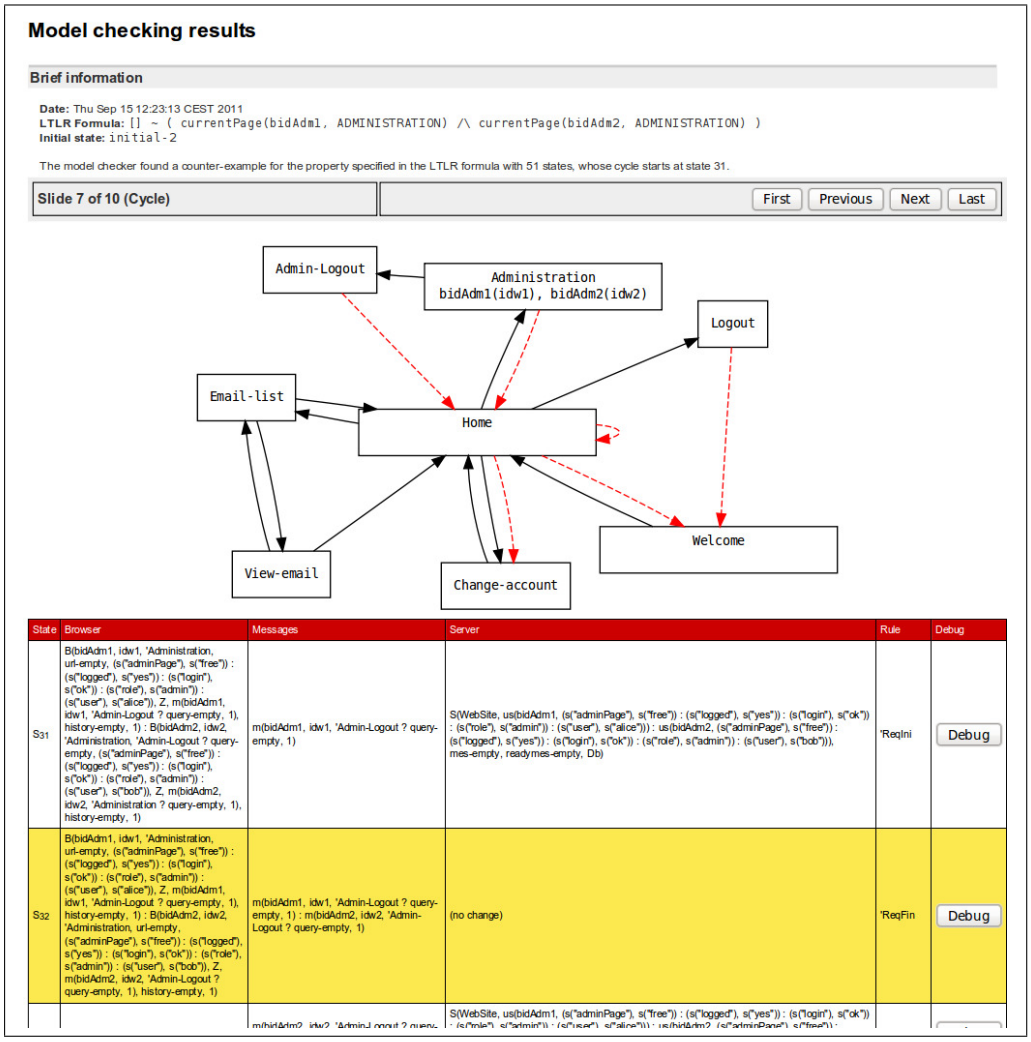

Figure 8: A snapshot of the interactive slideshow of WEB-TLR

7.1. Formal Verification of a Webmail system

Let us consider again the webmail application of Example 5 and a scenario that engages two administrator users, one registered user, and one intruder who does not have permission to access the application. Assume that these four actors are identified as bidAdm1, bidAdm2, bidRegUsr, and bidIntruder, respectively. We consider the following properties.

IncorrectLogin: Incorrect login info is allowed only 3 times, and then login is

Property	Time (ms)	Outcome	# states	Size (Kb)	Slice (Kb)
IncorrectLogin	1	Yes	–	–	–
NoTwoAdmin	4	No	51	178	[10 – 16]
Link accessibility	4	Yes	–	–	–
BannedUser	8	Yes	–	–	–
EmailLiveness	8	Yes	–	–	–
EmailFakeFairness	1	No	25	73	[6 – 8]

Table 1: Experimental results for the webmail application

forbidden (security policy)
$\Box(\mathsf{failedAttempt}(\mathsf{bidIntruder}, 3) \rightarrow \bigcirc(\Box\mathsf{userForbidden}(\mathsf{bidIntruder})))$

NoTwoAdmin: No two administrators can access the administration page simultaneously (mutual exclusion)
$\Box \neg (\mathsf{curPage}(\mathsf{bidAdm1}, \mathsf{administration}) \wedge \mathsf{curPage}(\mathsf{bidAdm2}, \mathsf{administration}))$

LinksAccess: Link accessibility
$\Box \neg \mathsf{curPage}(\mathsf{bidRegUsr}, \mathsf{PageNotFound})$

BannedUser: Banned user cannot access the home page (access control)
$\Box \neg (\mathsf{curPage}(\mathsf{bidIntruder}, \mathsf{home}) \wedge \mathsf{userForbidden}(\mathsf{bidIntruder}))$

EmailLiveness: A registered user always succeeds in accessing the home page from the welcome page (liveness)
$\Box([\mathsf{ReqIni}(\mathsf{Id_b} \mapsto \mathsf{bidRegUsr}, \mathsf{p_c} \mapsto \mathsf{welcome}, \mathsf{np} \mapsto \mathsf{home})] \rightarrow \Diamond\mathsf{curPage}(\mathsf{bidRegUsr}, \mathsf{home}))$

EmailFakeFairness: A registered user accesses her email infinitely often in any infinite computation[7] (*fake* fairness)
$\Box\Diamond\mathsf{curPage}(\mathsf{bidRegUsr}, \mathsf{email\text{-}list})$

Table 1 summarizes the results that we achieved. For each property, Table 1 shows the average execution time, which is measured in milliseconds, for a sufficiently large number of executions, and the outcome of the verification process, which is *Yes* if the property is satisfied, or *No* if the property is refuted and a counter-example is delivered. In the case when the property is falsified, we show the number of states and the size (in Kb) of the provided

[7]This property is refuted by the Maude model-checker, since a registered user may access only a finite number of times her email across an infinite computation.

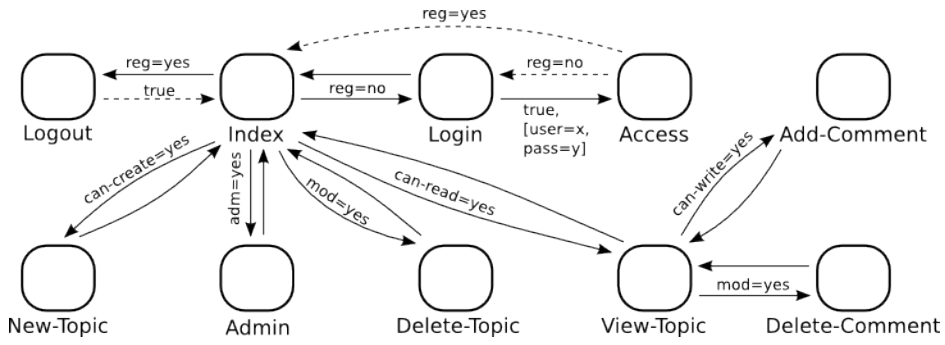

Figure 9: Virtual forum navigation model

counter-example. The figures we obtain are very encouraging and show that the execution times are reasonable in all cases. Indeed, since Web-TLR is based on the Maude model checker, the performance of our tool is ensured by the high-performance of the Maude model checker itself (see [30] for an experimental evaluation of the Maude model checker regarding time and memory consumption).

Moreover, in order to reduce the size of the counter-example traces, we are able to apply the slicing facility for rewrite traces provided within our tool. The improved results show impressive reduction rates for distinct slicing criteria, ranging from 80% to 95%, as shown in the last column of the table. Actually, sometimes the trace slices are so small that they can be manually inspected with little effort in order to explain the reasons of the detected wrong behavior. A thorough experimental evaluation of our trace slicing technique can also be found in [31].

7.2. Formal Verification of a Virtual Forum System

We consider a virtual forum web application that is equipped with some typical features such as topic and comment management. The security of the system is enforced by a role-based access control engine that includes moderator, administrator, registered user and guest roles.

A high-level graphical representation of the navigation model for the considered application is depicted in Figure 9. The application allows the web administrator to set different privilege levels for each role and comes with four predefined forum security models.

Standard Forum. Guest users, registered users and moderators can read

comments, while only registered users and moderators can write comments and create new topics.

Open Forum. Guest users, registered users and moderators can read/write comments and create new topics.

Closed Forum. Only registered users, and moderators can read/write comments and create new topics. Guest users are not allowed.

Newspaper-like Forum. Guest users, registered users and moderators can read/write comments, while only moderators can create new topics.

In all considered models, comments and topics can be removed only by moderators; furthermore, the administration page can be accessed only by administrators.

The access control policies implemented by the four different forum models have been fully verified by using the WEB-TLR system. Table 2 shows the experimental results that we obtained. Basically, we have checked 76 properties, partitioned in 19 properties for each forum model. Moreover, several initial states have been defined to rigorously specify the different forum models and the users acting on the system. Each property is a LTLR unreachability formula named $\langle R \rangle$-No-$\langle P \rangle$ that formally models the question: *Is it impossible for the user with role R to access the web page P?* By instantiating the parameters R and P with suitable roles and web pages, we were able to formalize questions about the privilege level of each role in each forum model. For instance, the property *Guest-No-AddComment* can be used to establish whether a guest user can write comments in a given forum model.

For each property, Table 2 records the time needed to model-check it as well as the outcome of the verification (*Yes/No*). In the case when the property is refuted, the table also contains the number of states and the size of the generated counter-example as well as the size of the minimum and maximum counter-example trace slice produced for that specific experiment.

We observed that the virtual forum specification correctly implements the considered access control policies in all cases but one. Specifically, we detected that moderators were not able to create new topics in the newspaper-like forum (see Property Mod-No-NewTopic in Table 2), while they should be allowed to perform this task. The wrong behavior was due to an erroneous definition of the privileges for the moderator role that was easily recognized

by inspecting the database specification encoding the newspaper-like forum model.

Finally, it is worth noting that the WEB-TLR system shows a good performance even for the more complex virtual forum case study, where checking the considered LTLR properties takes less than a few milliseconds.

8. Related Work

Web applications are complex software systems that today play a major role. Not surprisingly many different authors have already addressed the modeling and verification of these systems. A variant of the μ-calculus (called constructive μ-calculus) is proposed in [32] which allows connectivity properties to be model checked over the *static* graph-structure of a web system. However, this methodology does not support the verification of dynamic properties— e.g., reachability over sequences of web pages that are 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, [24] and [33] support the model-checking of LTL properties w.r.t. web application models that are represented as Kripke structures. In particular, [33] defines two automata, encoded in PROMELA [34], that respectively model the web application business logic and the transitions among web pages. Properties of interest are expressed as LTL formulae, and checked on the PROMELA specifications by using the SPIN model-checker [34]. Similar methodologies have been developed in [35] and [36] to verify web applications by using CTL formulae. All these model-checking approaches are based on coarse web application models that are not concerned with the communication protocol underlying the web interactions or the browser navigation features. Moreover, Section 5 shows CTL and LTL property specifications are very often textually large and hence difficult to formulate and understand. [18] presents a modeling and verification methodology that uses CTL and considers some basic adaptive navigation features. In contrast, our framework provides a complete formalization that supports more advanced adaptive navigation capabilities.

Finally, both [2] and [37] provide accurate analyses of web interactions that point out typical unexpected application behaviors that are essentially caused by the uncontrolled use of the browser navigation buttons as well as the shortcomings of HTTP. However, their approach is different from ours

Property	Forum Type	Time (ms)	Outcome	# States	Size (Kb)	Slice (Kb)
Guest-No-AddComment	Standard	12	Yes	–	–	–
	Open	19	No	30	140	14–35
	Closed	14	Yes	–	–	–
	Newspaper-like	11	No	25	116	8–32
Guest-No-DelComment	Standard	18	Yes	–	–	–
	Open	13	Yes	–	–	–
	Closed	15	Yes	–	–	–
	Newspaper-like	20	Yes	–	–	–
Guest-No-DelTopic	Standard	7	Yes	–	–	–
	Open	6	Yes	–	–	–
	Closed	10	Yes	–	–	–
	Newspaper-like	11	Yes	–	–	–
Guest-No-AdminPage	Standard	19	Yes	–	–	–
	Open	19	Yes	–	–	–
	Closed	18	Yes	–	–	–
	Newspaper-like	14	Yes	–	–	–
Guest-No-ViewTopic	Standard	18	No	30	140	14–42
	Open	13	No	25	116	12–32
	Closed	7	Yes	–	–	–
	Newspaper-like	19	No	25	184	6–46
Guest-No-NewTopic	Standard	6	Yes	–	–	–
	Open	14	No	25	116	16–35
	Closed	8	Yes	–	–	–
	Newspaper-like	8	Yes	–	–	–
Reg-No-AddComment	Standard	16	No	55	256	18–77
	Open	11	No	40	188	6–56
	Closed	9	No	40	184	9–52
	Newspaper-like	19	No	35	184	4–55
Reg-No-DelComment	Standard	15	Yes	–	–	–
	Open	15	Yes	–	–	–
	Closed	13	Yes	–	–	–
	Newspaper-like	17	Yes	–	–	–
Reg-No-DelTopic	Standard	8	Yes	–	–	–
	Open	12	Yes	–	–	–
	Closed	17	Yes	–	–	–
	Newspaper-like	16	Yes	–	–	–
Reg-No-AdminPage	Standard	12	Yes	–	–	–
	Open	13	Yes	–	–	–
	Closed	7	Yes	–	–	–
	Newspaper-like	19	Yes	–	–	–
Reg-No-ViewTopic	Standard	21	No	50	232	9–60
	Open	18	No	35	164	8–48
	Closed	15	No	35	164	16–41
	Newspaper-like	25	No	55	256	23–77
Reg-No-NewTopic	Standard	16	No	50	232	5–70
	Open	17	No	35	164	3–43
	Closed	18	No	35	164	7–43
	Newspaper-like	24	Yes	–	–	–
Mod-No-AddComment	Standard	25	No	65	300	21–78
	Open	17	No	50	232	20–70
	Closed	15	No	45	232	21–58
	Newspaper-like	13	No	60	208	8–54
Mod-No-DelComment	Standard	28	No	70	324	6–94
	Open	22	No	55	256	20–77
	Closed	24	No	55	256	5–67
	Newspaper-like	25	No	60	280	28–81
Mod-No-DelTopic	Standard	12	No	30	140	14–41
	Open	16	No	30	140	13–41
	Closed	15	No	30	140	3–35
	Newspaper-like	19	No	30	140	7–38
Mod-No-AdminPage	Standard	6	Yes	–	–	–
	Open	7	Yes	–	–	–
	Closed	13	Yes	–	–	–
	Newspaper-like	9	Yes	–	–	–
Mod-No-ViewTopic	Standard	23	No	60	280	11–73
	Open	13	No	45	208	6–58
	Closed	13	No	45	208	4–52
	Newspaper-like	17	No	50	232	14–70
Mod-No-NewTopic	Standard	21	No	60	280	11–76
	Open	14	No	45	208	6–56
	Closed	12	No	45	208	12–56
	Newspaper-like	16	Yes	–	–	–
Admin-No-AdminPage	Standard	16	No	30	140	13–35
	Open	16	No	30	140	14–36
	Closed	13	No	30	140	6–42
	Newspaper-like	19	No	30	140	8–38

Table 2: Experimental results for the virtual forum application

since it is based on defining a novel web programming language that allows safe web applications to be written: [2] exploits type-checking techniques to ensure application correctness, whereas [37] adopts a semantic approach that is based on program continuations.

9. Conclusion

In this paper, we have developed a detailed navigation model that accurately formalizes the behavior of web applications by means of rewriting logic. The proposed model allows several critical aspects of web applications such as concurrent web interactions, browser navigation features, and web scripts evaluations to be specified as an elegant, high-level rewrite theory.

We have also coupled our formal specification framework with a model-checking technique based on LTLR, which is a linear temporal logic designed to model-check rewrite theories. This marriage has proven to be particularly beneficial for the verification task, since it allows quite sophisticated properties that are very often difficult to express (or are even inexpressible) to be specified and efficiently verified within our framework.

Our methodology has been fully implemented in the Web-TLR system and tested on several complex cases, including a webmail application and a virtual forum. All the examples considered are available at the Web-TLR web site [25] and can be tested online through the graphical web interface (GWI) provided with our tool. The results obtained are very encouraging and demonstrate the practicality of our approach.

Web-TLR distinguishes itself from related tools in a number of salient aspects.

(*i*) It encompasses a rich web application core model which considers the communication protocol underlying web interactions as well as common browser navigation features.

(*ii*) It provides efficient and accurate model-checking of dynamic properties at low cost (e.g., unreachability of dynamic web pages). The verification process includes the automatic generation of diagnostic information (in the form of counter-example traces) for those properties that are refuted. This information demonstrates why a property does not hold and can be fruitfully exploited to debug faulty web applications.

(*iii*) It is equipped with a slicing facility that greatly reduces the size of counter-example traces, thus making their analysis feasible even in the case of complex, real-size web systems.

(*iv*) Counter-examples are visualized via an interactive slideshow, which allows the user to explore the model by performing forward and backward transitions. At each slide, the interface shows both a graphical representation of the web application state and the values of the more relevant variables. This on-the-fly exploration does not require the local installation of the verification system itself since it is entirely provided by the WEB-TLR's GWI.

Let us conclude by mentioning some directions for future work. Our priority is to improve the usability of WEB-TLR and we plan to do this by addressing the following tasks.

- Supplementing the system with proper support to the correct-by-construction synthesis of web applications [16, 17]. Actually, we are currently investigating a model-driven transformation technique that allow correct PHP-based web applications to be automatically generated from Maude specifications.

- Providing WEB-TLR with a model extraction facility to help web engineers verify existing web applications. The facility aims at deriving formal Maude models, which can be analyzed by our tool, from PHP-based web applications.

- Enriching the WEB-TLR user interface by implementing an editor for the specification of LTLR properties based on temporal patterns in the style of [24, 38, 39]. Basically, the idea is to define a catalogue of temporal LTLR templates (expressed via a simplified structured English grammar) that can ease the definition and reuse of LTLR properties even for those practitioners that are unfamiliar with formal method idioms.

We also intend to extend our framework in order to deal with more sophisticated web systems that support client-side scripts (defined for example by JavaScript-like languages) and that are based on web service architectures conforming to the REST (*REpresentational State Transfer*) framework [40].

Finally, in order to improve the scalability of our technique, we plan to consider the approach of encoding model-checking problems into SAT, and the related issues of determining the efficiency of solving different encodings in Maude.

Acknowledgments

We would like to thank María del Mar Gallardo for many useful comments and suggestions. We also gratefully acknowledge Javier Espert and Francisco Frechina for their helpful discussions concerning the design of Web-TLR and their valuable involvement in the implementation.

References

[1] R. Message, A. Mycroft, Controlling Control Flow in Web Applications, in: 4th Int'l Workshop on Automated Specification and Verification of Web Sites, WWV 2008, volume 200(3) of *Electronic Notes in Theoretical Computer Science*, pp. 119–131.

[2] P. Graunke, R. Findler, S. Krishnamurthi, M. Felleisen, Modeling Web Interactions, in: 12th European Symposium on Programming, ESOP 2003, volume 2618 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 238–252.

[3] Open Web Application Security Project, Top ten security flaws, 2007. Available at `http://www.owasp.org/index.php/OWASP_Top_Ten_Project`.

[4] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach, in: Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, POPL 1983, ACM Press, 1983, pp. 117–126.

[5] K. Bae, J. Meseguer, A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting, in: 9th International Workshop on Rule-Based Programming, RULE 2008, Electronic Notes in Theoretical Computer Science, Elsevier, 2008.

[6] M. Huth, M. Ryan, Logic in Computer Science, Cambridge University Press, 2004.

[7] R. D. Nicola, F. Vaandrager, Action versus State Based Logics for Transition Systems, in: LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes, Springer, 1990, pp. 407–419.

[8] J. Meseguer, The Temporal Logic of Rewriting: A Gentle Introduction, in: Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of his 65th Birthday, volume 5065, Springer, 2008, pp. 354–382.

[9] J. Meseguer, Conditional Rewriting Logic as a Unified Model of Concurrency, Theoretical Computer Science 96 (1992) 73–155.

[10] N. Martí-Oliet, J. Meseguer, Rewriting Logic: Roadmap and Bibliography, Theoretical Computer Science 285(2) (2002) 121–154.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude: A High-Performance Logical Framework, volume 4350 of *Lecture Notes in Computer Science*, Springer, 2007.

[12] K. Bae, J. Meseguer, The linear temporal logic of rewriting maude model checker, in: P. C. Ölveczky (Ed.), Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Revised Selected Papers, volume 6381 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 208–225.

[13] M. Alpuente, D. Ballis, D. Romero, Specification and Verification of Web Applications in Rewriting Logic, in: Formal Methods, Second World Congress, FM 2009, volume 5850 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 790–805.

[14] M. Alpuente, D. Ballis, J. Espert, D. Romero, Model-checking Web Applications with Web-TLR, in: 8th Int'l Symp. on Automated Technology for Verification and Analysis, ATVA 2010, volume 6252 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 341–346.

[15] M. Alpuente, D. Ballis, J. Espert, F. Frechina, D. Romero, Debugging of Web Applications with Web-TLR, in: 7th International Workshop on

Automated Specification and Verification of Web Systems, WWV 2011, volume 61 of *Electronic Proceedings in Theoretical Computer Science*, pp. 66–80.

[16] M. Bordin, T. Vardanega, Correctness by Construction for High-Integrity Real-Time Systems: A Metamodel-Driven Approach, in: 12th International Conference on Reliable Software Technologies, volume 4498 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 114–127.

[17] I. Poernomo, J. Terrell, Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq, in: 12th Int'l Conference on Formal Engineering Methods, ICFEM 2010, volume 6447 of *Lecture Notes in Computer Science*, Springer-Verlag, 2010, pp. 56–73.

[18] M. Han, C. Hofmeister, Modeling and Verification of Adaptive Navigation in Web Applications, in: 6th International Conference on Web Engineering, ICWE 2006, ACM, 2006, pp. 329–336.

[19] TeReSe (Ed.), Term Rewriting Systems, Cambridge University Press, Cambridge, UK, 2003.

[20] J. P. Jouannaud, C. Kirchner, H. Kirchner, Incremental Construction of Unification Algorithms in Equational Theories, in: Automata, Languages and Programming, 10th Colloquium, ICALP 1983, volume 154 of *Lecture Notes in Computer Science*, Springer, 1983, pp. 361–373.

[21] P. Viry, Equational Rules for Rewriting Logic, Theoretical Computer Science 285 (2002) 487–517.

[22] J. Meseguer, M. Palomino, N. Martí-Oliet, Equational abstractions, Theoretical Computer Science 403 (2008) 239–264.

[23] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer, 1992.

[24] M. Haydar, H. Sahraoui, A. Petrenko, Specification Patterns for Formal Web Verification, in: 2008 Eighth International Conference on Web Engineering, ICWE 2008, IEEE Computer Society, 2008, pp. 240–246.

[25] The Web-TLR Web site, 2012. Available at: `http://users.dsic.upv.es/grupos/elp/soft.html`.

[26] M. Alpuente, D. Ballis, J. Espert, D. Romero, Backward Trace Slicing for Rewriting Logic Theories, in: 23rd International Conference on Automated Deduction CADE 23, volume 6803 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 34–48.

[27] M. Alpuente, D. Ballis, F. Frechina, D. Romero, Backward trace slicing for conditional rewrite theories, in: Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR 18, volume 7180 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 62–76.

[28] M. Alpuente, D. Ballis, F. Frechina, D. Romero, Julienne: A Trace Slicer for Conditional Rewrite Theories, in: FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings, volume 7436 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 28–32.

[29] M. Alpuente, D. Ballis, F. Frechina, D. Romero, Using Conditional Trace Slicing for improving Maude programs, Science of Computer Programming (2012). Submitted.

[30] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL Model Checker, Electronic Notes in Theoretical Computer Science 71 (2002) 162–187.

[31] M. Alpuente, D. Ballis, J. Espert, D. Romero, Dynamic Backward Slicing of Rewriting Logic Computations, ArXiv e-prints. (2011).

[32] L. Alfaro, Model Checking the World Wide Web, in: 13th Int'l. Conference on Computer Aided Verification, CAV 2001, Paris, France, Lecture Notes in Computer Science, Springer, 2001, pp. 337–349.

[33] K. Homma, S. Izumi, K. Takahashi, A. Togashi, Modeling, Verification and Testing of Web Applications Using Model Checker, IEICE Transactions 94-D (2011) 989–999.

[34] G. J. Holzmann, The Spin Model Checker, Addison-Wesley, 2003.

[35] H. Miao, H. Zeng, Model Checking-based Verification of Web Application, in: 12th IEEE International Conference on Engineering Complex Computer Systems, ICECCS 2007, IEEE Computer Society, 2007, pp. 47–55.

[36] F. M. Donini, M. Mongiello, M. Ruta, R. Totaro, A Model Checking-based Method for Verifying Web Application Design, Electronic Notes in Theoretical Computer Science 151 (2006) 19–32.

[37] C. Queinnec, Continuations and Web Servers, Higher-Order and Symbolic Computation 17 (2004) 277–295.

[38] S. Konrad, B. H. C. Cheng, Real-time Specification Patterns, in: 27th International Conference on Software Engineering, ICSE 2005, ACM, 2005, pp. 372–381.

[39] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, Patterns in Property Specifications for Finite-State Verification, in: 1999 International Conference on Software, ICSE 1999, ACM, 1999, pp. 411–420.

[40] T. R. Fielding, R. N. Taylor, Principled Design of the Modern Web Architecture, ACM Trans. Internet Technol. 2 (2002) 115–150.

Appendix A. Formal Specification of the Operational Semantics of the Web Scripting Language

The equational theory (Σ_s, E_s), which we presented in Section 4, is formally defined by means of the following Maude specification that consists of two functional modules. The former module (called `EXPRESSION`) specifies the syntax as well as the semantics of the language *expressions*. The latter module (called `SCRIPT`) formalizes the syntax and semantics of the language *statements*. The evaluation function $[\![\_]\!]$: ScriptState $\rightarrow$ ScriptState, described in Section 4, is encoded via the operator `evlSt : ScriptState -> ScriptState` which is contained in the functional module `SCRIPT`.

```
(fmod EXPRESSION is inc MEMORY  + QUERY + SESSION + DATABASE .
  sorts Expression Test .
  subsorts Test Value Qid < Expression .

  --- Signature of the Expression operators

  op TRUE : -> Test .
  op FALSE : -> Test .
  op _=_  : Expression Expression -> Test .
  op _!=_  : Expression Expression -> Test .
  op _'+_ : Expression Expression -> Expression .
  op _'*_ : Expression Expression -> Expression .
  op _'._ : Expression Expression -> Expression .
  op getSession : Expression -> Expression  .
  op getQuery : Qid -> Expression .
  op selectDB : Expression -> Expression .
  op updateDB : Expression Expression -> Script .
  op evlEx : Expression Memory Session Query DB -> Expression  .

  --- Semantics of the Expression operators

  vars ex ex1 ex2 : Expression .
  vars m ms  : Memory .
  vars db dbs : DB .
  vars s ss : Session .
  vars q qs : Query .
  vars x y : Int .
  vars qid : Qid .
  vars v  : Value .
  vars str : String .
  vars sql : SqlDB .
  vars t : Test .

  --- Exp: value

  eq evlEx ( v , m , s , q , db ) = v .

  --- Exp: boolean conditions = and !=

  ceq evlEx ( ex1 = ex2 , m , s, q, db ) = TRUE
```

```
                        if ((evlEx(ex1, m, s, q, db)) ==  (evlEx(ex2, m, s, q, db)))  .
  ceq evlEx ( ex1 = ex2 , m , s, q, db ) = FALSE
                        if ((evlEx(ex1, m, s, q, db)) =/= (evlEx(ex2, m, s, q, db)))  .
  ceq evlEx ( ex1 != ex2 , m , s, q, db ) = FALSE
                        if ((evlEx(ex1, m, s, q, db)) ==  (evlEx(ex2, m, s, q, db)))  .
  ceq evlEx ( ex1 != ex2 , m , s, q, db ) = TRUE
                        if ((evlEx(ex1, m, s, q, db)) =/= (evlEx(ex2, m, s, q, db)))  .

  --- Exp: evaluation of private memory identifiers

  eq  evlEx ( qid , ([qid, v] : ms), s, q , db ) = v .
  ceq evlEx ( qid ,  m, s, q, db ) = null   if qid in m =/= true .

  --- Exp: arithmetic operators

  eq evlEx ( ex1 '+ ex2 , m , s, q, db ) = evlEx(ex1, m, s, q, db) + evlEx(ex2, m, s, q, db) .
  eq evlEx ( ex1 '* ex2 , m , s, q, db ) = evlEx(ex1, m, s, q, db) * evlEx(ex2, m, s, q, db) .


  --- Exp: attribute selector

  eq evlEx ( ex1 '. ex2 , m , s, q, db ) = evlEx(ex1, m, s, q, db) v+ evlEx(ex2, m, s, q, db) .


  -- Exp: getSession

  eq  evlEx ( getSession(ex) , m , s , q, db ) =  getSessionValue( s, evlEx(ex, m, s, q, db) )  .

  -- Exp: getQuery

  eq  evlEx ( getQuery(qid) , m , s , (qid '= str) : qs , db ) = s(str) .
  ceq evlEx ( getQuery(qid) , m ,  s , q , db ) = null  if qid in q =/= true .

  --- Exp: selectDB

  eq  evlEx ( selectDB(ex) , m ,  s , q, db ) = select(db,  evlEx(ex, m, s, q, db)) .

  --- Exp: null value

  eq evlEx ( ex , m , s , q , db ) = null [owise] .

endfm)

(fmod SCRIPT is inc EXPRESSION .

 --- Signature of the Statement operators

 sorts Script ScriptState .

 op skip : -> Script .
 op _;_ : Script Script -> Script [prec 61 assoc id: skip] .
 op _:=_ : Qid Expression -> Script .
 op if_then_else_fi : Test Script Script -> Script .
 op if_then_fi : Test Script -> Script .
 op while_do_od : Test Script -> Script .
 op repeat_until_od : Script Test -> Script .
 op `[_`,_`,_`,_`,_`] :  Script Memory Session Query DB  -> ScriptState .
```

```
 op setSession : Expression Expression -> Script .
 op clearSession : -> Script .
 op evlSt : ScriptState -> ScriptState .

 --- Semantics of the Statement operators

 vars ex ex1 ex2 : Expression .
 vars m ms  : Memory .
 vars db dbs : DB .
 vars s ss : Session .
 vars q qs : Query .
 vars x y : Int .
 vars qid : Qid .
 vars v  : Value .
 vars str : String .
 vars p p1 p2 ps : Script .
 vars t : Test .
 vars sql : SqlDB .

 --- Statement: skip

 eq evlSt ( [ skip , m , s , q , db ] ) = [ skip , m , s , q , db ] .

 --- Statement: assignment (:=)

 eq evlSt ( [ (qid := ex ); ps , [qid, v] : ms, s, q , db ] ) =
                evlSt ( [ ps , [qid, evlEx(ex, [qid, v] : ms, s, q , db ) ] : ms, s, q , db ] ) .
 ceq evlSt ( [ (qid := ex ); ps ,  ms, s, q , db ] ) =
                evlSt ( [ ps , [qid, evlEx(ex, ms, s, q, db ) ] : ms, s, q, db ] )
                                                                        if qid in ms =/= true .
--- Statement: if then else fi

 ceq evlSt ( [ ( if t then p1 else p2 fi ) ; ps , m, s, q , db ] ) =
                 evlSt ([ p1 ; ps , m, s, q, db ]) if (TRUE == evlEx(t, m, s, q, db)) == true  .
 ceq evlSt ( [ ( if t then p1 else p2 fi ) ; ps , m, s, q , db ] ) =
                evlSt ([ p2 ; ps , m, s, q, db ]) if (TRUE == evlEx(t, m, s, q, db)) =/= true  .

--- Statement: while do od

 ceq evlSt ( [ ( while t do p od ); ps , m, s, q , db ] ) =
                     evlSt ([ p ; while t do p od ; ps , m, s, q, db ])
                                                 if  (TRUE == evlEx(t, m, s, q, db)) ==  true .
 ceq evlSt ( [ ( while t do p od ); ps , m, s, q , db ] ) = evlSt ([  ps , m, s, q, db ])
                                                 if  (TRUE == evlEx(t, m, s, q, db)) =/= true .

--- Statement: setSession

 eq evlSt ([ ( setSession(ex1, ex2) ); ps , m, s , q , db ]) =
                evlSt ([ ps , m, setSessionValue (s , evlEx(ex1, m, s, q, db) ,
                                                    evlEx(ex2, m, s, q, db) )  , q , db ])  .

--- Statement: clearSession

  eq evlSt ([ clearSession ; ps , m, s , q , db ]) = evlSt ([ ps , m, session-empty , q , db ])  .

--- Statement: updateDB
```

```
eq evlSt ([ ( updateDB (ex1, ex2) ); ps , m, s , q , db ]) =
                              evlSt ([ ps , m, s , q , update (db, evlEx(ex1, m, s, q, db) ,
                                                                  evlEx(ex2, m, s, q, db) ) ])  .

endfm)
```

Appendix B. Formal Specification of the Evaluation Protocol Function

The protocol evaluation function eval, which we presented in Section 3, is formally specified by means of the following Maude functional module.

```
(fmod EVAL is inc WEB_MODEL  .

  vars page wapp wapps w : Page .
  vars np qid np1 np2 nextPage : Qid .
  vars q q1 : Query .
  vars sc sc1 : Script .
  vars cont conts : Continuation .
  vars nav : Navigation .
  vars ss nextS : Session .
  vars cond conds : Condition .
  vars url urls nextURLs : URL .
  vars id idw : Id .
  vars uss : UserSession .
  vars db nextDB : DB .
  vars m : Memory .
  vars idmes : Nat .

  op pageNotFound : -> Qid .
  op pageNotContinuaton : -> Qid .
  op holdContinuation : Qid Continuation Session -> Qid .
  op holdNavigation : Qid Page Session -> URL .
  op holdCont : Qid Continuation Session -> Qid .
  op whichQid : Qid Qid -> Qid .
  op getURLs : Navigation Session -> URL .
  op evalScript : Page UserSession Message DB -> ReadyMessage .

  --- Evaluation of the enabled continuations

  eq  holdContinuation(np, (cond => np) : conts, ss) = holdCont (np, (cond => np) : conts , ss) .
  ceq holdContinuation(np, conts, ss) = qid
                       if np1 :=  holdCont (np, conts , ss) /\
   qid := whichQid ( np , np1 ) [owise] .
  eq  holdCont (np, cont-empty, ss) = pageNotContinuaton .
  ceq holdCont (np, (cond => qid) : conts, ss) = qid  if ( holdCondition(cond,ss) ) == true .
  eq  holdCont (np, (cond => qid) : conts, ss) = holdCont (np, conts, ss) [owise] .
  eq whichQid ( np , pageNotContinuaton ) = np .
  eq whichQid ( np , np1 ) = np1 [owise] .


  --- Evaluation of the enabled navigations

  eq holdNavigation(np , (( np , sc , { cont } , { nav } ) : wapp ), ss  ) = getURLs (nav, ss) .
  eq holdNavigation(np ,  wapp , ss  ) = url-empty [owise] .
  eq  getURLs ( nav-empty , ss ) = url-empty .
  ceq getURLs ( ( cond -> url) : nav , ss ) = url : getURLs ( nav , ss )
                                                  if ( holdCondition(cond,ss) ) == true .
  eq  getURLs ( ( cond -> url) : nav , ss ) = getURLs ( nav , ss )  [owise] .


  --- Eval definition
```

```
  ceq eval ((( np , sc , { cont } , { nav } ) : wapps ) , us( id, ss ) : uss ,
            m( id , idw , (np ? q) , idmes ) , db)
                                      = rm( m( id, idw, nextPage, nextURLs, idmes), nextS, nextDB)
      if [sc1, m, nextS, q1, nextDB] := eval([sc, none, ss, q, db] )  /\
         nextPage := holdContinuation (np, cont, nextS)               /\
         nextURLs := holdNavigation (nextPage, (( np , sc , { cont } , { nav } ) : wapps ) , nextS)
  .

  eq eval ( wapp , us( id, ss ) : uss , m( id , idw , (np ? q) , idmes ) , db  )  =
                     rm( m( id , idw , pageNotFound , url-empty , idmes ) , ss , db ) [owise] .

endfm)
```

Appendix C. Virtual Forum Formal Specification

This WEB-TLR specification models a typical virtual forum. It allows four different privilege levels (guest, registered user, moderator and administrator), each being a superset of the preceding levels.

Some missing features are a profile page for registered users, and the capability to manage subfora. These have been omitted for the sake of simplicity, and are left as an open exercise for adventurous WEB-TLR users.

```
mod WEBAPP is inc PROTOCOL .

  --- Names of the web pages
  ops INDEX LOGIN ACCESS LOGOUT ADMIN ADDCOMMENT
      DELCOMMENT VIEWTOPIC NEWTOPIC DELTOPIC : -> Qid .

  eq INDEX = 'Index .
  eq LOGIN = 'Login .
  eq ACCESS = 'Access .
  eq LOGOUT = 'Logout .
  eq ADMIN = 'Admin .
  eq ADDCOMMENT = 'Add-Comment .
  eq DELCOMMENT = 'Delete-Comment .
  eq VIEWTOPIC = 'View-Topic .
  eq NEWTOPIC = 'New-Topic .
  eq DELTOPIC = 'Delete-Topic .

  --- INDEX page definition ---
  --- The index page lists the available topics.
  --- In addition, it allows guest users to login, logged users to
  --- log out, moderators to remove threads and administrators to
  --- enter the administration page.

  op indexPage : -> Page .
  eq indexPage = ( INDEX,
                   indexScript,
                   {cont-empty},
                   { ( (s("reg") '== s("no")) -> ( LOGIN ? query-empty ) )
                   : ( (s("reg") '== s("yes")) -> ( LOGOUT ? query-empty ) )
                   : ( (s("adm") '== s("yes")) -> ( ADMIN ? query-empty ) )
                   : ( (s("can-read") '== s("yes")) -> ( VIEWTOPIC ? ('topic '= "") ) )
                   : ( (s("can-create") '== s("yes")) -> ( NEWTOPIC ? ('topic '= "") ) )
                   : ( (s("mod") '== s("yes")) -> ( DELTOPIC ? ('topic '= "") ) ) }
                 ) .


  --- set initial session values

  op indexScript : -> Script .
  eq indexScript =
    setSession( s("adminPage"), s("free") ) ;
    --- Set default levels
    'r := getSession( s("reg") ) ;
    if ( 'r = null ) then
      setSession( s("reg"), s("no") ) ;
```

```
    setSession( s("mod"), s("no") ) ;
    setSession( s("adm"), s("no") ) ;
    setSession( s("can-create"), s("no") ) ;
    setSession( s("can-write"), s("no") ) ;
    setSession( s("can-read"), s("no") )
  fi ;

  --- Set capabilities available to all users (guests included)

  'createlvl := selectDB( s("create-level") ) ;
  'writelvl := selectDB( s("write-level") ) ;
  'readlvl := selectDB( s("read-level") ) ;
  if ( 'createlvl = s("all") ) then
    setSession( s("can-create"), s("yes") )
  fi ;
  if ( 'writelvl = s("all") ) then
    setSession( s("can-write"), s("yes") )
  fi ;
  if ( 'readlvl = s("all") ) then
    setSession( s("can-read"), s("yes") )
  fi .

--- LOGIN page definition---
--- The login page asks the user for a username and a password in
--- a form which can be summited to the access page. It can also
--- go back to the index page.

op loginPage : -> Page .
eq loginPage = ( LOGIN,
                 skip,
                 {cont-empty},
                 { ( TRUE -> ( INDEX ? query-empty ) )
                 : ( TRUE -> ( ACCESS ? ('user '= "")
                                      : ('pass '= "") ) ) }
               ) .


--- ACCESS page definition ---
--- The access page processes the login request. On success, the
--- user is redirected to the index page. Otherwise, it is
--- redirected back to the login page with an error message.

op accessPage : -> Page .
eq accessPage = ( ACCESS,
                  accessScript,
                  { ( (s("reg") '== s("yes")) => INDEX )
                  : ( (s("reg") '== s("no")) => LOGIN ) },
                  {nav-empty}
                ) .


op accessScript : -> Script .
eq accessScript =
  setSession( s("adm"), s("no") ) ;             --- no capabilities by default
  setSession( s("mod"), s("no") ) ;
  setSession( s("reg"), s("no") ) ;
  'u := getQuery('user) ;                       --- get submitted user name
```

```
  'p := getQuery('pass) ;                        --- get submitted password
  'p1 := selectDB('u) ;                          --- get the actual password
  'createlvl := selectDB( s("create-level") ) ; --- get minimum privilege levels
  'writelvl := selectDB( s("write-level") ) ;
  'readlvl := selectDB( s("read-level") ) ;
  if ( 'p = 'p1 ) then                           --- check password
    'r := selectDB( 'u '. s("-role") ) ;         --- get user role (adm, mod, reg)
    setSession( s("reg"), s("yes") ) ;           --- set user capabilities
    if ( 'createlvl = s("reg") ) then
      setSession( s("can-create"), s("yes") )
    fi ;
    if ( 'writelvl = s("reg") ) then
      setSession( s("can-write"), s("yes") )
    fi ;
    if ( 'readlvl = s("reg") ) then
      setSession( s("can-read"), s("yes") )
    fi ;
    if ( 'r = s("adm") ) then
      setSession( s("adm") , s("yes") ) ;
      setSession( s("mod") , s("yes") ) ;
      setSession( s("can-create"), s("yes") ) ;
      setSession( s("can-write"), s("yes") ) ;
      setSession( s("can-read"), s("yes") )
    else
      setSession( s("adm") , s("no") ) ;
      if ( 'r = s("mod") ) then
        setSession( s("mod"), s("yes") ) ;
        if ( 'createlvl = s("mod") ) then
          setSession( s("can-create"), s("yes") )
        fi ;
        if ( 'writelvl = s("mod") ) then
          setSession( s("can-write"), s("yes") )
        fi ;
        if ( 'readlvl = s("mod") ) then
          setSession( s("can-read"), s("yes") )
        fi
      else
        setSession( s("mod"), s("no") )
      fi
    fi
  fi
  .


--- LOGOUT page definition ---
--- The logout page processes the log out action and redirects
--- the user to the index page.

op logoutPage : -> Page .
eq logoutPage = ( LOGOUT,
                  logoutScript,
                  { ( TRUE => INDEX ) },
                  {nav-empty}
                ) .

op logoutScript : -> Script .
eq logoutScript =
```

```
  setSession( s("reg"), s("no") ) ;   --- remove all capabilities
  setSession( s("mod"), s("no") ) ;
  setSession( s("adm"), s("no") ) ;
  setSession( s("can-create"), s("no") ) ;
  setSession( s("can-write"), s("no") ) ;
  setSession( s("can-read"), s("no") )
  .

--- ADMIN page definition
--- The administration page is only available to administrators.
--- It should provide actions affecting fora and user accounts,
--- but they have made not explicit for the sake of simplicity.

op adminPage : -> Page .
eq adminPage = ( ADMIN,
                 adminScript,
                 {cont-empty},
                 { ( TRUE -> ( INDEX ? query-empty ) ) }
               ) .


op adminScript : -> Script .
eq adminScript =
  setSession( s("adminPage"), s("busy") )
  .


--- ADDCOMMENT page definition ---
--- Processes the action of adding a comment to a topic. The comment
--- dialog is supposed to be integrated with the view topic page.

op addCommentPage : -> Page .
eq addCommentPage = ( ADDCOMMENT,
                      skip,
                      { cont-empty },
                      { ( TRUE -> VIEWTOPIC  ? query-empty) }
                    ) .

--- DELCOMMENT page definition ---
--- Processes the action of deleting a comment from a topic. In a
--- practical implementation, a button in the view topic page,
--- next to each message, would trigger this action for the selected
--- message. It is only available to moderators.

op delCommentPage : -> Page .
eq delCommentPage = ( DELCOMMENT,
                      skip,
                      { cont-empty },
                      { ( TRUE -> VIEWTOPIC ? query-empty ) }
                    ) .


--- VIEWTOPIC page definition ---
--- Pages that shows the messages in a given topic

op viewTopicPage : -> Page .
eq viewTopicPage = ( VIEWTOPIC,
```

```
                    skip,
                    { cont-empty },
                    { ( TRUE -> ( INDEX ? query-empty ) )
                    : ( (s("can-write") '== s("yes")) -> ( ADDCOMMENT ? query-empty ) )
                    : ( (s("mod") '== s("yes")) -> (DELCOMMENT ? query-empty) ) }
                  ) .


  --- NEWTOPIC page definition ---
  --- Processes the action of creating a new topic. The actual dialog is
  --- supposed to be in the index page.
  op newTopicPage : -> Page .
  eq newTopicPage = ( NEWTOPIC,
                      skip,
                      { cont-empty },
                      { ( TRUE -> VIEWTOPIC ? query-empty ) }
                    ) .


  --- DELTOPIC page definition ---
  --- Processes the action of deleting a new topic. The actual button is
  --- supposed to be in the index page.

  op delTopicPage : -> Page .
  eq delTopicPage = ( DELTOPIC,
                      skip,
                      { cont-empty },
                      { ( TRUE -> INDEX ? query-empty) }
                    ) .


  --- VIRTUAL FORUM Web Application: soup containing the pages of the web application

  op wapp : -> Page .
  eq wapp =  adminPage : addCommentPage : delCommentPage
          : indexPage : loginPage : accessPage : logoutPage
          : viewTopicPage : newTopicPage : delTopicPage .

endm

mod WEBAPP-PROPERTIES is inc WEBAPP-CHECK .

  ------------------------------------------------------------------------
  ---                          D A T A B A S E                         ---
  ------------------------------------------------------------------------

  op users : -> DB .
  eq users =
    --- administrators
    (s("alfred") ; s("secretAlfred"))
    (s("alfred-role") ; s("adm"))

    (s("anna") ; s("secretAnna"))
    (s("anna-role") ; s("adm"))

    --- moderators
    (s("maude") ; s("secretMaude"))
```

```
  (s("maude-role") ; s("mod"))

  (s("marc") ; s("secretMarc"))
  (s("marc-role") ; s("mod"))

  --- registered users
  (s("robert") ; s("secretRobert"))
  (s("robert-role") ; s("reg"))

  (s("rachel") ; s("secretRachel"))
  (s("rachel-role") ; s("reg"))
  .

--- standard forum
op db1 : -> DB .
eq db1 =
  --- include user data
  users

  --- required privilege level to read comments in a topic
  (s("read-level") ; s("all"))

  --- required privilege level to write a comment in a topic
  (s("write-level") ; s("reg"))

  --- required privilege level to start a new topic
  (s("create-level") ; s("reg"))
  .

--- full-open forum
op db2 : -> DB .
eq db2 =
  --- include user data
  users

  --- required privilege level to read comments in a topic
  (s("read-level") ; s("all"))

  --- required privilege level to write a comment in a topic
  (s("write-level") ; s("all"))

  --- required privilege level to start a new topic
  (s("create-level") ; s("all"))
  .

--- closed forum
op db3 : -> DB .
eq db3 =
  --- include user data
  users

  --- required privilege level to read comments in a topic
  (s("read-level") ; s("reg"))

  --- required privilege level to write a comment in a topic
  (s("write-level") ; s("reg"))
```

```
  --- required privilege level to start a new topic
  (s("create-level") ; s("reg"))
  .

--- newspaper-like forum
op db4 : -> DB .
eq db4 =
  --- include user data
  users

  --- required privilege level to read comments in a topic
  (s("read-level") ; s("all"))

  --- required privilege level to write a comment in a topic
  (s("write-level") ; s("all"))

  --- required privilege level to start a new topic
  (s("create-level") ; s("adm"))
  .

------------------------------------------------------------------------
---                        B R O W S E R S                           ---
------------------------------------------------------------------------

--- browser IDs for each user registered user
ops bidAlfred bidAnna bidMaude bidMarc bidRobert bidRachel : -> Id .
ops bidGuido bidGreta : -> Id .

--- tab IDs for each user
ops tidAlfred tidAnna tidMaude tidMarc tidRobert tidRachel : -> Id .
ops tidGuido tidGreta : -> Id .

--- sigmas for each user
ops zAlfred zAnna zMaude zMarc zRobert zRachel zGuest : -> Sigma .
eq zAlfred = ('user / "alfred") : ('pass / "secretAlfred") .
eq zAnna   = ('user / "anna")   : ('pass / "secretAnna")   .
eq zMaude  = ('user / "maude")  : ('pass / "secretMaude")  .
eq zMarc   = ('user / "marc")   : ('pass / "secretMarc")   .
eq zRobert = ('user / "robert") : ('pass / "secretRobert") .
eq zRachel = ('user / "rachel") : ('pass / "secretRachel") .
eq zGuest  = sigma-empty .

--- browser definitions
ops brAlfred brAnna brMaude brMarc brRobert brRachel : -> Browser .
eq brAlfred = newBrowser (bidAlfred, tidAlfred, (INDEX ? query-empty), zAlfred) .
eq brAnna   = newBrowser (bidAnna,   tidAnna,   (INDEX ? query-empty), zAnna  ) .
eq brMaude  = newBrowser (bidMaude,  tidMaude,  (INDEX ? query-empty), zMaude ) .
eq brMarc   = newBrowser (bidMarc,   tidMarc,   (INDEX ? query-empty), zMarc  ) .
eq brRobert = newBrowser (bidRobert, tidRobert, (INDEX ? query-empty), zRobert) .
eq brRachel = newBrowser (bidRachel, tidRachel, (INDEX ? query-empty), zRachel) .

ops brGuido brGreta : -> Browser .
eq brGuido = newBrowser (bidGuido, tidGuido, (INDEX ? query-empty), zGuest) .
eq brGreta = newBrowser (bidGreta, tidGreta, (INDEX ? query-empty), zGuest) .
```

```
  ------------------------------------------------------------------------
  ---                          S E R V E R                               ---
  ------------------------------------------------------------------------

  --- default user session
  op uss : -> UserSession .
  eq uss = usersession-empty .

  --- server
  ops server-1 server-2 server-3 server-4 : -> Server .
  eq server-1 = S( wapp, uss, mes-empty, readymes-empty, db1 ) .
  eq server-2 = S( wapp, uss, mes-empty, readymes-empty, db2 ) .
  eq server-3 = S( wapp, uss, mes-empty, readymes-empty, db3 ) .
  eq server-4 = S( wapp, uss, mes-empty, readymes-empty, db4 ) .

endm
```