Universidad Politécnica de Valencia
E.T.S. Ingeniería Informática

Proyecto Final de Carrera

Verificación de Aplicaciones Web Dinámicas con Web-TLR

Alumno:
Javier Espert Real

Directores:
María Alpuente Frasnedo
Daniel Omar Romero

Julio 2011

Dirección del autor:

E.T.S. Ingeniería Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

Abstract

WEB-TLR is a software tool designed for model-checking Web applications that is based on rewriting logic. Web applications are expressed as rewrite theories that can be formally verified by using the Maude built-in LTLR model-checker. Whenever a property is refuted, it produces a counterexample trace that underlies the failing model checking computation. However, the analysis (or even the simple inspection) of large counterexamples may prove to be unfeasible due to the size and complexity of the traces under examination.

This work aims to improve the understandability of the counterexamples generated by WEB-TLR by developing an integrated framework for debugging Web applications that integrates a trace-slicing technique for rewriting logic theories that is particularly tailored to WEB-TLR. The verification environment is also provided with a user-friendly, graphical Web interface that shields the user from unnecessary information.

Trace slicing is a widely used technique for execution trace analysis that is effectively used in program debugging, analysis and comprehension. Our trace slicing technique allows us to systematically trace back rewrite sequences modulo equational axioms (such as associativity and commutativity) by means of an algorithm that dynamically simplifies the traces by detecting control and data dependencies, and dropping useless data that do not influence the final result. Our methodology is particularly suitable for analyzing complex, textually-large system computations such as those delivered as counter-example traces by Maude model-checkers.

The slicing facility implemented in WEB-TLR allows the user to select the pieces of information that she is interested into by means of a suitable pattern-matching language supported by wildcards. The selected information is then traced back through inverse rewrite sequences. The slicing process drastically simplifies the computation trace by dropping useless data that do not influence the final result. By using this facility, 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.

Keywords: Model checking, Web verification, Trace Slicing, Rewriting Logic, Counterexample generation, Linear Temporal Logic of Rewriting, Web debugging.

Resumen

WEB-TLR es una herramienta software basada en lógica de reescritura diseñada para la comprobación basada en modelos (*model checking*) de aplicaciones Web. Las aplicaciones Web se expresan como teorías de reescritura que pueden ser verificadas formalmente utilizando el *model checker* LTLR integrado en Maude. Cuando una propiedad es refutada, se genera la traza de un contraejemplo que subyace a la fallida comprobación del modelo. Sin embargo, el análisis (o la mera inspección) de contraejemplos de gran tamaño resulta inviable debido a las dimensiones y complejidad de las trazas bajo examen.

Este proyecto final de carrera pretende facilitar la comprensión de los contraejemplos generados por WEB-TLR mediante el desarrollo de un entorno integrado de depuración de aplicaciones Web que incorpora una técnica de rebanado de trazas (*trace slicing*) para teorías de lógica de reescritura especialmente adaptada a WEB-TLR. El entorno de verificación también es provisto de una interfaz gráfica Web amigable que oculta la información innecesaria para el usuario.

El rebanado de trazas es una técnica de análisis de trazas de ejecución ampliamente utilizada en la depuración, análisis y comprensión de programas. Nuestra técnica de rebanado nos permite obtener de manera sistemática la traza inversa de secuencias de reescritura módulo axiomas ecuacionales (como la asociatividad y la conmutatividad) mediante un algoritmo que simplifica las dinámicamente trazas mediante la detección de dependencias de control y datos, descartando los datos que no influyen en el resultado final. Nuestra metodología es especialmente adecuada para el análisis de computaciones en sistemas complejos y textualmente extensos, tales como los contraejemplos generados por el *model checker* de Maude.

La funcionalidad de rebanado implementada en WEB-TLR permite al usuario seleccionar los fragmentos de información que le interesen mediante un lenguaje de ajuste de patrones (*pattern matching*) con soporte para comodines. La información seleccionada es luego trazada hacia atrás mediante secuencias de reescritura invertidas. El proceso de rebanado simplifica drásticamente la traza de la computación al descartar datos prescindibles que no afectan al resultado final. Mediante esta funcionalidad, el ingeniero Web puede centrarse en las partes relevantes de la aplicación que falla, reduciendo en gran medida el esfuerzo de depuración y el número de iteraciones en la verificación.

Palabras clave: Model checking, Verificación Web, Rebanado de trazas, Lógica de Reescritura, Generación de Contraejemplos, Lógica Temporal de Reescritura, Depuración Web.

Resum

WEB-TLR és una ferramenta de programari basada en lògica de reescriptura dissenyada per a la comprovació basada en models WEB-TLR d'aplicacions Web. Les aplicacions Web s'expressen com a teories de reescriptura que poden ser verificades formalment utilitzant el *model checker* LTLR integrat en Maude. Quan una propietat és refutada, es genera la traça d'un contraexemple que subjau a la fallida comprovació del model. No obstant això, l'anàlisi (o la mera inspección) de contraexemples de grans mides resulta inviable a causa de les dimensions i complexitat de les traces sota examen.

Este projecte final de carrera pretén facilitar la comprensió dels contraexemples generats per WEB-TLR per mitjà del desenrotllament d'un entorn integrat de depuració d'aplicacions Web que incorpora una tècnica de llescat de traces (*trace slicing*) per a teories de lògica de reescriptura especialment adaptada a WEB-TLR. L'entorn de verificació també és proveït d'una interfície gràfica Web amigable que oculta la informació innecessària per a l'usuari.

El llescat de traces és una tècnica d'anàlisi de traces d'execució àmpliament utilitzada en la depuració, anàlisi i comprensió de programes. La nostra tècnica de llescat ens permet obtindre de manera sistemàtica la traça inversa de seqüències de reescriptura mòdul axiomes equacionals (com l'asociatividad i la conmutatividad) per mitjà d'un algoritme que simplifica les dinàmicament traces per mitjà de la detecció de dependències de control i dades, descartant les dades que no influïxen en el resultat final. La nostra metodologia és especialment adequada per a l'anàlisi de computacions en sistemes complexos i textualment extensos, com ara els contraexemples generats pel *model checker* de Maude.

La funcionalitat de llescat implementada en WEB-TLR permet a l'usuari seleccionar els fragments d'informació que li interessen per mitjà d'un llenguatge d'ajust de patrons (*pattern matching*) amb suport per a comodins. La informació seleccionada és després traçada cap arrere per mitjà de seqüències de reescriptura invertides. El procés de llescat simplifica dràsticament la traça de la computació al descartar dades prescindibles que no afecten el resultat final. Per mitjà d'esta funcionalitat, l'enginyer Web pot centrar-se en les parts rellevants de l'aplicació que falla, reduint en gran manera l'esforç de depuració i el nombre d'iteracions en la verificació.

Paraules clau: Model checking, Verificació Web, Llescat de traces, Lògica de Reescriptura, Generació de Contraexemples, Lògica Temporal de Reescriptura, Depuració Web.

Thanks

It's been two years. Or maybe five. Or a lifetime.

Who cares. It's been great.

To those of you who have given me the opportunity to turn interest into work, and work into passion. Who have offered me everything, from technical counselling to unsolicited esthetic advice. Who have made me feel at home at ELP.

To those of you who have proven to be there in the worst moments, and never fail to be in the greatest. Who put up with me, when I wouldn't. Who supported me, when I couldn't.

To those of you who have always been by my side, and for whom words are simply not enough.

Thank you.

Contents

List of Figures

1 Introduction

In recent years, the automated verification of Web applications has become a major field of research. Nowadays, a large number of corporations (including book retailers, auction sites, travel reservation services, *etc.*) interact primarily through the Web by means of Web applications that combine static content with dynamic data produced "on-the-fly" by the execution of Web scripts (e.g., Java servlets, Microsoft ASP.NET and PHP code). The inherent complexity of such highly concurrent systems has turned their verification into a challenge [1, 23, 29].

In [8], a rich and accurate navigation model that formalizes the behavior of Web applications in rewriting logic was formulated. That formulation allows one to specify critical aspects of Web applications such as concurrent Web interactions, browser navigation features (i.e., forward/backward navigation, page refreshing, and window/tab openings), and Web script evaluations by means of a concise, high-level rewrite theory. Such a formalization is particularly suitable for verification purposes since it allows in-depth analyses of several subtle aspects of Web interactions to be carried out. As shown in [8], real-size, dynamic Web applications can be efficiently model-checked using the *Linear Temporal Logic of Rewriting* (LTLR), which is a temporal logic specifically designed to model-check rewrite theories that combines all the advantages of the state-based and event-based logics, while avoiding their respective disadvantages [28].

Web-TLR is a model-checking tool written in Maude that implements the theoretical framework of [8]. Web-TLR focuses on the Web application tier (business logic, and thus handles server-side scripts; no support is given for GUI verification with Flash technology or other kinds of client-side computations). This work aims to enhance Web-TLR, equipping it with a freely accessible graphical Web interface (GWI) written in Java, which allows users to introduce and check their own specification of a Web application, together with the properties to be verified. In the case when the property is proven to be false (refuted), an online facility can be invoked that dynamically generates a counterexample (expressed as a navigation trace), which is ultimately responsible for the erroneous Web application behavior. In order to improve the understandability and usability of the system and since the textual information associated to counterexamples is usually rather large and poorly readable, the checker has been endowed with the capability to generate and display on-the-fly slideshows that allow the erroneous navigation trace to be visually reproduced step by step. This graphical facility, provides deep insight into Web application behavior and is extremely effective for debugging purposes.

Additionally, in order to improve the understandability of the counterexamples generated by Web-TLR, we have developed a complementary Web debugging facility that supports both the efficient manipulation of counterexample traces and the interactive exploration of error scenarios. This facility is based on a backward trace-slicing technique for rewriting logic theories we formalized in [5] that allows the pieces of information that we are interested to be traced back through the inversed rewrite sequence. The slicing process drastically simplifies the computation trace by dropping useless data that do not influence the final result.

We provide a convenient, handy notation for specifying the slicing criterion that is successively propagated backwards at locations selected by the user. Preliminary experiments reveal that the novel slicing facility of the extended version of Web-TLR is fast enough to enable smooth interaction and helps the users to locate the cause of errors accurately without overwhelming them with bulky information. By using the slicing facility, the Web engineer can focus on the relevant fragments of the failing application, which greatly reduces the manual debugging effort.

The results in this work have been presented in several international fora. The extended Web-TLR system including the novel GWI was published in Proc. of *8th International Symposium on Automated Technology for Verification and Analysis* (ATVA10, CORE A), held in September 2010 in Singapore [4]. The backward trace-slicing technique formalized in Chapter 5 has been accepted at the *23nd International Conference on Automated Deduction* (CADE 2011, CORE A, CSCR *impact factor* 0.93), to be held in August 2011 in Wroclaw, Poland [5]. Finally, the adaptation of the slicing algorithm and subsequent development of a Web debugging technique was presented at the *7th International Workshop on Automated Specification and Verification of Web Systems* (WWV 2011), held in June 2011 in Reykjavik, Iceland [3], published in the EPTCS. A special issue of the best papers of WWV is forthcoming.

Plan of the manuscript. In Chapter 2, we recall the essential notions regarding Term Rewriting Systems, Rewriting Logic, the Linear Temporal Logic of Rewriting, and Model Checking. In Chapter 3, we introduce a leading example that allows us to illustrate the use of rewriting logic for Web system specification, and use it to give an overview of the Web-TLR verification framework. Chapter 4 presents the enhanced Web-TLR system including the novel GWI interface. In Chapter 5, we formalize a backward trace-slicing technique for the analysis of Rewriting Logic theories. Chapter 6 describes an adaptation of the backward slicing technique to Web-TLR and develops a debugging technique based on it. Chapter 7 contains a high-level description of the design and the implementation of the improved Web-TLR system. Chapter 8 concludes. Appendix A uses a Webmail application to exemplify how to define a Web application model in WEB-TLR.

2 Preliminaries

2.1 Term Rewriting Systems

A *many-sorted signature* (Σ, S) consists of a set of sorts S and a $S^* \times S$-indexed family of sets $\Sigma = \{\Sigma_{\bar{s} \times s}\}_{(\bar{s},s) \in S^* \times S}$, which are sets of *function symbols* (or perators) with a given string of argument sorts and result sort. Given an S-sorted set $\mathcal{V} = \{\mathcal{V}_s \mid s \in S\}$ of disjoint sets of variables, $T_\Sigma(\mathcal{V})_s$ and $T_{\Sigma s}$ are the sets of terms and ground terms of sorts s, respectively. We write $T_\Sigma(\mathcal{V})$ and T_Σ for the corresponding term algebras.

An *equation* is a pair of terms of the form $s = t$, with $s, t \in T_\Sigma(\mathcal{V})_s$. In order to simplify the presentation, we often disregard sorts when no confusion can arise.

Terms are viewed as labelled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term. The empty sequence Λ denotes the root position. By $root(t)$, we denote the symbol that occurs at the root position of t. We let $\mathcal{P}os(t)$ denote the set of positions of t. By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. Positions are ordered by the prefix ordering, that is, given the positions w_1, w_2, $w_1 \leq w_2$ if there exists a position x such that $w_1.x = w_2$. is the subterm at the position u of t. $t[r]_u$ is the term t with the subterm rooted at the position u replaced by r. Given a term t, we say that t is *ground* if no variables occur in t.

A *substitution* σ is a mapping from variables to terms $\{x_1/t_1, \ldots, x_n/t_n\}$ such that $x_i\sigma = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $x\sigma = x$ for any other variable x. By ε, we denote the *empty* substitution. Given a substitution σ, the *domain* of σ is the set $Dom(\sigma) = \{x | x\sigma \neq x\}$. By $Var(t)$ (resp. $FSymbols(t)$), we denote the set of variables (resp. function symbols) occurring in the term t.

A *context* is a term $\gamma \in T_{\Sigma \cup \{\square\}}(\mathcal{V})$ with zero or more holes $\square$, and $\square \notin \Sigma$. We write $\gamma[\ ]_u$ to denote that there is a hole at position u of γ. By notation $\gamma[\ ]$, we define an arbitrary context (where the number and the positions of the holes are clarified *in situ*), while we write $\gamma[t_1, \ldots t_n]$ to denote the term obtained by filling the holes appearing in $\gamma[\ ]$ with terms $t_1, \ldots, t_n$. By notation $t^\square$, we denote the context obtained by applying the substitution $\sigma = \{x_1/\square, \ldots, x_n/\square\}$ to t, where $Var(t) = \{x_1 \ldots, x_n\}$ (i.e., $t^\square = t\sigma$).

A *term rewriting system* (TRS for short) is a pair (Σ, R), where Σ is a signature and R is a finite set of reduction (or rewrite) rules of the form $\lambda \to \rho$, $\lambda, \rho \in T_\Sigma(\mathcal{V})$, $\lambda \notin \mathcal{V}$ and $Var(\rho) \subseteq Var(\lambda)$. We often write just R instead of (Σ, R). A rewrite step is the application of a rewrite rule to an expression. A term s *rewrites* to a term t via $r \in R$, $s \overset{r}{\to}_R t$ (or $s \overset{r,\sigma}{\to}_R t$), if there exists a position q in s such that λ *matches* $s_{|q}$ via a substitution σ (in symbols, $s_{|q} = \lambda\sigma$), and t is obtained from s by replacing the subterm $s_{|q} = \lambda\sigma$ with the term $\rho\sigma$, in symbols $t = s[\rho\sigma]_q$. The rule $\lambda \to \rho$ (or equation $\lambda = \rho$) is *collapsing* if $\rho \in \mathcal{V}$; it is *left-linear* if no variable occurs in λ more than once. We denote the transitive and reflexive closure of $\to$ by $\to^*$.

2.2 Rewriting Logic

We assume some basic knowledge of term rewriting [38] and Rewriting Logic (RWL) [27]. Let us first recall some fundamental notions which are relevant to this work. The static state structure as well as the dynamic behavior of a concurrent system can be encoded into a RWL specification encoding a *rewrite theory*. More specifically, a *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$, where:

(i) (Σ, E) is an order-sorted equational theory equipped with a partial order $<$ modeling the usual subsort relation. Σ, which is called the *signature*, specifies the operators and sorts defining the type structure of $\mathcal{R}$, while E is a set of (possibly conditional) equational axioms which may include commutativity (comm), associativity (assoc) and unity (id). Intuitively, the sorts and operators contained in the signature Σ allow one to formalize system states as ground terms of a term algebra $\tau_{\Sigma,E}$ which is built upon Σ and E.

(ii) R defines a set of (possibly conditional) labeled rules of the form $(l : t \Rightarrow t'$ if $c)$ such that l is a label, t, t' are terms, and c is an optional boolean term representing the rule condition. Basically, rules in R specify general patterns modeling state transitions. In other words, R formalizes the dynamics of the considered system.

Variables may appear in both equational axioms and rules. By notation $x : S$, we denote that variable x has sort S. A *context* C is a term with a single hole, denoted by $[\,]$, which is used to indicate the location of a rewrite application. $C[t]$ is the result of placing t in the hole of C. A *substitution* σ is a finite mapping from variables to terms, $t\sigma$ is the result of applying σ to term t.

The system evolves by applying the rules of the rewrite theory to the system states by means of *rewriting modulo* E, where E is the set of equational axioms. This is accomplished by means of *pattern matching modulo* E. More precisely, given an equational theory (Σ, E), a term t and a term t', we say that t *matches* t' *modulo* E (or that t E*-matches* t') via substitution σ if there exists a context C such that $C[t\sigma] =_E t'$, where $=_E$ is the congruence relation induced by the equational theory

(Σ, E). Hence, given a rule $r = (l : t \Rightarrow t' \text{ if } c)$, and two ground terms s_1 and s_2 denoting two system states, we say that s_1 *rewrites* to s_2 modulo E via r (in symbols $s_1 \xrightarrow{r} s_2$), if there exists a substitution σ such that s_1 E-matches t via σ, $s_2 = C[t'\sigma]$ and $c\sigma$ holds (i.e. it is equal to *true* modulo E). A computation over $\mathcal{R}$ is a sequence of rewrites of the form $s_0 \xrightarrow{r_1} s_1 \ldots \xrightarrow{r_k} s_k$, with $r_1, \ldots, r_k \in R$, $s_0, \ldots, s_k \in \tau_{\Sigma,E}$.

2.3 Linear Temporal Logic of Rewriting (LTLR)

LTLR is a sublogic of the family of the Temporal Logics of Rewriting TLR* [28], which allows one to specify properties of a given rewrite theory in a simple and natural way. In particular, we chose the "tandem" $LTLR/(\Sigma_p, E_p, R_p)$.

LTLR extends the traditional Linear Temporal Logic (LTL) with *state predicates* (SP) and *spatial action patterns* (Π). As explained in [9], state predicates are predicates that are locally evaluated on the states of the system. Spatial actions are the action atoms of TLR*. They generalize one-step proof terms, which can be thought of as ground-instantiated spatial actions. Spatial actions describe patterns, which in general specify not just a single one-step proof term, but a possibly infinite set of such proof terms.

A LTLR formulae w.r.t. SP and Π can be defined by means of the following BNF-like syntax:

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

Since LTLR generalizes LTL, the modalities and semantic definitions are entirely similar to those for LTL (see, e.g., [25]). The key new addition is the semantics of spatial actions; the relation $R, (\pi, \gamma) \models \delta$ holds if and only if the proof term $\gamma(0)$ of a current computation is an instance of a spatial action pattern δ.

In Section 3.3, we use a running example to give an intuitive introduction to state predicates and spatial actions. An in-depth study on LTLR can be found in [28].

2.4 Model Checking

Model Checking (MC) is an automatic technique for verifying finite state concurrent systems. This technique has a number of advantages over traditional approaches to this problem that are based on simulation, testing, and deductive reasoning.

Model checking is a powerful and efficient method that has been used to find flaws in hardware designs, business processes, object-oriented software, and hypermedia applications. One remaining major obstacle to a broader application of model checking is its limited usability for non-experts. In the case of specification violation, it requires much effort and insight to determine the root cause of errors from the counterexamples generated by model checkers [41]. In this work, we develop novel techniques to deal with this problem in the context of Web verification.

Maude [16] features built-in support for the rewriting-based LTL model checking of any computable rewrite theory. This facility is extended in [9], where a model checker for LTLR, a subset of the temporal logic of rewriting TLR*, is presented. The semantics of TLR* is given in terms of rewrite theories, so that the concurrent systems on which the LTLR properties are model checked can be specified at a very high level with rewrite rules. This LTLR model checker is the foundation of the Web-TLR verification back-end.

An thorough study of model checking can be found in [24].

3
An Overview of Web-TLR

Web-TLR is a Web verification engine that is based on the well-established *Rewriting Logic–Maude/LTLR* tandem for Web system specification and model-checking. In Web-TLR, Web applications are expressed as rewrite theories that can be formally verified by using the Maude built-in LTLR model checker. Whenever a property is refuted, the tool delivers a counterexample trace that reveals an undesired, erroneous navigation sequence. In this chapter, we use a running example to briefly recall the main features of the Web verification framework proposed in [8], which are essential for understanding this work. The original version of Web-TLR, which predates this work and is still preserved as the back-end of the newer versions, is assumed.

3.1 A Running Example: the Electronic Forum Application

Throughout this work, a Web application is thought of as a collection of related Web pages that are hosted by a Web server and contain a mixture of (X)HTML code and executable code (Web scripts), and links to other Web pages. A Web application is accessed over a network such as the Internet by using a Web browser that allows Web pages to be navigated by clicking and following links. Interactions between Web browsers and the Web server are driven by the HTTP protocol.

As a running example that allows us to illustrate the capabilities of our tool, let us introduce a Web application that implements an electronic forum. The electronic forum is a rather complex Web system that is equipped with a number of common features, such as user registration, role-based access control including moderator and administrator roles, and topic and comment management.

The navigation model (i.e., the intended semantics) of such an application can be specified by means of the graph-like structure given in Figure 3.1. Web pages are modeled as graph nodes. Each navigation link l is specified by a solid arrow that is labeled by a condition c and a query string q. Link l is enabled whenever c evaluates to *true*, while q represents the input parameters that are sent to the Web server once the link is clicked. For example, the navigation link that connects the Login

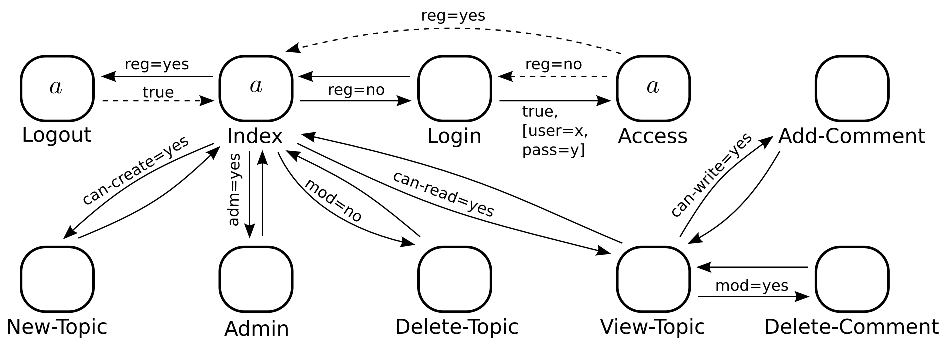


Figure 3.1: The navigation model of an Electronic Forum

and Access Web pages is always enabled and requires two input parameters (user and pass). The dashed arrows model Web application continuations, that is, arrows pointing to Web pages that are automatically computed by Web script executions. Each dashed arrow is labeled by a condition, which is used to select the continuation at runtime. For example, the Access Web page has got two possible continuations (dashed arrows) whose labels are reg=yes and reg=no, respectively. The former continuation specifies that the login attempt succeeds, and, thus, the Index Web page is delivered to the browser; in the latter case, the login fails and the Login page is sent back to the browser.

3.2 Web Application Specification

In our setting, Web applications are specified by means of a rewrite theory, which accurately formalizes the entities in play (e.g., Web server, Web browsers, Web scripts, Web pages, messages) by means of a rich Web state data structure that can be interpreted as a snapshot of the system that captures the current configurations of the active browsers (i.e., the browsers currently using the Web application), together with the server and the channel through which the browsers and the server communicate via message-passing. A part of the Maude specification that formalizes the internal representation of a generic Web state is given in Figure 3.2 (see [8] for a complete specification). Formally, a Web state WebState is a triple [Browsers] [Messages] [Server] where Browsers, Messages, and Server are encoded by using suitable Maude constructors. For instance, Browsers is a multiset of active browsers that is built using the associative and commutative constructor _;_. Each active browser is in turn formalized by a constructor term of the form

$$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{page}, \mathsf{urls}, \mathsf{session}, \mathsf{sigma}, \mathsf{lm}, \mathsf{h}, \mathsf{i})$$

where $\mathsf{id_b}$ is an identifier representing the browser; $\mathsf{id_t}$ is an identifier modeling an open windows/tab of browser $\mathsf{id_b}$; page is the name of the Web page that is currently displayed on the Web browser; urls represents the list of navigation links that appear in the Web page n; $\mathsf{session}$ is a list of pairs of the form (n, v) that is used to encode the last session that the server has sent to the browser, where each n represents a server-side variable whose value is v; sigma contains the information that is needed to automatically fill in the forms in the Web pages; lm is the last message sent to the server; h is a bidirectional list that records the history of the visited Web pages; and i is an internal counter used to distinguish among several response messages generated by repeated refresh actions (e.g., if a user pressed the refresh button twice, only the second refresh is displayed in the browser window).

A formal description of the Web pages is encoded in the Server data structure, together with the Web application's scripts. It is worth noting that our scripting language includes the main features of the most popular Web programming languages such as built-in primitives for reading/writing session data ($\mathsf{getSession}$, $\mathsf{setSession}$), accessing and updating data bases ($\mathsf{selectDB}$, $\mathsf{updateDB}$), capturing values contained in the query strings sent by browsers ($\mathsf{getQuery}$), *etc.*

The Web application behaviour is formalized by using labeled rewrite rules of the form $\mathsf{label} : \mathsf{WebState} \Rightarrow \mathsf{WebState}$ that model the application's navigation model through suitable state transitions. For instance, the rule Evl shown below consumes the first request message $\mathsf{m}_{\mathsf{idb,idt}}$ of the queue $\mathsf{fifo}_{\mathsf{req}}$, evaluates the message w.r.t. the corresponding browser session $(\mathsf{id}_{\mathsf{b}}, \{\mathsf{s}\})$, and generates the response message that is stored in the queue $\mathsf{fifo}_{\mathsf{res}}$, that is, the server queue that contains the responses to be sent to the browsers.

$$\begin{aligned}\mathsf{Evl}:\ & [\mathsf{br}][\mathsf{m}][\mathsf{S}(\mathsf{w}, \{\mathsf{BS}\underline{(\mathsf{id}_{\mathsf{b}}, \{\mathsf{s}\})}, \mathsf{bs}\}, \underline{\{\mathsf{db}\}, (\mathsf{m}_{\mathsf{idb,idt}}}, \mathsf{fifo}_{\mathsf{req}}), \mathsf{fifo}_{\mathsf{res}})] \Rightarrow \\ & \qquad [\mathsf{br}][\mathsf{m}][\mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\mathsf{id}_{\mathsf{b}}, \underline{\{\mathsf{s}'\}}), \mathsf{bs}\}, \underline{\{\mathsf{db}'\}}, \mathsf{fifo}_{\mathsf{req}}, (\mathsf{fifo}_{\mathsf{res}}, \underline{\mathsf{m}'}))] \\ \text{where } & (\mathsf{s}', \mathsf{db}', \mathsf{m}') = \mathsf{eval}(\mathsf{w}, \mathsf{s}, \mathsf{db}, \mathsf{m}_{\mathsf{idb,idt}})\end{aligned}$$

3.3 Model-checking Web Applications

Formal properties of the Web application can be specified by means of the *Linear Temporal Logic of Rewriting* (LTLR), which is a temporal logic that extends the traditional Linear Temporal Logic (LTL) with *state predicates* [28], i.e, atomic predicates that are locally evaluated on the states of the system. Let us see some examples. Consider again the electronic forum example of Section 3.1 along with the Maude code in Figure 3.2 that describes our Web state structure. We can define the state predicate $\mathsf{curPage}(\mathsf{id}_{\mathsf{b}}, \mathsf{page})$ by means of a boolean-value function as follows,

$$[\mathsf{B}(\underline{\mathsf{id}_{\mathsf{b}}}, \mathsf{id}_{\mathsf{t}}, \underline{\mathsf{page}}, \mathsf{urls}, \mathsf{session}, \mathsf{sigma}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br}][\mathsf{m}][\mathsf{sv}] \models \mathsf{curPage}(\mathsf{id}_{\mathsf{b}}, \mathsf{page}) = \mathsf{true}$$

which holds (i.e., evaluates to true) for any Web state such that page is the current Web page displayed in the browser with identifier id_{b}.

By defining elementary state predicates, we can build more complex LTLR formulas that express mixed properties containing dependencies among states, actions, and time. These properties intrinsically involve both action-based and state-based aspects that are either not expressible or are difficult to express in other temporal logic frameworks. For example, consider the administration Web page Admin of the electronic forum application. Let us consider two administrator users whose identifiers are bibAlfred and bidAnna, respectively. Then, the mutual exclusion property *"no two administrators can access the administration page simultaneously"* can be defined as follows.

$$\Box\neg(\mathsf{curPage}(\mathsf{bibAlfred},\mathsf{Admin}) \wedge \mathsf{curPage}(\mathsf{bibAnna},\mathsf{Admin})) \tag{3.1}$$

Any given LTLR property can be automatically checked by using the built-in LTLR model-checker [28]. If the property of interest is not satisfied, a counter-example that consists of the erroneous trace is returned. This trace is expressed as a sequence of rewrite steps that leads from the initial state to the state that violates the property. Unfortunately, the analysis (or even the simple inspection) of these traces may be unfeasible because of the size and complexity of the traces under examination. Typical counter-example traces in Web-TLR consist in a sequence of around 100 states, each of which contains more than 5.000 characters. As an example, one of the Web states that corresponds to the example given in Section 6.4 is shown in Figure 3.4, which demonstrates that the manual analysis of counterexample traces is generally impracticable for debugging purposes.

```
op `[_`]`[_`]`[_`] :
  Browser Message Server -> WebState [ctor] .

op B :
  Id Id Qid URL Session Sigma Message History Nat
  -> Browser [ctor] .
op br-empty : -> Browser [ctor] .
op _:_ : Browser Browser -> Browser
  [ctor assoc comm id:br-empty] .

--- message to server
op B2S : Id Id URL -> Message [ctor] .
--- message to browser
op S2B : Id Id Qid URL Session -> Message [ctor] .
op mes-empty : -> Message [ctor] .
op _:_ : Message Message -> Message
  [ctor assoc comm id:mes-empty] .

op S : Page UserSession DB -> Server [ctor] .

op `(_`,_`,`{_`}`,`{_`}`) :
  Qid Script Continuation Navigation -> Page [ctor] .
op page-empty : -> Page .
op _:_ : Page Page -> Page
  [ctor assoc comm id:page-empty] .
```

Figure 3.2: Maude representation of a Web state.

Formal description of the electronic forum Web pages:

$$\begin{array}{rl}
\mathsf{P_{Index}} = & (\mathsf{Index}, \alpha_{\mathsf{index}}, \{\emptyset\}, \{(\mathsf{reg} = \mathsf{no}) \rightarrow (\mathsf{Login?}[\emptyset]) : (\mathsf{reg} = \mathsf{yes}) \rightarrow (\mathsf{Logout?}[\emptyset]) : (\mathsf{adm} = \mathsf{yes}) \rightarrow (\mathsf{Admin?}[\emptyset]) \\
 & : (\mathsf{can\text{-}read} = \mathsf{yes}) \rightarrow (\mathsf{View\text{-}Topic?}[\mathsf{topic}]) : (\mathsf{can\text{-}create} = \mathsf{yes}) \rightarrow (\mathsf{New\text{-}Topic?}[\mathsf{topic}])) \\
 & : (\mathsf{mod} = \mathsf{yes}) \rightarrow (\mathsf{Del\text{-}Topic?}[\mathsf{topic}])\}) \\
\mathsf{P_{Login}} = & (\mathsf{Login}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow (\mathsf{Index?}[\emptyset])) : (\emptyset \rightarrow (\mathsf{Access?}[\mathsf{user}, \mathsf{pass}]))\}) \\
\mathsf{P_{Access}} = & (\mathsf{Access}, \alpha_{\mathsf{accessScript}}, \{((\mathsf{reg} = \mathsf{yes}) \Rightarrow \mathsf{Index}) : ((\mathsf{reg} = \mathsf{no}) \Rightarrow \mathsf{Login})\}, \{\emptyset\}) \\
\mathsf{P_{Logout}} = & (\mathsf{Logout}, \alpha_{\mathsf{logout}}, \{(\emptyset \Rightarrow \mathsf{Index})\}, \{\emptyset\}) \\
\mathsf{P_{Admin}} = & (\mathsf{Admin}, \alpha_{\mathsf{admin}}, \{\emptyset\}, \{(\emptyset \rightarrow (\mathsf{Index?}[\emptyset]))\}) \\
\mathsf{P_{AddComment}} = & (\mathsf{AddComment}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{ViewTopic?}[\emptyset])\}) \\
\mathsf{P_{DelComment}} = & (\mathsf{DelComment}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{ViewTopic?}[\emptyset])\}) \\
\mathsf{P_{ViewTopic}} = & (\mathsf{ViewTopic}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow (\mathsf{Index?}[\emptyset])) : ((\mathsf{can\text{-}write} = \mathsf{yes}) \rightarrow (\mathsf{AddComment?}[\emptyset])) \\
 & : ((\mathsf{mod} = \mathsf{yes}) \rightarrow (\mathsf{DelComment?}[\emptyset]))\}) \\
\mathsf{P_{NewTopic}} = & (\mathsf{NewTopic}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{ViewTopic?}[\emptyset])\}) \\
\mathsf{P_{DelTopic}} = & (\mathsf{DelTopic}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{Index?}[\emptyset])\})
\end{array}$$

Electronic forum Web scripts:

$\boxed{\alpha_{\mathsf{access}}}$

```
setSession("adm","no");
setSession("mod", "no") ;
setSession("reg", "no") ;
'u := getQuery('user) ;
'p := getQuery('pass) ;
'p1 := selectDB('u) ;
'createlvl := selectDB("create-level") ;
'writelvl := selectDB("write-level") ;
'readlvl := selectDB("read-level") ;
if ('p = 'p1) then
setSession("user", 'u) ;
'r := selectDB('u '.  "-role") ;
setSession("reg", "yes") ;
if ('createlvl = "reg") then
setSession("can-create", "yes") fi ;
if ('writelvl = "reg") then
setSession("can-write", "yes") fi ;
if ('readlvl = "reg") then
setSession("can-read", "yes") fi ;
if ('r = "adm") then
setSession("adm" , "yes") ;
setSession("mod" , "yes") ;
setSession("can-create", "yes") ;
setSession("can-write", "yes") ;
setSession("can-read", "yes")
else
setSession("adm" , "no") ;
if ('r = "mod") then
setSession("mod", "yes") ;
if ('createlvl = "mod") then
setSession("can-create", "yes") fi ;
if ('writelvl = "mod") then
setSession("can-write", "yes") fi ;
if ('readlvl = "mod") then
setSession("can-read", "yes") fi
else
setSession("mod", "no")
fi fi fi
```

$\boxed{\alpha_{\mathsf{index}}}$

```
setSession("adminPage", "free") ;
--- Set default levels
'r := getSession("reg") ;
if ('r = null) then
setSession("reg", "no") ;
setSession("mod", "no") ;
setSession("adm", "no") ;
setSession("can-create", "no") ;
setSession("can-write", "no") ;
setSession("can-read", "no")
fi ;
--- Set capabilities available
'createlvl := selectDB("create-level") ;
'writelvl := selectDB("write-level") ;
'readlvl := selectDB("read-level") ;
if ('createlvl = "all") then
setSession("can-create", "yes")
fi ;
if ('writelvl = "all") then
setSession( "can-write","yes")
fi ;
if ('readlvl = "all") then
setSession("can-read", "yes")
fi
```

$\boxed{\alpha_{\mathsf{logout}}}$

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

$\boxed{\alpha_{\mathsf{admin}}}$

```
setSession("adminPage", "busy")
```

Figure 3.3: Specification of the electronic forum application in WEB-TLR

```
{[ B(bidAlfred, tidAlfred, 'Admin, 'Index ?  query-empty, (s("adm"), s("yes")) :  (s("adminPage"), s("busy")) :
(s("can-create"), s("yes")) :  (s("can-read"), s("yes")) :  (s("can-write"), s("yes")) :  (s("mod"), s("yes")) :  (s("reg"),
s("yes")), ('pass / "secretAlfred") :  'user / "alfred", m(bidAlfred, tidAlfred, 'Admin ?  query-empty, 1), history-empty, 1) :
B(bidAnna, tidAnna, 'Admin, 'Index ?  query-empty, (s("adm"), s("yes")) :  (s("adminPage"), s("busy")) :  (s("can-create"),
s("yes")) :  (s("can-read"), s("yes")) :  (s("can-write"), s("yes")) :  (s("mod"), s("yes")) :  (s("reg"), s("yes")), ('pass /
"secretAnna") :  'user / "anna", m(bidAnna, tidAnna, 'Admin ?  query-empty, 1), history-empty, 1)]bra-empty[mes-empty][S(('Access,
setSession(s("adm"), s( "no")); setSession(s("mod"), s("no")); setSession(s("reg"), s("no")); 'u := getQuery('user); 'p :=
getQuery('pass); 'p1 := selectDB('u); 'createlvl := selectDB(s("create-level")); 'writelvl := selectDB(s("write-level"));
'readlvl := selectDB(s("read-level")); if 'p = 'p1 then 'r := selectDB('u '.  s("-role")); setSession(s("reg"), s("yes")); 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,
{(s("reg") '== s("no") => 'Login) :  (s("reg") '== s("yes") => 'Index)}, { nav-empty}) :  ('Add-Comment, skip, {cont-empty},
{(TRUE -> 'View-Topic ?  query-empty)}) :  ('Admin, setSession(s("adminPage"), s("busy")), {cont-empty}, {( TRUE -> 'Index
?  query-empty)}) :  ('Delete-Comment, skip, {cont-empty}, {(TRUE -> 'View-Topic ?  query-empty)}) :  ('Delete-Topic, skip,
{cont-empty}, {(TRUE -> 'Index ?  query-empty)}) :  ('Index, setSession(s("adminPage"), s("free")); '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 ; '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, {cont-empty}, {(s("adm") '== s("yes") -> 'Admin ?  query-empty)
:  (s( "can-create") '== s("yes") -> 'New-Topic ?  'topic '= "") :  (s("can-read") '== s( "yes") -> 'View-Topic ?  'topic '= "")
:  (s("mod") '== s("yes") -> 'Delete-Topic ?  'topic '= "") :  (s("reg") '== s("no") -> 'Login ?  query-empty) :  (s("reg") '==
s("yes") -> 'Logout ?  query-empty)}) :  ('Login, skip, {cont-empty}, {(TRUE -> 'Access ?  ('pass '= "") :  'user '= "") :  (TRUE ->
'Index ?  query-empty)}) :  ( 'Logout, 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")), {( TRUE => 'Index)},
{nav-empty}) :  ('New-Topic, skip, {cont-empty}, {(TRUE -> 'View-Topic ?  query-empty)}) :  ('View-Topic, skip, {cont-empty},
{(TRUE -> 'Index ?  query-empty) :  (s("can-write") '== s("yes") -> 'Add-Comment ?  query-empty) :  (s("mod") '== s("yes") ->
'Delete-Comment ?  query-empty)}), us(bidAlfred, (s("adm"), s("yes")) :  (s("adminPage"), s("busy")) :  (s("can-create"), s("yes"))
:  (s("can-read"), s("yes")) :  (s("can-write"), s("yes")) :  (s("mod"), s("yes")) :  (s("reg"), s("yes"))) :  us(bidAnna, (s("adm"),
s("yes")) :  (s("adminPage"), s("busy")) :  (s("can-create"), s("yes")) :  (s("can-read"), s("yes")) :  (s("can-write"), s("yes"))
:  (s("mod"), s("yes")) :  (s("reg"), s("yes"))), mes-empty, readymes-empty, (s("alfred") ; s("secretAlfred")) (s("alfred-role") ;
s("adm")) (s("anna") ; s("secretAnna")) (s("anna-role") ; s("adm")) (s("create-level") ; s("reg")) (s("marc") ; s("secretMarc"))
(s("marc-role") ; s("mod")) (s("maude") ; s("secretMaude")) (s("maude-role") ; s("mod")) (s("rachel") ; s("secretRachel"))
(s("rachel-role") ; s("reg")) (s("read-level") ; s("all")) (s("robert") ; s("secretRobert")) (s("robert-role") ; s("reg"))
(s("write-level") ; s("reg")))] , 'ReqFin}
```

Figure 3.4: One Web state of the counter-example trace of Section 6.4.

4 An Interactive WEB-TLR System

WEB-TLR is a software tool designed for model-checking Web applications which is based on rewriting logic. Web applications are expressed as rewrite theories which can be formally verified by using the Maude built-in LTLR model checker. This chapter describes a first improved version of WEB-TLR (as published in [4]) that promoted it from a console-line based, manual tool to a full-fledged Web application. For covenience, we will refer to this version as Interactive WEB-TLR. In this enhanced version, whenever a property is refuted, an interactive slideshow is generated that allows the user to visually reproduce, step by step, the erroneous navigation trace that underlies the failing model checking computation. This provides deep insight into the system behavior, which helps to debug Web applications.

4.1 Introduction

In recent years, the automated verification of Web applications has become a major field of research. Nowadays, a number of corporations (including book retailers, auction sites, travel reservation services, *etc.*) interact primarily through the Web by means of Web applications that combine static content with dynamic data produced "on-the-fly" by the execution of Web scripts (e.g. Java servlets, Microsoft ASP.NET and PHP code). The inherent complexity of such highly concurrent systems has turned their verification into a challenge [1, 23, 29].

In [8], a rich and accurate navigation model that formalizes the behavior of Web applications in rewriting logic was formulated. This formulation allows to specify critical aspects of Web applications such as concurrent Web interactions, browser navigation features (i.e., forward/backward navigation, page refreshing, and window/tab openings), and Web script evaluations by means of a concise, high-level rewrite theory. This formalization is particularly suitable for verification purposes since it allows in-depth analyses of several subtle aspects of Web interactions to be carried out. It has been shown how real-size, dynamic Web applications can be efficiently model-checked using the *Linear Temporal Logic of Rewriting* (LTLR), which is a temporal logic specifically designed to model-check rewrite theories that combines all the advantages of the state-based and event-based logics, while avoiding their respective disadvantages [28].

This chapter describes WEB-TLR, which is a model-checking tool that implements the theoretical framework of [8]. WEB-TLR is written in Maude and is equipped with a freely accessible graphical Web interface (GWI) written in Java, which allows users to introduce and check their own specification of a Web application, together with the properties to be verified. In the case when the property is proven to be false (refuted), an online facility can be invoked that dynamically generates a counter-example (expressed as a navigation trace), which is ultimately responsible for the erroneous Web application behavior. In order to improve the understandability and usability of the system and since the textual information associated to counter-examples is usually rather large and poorly readable, the checker has been endowed with the capability to generate and display on-the-fly slideshows that allow the erroneous navigation trace to be visually reproduced step by step. This graphical facility, provides deep insight into Web application behavior and is extremely effective for debugging purposes.

WEB-TLR focuses on the Web application tier (business logic, and thus handles server-side scripts; no support is given for GUI verification with Flash technology or other kinds of client-side computations.

4.2 An overview of the Web verification framework

In this section, we briefly recall the main concepts of the Web verification framework proposed in [8], which are essential for understanding this tool description. A Web application is thought of as a collection of related Web pages that are hosted by a Web server and contain a mixture of (X)HTML code, executable code (Web scripts), and links to other Web pages. A Web application is accessed over a network such as the Internet by using a Web browser which allows Web pages to be navigated by clicking and following links. Interactions between Web browsers and the Web server are driven by the HTTP protocol.

A Web application is specified in our setting by means of a rewrite theory, which accurately formalizes the entities in play (e.g., Web server, Web browsers, Web scripts, Web pages, messages) as well as the dynamics of the system (that is, how the computation evolves through HTTP interactions). More specifically, the Web application behavior is formalized by using labeled rewrite rules of the form label: $\mathsf{WebState} \Rightarrow \mathsf{WebState}$, where $\mathsf{WebState}$ is a triple[1] $\_\|\_\|\_ :(\mathsf{Browsers} \times \mathsf{Message} \times \mathsf{Server}) \to \mathsf{WebState}$ that can be interpreted as a snapshot of the system that captures the current configurations of the active browsers (i.e., the browsers currently using the Web application), together with the server and the channel through which the browsers and the server communicate via message-passing. Given an initial Web state st_0, a computation is a rewrite sequence starting

[1]A detailed specification of Browsers, Message, and Server can be found in [8].

from $\mathsf{st_0}$ that is obtained by non-deterministically applying (labeled) rewrite rules to Web states.

Also, formal properties of the Web application can be specified by means of the *Linear Temporal Logic of Rewriting* (LTLR), which is a temporal logic that extends the traditional Linear Temporal Logic (LTL) with *state predicates*[28], i.e, atomic predicates that are locally evaluated on the states of the system. Let us see some examples. Assume that forbid is a session variable that is used to establish whether a login event is possible at a given configuration. In LTLR, we can define the state predicate userForbidden(bid), which holds in a Web state when a browser bid[2] is prevented from logging on to the system, by simply inspecting the value of the variable forbid appearing in the server session that is recorded in the considered state. More formally,

$$\mathsf{browsers}\|\mathsf{channel}\|\mathsf{server}(\mathsf{session}(\underline{(\mathsf{bid}, \{\mathsf{forbid} = \mathsf{true}\})})) \models \mathsf{userForbidden}(\mathsf{bid}) = \mathsf{true}$$

In LTLR, we can also define the following state predicates as boolean functions: failedAttempt(bid,n), which holds when browser bid has performed n failed login attempts (this is achieved by recording in the state a counter n with the number of failed attempts); curPage(bid,p), which holds when browser bid is currently displaying the Web page p; and inconsistentState, which holds when two browser windows or tabs of the same browser refer to distinct user sessions. These elementary state predicates are used below to build more complex LTLR formulas expressing mixed properties that include dependencies among states, actions, and time. These properties intrinsically involve both action-based and state-based aspects which are either not expressible or are difficult to express in other temporal logic frameworks.

4.3 The Web-TLR system

Our verification methodology has been implemented in the Web-TLR system using the high-performance, rewriting logic language Maude [16] (around 750 lines of code not including third-party components). Web-TLR is available online via its friendly Web interface at `http://users.dsic.upv.es/grupos/elp/soft.html`. The Web interface frees users from having to install applications on their local computer and hides a lot of technical details of the tool operation. After introducing the (or customizing a default) Maude specification of a Web application, together with an initial Web state $\mathsf{st_0}$ and the LTLR formula φ to be verified, φ can be automatically checked at $\mathsf{st_0}$. Once all inputs have been entered in the system (see Figure 4.2 for an example), we can automatically check the property by just clicking the button Check, which invokes the Maude built-in operator tlr check[9] that supports model checking of rewrite theories w.r.t. LTLR formulas.

In the case when φ is refuted by the model-checker, a counter-example is provided that is expressed as a model-checking computation trace starting from $\mathsf{st_0}$.

[2]We assume that the browser identifier uniquely identifies the user.

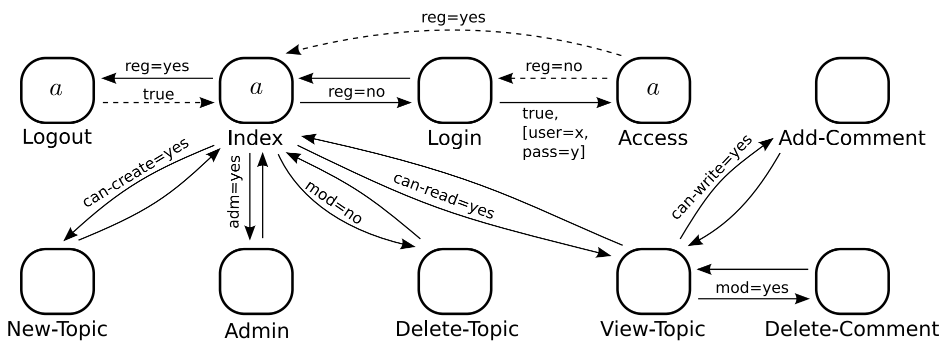


Figure 4.1: The navigation model of an Electronic Forum

The counter-example is graphically displayed by means of an interactive slideshow that allows forward and backward navigation through the computation's Web states (see Figure 4.3). Each slide contains a graph that models the structure of (a part of) the Web application. The nodes of the graph represent the Web pages, and the edges that connect the Web pages specify Web links or Web script continuations[3]. The graph also shows the current Web page of each active Web browser. The graphical representation is combined with a detailed textual description of the current configurations of the Web server and the active Web browsers.

4.4 A case study of Web verification.

We tested our tool on several complex case studies that are available at the WEB-TLR web page and distribution package. In order to illustrate the capabilities of the tool, in the following we discuss the verification of an electronic forum equipped with a number of common features, such as user registration, role-based access control including moderator and administrator roles, and topic and comment management.

The navigation model of such an application is formalized by means of the graph-like structure given in Figure 4.1. Web pages are modeled as graph nodes. Each navigation link l is specified by a solid arrow that is labeled by a condition c and a query string q. l is enabled whenever c evaluates to *true*, while q represents the input parameters that are sent to the Web server once the link is clicked. For example, the navigation link connecting the Login and Access Web pages is always enabled and requires two input parameters (user and pass). The dashed arrows model Web application continuations, that is, arrows pointing to Web pages that are automatically computed by Web script executions. Each dashed arrow is labeled by a condition, which is used to select the continuation at runtime. For example, the Access Web page has got two possible continuations (dashed arrows) whose labels are reg=yes and reg=no, respectively. The former continuation specifies that the

[3]To obey the stateless nature of the Web, the structure of Web applications has traditionally been "inverted", resembling programs written in a continuation–passing style [23].

login succeeds, and thus the Index Web page is delivered to the browser; in the latter case, the login fails and the Login page is sent back to the browser.

Using the state predicates given in Section 4.2, we are able to define and check sophisticated LTLR properties w.r.t. the considered Web application model. In the following, we discuss a selection of the properties that we considered.

Concise and parametric properties. We can define and verify the login property *"Incorrect login attempts are allowed only k times; then login is denied"*, which is defined parametrically w.r.t. the number of login attempts:

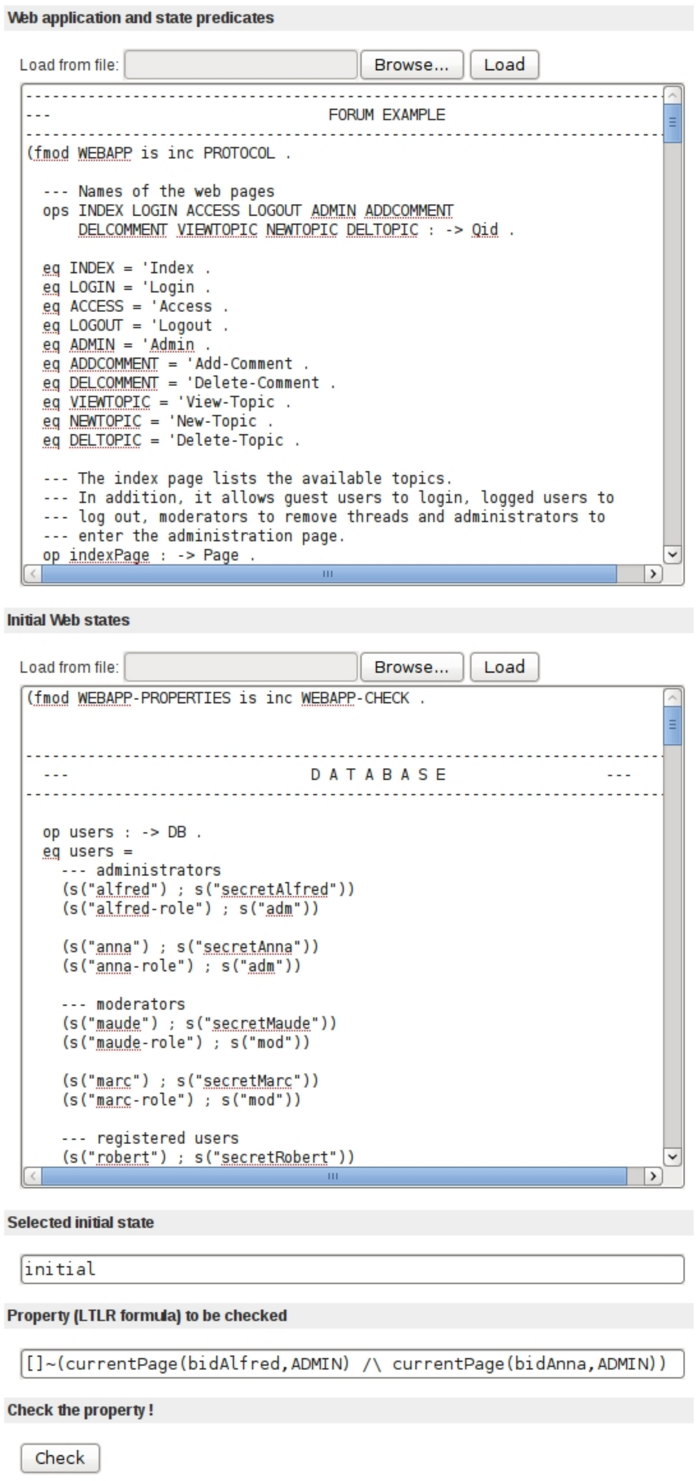
$$\Diamond(\mathsf{curPage}(\mathsf{A},\mathsf{Login}) \wedge \bigcirc(\Diamond\mathsf{failedAttempt}(\mathsf{A},\mathsf{k}))) \rightarrow \Box\mathsf{userForbidden}(\mathsf{A})$$

Note the sugared syntax (which is allowed in LTLR) when using relational notation for the state predicates which were defined as boolean functions above.

Unreachability properties. Unreachability properties can be specified as LTLR formulas of the form $\Box\neg\langle\mathsf{State}\rangle$, where State is an undesired configuration that the system should not reach. This unreachability pattern allows us to specify and verify a wide range of interesting properties such as the absence of conflict due to multiple windows, mutual exclusion, link accessibility, *etc.*

- Mutual exclusion: *"No two administrators can access the administration page simultaneously"*.
  $\Box\neg(\mathsf{curPage}(\mathsf{A},\mathsf{Admin}) \wedge \mathsf{curPage}(\mathsf{B},\mathsf{Admin}))$.
- Link accessibility: *"All links refer to existing Web pages"* (absence of broken links).
  $\Box\neg\mathsf{curPage}(\mathsf{A},\mathsf{PageNotFound})$.
- No multiple windows problem: *"We do not want to reach a Web application state in which two browser windows refer to distinct user sessions"*.
  $\Box\neg\mathsf{inconsistentState}$.

The detailed specification of the electronic forum, together with some example properties are available at `http://users.dsic.upv.es/grupos/elp/soft.html`.


Web application and state predicates

```
---------------------------------------------------------------------
---                              FORUM EXAMPLE
---------------------------------------------------------------------
(fmod 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 .

  --- 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 .
```

Initial Web states

```
(fmod 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"))
```

Selected initial state

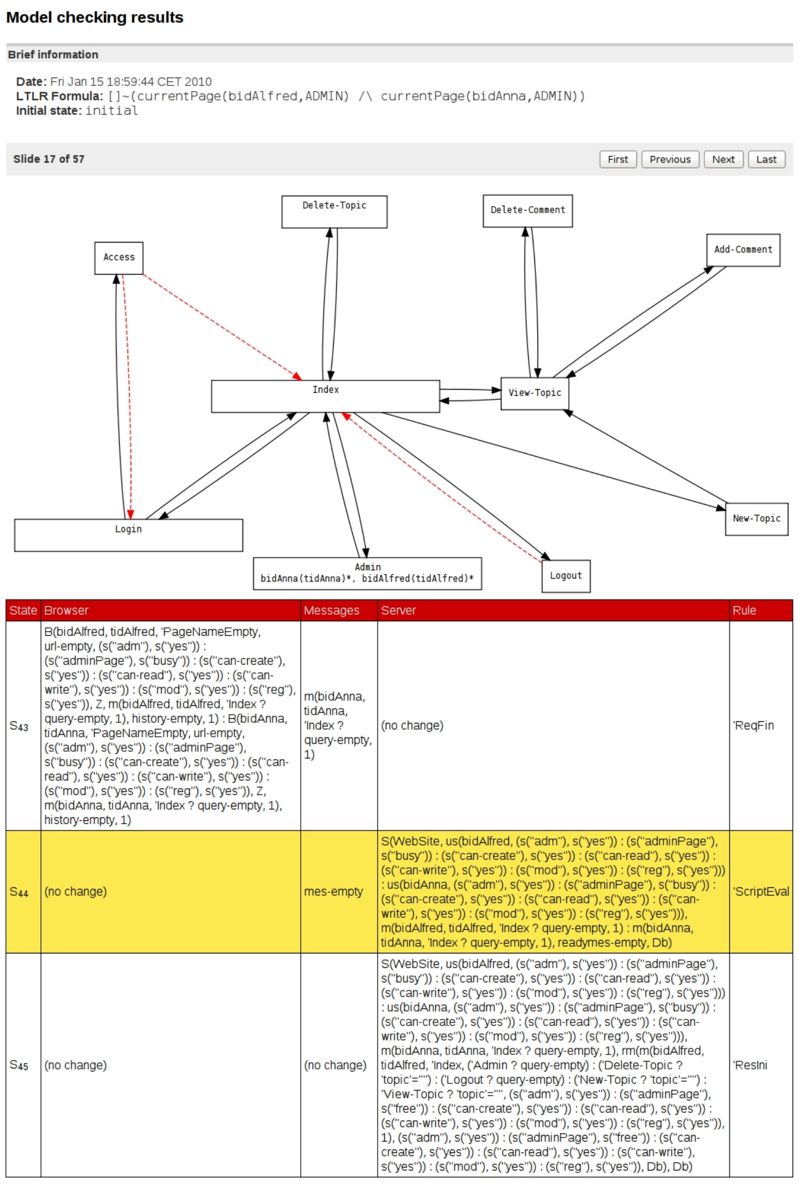
```
initial
```

Property (LTLR formula) to be checked

```
[]~(currentPage(bidAlfred,ADMIN) /\ currentPage(bidAnna,ADMIN))
```

Check the property !

Figure 4.2: The model definition page



Model checking results

Brief information

Date: Fri Jan 15 18:59:44 CET 2010
LTLR Formula: `[]~(currentPage(bidAlfred,ADMIN) /\ currentPage(bidAnna,ADMIN))`
Initial state: `initial`

Slide 17 of 57

State	Browser	Messages	Server	Rule
S_{43}	B(bidAlfred, tidAlfred, 'PageNameEmpty, url-empty, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes")), Z, m(bidAlfred, tidAlfred, 'Index ? query-empty, 1), history-empty, 1) : B(bidAnna, tidAnna, 'PageNameEmpty, url-empty, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes")), Z, m(bidAnna, tidAnna, 'Index ? query-empty, 1), history-empty, 1)	m(bidAnna, tidAnna, 'Index ? query-empty, 1)	(no change)	'ReqFin
S_{44}	(no change)	mes-empty	S(WebSite, us(bidAlfred, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes"))) : us(bidAnna, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes"))), m(bidAlfred, tidAlfred, 'Index ? query-empty, 1) : m(bidAnna, tidAnna, 'Index ? query-empty, 1), readymes-empty, Db)	'ScriptEval
S_{45}	(no change)	(no change)	S(WebSite, us(bidAlfred, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes"))) : us(bidAnna, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes"))), m(bidAnna, tidAnna, 'Index ? query-empty, 1), rm(m(bidAlfred, tidAlfred, 'Index, ('Admin ? query-empty) : ('Delete-Topic ? 'topic'="") : ('Logout ? query-empty) : ('New-Topic ? 'topic'="") : 'View-Topic ? 'topic'="", (s("adm"), s("yes")) : (s("adminPage"), s("free")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes")), 1), (s("adm"), s("yes")) : (s("adminPage"), s("free")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes")), Db), Db)	'ResIni

Figure 4.3: Snapshot of the slide show

5

Backward Trace Slicing for Rewriting Logic Theories

Trace slicing is a widely used technique for execution trace analysis that is effectively used in program debugging, analysis and comprehension. In this chapter, we present a backward trace slicing technique that can be used for the analysis of Rewriting Logic theories. Our trace slicing technique allows us to systematically trace back rewrite sequences modulo equational axioms (such as associativity and commutativity) by means of an algorithm that dynamically simplifies the traces by detecting control and data dependencies, and dropping useless data that do not influence the final result. Our methodology is particularly suitable for analyzing complex, textually-large system computations such as those delivered as counter-example traces by Maude model-checkers.

5.1 Introduction

The analysis of execution traces plays a fundamental role in many program manipulation techniques. Trace slicing is a technique for reducing the size of traces by focusing on selected aspects of program execution, which makes it suitable for trace analysis and monitoring [13].

Rewriting Logic (RWL) is a very general *logical* and *semantic framework*, which is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [11, 37] and Web systems [4, 8]). RWL is efficiently implemented in the high-performance system Maude [16]. Roughly speaking, a *rewriting logic theory* seamlessly combines a *term rewriting system* (TRS) together with an *equational theory* that may include sorts, functions, and algebraic laws (such as commutativity and associativity) so that rewrite steps are applied *modulo* the equations. Within this framework, the system states are typically represented as elements of an algebraic data type that is specified by the equational theory, while the system computations are modeled via the rewrite rules, which describe transitions between states.

Due to the many important applications of RWL, in recent years, the debugging and optimization of RWL theories have received growing attention [2, 26, 32, 33]. However, the existing tools provide hardly support for execution trace analysis.

The original motivation for our work was to reduce the size of the counterexample traces delivered by Web-TLR, which is a RWL-based model-checking tool for Web applications proposed in [4, 8]. As a matter of fact, the analysis (or even the simple inspection) of such traces may be unfeasible because of the size and complexity of the traces under examination. Typical counterexample traces in Web-TLR are 75 Kb long for a model size of 1.5 Kb, that is, the trace is in a ratio of 5.000% w.r.t. the model.

To the best of our knowledge, the presented algorithm is the first trace slicing technique for RWL theories. The basic idea is to take a trace produced by the RWL engine and traverse and analyze it backwards to filter out events that are irrelevant for the rewritten task. The trace slicing technique that we propose is fully general and can be applied to optimizing any RWL-based tool that manipulates rewrite logic traces. Our technique relies on a suitable mechanism of backward tracing that is formalized by means of a procedure that labels the calls (terms) involved in the rewrite steps. The backward traversal is preferred to a forward one because a causal relation is computed. This allows us to infer, from a term t and positions of interest on it, positions of interest of the term that was rewritten to t. Our labeling procedure extends the technique in [12], which allows descendants and origins to be traced in orthogonal (i.e., left-linear and overlap-free) term rewriting systems in order to deal with rewrite theories that may contain commutativity/associativity axioms, as well as nonleft-linear, collapsing equations and rules.

Plan of the chapter. In Section 5.2, we recall the essential notions concerning rewriting modulo equational theories. In Section 5.3, we formalize our backward trace slicing technique for rewriting logic theories, which computes the reverse dependence among the symbols involved in a rewrite step and removes all data that are irrelevant with respect to a given slicing criterion. Section 5.4 extends the trace slicing technique of Section 5.3 by considering extended rewrite theories, i.e., rewrite theories that may include collapsing, nonleft-linear rules, associative/commutative equational axioms, and built-in operators. Section 5.5 describes a software tool that implements the proposed backward slicing technique and presents an experimental evaluation of the tool that allows us to assess the practical advantages of the trace slicing technique. In Section 5.6, we discuss some related work.

5.2 Rewriting Modulo Equational Theories

An *equational theory* is a pair (Σ, E), where Σ is a signature and $E = \Delta \cup B$ consists of a set of (oriented) equations Δ together with a collection B of equational axioms (e.g., associativity and commutativity axioms) that are associated with some operator of Σ. The equational theory E induces a least congruence relation on the term algebra $T_\Sigma(\mathcal{V})$, which is usually denoted by $=_E$.

A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory, and R is a TRS. Examples of rewrite theories can be found in [16].

Rewriting modulo equational theories [26] can be defined by lifting the standard rewrite relation $\to_R$ on terms to the E-congruence classes induced by $=_E$. More precisely, the rewrite relation $\to_{R/E}$ for rewriting modulo E is defined as $=_E \circ \to_R \circ =_E$. A computation in $\mathcal{R}$ using $\to_{R\cup\Delta,B}$ is a *rewriting logic deduction*, in which the *equational simplification* with Δ (i.e., applying the oriented equations in Δ to a term t until a canonical form $t\downarrow_E$ is reached where no further equations can be applied) is intermixed with the rewriting computation with the rules of R, using an *algorithm of matching modulo*[1] B in both cases.

Formally, given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where $E = \Delta \cup B$, a *rewrite step modulo* E on a term s_0 by means of the rule $r : \lambda \to \rho \in R$ (in symbols, $s_0 \xrightarrow{r}_{R\cup\Delta,B} s_1$) can be implemented as follows: (i) apply (modulo B) the equations of Δ on s_0 to reach a canonical form $(s_0\downarrow_E)$; (ii) rewrite (modulo B) $(s_0\downarrow_E)$ to term v by using $r \in R$; and (iii), apply (modulo B) the equations of Δ on v again to reach a canonical form for v, $s_1 = v\downarrow_E$.

Since the equations of Δ are implicitly oriented (from left to right), the equational simplification can be seen as a sequence of (equational) rewrite steps $(\to_{\Delta/B})$. Therefore, a *rewrite step modulo* E $s_0 \xrightarrow{r}_{R\cup\Delta,B} s_1$ can be expanded into a sequence of rewrite steps as follows:

$$s_0 \overbrace{\to_{\Delta/B} \cdot\cdot \to_{\Delta/B} s_0{\downarrow_E}}^{\text{equational simplification}} \overbrace{=_B u \xrightarrow{r}_R v}^{\text{rewrite step}/_B} \overbrace{\to_{\Delta/B} \cdot\cdot \to_{\Delta/B} v{\downarrow_E}}^{\text{equational simplification}} = s_1$$

Given a finite rewrite sequence $\mathcal{S} = s_0 \to_{R\cup\Delta,B} s_1 \to_{R\cup\Delta,B} \ldots \to s_n$ in the rewrite theory $\mathcal{R}$, the *execution trace* of $\mathcal{S}$ is the rewrite sequence $\mathcal{T}$ obtained by expanding all the rewrite steps $s_i \to_{R\cup\Delta,B} s_{i+1}$ of $\mathcal{S}$ as is described above.

The computability of $\to_{R\cup\Delta,B}$ as well as its equivalence w.r.t. $\to_{R/E}$ are assured by enforcing some conditions on the considered rewrite theories [26, 40], specifically, *coherence* between the rules and the equations as well as the assumption of *Church-Rosser* and *termination* properties of Δ modulo the equational axioms B[2].

A rewrite theory $\mathcal{R} = (\Sigma, B \cup \Delta, R)$ is called *elementary* if $\mathcal{R}$ does not contain equational axioms ($B = \emptyset$) and both rules and equations are left-linear and not collapsing.

[1]A subterm of t matches l (*modulo* B) via the substitution σ if $t =_B u$ and $u_{|q} = l\sigma$ for a position q of u.

[2]These conditions are quite natural in practical rewriting logic specifications, and can generally be checked by using the Maude Church-Rosser, Termination, and Coherence tools [16].

5.3 Backward Trace Slicing for Elementary Rewrite Theories

In this section, we formalize a backward trace slicing technique for *elementary rewrite theories* that is based on a term labeling procedure that is inspired by [12]. Since equations in Δ are treated as rewrite rules that are used to simplify terms, our formulation for the trace slicing technique is purely based on standard rewriting. In Section 5.4, we will drop all these restrictions in order to consider more expressive rewrite theories.

5.3.1 Labeling procedure for rewrite theories

Let us define a labeling procedure for rules similar to [12] that allows us to trace symbols involved in a rewrite step. First, we provide the notion of labeling for terms, and then we show how it can be naturally lifted to rules and rewrite steps.

Consider a set $\mathcal{A}$ of *atomic labels*, which are denoted by Greek letters $\alpha, \beta, \ldots$. *Composite labels* (or simply *labels*) are defined as finite sets of elements of $\mathcal{A}$. By abuse, we write the label $\alpha\beta\gamma$ as a compact denotation for the set $\{\alpha, \beta, \gamma\}$.

A *labeling* for a term $t \in T_{\Sigma\cup\{\square\}}(\mathcal{V})$ is a map L that assigns a label to (the symbol occurring at) each position w of t, provided that $root(t_{|w}) \neq \square$. If t is a term, then t^L denotes the labeled version of t. Note that, in the case when t is a context, occurrences of symbol $\square$ appearing in the labeled version of t are not labeled. The *codomain* of a labeling L is denoted by $Cod(L) = \{l \mid (w \mapsto l) \in L\}$.

An *initial labeling* for the term t is a labeling for t which assigns distinct fresh atomic labels to each position of the term. For example, given $t = f(g(a, a), \square)$, then $t^L = f^\alpha(g^\beta(a^\gamma, a^\delta), \square)$ is the labeled version of t via the initial labeling $L = \{\Lambda \mapsto \alpha,$ $1 \mapsto \beta,\ 1.1 \mapsto \gamma,\ 1.2 \mapsto \delta\}$. This notion extends to rules and rewrite steps in a natural way as shown below.

Labeling of Rules.

Let us introduce the notions of *redex pattern* and *contractum pattern* of a rule. Let $r : \lambda \to \rho$ be a rule. We call the context $\lambda^\square$ (resp. $\rho^\square$) *redex pattern* (resp. *contractum pattern*) of r.

Example Given the rule $r : f(g(x, y), a)) \to d(s(y), y)$, where a is a constant symbol, the redex pattern of r is the context $f(g(\square, \square), a)$, while the contractum pattern of r is the context $d(s(\square), \square)$.

Definition (rule labeling) [12] Given a rule $r : \lambda \to \rho$, a labeling L_r for r is defined by means of the following procedure.

r_1. The redex pattern $\lambda^\square$ is labeled by means of an initial labeling L.

r_2. A new label l is formed by joining all the labels that occur in the labeled redex pattern $\lambda^{\square}$ (say in alphabetical order) of the rule r. Label l is then associated with each position w of the contractum pattern $\rho^{\square}$, provided that $root(\rho^{\square}_{|w}) \neq \square$.

Example Consider the rule r of Example 5.3.1. The labeled version of rule r using the initial labeling $L = \{(\Lambda \mapsto \alpha, 1 \mapsto \beta, 2 \mapsto \gamma\}$ is as follows: $f^{\alpha}(g^{\beta}(x,y), a^{\gamma}) \to d^{\alpha\beta\gamma}(s^{\alpha\beta\gamma}(y), y)$.

The labeled version of r w.r.t. L_r is denoted by r^{L_r}. Note that the labeling procedure shown in Definition 5.3.1 does not assign labels to variables but only to the function symbols occurring in the rule.

Labeling of Rewrite Steps.

Before giving the definition of labeling for a rewrite step, we need to formalize the auxiliary notion of substitution labeling.

Definition (substitution labeling) Let $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ be a substitution. A labeling L_σ for the substitution σ is defined by a set of initial labelings $L_\sigma = \{L_{x_1/t_1}, \ldots, L_{x_n/t_n}\}$ such that (i) for each binding (x_i/t_i) in the substitution σ, t_i is labeled using the corresponding initial labeling L_{x_i/t_i}, and (ii) the sets $Cod(L_{x_1/t_1}), \ldots, Cod(L_{x_n/t_n})$ are pairwise disjoint.

By using Definition 5.3.1, we can formulate a labeling procedure for rewrite steps as follows.

Definition (rewrite step labeling) Let $r : \lambda \to \rho$ be a rule, and $\mu : t \overset{r,\sigma}{\to} s$ be a rewrite step using r such that $t = C[\lambda\sigma]_q$ and $s = C[\rho\sigma]_q$, for a context C and position q. Let $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. Let L_r be a labeling for the rule r, L_C be an initial labeling for the context C, and $L_\sigma = \{L_{x_1/t_1}, \ldots, L_{x_n/t_n}\}$ be a labeling for the substitution σ such that the sets $Cod(L_C)$, $Cod(L_r)$, and $Cod(\sigma)$ are pairwise disjoint, where $Cod(\sigma) = \bigcup_{i=1}^{n} Cod(L_{x_i/t_i})$.

The *rewrite step* labeling L_μ for μ is defined by successively applying the following steps:

s_1. First, positions of t or s that belong to the context C are labeled by using the initial labeling L_C.

s_2. Then positions of $t_{|q}$ (resp. $s_{|q}$) that correspond to the redex pattern (resp. contractum pattern) of the rule r rooted at the position q are labeled according to the labeling L_r.

s_3. Finally, for each term t_j, $j = \{1, \ldots, n\}$, which has been introduced in t or s via the binding $x_j/t_j \in \sigma$, with $x_j \in Var(\lambda)$, t_j is labeled using the corresponding labeling $L_{x_j/t_j} \in L_\sigma$

The labeled version of a rewrite step μ w.r.t. L_μ is denoted by μ^{L_μ}. Let us illustrate it by means of a rather intuitive example.

Example Consider again the rule $r : f(g(x,y),a)) \to d(s(y),y)$ of Example 5.3.1, and let $\mu : C[\lambda\sigma] \xrightarrow{r} C[\rho\sigma]$ be a rewrite step using r, where $C[\lambda\sigma] = d(f(g(a,h(b)),a),a)$, $C[\rho\sigma] = d(d(s(h(b)),h(b)),a)$, and $\sigma = \{x/a, y/h(b)\}$.

Assume that r is labeled by means of the rule labeling of Example 5.3.1, that is

$$r^L : f^\alpha(g^\beta(x,y),a^\gamma) \to d^{\alpha\beta\gamma}(s^{\alpha\beta\gamma}(y),y)$$

Let $L_C = \{\Lambda \mapsto \delta,\ 2 \mapsto \epsilon\}$, $L_{x/a} = \{\Lambda \mapsto \zeta\}$, and $L_{y/h(b)} = \{\Lambda \mapsto \eta, 1 \mapsto \theta\}$ be the labelings for C and the bindings in σ, respectively. Then, the corresponding labeled rewrite step μ^L is as follows

$$\mu^{\mathtt{L}} : \mathtt{d}^\delta(\mathtt{f}^\alpha(\mathtt{g}^\beta(\mathtt{a}^\zeta,\mathtt{h}^\eta(\mathtt{b}^\theta)),\mathtt{a}^\gamma),\mathtt{a}^\epsilon) \to \mathtt{d}^\delta(\mathtt{d}^{\alpha\beta\gamma}(\mathtt{s}^{\alpha\beta\gamma}(\mathtt{h}^\eta(\mathtt{b}^\theta)),\mathtt{h}^\eta(\mathtt{b}^\theta)),\mathtt{a}^\epsilon)$$

Now, we are ready to define our labeling-based, *backward tracing relation* on rewrite steps.

Definition (origin positions) Let $\mu : t \xrightarrow{r} s$ be a rewrite step and L be a labeling for μ where L_t (resp. L_s) is the labeling of t (resp. s). Given a position w of s, the set of origin positions of w in t w.r.t. μ and L (in symbols, $\lhd^L_\mu w$) is defined as follows:

$$\lhd^L_\mu w = \{v \in \mathcal{P}os(t) \mid \exists p \in \mathcal{P}os(s), (v \mapsto l_v) \in L_t, (p \mapsto l_p) \in L_s \text{ s.t. } p \leq w \text{ and } l_v \subseteq l_p\}$$

Roughly speaking, a position v in t is an origin of w, if the label of the symbol occurring in t^L at position v is contained in the label of a symbol occurring in s^L in the path from its root to the position w.

Example Consider again the rewrite step $\mu^L : t^L \to s^L$ of Example 5.3.1, and let w be the position 1.2 of s^L. The set of labeled symbols occurring in s^L in the path from its root to position w is the set $\mathtt{z} = \{\mathtt{h}^\eta, \mathtt{d}^{\alpha\beta\gamma}, \mathtt{d}^\delta\}$. Now, the labeled symbols occurring in t^L whose label is contained in the label of one element of $\mathtt{z}$ is the set $\{\mathtt{h}^\eta, \mathtt{f}^\alpha, \mathtt{g}^\beta, \mathtt{a}^\gamma, \mathtt{d}^\delta\}$. By Definition 5.3.1, the set of origin positions of w in μ^L is $\lhd^{\mathtt{L}}_\mu \mathtt{w} = \{\mathtt{1.1.2},\ \mathtt{1},\ \mathtt{1.1},\ \mathtt{1.2},\ \Lambda\}$.

Note that the origin positions of w in the rewrite step $\mu : t \xrightarrow{r} s$ are not the antecedent positions of w in μ [31]; one main difference is the fact that we consider all positions of s in the path from its root to w for computing the origins, and we use the labeling to trace back every relevant piece of information involved in the step μ.

5.3.2 The Backwards Trace Slicing Algorithm

First, let us formalize the slicing criterion, which basically represents the information we want to trace back across the execution trace in order to find out the "origins" of the data we observe. Given a term t, we denote by $\mathcal{O}_t$ the set of *observed* positions of t, which point to the symbols of t that we want to trace/observe.

Definition (slicing criterion) Given a rewrite theory $\mathcal{R} = (\Sigma, \Delta, R)$ and an execution trace $\mathcal{T} : s \to^* t$ in $\mathcal{R}$, a slicing criterion for $\mathcal{T}$ is any set $\mathcal{O}_t$ of positions of the term t.

In the following, we show how backward trace slicing can be performed by exploiting the backward tracing relation $\lhd^L_\mu$ that was introduced in Definition 5.3.1. Informally, given a slicing criterion $\mathcal{O}_{t_n}$ for $\mathcal{T} : t_0 \to t_2 \to \ldots \to t_n$, at each rewrite step $t_{i-1} \to t_i$, $i = 1, \ldots, n$, our technique inductively computes the backward tracing relation between the relevant positions of t_i and those in t_{i-1}. The algorithm proceeds backwards, from the final term t_n to the initial term t_0, and recursively generates at step i the corresponding set of relevant positions, $P_{t_{n-i}}$. Finally, by means of a removal function, a simplified trace is obtained where each t_j is replaced by the corresponding *term slice* that contains only the relevant information w.r.t. P_{t_j}.

Definition (sequence of relevant position sets) Let $\mathcal{R} = (\Sigma, \Delta, R)$ be a rewrite theory, and $\mathcal{T} : t_0 \xrightarrow{r_1} t_1 \ldots \xrightarrow{r_n} t_n$ be an execution trace in $\mathcal{R}$. Let L_i be the labeling for the rewrite step $t_i \to t_{i+1}$ with $0 \le i < n$. The sequence of relevant position sets in $\mathcal{T}$ w.r.t. the slicing criterion $\mathcal{O}_{t_n}$ is defined as follows:

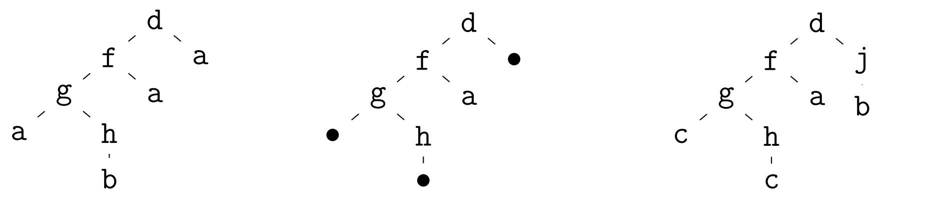
$$relevant\_positions(\mathcal{T}, \mathcal{O}_{t_n}) = [P_0, \ldots, P_n]$$
$$\text{where } \begin{cases} P_n = \mathcal{O}_{t_n} \\ P_j = \bigcup_{p \in P_{j+1}} \lhd^{L_j}_{(t_j \to\, t_{j+1})} p, \text{ with } 0 \le j < n \end{cases}$$

Now, it is straightforward to formalize a procedure that obtains a term slice from each term t in $\mathcal{T}$ and the corresponding set of relevant positions of t. We introduce the fresh symbol $\bullet \notin \Sigma$ to replace any information in the term that is not relevant (i.e., those symbols that occur at any position of t that is not above a relevant position of the term), hence does not affect the observed criterion.

Definition (term slice) Let $t \in T_\Sigma$ be a term and P be a set of positions of t. A *term slice* of t with respect to P is defined as follows:

$$slice(t, P) = sl\_rec(t, P, \Lambda),\ where$$

$$sl\_rec(t, P, p) = \begin{cases} f(sl\_rec(t_1, P, p.1), \ldots, sl\_rec(t_n, P, p.n)) \\ \quad \text{if } t = f(t_1, \ldots, t_n) \text{ and there exists } w \text{ s.t. } (p.w) \in P \\ \bullet \quad \text{otherwise} \end{cases}$$



term t | term slice $t^\bullet$ of t w.r.t. {1.1.2, 1.2} | a concretization of $t^\bullet$

Figure 5.1: A term slice and a possible concretization.

In the following, we use the notation $t^\bullet$ to denote a term slice of the term t. Roughly speaking, the symbol $\bullet$ can be thought of as a variable, so that any term $t' \in \tau(\Sigma)$ can be considered as a possible concretization of $t^\bullet$ if it is an "instance" of $[t^\bullet]$, where $[t^\bullet]$ is the term that is obtained by replacing all occurrences of $\bullet$ in $t^\bullet$ with fresh variables.

Definition (term slice concretization) Given $t' \in T_\Sigma$ and a term slice $t^\bullet$, we define $t^\bullet \propto t'$ if $[t^\bullet]$ is (syntactically) more general than t' (i.e. $[t^\bullet]\sigma = t'$, for some substitution σ). We also say that t' is a concretization of $t^\bullet$.

Figure 5.1 illustrates the notions of term slice and term slice concretization for a given term t w.r.t. the set of positions $\{1.1.2, 1.2\}$.

Let us define a *sliced rewrite step* between two term slices as follows.

Definition (sliced rewrite step) Let $\mathcal{R} = (\Sigma, \Delta, R)$ be a rewrite theory and r a rule of $\mathcal{R}$. The term slice $s^\bullet$ rewrites to the term slice $t^\bullet$ via r (in symbols, $s^\bullet \xrightarrow{r} t^\bullet$) if there exist two terms s and t such that $s^\bullet$ is a term slice of s, $t^\bullet$ is a term slice of t, and $s \xrightarrow{r} t$.

Using Definition 5.3.2, backward trace slicing is formalized as follows.

Definition (backward trace slicing) Let $\mathcal{R} = (\Sigma, \Delta, R)$ be a rewrite theory, and $\mathcal{T} : t_0 \xrightarrow{r_1} t_1 \ldots \xrightarrow{r_n} t_n$ be an execution trace in $\mathcal{R}$. Let $\mathcal{O}_{t_n}$ be a slicing criterion for $\mathcal{T}$, and let $[P_0, \ldots, P_n]$ be the sequence of the relevant position sets of $\mathcal{T}$ w.r.t. $\mathcal{O}_{t_n}$. A trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ w.r.t. $\mathcal{O}_{t_n}$ is defined as the sliced rewrite sequence of term slices $t_i^\bullet = \mathit{slice}(t_i, P_i)$ which is obtained by glueing together the sliced rewrite steps in the set

$$\mathcal{K}^\bullet = \{t_{k-1}^\bullet \xrightarrow{r_k} t_k^\bullet \mid 0 < k \leq n \;\wedge\; t_{k-1}^\bullet \neq t_k^\bullet\}.$$

Note that in Definition 5.3.2, the sliced rewrite steps that do not affect the relevant positions (i.e., $t^\bullet_{k-1} \stackrel{r_k}{\to} t^\bullet_k$ with $t^\bullet_{k-1} = t^\bullet_k$) are discarded, which further reduces the size of the trace.

A desirable property of a slicing technique is to ensure that, for any concretization of the term slice $t^\bullet_0$, the trace slice $\mathcal{T}^\bullet$ can be reproduced. This property ensures that the rules involved in $\mathcal{T}^\bullet$ can be applied again to every concrete trace $\mathcal{T}'$ that we can derive by instantiating all the variables in $[t^\bullet_0]$ with arbitrary terms.

Theorem 5.3.1 *(soundness) Let $\mathcal{R}$ be an elementary rewrite theory. Let $\mathcal{T}$ be an execution trace in the rewrite theory $\mathcal{R}$, and let $\mathcal{O}$ be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^\bullet : t^\bullet_0 \stackrel{r_1}{\to} t^\bullet_1 \dots \stackrel{r_n}{\to} t^\bullet_n$ be the corresponding trace slice w.r.t. $\mathcal{O}$. Then, for any concretization t'_0 of $t^\bullet_0$, it holds that $\mathcal{T}' : t'_0 \stackrel{r_1}{\to} t'_1 \dots \stackrel{r_n}{\to} t'_n$ is an execution trace in $\mathcal{R}$, and $t^\bullet_i \propto t'_i$, for $i = 1, \dots, n$.*

The proof of Theorem 5.3.1 relies on the fact that redex patterns are preserved by backward trace slicing. Therefore, for $i = 1, \dots, n$, the rule r_i can be applied to any concretization t'_{i-1} of term $t^\bullet_{i-1}$ since the redex pattern of r_i does appear in $t^\bullet_{i-1}$, and hence in t'_{i-1}. A detailed proof of Theorem 5.3.1 is included in [34].

5.4 Backward Trace Slicing for Extended Rewrite Theories

In this section, we consider an extension of our basic slicing methodology that allows us to deal with extended rewrite theories. An extended rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is a rewrite theory where the equational theory (Σ, E) may contain associativity and commutativity axioms, and R may contain collapsing as well as nonleft-linear rules. Moreover, we provide a further extension to deal with the built-in operators existing in Maude, that is, operators that are not equipped with an explicit functional definition (e.g., Maude arithmetical operators and if-then-else conditional operators).

It is worth noting that all the proposed extensions are restricted to the labeling procedure of Section 5.3.1, leaving the backbone of our slicing technique unchanged.

5.4.1 Dealing with collapsing and nonleft-linear rules

Collapsing Rules. The main difficulty with collapsing rules is that they have a trivial contractum pattern, which consists in the empty context $\square$; hence, it is not possible to propagate labels from the left-hand side of the rule to its right-hand side. This makes the rule labeling procedure of Definition 5.3.1 completely unproductive for trace slicing.

In order to overcome this problem, we keep track of the labels in the left-hand side of the collapsing rule r, whenever a rewrite step involving r takes place. This amounts to extending the labeling procedure of Definition 5.3.1 as follows.

Definition (rewrite step labeling for collapsing rules) Let $\mu : t \overset{r,\sigma}{\to} s$ be a rewrite step s.t. $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. Let L_r be a labeling for the rule r. For the case of a rewrite step given by using a collapsing rule $r : \lambda \to x_i$, the labeling procedure formalized in Definition 5.3.1 is extended as follows:

s_4. Let t_i be the term introduced in s via the binding $x_i/t_i \in \sigma$, for some $i \in \{1, \ldots, n\}$. Then, the label l_i of the root symbol of t_i in s is replaced by a new composite label $l_c l_i$, where l_c is formed by joining all the labels appearing in the redex pattern of r^{L_r}.

Example Consider again the labeled collapsing rule $f^{\alpha}(a^{\beta}, x) \to x$, together with the rewrite step $\mu : f(a, h(b)) \to h(b)$ and matching substitution $\sigma = \{x/h(b)\}$. Let $L_\sigma = \{\{\Lambda \mapsto \gamma,\ 1 \mapsto \delta\}\}$ be the labeling for σ. Then, by applying Definition 5.4.1, the labeling of μ is

$$f^{\alpha}(a^{\beta}, h^{\gamma}(b^{\delta})) \to h^{\alpha\beta\gamma}(b^{\delta})$$

and the trace slice for $f(a, h(b)) \to h(b)$ w.r.t. the slicing criterion $\{\Lambda\}$ is $f(a, h(\bullet)) \to h(\bullet)$.

Note that if we had merely applied Definition 5.3.1 instead of Definition 5.4.1, we would have got the following labeling for μ: $f^{\alpha}(a^{\beta}, h^{\gamma}(b^{\delta})) \to h^{\gamma}(b^{\delta})$, which is undesirable since it does not correctly record the redex pattern information that we need for backward trace slicing: e.g. if we slice the rewriting step μ w.r.t. $\{\Lambda\}$ using this wrong labeling, we would get $f(\bullet, h(\bullet)) \to h(\bullet)$.

Nonleft-linear Rules. The trace slicing technique we described in Section 5.3 does not work for nonleft-linear TRS. Consider the rule: $r : f(x, y, x) \to g(x, y)$ and the one-step trace $\mathcal{T} : f(a, b, a) \to g(a, b)$. If we are interested in tracing back the symbol g that occurs in the final state $g(a, b)$, we would get the following trace slice $\mathcal{T}^{\bullet} : f(\bullet, \bullet, \bullet) \to g(\bullet, \bullet)$. However, $f(a, b, b)$ is a concretization of $f(\bullet, \bullet, \bullet)$ that cannot be rewritten by using r. In the following, we augment Definition 5.4.1 in order to also deal with nonleft-linear rules.

Definition (rewrite step labeling procedure for nonleft-linear rules) Let $\mu : t \overset{r,\sigma}{\to} s$ be a rewrite step s.t. $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. Let $L_\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ be a labeling for the substitution σ. For the case of a rewrite step given by using a nonleft-linear rule r, the labeling procedure formalized in Definition 5.4.1 is extended as follows:

s_5. For each variable x_j that occurs more than once in the left-hand side of the rule r, the following steps should be performed:

- we form a new label l_{x_j} by joining all the labels in $Cod(L_{x_j/t})$ where $L_{x_j/t} \in L_\sigma$;
- let l_s be the label of the root symbol of s. Then, l_s is replaced by a new composite label $l_{x_j} l_s$.

Example Consider the nonleft-linear (labeled) rule $f^\alpha(x, y, x) \to g^\alpha(x, y)$ together with the rewrite step $\mu : f(g(a), b, g(a)) \to g(g(a), b)$, and matching substitution $\sigma = \{x/g(a), y/b\}$. Then, for the labeling $L_\sigma = \{L_{x/g(a)},\ L_{y/b}\}$, with $L_{x/g(a)} = \{\Lambda \mapsto \beta,\ 1 \mapsto \gamma\}$ and $L_{y/b} = \{\Lambda \mapsto \delta\}$, the labeled version of μ is

$$f^\alpha(g^\beta(a^\gamma), b^\delta, g^\beta(a^\gamma)) \to g^{\alpha\beta\gamma}(g^\beta(a^\gamma), b^\delta).$$

Finally, by considering the criterion $\{1\}$, we can safely trace back the symbol g at the position 1 of the term $g(g(a), b)$ and obtain the following trace slice

$$f(g(a), \bullet, g(a)) \to g(g(\bullet), \bullet).$$

5.4.2 Built-in Operators

In practical implementations of RWL (e.g., Maude [16]), several commonly used operators are pre-defined (e.g., arithmetic and boolean operators, if-then-else constructs). Obviously, backward trace slicing of function calls involving built-in operators is not supported by our basic technique. This would require an explicit (rule-based or equational) specification of every single operator involved in the execution trace. To overcome this limitation, we further extend the labeling procedure of Definition 5.4.1 in order to deal with built-in operators.

Definition (rewrite step labeling procedure for built-in operators) For the case of a rewrite step $\mu : C[op(t_1, \ldots, t_n)] \to C[t']$ involving a call to a built-in, n-ary operator op, we extend Definition 5.4.1 by introducing the following additional case:

s_6. Given an initial labeling L_{op} for the term $op(t_1, \ldots, t_n)$,

- each symbol occurrence in t' is labeled with a new label that is formed by joining the labels of all the (labeled) arguments $t_1, \ldots, t_n$ of op;
- the remaining symbol occurrences of $C[t']$ that are not considered in the previous step inherit all the labels appearing in $C[op(t_1, \ldots, t_n)]$.

For example, by applying Definition 5.4.2, the addition of two natural numbers implemented through the built-in operator $+$ might be labeled as $+^\alpha(7^\beta, 8^\gamma) \to 15^{\beta\gamma}$.

5.4.3 Associative-Commutative Axioms

Let us finally consider an extended rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where B is a set of associativity (A) and commutativity (C) axioms that hold for some function symbols in Σ. As described in Section 5.2, an execution trace in $\mathcal{R}$ may contain rewrite steps modulo B that have the form $t =_B t' \to t''$, where $=_B$ is the congruence relation induced by the set of axioms B. Now, since B only contains associativity/commutativity (AC) axioms, terms can be represented by means of a single representative of their AC congruence class, called *AC canonical form* [18]. This representative is obtained by replacing nested occurrences of the same AC operator by a flattened argument list under a variadic symbol, whose elements are sorted by means of some linear ordering. In other words, if a function symbol f is declared to be associative, then the subterms rooted by f of any term t are flattened; and if f is also commutative, the subterms are sorted with respect to a fixed (internal) ordering [3].

The inverse process to flat transformation is unflat transformation, which is nondeterministic in the sense that it generates all the unflattended terms that are equivalent (modulo AC) to the flattened term.

For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the B-equivalence $f(b, f(f(b, a), c)) =_B f(f(b, c), f(a, b))$, we can represent it by using the "internal sequence" $f(b, f(f(b, a), c)) \rightarrow^*_{\mathit{flat}_B} f(a, b, b, c) \rightarrow^*_{\mathit{unflat}_B} f(f(b, c), f(a, b))$, where the first one corresponds to the *flattening* transformation sequence that obtains the AC canonical form, while the second one corresponds to the inverse, unflattening one.

These two processes are typically hidden inside the B-matching algorithms[4] that are used to implement rewriting modulo B.

The key idea for extending our labeling procedure in order to cope with B-equivalence $=_B$ is to exploit the flat transformation ($\rightarrow^*_{\mathit{flat}_B}$) and unflat transformation ($\rightarrow^*_{\mathit{unflat}_B}$) mentioned above. Without loss of generality, we assume that flat/unflat transformations are stable w.r.t. the lexicographic ordering over positions $\sqsubseteq$[5] (i.e., the relative ordering among the positions of multiple occurrences of a term is preserved).

This assumption allows us to trace back arguments of commutative operators, since multiple occurrences of the same symbol can be precisely identified.

Definition (AC Labeling.) Let f be an associative-commutative operator and B be the AC axioms for f. Consider the B-equivalence $t_1 =_B t_2$ and the corresponding (internal) flat/unflat transformation $\mathcal{T} : t_1 \rightarrow^*_{\mathit{flat}_B} s \rightarrow^*_{\mathit{unflat}_B} t_2$. Let L be an initial labeling for t_1. The labeling procedure for $t_1 =_B t_2$ is as follows.

1. (flattening) For each flattening transformation step $t_{|v} \rightarrow_{\mathit{flat}_B} t'_{|v}$ in $\mathcal{T}$ for the symbol f, a new label l_f is formed by joining all the labels attached to the symbol f in any position w of t^L such that $w = v$ or $w \geq v$, and every symbol on the path from v to w is f; then, label l_f is attached to the root symbol of $t'_{|v}$.

[3]Specifically, Maude uses the lexicographic order of symbols.

[4]See [15] (Section 4.8) for an in-depth discussion on matching and simplification modulo AC in Maude.

[5]The lexicographic ordering $\sqsubseteq$ is defined as follows: $\Lambda \sqsubseteq w$ for every position w, and given the positions $w_1 = i.w'_1$ and $w_2 = j.w'_2$, $w_1 \sqsubseteq w_2$ iff $i < j$ or ($i = j$ and $w'_1 \sqsubseteq w'_2$). Obviously, in a practical implementation of our technique, the considered ordering among the terms should be chosen to agree with the ordering considered by flat/unflat transformations in the RWL infrastructure.

2. (unflattening) For each unflattening transformation step $t_{|v} \to_{\mathit{unflat}_B} t'_{|v}$ in $\mathcal{T}$ for the symbol f, the label of the symbol f in the position v of t^L is attached to the symbol f in any position w of t' such that $w = v$ or $w \geq v$, and every symbol on the path from v to w is f.

3. The remaining symbol occurrences in t' that are not considered in cases 1 or 2 above inherit the label of the corresponding symbol occurrence in t.

Example Consider the transformation sequence

$$f(b, f(b, f(a, c))) \to^*_{\mathit{flat}_B} f(a, b, b, c) \to^*_{\mathit{unflat}_B} f(f(b, c), f(a, b))$$

by using Definition 5.4.3, the associated transformation sequence can be labeled as follows:

$$\begin{aligned} f^{\alpha}(b^{\beta}, f^{\gamma}(b^{\delta}, f^{\epsilon}(a^{\zeta}, c^{\eta}))) \to^*_{\mathit{flat}_B} \quad & f^{\alpha\gamma\epsilon}(a^{\zeta}, b^{\beta}, b^{\delta}, c^{\eta}) \to^*_{\mathit{unflat}_B} \\ & f^{\alpha\gamma\epsilon}(f^{\alpha\gamma\epsilon}(b^{\beta}, c^{\eta}), f^{\alpha\gamma\epsilon}(a^{\zeta}, b^{\delta})) \end{aligned}$$

Note that the original order between the two occurrences of the constant b is not changed by the flat/unflat transformations. For example, in the first term, b^{β} is in position 1 and b^{δ} is in position 2.1 with $1 \sqsubseteq 2.1$, whereas, in the last term, b^{β} is in position 1.1 and b^{δ} is in position 2.2 with $1.1 \sqsubseteq 2.2$.

Finally, observe that the methodology described in this section can be easily extended to deal with other equational attributes such as identity (U) by explicitly encoding the internal transformations performed by Maude via suitable rewrite rules.

5.4.4 Extended Soundness

Soundness of the backward trace slicing algorithm for the extended rewrite theories is established by the following theorem which properly extends Theorem 5.3.1. The proof of such an extension can be found in [34].

Theorem 5.4.1 *(extended soundness) Let $\mathcal{R} = (\Sigma, E, R)$ be an extended rewrite theory. Let $\mathcal{T}$ be an execution trace in the rewrite theory $\mathcal{R}$, and let $\mathcal{O}$ be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^{\bullet} : t_0^{\bullet} \xrightarrow{r_1} t_1^{\bullet} \dots \xrightarrow{r_n} t_n^{\bullet}$ be the corresponding trace slice w.r.t. $\mathcal{O}$. Then, for any concretization t'_0 of $t_0^{\bullet}$, it holds that $\mathcal{T}' : t'_0 \xrightarrow{r_1} t'_1 \dots \xrightarrow{r_n} t'_n$ is an execution trace in $\mathcal{R}$, and $t_i^{\bullet} \propto t'_i$, for $i = 1, \dots, n$.*

5.5 Experimental Evaluation

We have developed a prototype implementation of our slicing methodology which is publicly available at `http://users.dsic.upv.es/grupos/elp/soft.html`. The implementation is written in Maude and consists of approximately 800 lines of code.

Maude is a high-performance, reflective language that supports both equational and rewriting logic programming, which is particularly suitable for developing domain-specific applications [19, 20]. The reflection capabilities of Maude allow metalevel computations in RWL to be handled at the object-level. This facility allows us to easily manipulate computation traces of Maude itself and eliminate the irrelevant contents by implementing the backward slicing procedures that we have defined in this chapter. Using reflection to implement the slicing tool has one important additional advantage, namely, the ease to rapidly integrate the tool within the Maude formal tool environment [17], which is also developed using reflection.

The prototype takes a Maude execution trace and a slicing criterion as input, and delivers a trace slice together with some quantitative information regarding the reduction achieved. The outcome is formatted in HTML, so it can be easily inspected by means of a Web browser.

In order to evaluate the usefulness of our approach, we benchmarked our prototype with several examples of Maude applications:

War of Souls (WoS). WoS is a nontrivial producer/consumer example that is modeled as a game in which an angel and a daemon fight to conquer the souls of human beings. Basically, when a human being passes away, his/her soul is sent to heaven or to hell depending on his/her faith as well as the strength of the angel and the daemon in play.

Fault-Tolerant Communication Protocol (FTCP). FTCP is a Maude specification borrowed from [28] that models a fault-tolerant, client-server communication protocol. There can be many clients and many servers, where a server can serve many clients; however, each client communicates with a single server. Also, the communication environment might be faulty —that is, messages can arrive out of order, can be duplicated, or can be lost.

Web-TLR: the Web application verifier. Web-TLR [8, 4] is a software tool designed for model-checking real-size Web applications (Web-mailers, Electronic forums, *etc.*) which is based on rewriting logic. Web applications are expressed as rewrite theories which can be formally verified by using the Maude built-in LTL(R) model checker [9].

A detailed description of these Maude applications and the Maude code are available at the URL mentioned above.

We have tested our tool on some execution traces which were generated by the Maude applications described above by imposing different slicing criteria. For each application, we considered two execution traces that were sliced using two different criteria. Table 5.1 summarizes the results we achieved.

As for the `WoS` example, we have chosen criteria that allow us to backtrace both the values produced and the entities in play — e.g., the criterion $\texttt{WoS}.\mathcal{T}_1.O_2$ isolates the angel and daemon behaviours along the trace $\mathcal{T}_1$.

Example	Example trace	Original trace size	Slicing criterion	Sliced trace size	% *reduction*
WoS	WoS.$\mathcal{T}_1$	776	WoS.$\mathcal{T}_1.O_1$	201	74.10%
			WoS.$\mathcal{T}_1.O_2$	138	82.22%
	WoS.$\mathcal{T}_2$	997	WoS.$\mathcal{T}_2.O_1$	404	58.48%
			WoS.$\mathcal{T}_2.O_2$	174	82.55%
FTCP	FTCP.$\mathcal{T}_1$	2445	FTCP.$\mathcal{T}_1.O_1$	895	63.39%
			FTCP.$\mathcal{T}_1.O_2$	698	71.45%
	FTCP.$\mathcal{T}_2$	2369	FTCP.$\mathcal{T}_2.O_1$	364	84.63%
			FTCP.$\mathcal{T}_2.O_2$	707	70.16%
Web-TLR	Web-TLR.$\mathcal{T}_1$	31829	Web-TLR.$\mathcal{T}_1.O_1$	1949	93.88%
			Web-TLR.$\mathcal{T}_1.O_2$	1598	94.97%
	Web-TLR.$\mathcal{T}_2$	72098	Web-TLR.$\mathcal{T}_2.O_1$	9090	87.39%
			Web-TLR.$\mathcal{T}_2.O_2$	7119	90.13%

Table 5.1: Summary of the reductions achieved.

Execution traces in the FTCP example represent client-server interactions. In this case, the chosen criteria aim at (i) isolating a server and/or a client in a scenario which involves multiple servers and clients (FTCP.$\mathcal{T}_2.O_1$), and (ii) tracking the response generated by a server according to a given client request (FTCP.$\mathcal{T}_1.O_1$).

In the last example, we have used Web-TLR to verify two LTL(R) properties of a Webmail application. The considered execution traces are much bigger for this program, and correspond to the counterexamples produced as outcome by the built-in model-checker of Web-TLR. In this case, the chosen criteria allow us to monitor the messages exchanged by the Web browsers and the Webmail server, as well as to focus our attention on the data structures of the interacting entities (e.g., browser/server sessions, server database).

For each criterion, Table 5.1 shows the size of the original trace and that of the computed trace slice, both measured as the length of the corresponding string. The *%reduction* column shows the percentage of reduction achieved. These results are very encouraging, and show an impressive reduction rate (up to $\sim 95\%$). Actually, sometimes the trace slices are small enough to be easily inspected by the user, who can restrict her attention to the part of the computation she wants to observe getting rid of those data which are useless or even noisy w.r.t. the considered slicing criterion.

5.6 Related Work

Our backward tracing relation (Definition 5.3.1) extends a previous tracing relation that was formalized in [12] for orthogonal TRSs. In [12], a label is formed from atomic labels by using the operations of sequence concatenation and underlining, which are used to record every rule application (e.g., a, b, ab, $\underline{\underline{ab}c}d$, are labels). Collapsing rules are simply avoided by coding them away. This is done by replacing each collapsing rule $\lambda \to x$ with the rule $\lambda \to \varepsilon(x)$, where ε is a unary dummy symbol. Then, in order to lift the rewrite relation to terms containing ϵ occurrences, infinitely many new extra-rules are added that are built by saturating all left-hand sides with $\varepsilon(x)$. In contrast to [12], we use a more sophisticated notion of labeling, where composite labels are interpreted as sets of atomic labels, which allows us to deal with collapsing as well as nonleft-linear rules in an effective way.

Tracing techniques have been extensively used in functional programming to implement debugging tools [14]. For instance, Hat [39] is an interactive system that enables exploring a computation backwards, starting at the program output or an error message (with which the computation aborted). Backward tracing in Hat is carried out by navigating a redex trail —that is, a graph-like data structure that records dependencies among function calls. Our approach is somehow lighter, since it does not require the construction of any complex, auxiliary data structure.

The work that is most closely related to ours is [21]. [21] formalizes a notion of dynamic dependence among symbols by means of contexts and studies its application to program slicing of TRSs that may include collapsing as well as nonleft-linear rules. Both the *creating* and the *created* contexts associated with a reduction (i.e., the minimal subcontext that is needed to match the left-hand side of a rule and the minimal context that is "constructed" by the right-hand side of the rule, respectively) are tracked. Intuitively, these concepts are similar to our notions of redex and contractum patterns. The main differences with respect to our work are as follows. First, in [21] the slicing is given as a context, while we consider term slices. Second, the slice is obtained only on the first term of the sequence by the transitive and reflexive closure of the dependence relation, while we slice the whole execution trace, step by step. Obviously, their notion of slice is smaller, but we think that our approach can be more useful for trace analysis and program debugging.

An extension of the method is described in [38], which provides a generic definition of labeling that works not only for orthogonal TRSs as is the case of [21] but for the wider class of all left-linear TRSs. Specifically, [38] describes a methodology of static and dynamic tracing which is mainly based on the notion of *sample of a traced proof term* —i.e., a pair (μ, P) that records a rewrite step $\mu = s \to t$, and a set P of reachable positions in t from a set of observed positions in s. The tracing proceeds forward, while ours employs a backward strategy which is particularly convenient for trace analysis.

Finally, [21] and [38] apply to TRSs whereas we deal with the richer framework of RWL that considers equations and equational axioms, namely rewriting modulo equational theories.

6 Debugging of Web Applications with Web-TLR

Web-TLR is a Web verification engine that is based on the well-established *Rewriting Logic–Maude/LTLR* tandem for Web system specification and model-checking. In Web-TLR, Web applications are expressed as rewrite theories that can be formally verified by using the Maude built-in LTLR model checker. Whenever a property is refuted, the tool delivers a counterexample trace that reveals an undesired, erroneous navigation sequence. Unfortunately, the analysis (or even the simple inspection) of such counterexamples may be unfeasible because of the size and complexity of the traces under examination. In this chapter, we delve into a debugging facility of Web-TLR especially focused on tackling these cases. This facility is based on the backward trace-slicing technique described in Chapter 5 for rewriting logic theories, which allows the pieces of information that we are interested into to be traced back through inverted rewrite sequences. The slicing process drastically simplifies the computation trace by dropping useless data that do not influence the final result. By using this facility, 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.

6.1 Introduction

Model checking is a powerful and efficient method for finding flaws in hardware designs, business processes, object-oriented software, and hypermedia applications. One remaining major obstacle to a broader application of model checking is its limited usability for non-experts. In the case of specification violation, it requires much effort and insight to determine the root cause of errors from the counterexamples generated by model checkers [41].

Web-TLR [8] is a software tool designed for model-checking Web applications that is based on rewriting logic [26]. Web applications are expressed as rewrite theories that can be formally verified by using the Maude built-in LTLR model-checker [9]. Whenever a property is refuted, a counterexample trace is delivered that reveals an undesired, erroneous navigation sequence. Web-TLR is endowed with support for user interaction in Chapter 4 (and [4]), including the successive exploration of error scenarios according to the user's interest by means of a slideshow facility that allows the user to incrementally expand the model states to the desired level of detail, thus avoiding the rather tedious task of inspecting the textual representation of the system. Although this facility helps the user to keep the overview of the model, the analysis (or even the simple inspection) of the delivered counterexamples is still unfeasible because of the size and complexity of the traces under examination. This is particularly serious in the rewriting logic context of Web-TLR because Web specifications may contain equations and algebraic laws that are internally used to simplify the system states, and temporal LTLR formulae may contain function symbols that are interpreted in the considered algebraic theory. All of this results in execution traces that may be difficult to understand for users who are not acquainted with rewriting logic technicalities.

This chapter aims at improving the understandability of the counterexamples generated by Web-TLR. This is achieved by means of a complementary Web debugging facility that supports both the efficient manipulation of counterexample traces and the interactive exploration of error scenarios. This facility is based on a backward trace-slicing technique for rewriting logic theories formalized in Chapter 5 (and [5]) that allows the pieces of information that we are interested into to be traced back through the inverse rewrite sequence. The slicing process drastically simplifies the computation trace by dropping useless data that do not influence the final result. We provide a convenient, handy notation for specifying the slicing criterion that is successively propagated backwards at locations selected by the user. Preliminary experiments reveal that the novel slicing facility of the extended version of Web-TLR is fast enough to enable smooth interaction and helps the users to locate the cause of errors accurately without overwhelming them with bulky information. By using the slicing facility, the Web engineer can focus on the relevant fragments of the failing application, which greatly reduces the manual debugging effort.

Plan of the chapter. In Section 6.2 we motivate the need for an improved Web-TLR and introduce a novel slicing facility. Section 6.3 introduces a pattern-matching language that allows the selection of the slicing criterion. In Section 6.4, we illustrate our methodology for interactive analysis of counterexample traces and debugging of Web Applications.

6.2 Extending the Web-TLR System

Web-TLR is a model-checking tool that implements the theoretical framework of [8]. The Web-TLR system is available online via its friendly Web interface at `http://users.dsic.upv.es/grupos/elp/soft.html`. The Web interface frees users from having to install applications on their local computer and hides unnecessary technical details of the tool operation. After introducing the (or customizing a default) Maude specification of a Web application, together with an initial Web state st_0 and the LTLR formula φ to be verified, φ can be automatically checked at st_0. Once all inputs have been entered in the system, we can automatically check the property by just clicking the button Check, which invokes the Maude built-in operator tlr check[9] that supports model checking of LTLR formulas in rewrite theories. If the property is not satisfied, an interactive slideshow that illustrates the corresponding counterexample (expressed in the form of an execution trace) is generated. The slideshow supports both forward and backward navigation through the execution trace and combines a graphical representation of the application's navigation model with a detailed textual description of the Web states.

Although Web-TLR provides a complete picture of both, the application model and the generated counterexample, this information is hardly exploitable for debugging Web applications. Actually, the graphical representation provides a very coarse-grained model of the application's dynamics, while the textual description conveys too much information (e.g., see Figure 3.4). Therefore, in several cases both representations may result in limited use.

In order to assist Web engineers in the debugging task, we extend Web-TLR by including a trace-slicing technique whose aim is to reduce the amount of information recorded by the textual description of counterexamples. Roughly speaking, this technique (originally described in [5]) consists in tracing back, along an execution trace, all the symbols of a (Web) state that are of interest (target symbols), while useless data are discarded. The basic idea is to take a Rewriting Logic execution trace and traverse it backwards in order to filter out data that are definitely related to the wrong behavior. This way, we can focus our attention on the most critical parts of the trace, which are eventually responsible for the erroneous application's behaviour. It is worth noting that our trace slicing procedure is sound in the sense that, given an execution trace $\mathcal{T}$, it automatically computes a trace slice of $\mathcal{T}$ that includes all the information needed to produce the target symbols of $\mathcal{T}$ we want to observe. In other words, there is no risk that our tool eliminates data from the original execution trace $\mathcal{T}$ which are indeed relevant w.r.t. the considered target symbols. Soundness of backward trace slicing has been formally proven in [6].

We have implemented the backward trace-slicing technique as a stand-alone application written in Maude that can be used to simplify general Maude traces (e.g., the ones printed when the trace is set on in a standard rewrite). Furthermore, we have coupled the on-line Web-TLR system with the slicing tool in order to optimize the counterexample traces delivered by Web-TLR. To achieve this, the external slicing routine is fed with the given counterexample, the selected Web state s where the backward-slicing process is required to start, and the slicing criterion for s —that is, the symbols of s we want to trace back. It is worth noting that, for model checking Web applications with Web-TLR, we have developed a specially–tailored, handy filtering notation that allows us to easily specify the slicing criterion and automatically select the desired information by exploiting the powerful, built-in pattern-matching mechanism of Rewriting Logic. The outcome of the slicing process is a sliced version of the textual description of the original counterexample trace which facilitates the interactive exploration of error scenarios when debugging Web applications.

6.3 Filtering Notation

In order to select the relevant information to be traced back, we introduce a simple, pattern-matching filtering language that frees the user from explicitly introducing the specific positions of the Web state that s/he wants to observe [1]. Roughly speaking, the user introduces an information pattern p that has to be detected inside a given Web state s. The information matching p that is recognized in s, is then identified by pattern matching and is kept in $s^\bullet$, whereas all other symbols of s are considered irrelevant and then removed. Finally, the positions of the Web state where the relevant information is located are obtained from $s^\bullet$. In other words, the slicing criterion is defined by the set of positions where the relevant information is located within the state s that we are observing and is automatically generated by pattern-matching the information pattern against the Web state s.

The filtering language allows us to define the relevant information as follows: (*i*) by giving the name of an operator (or constructor) or a substring of it; and (*ii*) by using the question mark "?" as a wildcard character that indicates the position where the information is considered relevant. On the other hand, the irrelevant information can be declared by using the wildcard symbol "_" as a placeholder for uninteresting arguments of an operator.

Let us illustrate this filtering notation by means of a rather intuitive example. Let us assume that the electronic forum application allows one to list some data about the available topics. Specifically, the following term t specifies the names of the topics available in our electronic forum together with the total number of posted messages for each topic.

$$\begin{array}{l}\mathsf{topic\_info}(\mathsf{topic}(\mathsf{astronomy}, \sharp\mathsf{posts}(520)), \mathsf{topic}(\mathsf{stars}, \sharp\mathsf{posts}(58)),\\ \qquad\qquad \mathsf{topic}(\mathsf{astrology}, \sharp\mathsf{posts}(20)), \mathsf{topic}(\mathsf{telescopes}, \sharp\mathsf{posts}(290))\ )\end{array}$$

Then, the pattern $\mathsf{topic}(\mathsf{astro}, \sharp\mathsf{posts}(?))$ defines a slicing criterion that allows us to observe the topic name as well as the total number of messages for all topics whose name includes the word astro. Specifically, by applying such a pattern to the term t, we obtain the following term slice

$$\mathsf{topic\_info}(\mathsf{topic}(\mathsf{astronomy}, \sharp\mathsf{posts}(520)), \bullet, \mathsf{topic}(\mathsf{astrology}, \sharp\mathsf{posts}(20)), \bullet)$$

which ignores the information related to the topics stars and telescopes, and induces the slicing criterion

$$\{\Lambda.1.1,\ \Lambda.1.2.1,\ \Lambda.3.1,\ \Lambda.3.2.1\}.$$

[1]Terms are viewed as labeled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term. The empty sequence Λ denotes the root position of the term.

Note that we have introduced the fresh symbol • to approximate any output information in the term that is not relevant with respect to a given pattern.

6.4 A Debugging Session with Web-TLR

In this section, we illustrate our methodology for interactive analysis of counterexample traces and debugging of Web applications.

Let us consider an initial state that consists of two administrator users whose identifiers are $\mathsf{bidAlfred}$ and $\mathsf{bidAnna}$, respectively. Let us also recall the mutual exclusion Property 3.1 of Chapter 3

$$\Box\neg(\mathsf{curPage}(\mathsf{bidAlfred},\mathsf{Admin})\wedge\mathsf{curPage}(\mathsf{bidAnna},\mathsf{Admin}))$$

which states that "no two administrators can access the administration page simultaneously". Note that the predicate state $\mathsf{curPage}(\mathsf{bidAlfred},\mathsf{Admin})$ holds when the user $\mathsf{bidAlfred}$ logs into the Admin page (a similar interpretation is given to predicate $\mathsf{curPage}(\mathsf{bidAnna},\mathsf{Admin})$). By verifying the above property with Web-TLR, we get a huge counterexample that proves that the property is not satisfied. The trace size weighs around $190kb$.

In the following, we show how the considered Web application can be debugged using Web-TLR. First of all, we specify the slicing criterion to be applied on the counterexample trace. This is done by using the wildcard notation on the terms introduced in Section 6.3. Then, the slicing process is invoked and the resulting trace slice is produced. Finally, we analyze the trace slice and outline a methodology that helps the user to locate the errors.

Slicing Criterion

The slicing criterion represents the information that we want to trace back through the execution trace $\mathcal{T}$ that is produced as the outcome of the Web-TLR model-checker.

For example, consider the final Web state s shown in Figure 3.4. In this Web state, the two users, `bidAlfred` and `bidAnna`, are logged into the `Admin` page. Therefore, the considered mutual exclusion property has been violated. Let us assume that we want to diagnose the erroneous pieces of information within the execution trace $\mathcal{T}$ that produce this bug. Then, we can enter the following information pattern as input,

$$\mathsf{B}(?,\_,?,\_,\_,\_,\_,\_,\_)$$

where the operator B restricts the search of relevant information inside the browser data structures, the first question symbol ? represents that we are interested in tracing the user identifiers, and the second one calls for the Web page name. Thus, by applying the considered information pattern to the Web state s, we obtain the slicing criterion $\{\Lambda.1.1.1,\ \Lambda.1.1.3,\ \Lambda.1.2.1,\ \Lambda.1.2.3\}$ and the corresponding sliced state

$$s^\bullet = [\mathsf{B}(\mathsf{bibAlfred}, \bullet, \mathsf{Admin}, \bullet, \bullet, \bullet, \bullet, \bullet, \bullet) : \mathsf{B}(\mathsf{bibAnna}, \bullet, \mathsf{Admin}, \bullet, \bullet, \bullet, \bullet, \bullet, \bullet)][\bullet][\bullet]$$

Note that $\Lambda.1.1.1$ and $\Lambda.1.2.1$ are the positions in $s^\bullet$ of the user identifiers `bidAlfred` and `bidAnna`, respectively, and $\Lambda.1.1.3$ and $\Lambda.1.2.3$ are the positions in $s^\bullet$ that indicate that the users are logged into the `Admin` page.

Trace Slice

Let us consider the counterexample execution trace $\mathcal{T} = s_0 \to s_1 \to \ldots \to s_n$, where $s_n = s$. The slicing technique proceeds backwards, from the observable state s_n to the initial state s_0, and for each state s_i recursively generates a sliced state $s_i^\bullet$ that consists of the relevant information with respect to the slicing criterion.

By running the backward-slicing tool with the execution trace $\mathcal{T}$ and the slicing criterion given above as input, we get the trace slice $\mathcal{T}^\bullet$ as outcome, where useless data that do not influence the final result are discarded. Figure 6.1 shows a part of the trace slice $\mathcal{T}^\bullet$.

It is worth observing that the slicing process greatly reduces the size of the original trace $\mathcal{T}$, and allows us to center on those data that are likely to be the source of an erroneous behavior.

Let $|\,\mathcal{T}\,|$ be the size of the trace $\mathcal{T}$, namely the sum of the number of symbols of all trace states. In this specific case, the size reduction that is achieved on the the subsequence $s_{(n-6)} \ldots s_n$ of $\mathcal{T}$, in symbols $\mathcal{T}_{[s_{(n-6)} .. s_n]}$ is:

$$\frac{|\,\mathcal{T}^\bullet_{[s^\bullet_{(n-6)} \cdots s^\bullet_n]}\,|}{|\,\mathcal{T}_{[s_{(n-6)} \cdots s_n]}\,|} = \frac{121}{1458} = 0.083 \text{ (i.e., a reduction of 91.7\%)}$$

Trace Slice Analysis

Let us analyze the information recorded in the trace slice $\mathcal{T}^\bullet$. In order to facilitate understanding, the main symbols involved in the description are underlined in Figure 6.1.

- The sliced state $s_n^\bullet$ is the observable state that records only the relevant information defined by the slicing criterion.

- The slice state $s_{n-1}^\bullet$ is obtained from $s_n^\bullet$ by the flat/unflat transformation.

- In the sliced state $s_{n-2}^\bullet$, the communication channel contains a response message for the user `bidAlfred`. This response message enables the user `bidAlfred` to log into the `Admin` page. Note that the identifier `tidAlfred` occurs in the Web state. This identifier signals the open window that the

$$\mathcal{T}^\bullet = s_0^\bullet \ldots \;\to s_{n-6}^\bullet \xrightarrow{\mathit{ScriptEval}} s_{n-5}^\bullet \xrightarrow{\mathit{flat/unflat}} s_{n-4}^\bullet \xrightarrow{\mathit{ResIni}} s_{n-3}^\bullet$$

$$\xrightarrow{\mathit{flat/unflat}} s_{n-2}^\bullet \xrightarrow{\mathit{ResFin}} s_{n-1}^\bullet \xrightarrow{\mathit{flat/unflat}} s_n^\bullet$$

where

$$\begin{aligned}
s_n^\bullet = {} & [\mathtt{B}(\underline{\mathtt{bidAlfred}}, *, \underline{\mathtt{Admin}}, *, *, *, *, *, *) : \mathtt{B}(\mathtt{bidAnna}, *, \mathtt{Admin}, *, *, *, *, *, *)] * [*][*] \\
s_{n-1}^\bullet = {} & [\mathtt{B}(\mathtt{bidAnna}, *, \mathtt{Admin}, *, *, *, *, *, *) : \mathtt{B}(\underline{\mathtt{bidAlfred}}, *, \underline{\mathtt{Admin}}, *, *, *, *, *, *)] * [*][*] \\
s_{n-2}^\bullet = {} & [\mathtt{B}(\mathtt{bidAnna}, *, \mathtt{Admin}, *, *, *, *, *, *) : \mathtt{B}(\underline{\mathtt{bidAlfred}}, \underline{\mathtt{tidAlfred}}, *, *, *, *, *, *, 1)] * \\
& [\underline{\mathtt{S2B}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, \mathtt{Admin}, *, *, 1)} : *][*] \\
s_{n-3}^\bullet = {} & [\mathtt{B}(\underline{\mathtt{bidAlfred}}, \underline{\mathtt{tidAlfred}}, *, *, *, *, *, *, 1) : \mathtt{B}(\mathtt{bidAnna}, *, \mathtt{Admin}, *, *, *, *, *, *)] * \\
& [* : \underline{\mathtt{S2B}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, \mathtt{Admin}, *, *, 1)}][*] \\
s_{n-4}^\bullet = {} & [\mathtt{B}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, *, *, *, *, *, *, 1) : \mathtt{B}(\mathtt{bidAnna}, *, \mathtt{Admin}, *, *, *, *, *, *)] * [*] \\
& [\mathtt{S}(*, * : \mathtt{us}(\mathtt{bidAlfred}, *), *, (\mathtt{rm}(\underline{\mathtt{S2B}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, \mathtt{Admin}, *, *, 1)}, *, *) : *), *] \\
s_{n-5}^\bullet = {} & [\mathtt{B}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, *, *, *, *, *, *, 1) : \mathtt{B}(\mathtt{bidAnna}, *, \mathtt{Admin}, *, *, *, *, *, *)] * [*] \\
& [\mathtt{S}(*, \mathtt{us}(\mathtt{bidAlfred}, *) : *, *, (* : \underline{\mathtt{evalScript}}(\mathtt{WEB\text{-}APP}, \mathtt{SESSION}, \\
& \underline{\mathtt{B2S}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, \mathtt{Admin?query\text{-}empty}, 1)}, \mathtt{DB})), *)] \\
s_{n-6}^\bullet = {} & [\mathtt{B}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, *, *, *, *, *, *, 1) : \mathtt{B}(\mathtt{bidAnna}, *, \mathtt{Admin}, *, *, *, *, *, *)] * [*] \\
& [\mathtt{S}(\mathtt{WEB\text{-}APP}, (* : \mathtt{us}(\mathtt{bidAlfred}, \mathtt{SESSION})), \\
& \underline{\mathtt{B2S}(\mathtt{bidAlfred}, \mathtt{tidAlfred}, \mathtt{Admin?query\text{-}empty}, 1)} : *, *, \mathtt{DB}]
\end{aligned}$$

Figure 6.1: Trace slice $\mathcal{T}^\bullet$.

response message refers to. Also, the number `1` that occurs in the sliced state $s_{n-2}^\bullet$ represents the *ack* (acknowledgement) of the response message. Finally, the reduction from $s_{n-2}^\bullet$ to $s_{n-3}^\bullet$ corresponds again to a flat/unflat transformation.

- In the sliced state $s_{n-4}^\bullet$, we can see the response message stored in the server that is ready to be sent, whereas, in the server configuration of the sliced state $s_{n-5}^\bullet$, the operator `evalScript` occurs. This operator takes the Web application (`WEB-APP`), the user session (`SESSION`), the request message, and the database (`DB`) as input. The request message contains the query string that has been sent by the user `bidAlfred` to ask for admission into the `Admin` page. Observe that the response message that is shown in the slice state $s_{n-4}^\bullet$ is the one given as the outcome of the evaluation of the operator `evalScript` in the sliced state $s_{n-5}^\bullet$.

- Finally, the sliced state $s_{n-6}^\bullet$ shows the request message waiting to be evaluated.

Note that the outcome delivered by the operator `evalScript`, when the script α_{admin} is evaluated, is not what the user would have expected, since it allows the user to log into the Admin page, which leads to the violation of the considered property. This identifies the script α_{admin} as the script that is responsible for the error. Note that this conclusion is correct because α_{admin} has not implemented a mutual exclusion control (see Figure 3.3). A snapshot of WEB-TLR that shows the slicing process is given in Figure 6.2.

This bug can be fixed by introducing the necessary control for mutual exclusion as follows. First, a continuation ($\text{"}\mathsf{adminPage}\text{"} = \text{"}\mathsf{busy}\text{"}) \rightarrow \mathsf{Index}?[\emptyset]$) is added to the Admin page, and the α_{admin} is replaced by a new Web script that checks whether there is another user in the Admin page. In the case when the Admin page is busy because it is being accessed by a given user, any other user is redirected to the Index page. If the Admin page is free, the user asking for permission to enter is authorized to do so (and the page gets locked). Furthermore, the control for unlocking the Admin page is added at the beginning of the script α_{index}. Hence, the fixed Web scripts are as follows:

$$\boxed{\begin{aligned}\mathsf{P}_{\mathsf{Admin}} = \ & (\mathsf{Admin}, \alpha_{\mathsf{admin}},\\ & \underline{\{(\text{"}\mathsf{adminPage}\text{"} = \text{"}\mathsf{busy}\text{"}) \rightarrow \mathsf{Index}?[\emptyset])\}},\\ & \{(\emptyset \rightarrow (\mathsf{Index}?[\emptyset]))\})\end{aligned}}$$

where the new α_{Admin} is:

$\boxed{\alpha_{\mathsf{admin}}}$

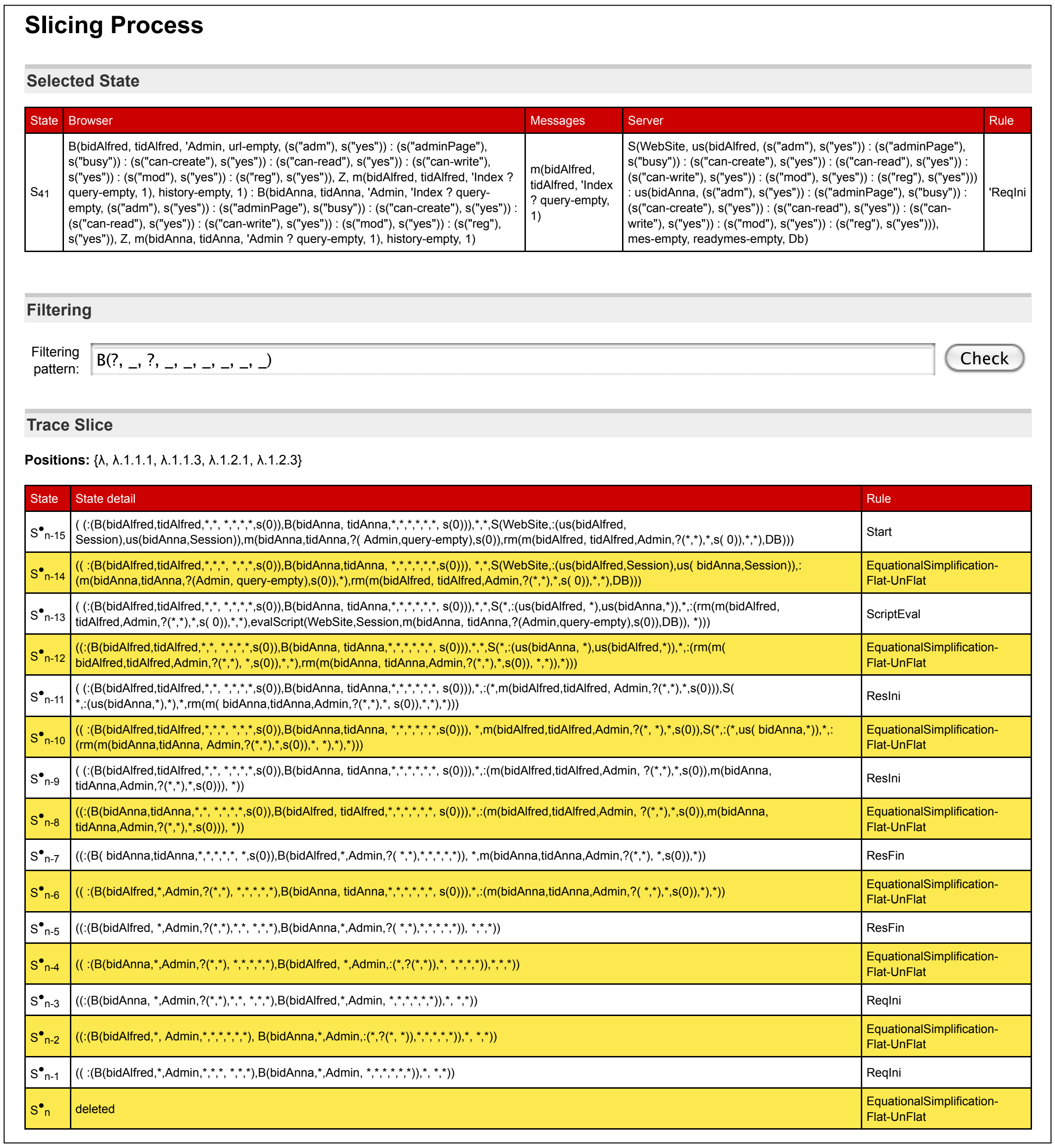
```
'u := getSession("user") ;
'adm := selectDB("adminPage") ;
if ( 'adm != 'u) then
setSession("adminPage", "busy")
else
setSession("adminPage", "free")) ;
updateDB( "adminPage", 'u) )
fi
```

and the piece of code that patches α_{index} is:

$\boxed{\alpha_{\mathsf{index}}}$

```
'adm := getSession("adminPage") ;
if ('adm = "free") then
updateDB("adminPage", "free")
fi ;
...
```

Finally, by using WEB-TLR again we get the outcome "Property is fulfilled, no counter-example given", which guarantees that now the Web application satisfies Property 3.1. Figure 6.3 shows a snapshot of WEB-TLR for the case when a property is fulfilled.



Slicing Process

Selected State

State	Browser	Messages	Server	Rule
S_{41}	B(bidAlfred, tidAlfred, 'Admin, url-empty, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes")), Z, m(bidAlfred, tidAlfred, 'Index ? query-empty, 1), history-empty, 1) : B(bidAnna, tidAnna, 'Admin, 'Index ? query-empty, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes")), Z, m(bidAnna, tidAnna, 'Admin ? query-empty, 1), history-empty, 1)	m(bidAlfred, tidAlfred, 'Index ? query-empty, 1)	S(WebSite, us(bidAlfred, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes"))) : us(bidAnna, (s("adm"), s("yes")) : (s("adminPage"), s("busy")) : (s("can-create"), s("yes")) : (s("can-read"), s("yes")) : (s("can-write"), s("yes")) : (s("mod"), s("yes")) : (s("reg"), s("yes"))), mes-empty, readymes-empty, Db)	'ReqIni

Filtering

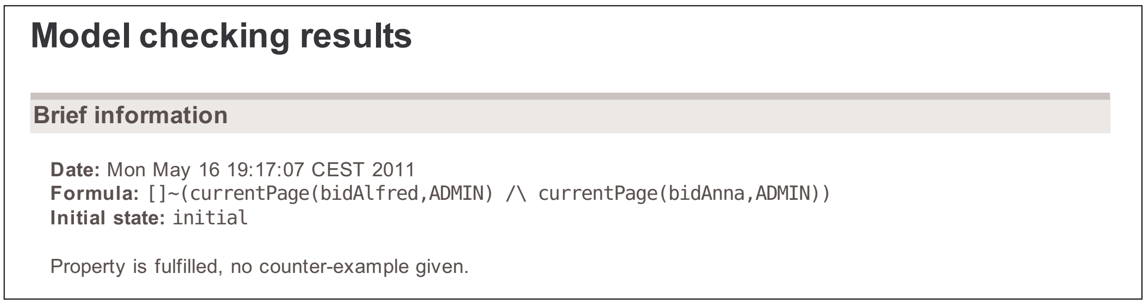
Filtering pattern: B(?, _, ?, _, _, _, _, _, _)

Trace Slice

Positions: {λ, λ.1.1.1, λ.1.1.3, λ.1.2.1, λ.1.2.3}

State	State detail	Rule
$S^{\bullet}_{n-15}$	((:(B(bidAlfred,tidAlfred,*,*, *,*,*,*,s(0)),B(bidAnna, tidAnna,*,*,*,*,*,*, s(0))),*,*,S(WebSite,:(us(bidAlfred, Session),us(bidAnna,Session)),m(bidAnna,tidAnna,?(Admin,query-empty),s(0)),rm(m(bidAlfred, tidAlfred,Admin,?(*,*),*,s(0)),*,*),DB)))	Start
$S^{\bullet}_{n-14}$	((:(B(bidAlfred,tidAlfred,*,*,*, *,*,*,s(0)),B(bidAnna,tidAnna, *,*,*,*,*,*,s(0))), *,*,S(WebSite,:(us(bidAlfred,Session),us(bidAnna,Session)),:(m(bidAnna,tidAnna,?(Admin, query-empty),s(0)),*),rm(m(bidAlfred, tidAlfred,Admin,?(*,*),*,s(0)),*,*),DB)))	EquationalSimplification-Flat-UnFlat
$S^{\bullet}_{n-13}$	((:(B(bidAlfred,tidAlfred,*,*, *,*,*,*,s(0)),B(bidAnna, tidAnna,*,*,*,*,*,*, s(0))),*,*,S(*,:(us(bidAlfred, *),us(bidAnna,*)),*,:(rm(m(bidAlfred, tidAlfred,Admin,?(*,*),*,s(0)),*,*),evalScript(WebSite,Session,m(bidAnna, tidAnna,?(Admin,query-empty),s(0)),DB)), *)))	ScriptEval
$S^{\bullet}_{n-12}$	((:(B(bidAlfred,tidAlfred,*,*, *,*,*,*,s(0)),B(bidAnna, tidAnna,*,*,*,*,*,*, s(0))),*,*,S(*,:(us(bidAnna, *),us(bidAlfred,*)),*,:(rm(m(bidAlfred,tidAlfred,Admin,?(*,*), *,s(0)),*,*),rm(m(bidAnna, tidAnna,Admin,?(*,*),*,s(0)), *,*)),*)))	EquationalSimplification-Flat-UnFlat
$S^{\bullet}_{n-11}$	((:(B(bidAlfred,tidAlfred,*,*, *,*,*,*,s(0)),B(bidAnna, tidAnna,*,*,*,*,*,*, s(0))),*,:(*,m(bidAlfred,tidAlfred, Admin,?(*,*),*,s(0))),S(*,:(us(bidAnna,*),*),*,rm(m(bidAnna,tidAnna,Admin,?(*,*),*, s(0)),*,*),*)))	ResIni
$S^{\bullet}_{n-10}$	((:(B(bidAlfred,tidAlfred,*,*,*, *,*,*,s(0)),B(bidAnna,tidAnna, *,*,*,*,*,*,s(0))), *,m(bidAlfred,tidAlfred,Admin,?(*, *),*,s(0)),S(*,:(*,us(bidAnna,*)),*,:(rm(m(bidAnna,tidAnna, Admin,?(*,*),*,s(0)),*, *),*),*)))	EquationalSimplification-Flat-UnFlat
$S^{\bullet}_{n-9}$	((:(B(bidAlfred,tidAlfred,*,*, *,*,*,*,s(0)),B(bidAnna, tidAnna,*,*,*,*,*,*, s(0))),*,:(m(bidAlfred,tidAlfred,Admin, ?(*,*),*,s(0)),m(bidAnna, tidAnna,Admin,?(*,*),*,s(0))), *))	ResIni
$S^{\bullet}_{n-8}$	((:(B(bidAnna,tidAnna,*,*, *,*,*,*,s(0)),B(bidAlfred, tidAlfred,*,*,*,*,*,*, s(0))),*,:(m(bidAlfred,tidAlfred,Admin, ?(*,*),*,s(0)),m(bidAnna, tidAnna,Admin,?(*,*),*,s(0))), *))	EquationalSimplification-Flat-UnFlat
$S^{\bullet}_{n-7}$	((:(B(bidAnna,tidAnna,*,*,*,*,*, *,s(0)),B(bidAlfred,*,Admin,?(*,*),*,*,*,*,*)), *,m(bidAnna,tidAnna,Admin,?(*,*), *,s(0)),*))	ResFin
$S^{\bullet}_{n-6}$	((:(B(bidAlfred,*,Admin,?(*,*), *,*,*,*,*),B(bidAnna, tidAnna,*,*,*,*,*,*, s(0))),*,:(m(bidAnna,tidAnna,Admin,?(*,*),*,s(0)),*),*))	EquationalSimplification-Flat-UnFlat
$S^{\bullet}_{n-5}$	((:(B(bidAlfred, *,Admin,?(*,*),*,*, *,*,*),B(bidAnna,*,Admin,?(*,*),*,*,*,*,*)), *,*,*))	ResFin
$S^{\bullet}_{n-4}$	((:(B(bidAnna,*,Admin,?(*,*), *,*,*,*,*),B(bidAlfred, *,Admin,:(*,?(*,*)),*, *,*,*,*)),*,*,*))	EquationalSimplification-Flat-UnFlat
$S^{\bullet}_{n-3}$	((:(B(bidAnna, *,Admin,?(*,*),*,*, *,*,*),B(bidAlfred,*,Admin, *,*,*,*,*,*)),*, *,*))	ReqIni
$S^{\bullet}_{n-2}$	((:(B(bidAlfred,*, Admin,*,*,*,*,*,*), B(bidAnna,*,Admin,:(*,?(*, *)),*,*,*,*,*)),*, *,*))	EquationalSimplification-Flat-UnFlat
$S^{\bullet}_{n-1}$	((:(B(bidAlfred,*,Admin,*,*,*, *,*,*),B(bidAnna,*,Admin, *,*,*,*,*,*)),*, *,*))	ReqIni
$S^{\bullet}_{n}$	deleted	EquationalSimplification-Flat-UnFlat

Figure 6.2: Snapshot of the Web-TLR System.



Model checking results

Brief information

Date: Mon May 16 19:17:07 CEST 2011
Formula: `[]~(currentPage(bidAlfred,ADMIN) /\ currentPage(bidAnna,ADMIN))`
Initial state: `initial`

Property is fulfilled, no counter-example given.

Figure 6.3: Snapshot of the WEB-TLR System for the case of no counter-examples.

7 Implementation

The Web-TLR system described along this work has been implemented as a Web application. This chapter gives an overview of the implementation of the latest version of Web-TLR, which includes all the features enumerated in this work.

Functionally, Web-TLR is composed of three parts. The first one contains Maude definitions of a DSL for Web applications and another DSL for Web scripts, combined with the provisions necessary to perform model checking over them; for convenience, we will refer to this part as simply 'the model checker'. The second part is the slicing algorithm, which has been completely programmed in Maude. Finally, the third part contains both the GWI and a set of post-processing filters that simplify the generated counterexamples before they are presented to the user.

Most of Web-TLR has been implemented in a combination of the programming languages Java and Maude. In figures, the code base of Web-TLR consists of approximately:

- 1,600 lines of Maude code for the model checker.
- 1,300 lines of Maude code for the slicing facility.
- 225 lines of Maude code for post-processing.
- 150 lines of JavaScript code for the client-side GWI.
- 2,700 lines of Java code for the server-side GWI and post-processing.

The latest version of Web-TLR (including its source code) can be found at `http://users.dsic.upv.es/grupos/elp/soft.html`. The system can be used from any modern web browser and examples are provided on-line so the user can test the platform without needing to perform any sort of installation or download.

7.1 Implementation of Interactive Web-TLR

Since the input/output capabilities of Maude are very limited, most of the new code in Interactive Web-TLR has been written in Java, including the server-side JSP scripts, parsing of the model checker output, and execution of external programs (Graphviz). However, we have taken benefit of Maude's high-level abstraction where practical, such as the implementation of a post-processing filter which removes redundant or static information.

The GWI design is focused on simplicity. A linear input form collects the information about the Web application model, the state predicates, and the formula to be checked. If the property is true, a message is shown. Otherwise, the tool presents a slide show representing a counterexample that refutes the property.

7.1.1 Maude

As presented in [36], Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. Maude has been influenced in important ways by the OBJ3 language, which can be regarded as an equational logic sublanguage. Besides supporting equational specification and programming, Maude also supports rewriting logic computation.

Since the original implementation of Web-TLR was coded completely in Maude, its selection as a development language was easy. Nevertheless, Maude has proven to be a worthy choice for our purposes. Thanks to having a built-in model checker, we have been able to combine in a single environment the specification of the dynamic parts (Web scripts) of our model and their model checking.

A new module was added to the Web-TLR model checker, in file `parser.maude`. It implements a post-processing filter which purges the generated counterexamples of redundant information, improving its readability.

7.1.2 Java Server Pages

Java Server Pages (JSP) is a technology for developing dynamic Web pages. They manage so by combining a plain HTML page with Java code which generates content depending on a variety of factors; in our case, the information provided by the user. This information represents the Web application model that is to be checked.

JSP was chosen out of the familiarity of the authors with the Java programming language. We use the Apache Tomcat server due to its simplicity.

The JSP files only contain simple logic. All heavy-weight processing is concentrated in `ParserModelChecking.jar`.

7.1.3 Graphviz and SVG

As introduced in [35], the Graphviz layout programs take descriptions of graphs in a simple text language, and make diagrams in useful formats, such as images and SVG for web pages, PDF or Postscript for inclusion in other documents; or display in an interactive graph browser. Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, rolland custom shapes.

The biggest handicap of using Graphviz was its inability to generate incremental graphs, which we needed. When advancing slides, the visual effect of non-incremental graphs is awful. We solved this issue with a combination of some pre-computation, unnecessary whitespace, and monospaced fonts, tricking Graphviz into repeating the same layout for every graph.

We opted for the SVG output of Graphviz in order to inline it in XHTML code, as explained in the next section.

7.1.4 XHTML

XHTML was chosen instead of HTML due to its ability to inline SVG images in its code. This feature was important to simplify the server-side implementation. Since temporal files to store the generated images were made unnecessary, so was the need to control their lifetime and deletion.

While SVG and XHTML are W3C standards, these choices pose a compatibility problem with old versions of Microsoft Internet Explorer (IE), which lack the required level of support. Thankfully, starting from version 9, IE fully supports XHTML and inline SVG. Of the remaining browsers, the most popular – Mozilla Firefox, Google Chrome, Apple Safari, and Opera – all render the page successfully.

7.2 Implementation of the Slicing Facility in RWL

The enhanced verification methodology described in Chapter 6 has been implemented in the Web-TLR system using the high-performance, rewriting logic language Maude [16]. In this section, we discuss some of the most important features of the Maude language that we have been conveniently exploited for the optimized implementation of Web-TLR.

Maude is a high-performance, reflective language that supports both equational and rewriting logic programming, which is particularly suitable for developing domain-specific applications [19, 20]. In addition, the Maude language is not only intended for system prototyping, but it has to be considered as a real programming language with competitive performance. The salient features of Maude that we used in the implementation of our framework are as follows.

7.2.1 Metaprogramming

Maude is based on rewriting logic [26], which is reflective in a precise mathematical way. In other words, there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) in $\mathcal{U}$ (as a datum), and then mimick the behavior of $\mathcal{R}$ in $\mathcal{U}$.

In the implementation of the extended Web-TLR system, we have exploited the metaprogramming capabilities of Maude in order to provide the system with our backward-tracing slicing tool for RWL theories in RWL itself. Specifically, during the backward-tracing slicing process, all input Web-TLR modules are raised to the meta-level and handled as meta-terms, which are meta-reduced and meta-matched by Maude operators.

7.2.2 AC Pattern Matching

The evaluation mechanism of Maude is based on rewriting modulo an equational theory E (i.e., a set of equational axioms), which is accomplished by performing *pattern matching modulo* the equational theory E. More precisely, given an equational theory E, a term t and a term u, we say that *t matches u modulo E* (or that t *E-matches u*) if there is a substitution σ such that $\sigma(t) =_E u$, that is, $\sigma(t)$ and u are equal modulo the equational theory E. When E contains axioms that express the associativity and commutativity of one operator, we talk about *AC pattern matching*. We have exploited the AC pattern matching to implement both the filtering language and the slicing process.

7.2.3 Equational Attributes

Equational attributes are a means of declaring certain kinds of equational axioms in a way that allows Maude to use these equations efficiently in a built-in way. Semantically, declaring a set of equational attributes for an operator is equivalent to declaring the corresponding equations for the operator. In fact, the effect of declaring equational attributes is to compute with equivalence classes modulo these equations. This avoids termination problems and leads to much more efficient evaluation.

In the signature presented in Figure 3.2, the overloaded operator $\_:\_$ is given with the equational attributes assoc, comm, and id. This allows Maude to handle simple objects and multisets of elements in the same way. For example, given two terms b_1 and b_2 of sort Browser, the term $\mathsf{b}_1 : \mathsf{b}_2$ belongs to the sort Browser as well. Also, these equational attributes allow us to get rid of parentheses and disregard the ordering among elements. For example, the communication channel is modeled as a term of sort Message where the messages among the browsers and the server can arrive out of order, which allows us to simulate the HTTP communication protocol.

7.2.4 Flat/unflat Transformations

In Maude, AC pattern matching is implemented by means of a special encoding of AC operators, which allows us to represent AC terms terms by means of single representatives that are obtained by replacing nested occurrences of the same AC operator by a flattened argument list under a variadic symbol, whose elements are sorted by means of some linear ordering[1]. The inverse of the flat transformation is the unflat transformation, which is nondeterministic in the sense that it generates all the unflattended terms that are equivalent (modulo AC) to the flattened term. For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the AC-equivalence $f(b, f(f(b, a), c)) =_{AC} f(f(b, c), f(a, b))$, we can represent it by using the "internal sequence" $f(b, f(f(b, a), c)) \to^*_{flat_{AC}} f(a, b, b, c) \to^*_{unflat_{AC}} f(f(b, c), f(a, b))$, where the first subsequence corresponds to the *flattening* transformation that obtains the AC canonical form of the term, whereas the second one corresponds to the inverse, unflattening transformation.

These two processes are typically hidden inside the AC-matching algorithms[2] that are used to implement the rewriting modulo relation. In order to facilitate the understanding of the sequence of rewrite steps, we exposed the flat and unflat transformations visibly in our slicing process. This is done by breaking up a rewrite step and adding the intermediate flat/unflat transformation sequences into the computation trace delivered by Maude.

7.3 Organization of the source code

The files composing the source code of Web-TLR are organized into four locations. The first is the root folder, which stores the presentation layer; it includes all the JSP, XHTML, and JavaScript files. The second is the folder `checker`, which stores the Maude code corresponding to the Web-TLR model-checking back-end. A third folder, named `slicing`, contains the Maude implementation of the slicing facility described in Chapter 5 and Chapter 6. Finally, there is `ParserModelChecking.jar`, a Java Archive (JAR) file that provides the facilities necessary for slide generation, including the safe execution of external programs.

7.3.1 Root folder

checker.css

Cascading Style Sheet (CSS) used by `checker.html`.

[1]Specifically, Maude uses the lexicographic order of symbols.

[2]See [15] (Section 4.8) for an in-depth discussion on matching and simplification modulo AC in Maude.

checker.html

This Web page introduces the user to the model checker, allows the user to define a Web application model, and explains how to codify LTLR formulas. The model is verified when the user presses the 'Check' button.

checker.js

JavaScript code used by `checker.html`.

debug.jsp

Graphical Web Interface for the debugging and slicing facility described in Chapter 6.

index.html

Home page of WEB-TLR. It describes the tool, but has no source code itself.

modcheck.css

Cascading Style Sheet (CSS) used by the slide shows generated by `modcheck.jsp`.

modcheck.js

JavaScript code used by `modcheck.jsp`.

7.3.2 Checker

parser.maude

Maude-based filter that simplifies the output of the model checker before it is passed to the Java front-end.

script.maude

Maude definition of a Domain Specific Language (DSL) that is powerful enough to model the dynamics of complex Web applications by encompassing the main features of the most popular Web scripting languages (e.g. PHP, ASP, Java Servlets).

A brief description of this DSL can be found in the preliminaries. The full formalization of the operational semantics of this scripting language can be found in [7].

tlr.maude

This file implements the LTLR model checker for Maude developed by Kyungmin Bae and José Meseguer. It has been described in [10] and [30].

web.maude

This file specifies a Domain Specific Language (DSL) by defining a set of operators that allow the algebraic modelling of Web applications in Maude. An introduction to this DSL can be found in the preliminaries. A more detailed description of this language can be found in [7].

7.3.3 Slicing

slice.maude

This is a large file, comprising about 1,300 lines of Maude code. Inside it resides a single module, `BackwardSlicing`, which implements the slicing algorithm presented in Chapter 6, which is itself an adaptation of the more general algorithm described in Chapter 5.

webtlrtrace.maude

This file contains the Maude module `webtrace`, which extends the module `BackwardSlicing` in `slice.maude` with definitions that improve its usability.

7.3.4 ParserModelChecking.jar

Browser.java

Defines a class that represents a browser.

A constructor `Browser(state)` has been defined which takes an string describing a browser state, in the format provided by the Maude WEB-TLR back-end. In addition, since our Web model allows for more than one browser per state, an static method `parseSection(section)` has been defined for convenience. It returns an array with all the browser objects that appear in the corresponding section of the model checker state, as provided by Maude. In order to do so, it splits the input into strings corresponding to each browser, which are then fed to the Browser constructor and returned as an array.

Access to the object attributes is done, as usual, with getter methods. As the object is immutable, setter methods are not provided.

BrowserGraph.java

This class includes the functionality needed to generate the series of incremental graphs which show which pages are being viewed by which browsers at any given time. Graphviz is used for graph rendering. The output is a set of SVG files, each representing one graph.

Broadly speaking, the `BrowserGraph` constructor takes the list of pages, nodes, continuations and navigations of the Web application as input arguments. Then, for each state, all the browsers are related to the pages they are currently navegating by means of the `addBrowser(page, browser)` method. When a state has been completely processed, `newStep()` is invoked, and we proceed with the next one. When all states have been processed, a call to the `draw()` method generates the SVG files.

InterruptScheduler.java

This is an auxiliary class used by `Utils.java` to specify timeouts for the invocation of external programs. This helps avoid crashes and hangs when dealing with excessively large data sets.

Parser.java

The `Parser` class contains methods for parsing the output of the Maude Web-TLR back-end, especially its counter-examples.

The `Parser(input)` constructor takes as argument the textual representation of the counter-example generated by the Web-TLR back-end, which is immediately processed by several private methods. Then, the graphs are generated.

The components of the counter-example and the generated SVG files can all be recovered using public getter methods:

- `getFirstStates` returns an array of an array containing the path of states in the counter-example leading to the cycle. The additional level of indirection is necessary because states sharing the same graph are grouped together.
- `getCycleStates` is analogous to `getFirstStates`, for the cyclical part of the counter-example.
- `getResultType` returns the result type of the model-checking process. It can be one of: property is true, a counter-example was found, and the model-checker deadlocked.
- `getNumberOfStates` returns the total number of states.
- `getStartOfCycle` returns the position of the state that starts the cycle.

State.java

Class representing a single state in the counter-example.

It has a constructor `State(data)` with a single argument: a string describing the state in the Web-TLR back-end output format. The string is decomposed into its components, which are then processed individually by a series of simple functions. Access to those components can be later done via getter methods. The object is immutable, so no setter methods are provided.

Utils.java

This file contains a variety of auxiliary functions.

- IO wrappers (`readFile`, `saveFile`).
- Output beautifiers (`beautify`).
- Execution of external programs (`executeCmd`).
- A hand-coded parser that explores a string for matching parenthesis, brackets and curly braces (`tryParseUntil`, `parseUntil`).
- A variety of string manipulation functions.

8 Conclusions

Web-TLR is the first verification engine based on the versatile and well-established Rewriting Logic/LTLR tandem for specifying Web systems and checking of Web properties. Web-TLR distinguishes itself from related tools in two salient aspects: (*i*) the rich Web application core model that considers the communication protocol underlying Web interactions as well as common browser navigation features; and (*ii*) efficient and accurate model-checking of dynamic properties —e.g., reachability of Web pages generated by means of Web script executions— at low cost. Verification includes both analysis (checking whether properties are satisfied) and diagnostic traces demonstrating why a property does not hold.

The contributions of this work are twofold and consist in two major enhancements to the Web-TLR system that are intended to improve its understandability and usability.

The first improvement is the promotion of Web-TLR from a console-line based, manual tool to a full-fledged Web application capable of visualizing counterexamples via an interactive slideshow generated on-the-fly. This slideshow allows the user to explore the model by performing forward and backward transitions. At each slide, the interface shows a graph that relates the browsers to the Web pages they are currently exploring together with the values of relevant variables of the Web state. This exploration does not require installation of the checker itself and is provided entirely by the GWI.

The second enhancement to Web-TLR is the inclusion of our backward trace-slicing facility that eases the interactive debugging of Web applications. The proposed slicing technique allows that the size of the counterexample trace be greatly reduced, making their analysis feasible even in the case when complex, real-size Web systems are considered.

In addition, we have presented a backward slicing technique for rewrite theories. The key idea behind our slicing technique consists in tracing back —through the rewrite sequence— all the relevant symbols, defined by the slicing criterion, of the final state that we are interested in. The trace slicing technique can be applied to execution trace analysis of sophisticated rewrite theories, which can include equations and equational axioms as well as nonleft-linear and collapsing rules.

While Web-TLR was the motivation behind the development of this algorithm, its applicability transcends Web-TLR itself. To the best of our knowledge, no trace slicing methodology for rewriting logic theories has yet been proposed.

We also provide a tool that implements our backward slicing technique. This tool provides a complete description of all events and locations involved in a selected error scenario, and the user can analyze different error scenarios in an incremental, step-by-step manner.

We have tested our tool on several complex case studies that are available at the Web-TLR Web page and within the distribution package, including a Webmail application, a producer/consumer system and a sophisticated formal specification of a fault-tolerant communication protocol. The results obtained are very encouraging and show impressive reduction rates in all cases, ranging from 90% to 95% in reasonable time (max. 0.5 s on a Linux laptop equipped with an Intel Core 2 Duo 2.26GHz and 4Gb of RAM memory). Moreover, sometimes the trace slices are so small that they can be easily inspected by the user who can keep a quick eye on what's going on behind the scenes.

As future work, we plan to extend Web-TLR by considering the problem of synthesizing correct-by-construction Web applications. We also plan to deal with client-side scripts defined for example by JavaScript-like languages. In addition, there has been external interest in adapting the framework to deal with Representational State Transfer (REST) architectures [22]. Respecting the slicing algorithm, we are working on increasing the efficiency of the tool. Also, as future work, we plan to deal with the execution traces of more sophisticated theories that may include membership and conditional equations.

This work has been developed in joint work with my supervisors M. Alpuente and D. Romero. The theoretical results are the core of D. Romero's Ph.D. Thesis [34]. The integration of the generic slicing techniques into Web-TLR has been done in joint work with F. Frechina.

To conclude, and the specific goals of this work notwithstanding, we also like to think of this work as part of a bigger, inspiring picture. Formal methods are a powerful toolset that has a great potential to improve the quality, security, and safety of Web applications. We hope that contributions like ours help to promote the use of formal methods in software engineering in general, and in the area of Web applications in particular.

Appendices

A
Web Application Model Development by Example

Throughout this work, we have used the Electronic Forum described in Chapter 3 as a running example. In this appendix, we will introduce a Webmail application to show by example how we can specify a Web application model in WEB-TLR that can be verified later by means of the technique described in Chapter 6.

Introduction

We will now introduce the Webmail application model presented in [8]. Starting from the *welcome* page, the user must enter his *user* and *password*. At *home*, the user can access his email list and then see a particular message. The user might also check another email account, after entering a new *user* and *password*. The users with role *admin* may access the *administration* page. Finally, the *logout* page is self-descriptive.

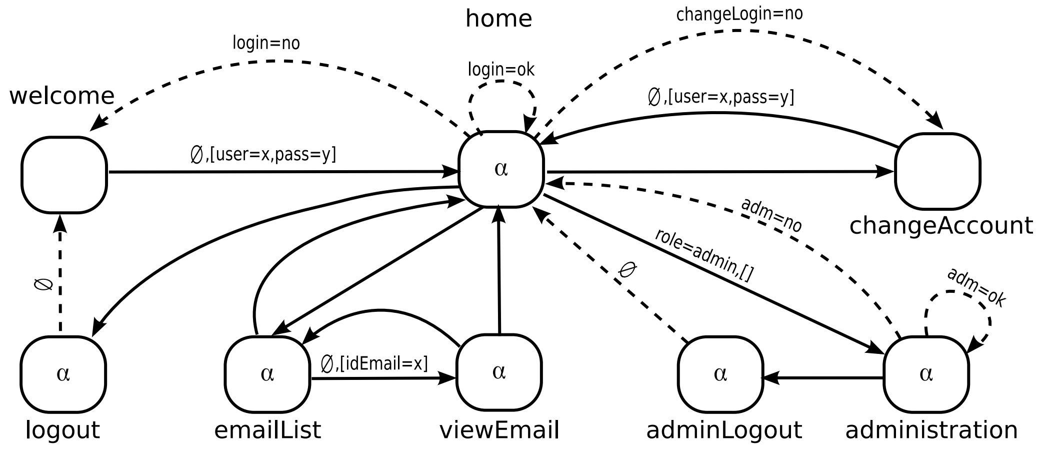


Figure A.1: The navigation model of an Webmail application

The Web application and state predicates

The first text area in the input form should have two functional modules. The first module contains the definition of the web application itself: the set of available pages, the continuations, the navigations, and the scripts associated to each Web page. The second module contains the definition of the state predicates, which are used in the property to be checked (a LTLR formula).

The following code listing shows the skeleton of this component.

```
mod WEBAPP is inc PROTOCOL .
  --- Your Web application definition here!
endm

mod WEBAPP-CHECK is including MODEL-CHECKER + WEBAPP .
  --- Your state predicates here!
end
```

Defining the page identifiers

It is customary to start by defining in the WEBAPP module the identifiers of all the pages that we are going to use. For our Webmail example, this would be:

```
ops WELCOME
  HOME
  EMAIL-LIST
  VIEW-EMAILCHANGE-ACCOUNT
  ADMINISTRATION
  ADMIN-LOGOUT
  LOGOUT
: -> Qid .
```

Notice that the identifiers have been defined as constants, but literals would have been as valid. Indeed, we can assign more readable values to the operators that we have just defined:

```
eq WELCOME = 'Welcome .
eq HOME = 'Home .
eq EMAIL-LIST = 'Email-list .
eq VIEW-EMAIL = 'View-email .
eq CHANGE-ACCOUNT = 'Change-account .
eq ADMINISTRATION = 'Administration .
eq ADMIN-LOGOUT = 'Admin-Logout .
eq LOGOUT = 'Logout .
```

Defining a script

We show here the script of the *home* page as an example. A thorough description of the scripting language can be found in [7].

```
op sHome : -> Script .
eq sHome =
   'login := getSession( s("login") ) ;
   if ( 'login = null ) then
      'u := getQuery('user) ;
          'p := getQuery('pass) ;
          'p1 := selectDB('u) ;
          if ( 'p = 'p1 ) then
             'r := selectDB('u '. s("-role") ) ;
             setSession( s("user") , 'u ) ;
                 setSession( s("role") , 'r ) ;
                 setSession( s("login") , s("ok") )
          else
             setSession( s("login") , s("no") )
          fi
   fi .
```

Defining a page

Recalling the DSL presented in Chapter 3, in Web-TLR a Web page is a tuple containing an identifier, an script that is executed on entry, a set of continuations, and a set of navigations. All fields must be set. If a Web page does not have an associated script, `skip` must be used. The operators `cont-empty` and `nav-empty` perform the same function for continuations and navigations, respectively. The most simple Web page is, therefore:

```
op trivialPage: -> Page .
eq trivialPage =
   ( TRIVIAL-PAGE , skip, { cont-empty } , { nav-empty } ) .
```

A more realistic example would be the *e-mail list page*:

```
 op emailListPage : -> Page .
 eq emailListPage =
 (
   EMAIL-LIST ,
   sEmailList ,
   { cont-empty } ,
   {
     ( TRUE  -> ( VIEW-EMAIL ? ('idEmail '= "") ) )
   : ( TRUE  -> ( HOME ? query-empty ) )
   }
 ) .
```

Notice how the colon operator is used to separate navigations. Albeit not shown in the example, it is also used to separate continuations.

Defining a state predicate

State predicates should be defined in the `WEBAPP-CHECK` module. We can define state predicates in order to evaluate propositions on the states. For example, consider the two following state predicates where: (*i*) `currentPage` evaluates to true when a given page passed as parameter is being displayed in the browser; (*ii*) `requestHome` holds when there is a requirement to go to the home page. These predicates can be defined as follows:

```
subsorts WebState < State .

vars idb idw : Id .
vars page : Qid .
vars url  : URL .
vars z : Sigma .
vars lms ms  : Message .
vars brs : Browser .
vars sv : Server .
vars q : Query .
vars i : Nat .

--- current page
op currentPage : Id Qid -> Prop .
eq [ B(idb, idw, page, urls, z, lms, h, n, idlm) : brs ]
   [ ms ] [ sv ]
   |= currentPage(idb, page) = true .
eq [ brs ] [ ms ] [ sv ]
   |= currentPage(idb, page) = false [owise] .

--- request to go to home
op requestHome : Id -> Prop .
eq [ brs ] [ m(idb, idw, (HOME ? q ), i ) : ms ] [ sv ]
   |= requestHome(idb) = true .
eq [ brs ] [ ms ] [ sv ]
   |= requestHome(idb) = false [owise] .
```

Initial Web states

This section includes the definition of the initial states, including the available browsers, the substitution (predefined answers used to simulate user input in forms) associated to each of them, and its database configuration. The initial state describes the system at the start of the model checking process. More than one initial state can be defined but only one can be selected for checking the property. The initial Web states should be contained in a module named `WEBAPP-PROPERTIES`.

```
mod WEBAPP-PROPERTIES is inc WEBAPP-CHECK .
  ops idA idw1 : -> Id .

  op brA : -> Browser .
  eq brA = newBrowser(idA,idw1,(WELCOME ? query-empty),zA) .

  op zA : -> Sigma  .
  eq zA = ('user / "alice")
        : ('pass / "secretAlice" )
        : ('idEmail / "email2") .

  op initial-1 : -> WebState .
  eq initial-1 = [ brA ] bra-empty [ mes-empty ] [ sv ] .
endm
```

Property specification

The translation of LTLR formulas into their Maude representation is straightforward. A LTLR formula can include the following operators: `[]` means “always”; `<>` means “eventually”; `O` (capital letter) means “in the next state.”; `/\` expresses conjuntion; `\/` expresses disjuntion; and expresses negation. More information can be found in The Linear Temporal Logic of Rewriting Model Checker by Kyungmin Bae and José Meseguer.

As an example, the following formula verifies (or not) if only one user can visit the *administration* page at the same time:

```
[] (~ ((currentPage(idA,ADMIN) /\ currentPage(idB,ADMIN))
```

Bibliography

[1] M. H. Alalfi, J. R. Cordy & T. R. Dean (2009): *Modelling methods for web application verification and testing: state of the art. Software Testing, Verification and Reliability* 19, pp. 265–296.

[2] M. Alpuente, D. Ballis, M. Baggi & M. Falaschi (2010): *A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT*. In: *Proc. 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, (PEPM 2010)*, ACM, pp. 43–52.

[3] M. Alpuente, D. Ballis, J. Espert, F. Frechina & D. Romero (2011): *Debugging of Web Applications with* WEB-TLR. In: *Proc. 7th Int'l Workshop on Automated Specification and Verification of Web Systems (WWV 2011)*.

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

[5] M. Alpuente, D. Ballis, J. Espert & D. Romero (2011): *Backward Trace Slicing for Rewriting Logic Theories*. In: *The 23rd Int'l Conference on Automated Deduction (CADE 2011)*, LNCS/LNAI, Springer. To appear.

[6] M. Alpuente, D. Ballis, J. Espert & D. Romero (2011): *Dynamic Backward Slicing of Rewriting Logic Computations*. *ArXiv e-prints*. Available at http://arxiv.org/abs/1105.2665v1.

[7] M. Alpuente, D. Ballis & D. Romero (2009): *A Rewriting Logic Framework for the Specification and the Analysis of Web Applications*. Technical Report DSIC-II/01/09, Technical University of Valencia. Available at: http://www.dsic.upv.es/~dromero/web-tlr.html.

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

[9] K. Bae & J. Meseguer (2008): *A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting*. In: *Proc. of the 9th Int'l Workshop on Rule-Based Programming (RULE'08)*, Electronic Notes in Theoretical Computer Science, Elsevier.

[10] Kyungmin Bae & José Meseguer (2010): *The Linear Temporal Logic of Rewriting Maude Model Checker.* In Ölveczky [30], pp. 208–225. Available at http://dx.doi.org/10.1007/978-3-642-16310-4_14.

[11] M. Baggi, D. Ballis & M. Falaschi (2009): *Quantitative Pathway Logic for Computational Biology.* In: *Proc. of 7th Int'l Conference on Computational Methods in Systems Biology (CMSB'09)*, *Lecture Notes in Computer Science* 5688, Springer, pp. 68–82.

[12] I. Bethke, J. W. Klop & R. de Vrijer (2000): *Descendants and origins in term rewriting. Inf. Comput.* 159(1-2), pp. 59–124.

[13] F. Chen & G. Rosu (2009): *Parametric Trace Slicing and Monitoring.* In: *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, *Lecture Notes in Computer Science* 5505, Springer, pp. 246–261.

[14] O. Chitil, C. Runciman & M. Wallace (2000): *Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs.* In: *Implementation of Functional Languages, 12th International Workshop (IFL 2000)*, *Lecture Notes in Computer Science* 2011, Springer, pp. 176–193.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C. Talco (2009): *Maude Manual (Version 2.4).* Technical Report, SRI Int'l Computer Science Laboratory. Available at: http://maude.cs.uiuc.edu/maude2-manual/.

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

[17] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer & P. C. Ölveczky (2007): *The Maude Formal Tool Environment.* In: *Algebra and Coalgebra in Computer Science (CALCO'07)*, *Lecture Notes in Computer Science* 4624, Springer, pp. 173–178.

[18] S. Eker (2003): *Associative-Commutative Rewriting on Large Terms.* In: *Proc. of 14th Int'l Conference, Rewriting Techniques and Applications (RTA '03)*, *Lecture Notes in Computer Science* 2706, Springer, pp. 14–29.

[19] S. Eker, J. Meseguer & A. Sridharanarayanan (2003): *The Maude LTL model checker and its implementation.* In: *Model Checking Software: Proc. 10 th Intl. SPIN Workshop*, *Lecture Notes in Computer Science* 2648, Springer, pp. 230–234.

[20] S. Escobar, C. Meadows & J. Meseguer (2006): *A Rewriting-Based Inference System for the NRL Protocol Analyzer and its Meta-Logical Properties*. *Theoretical Computer Science* 367(1-2), pp. 162–202.

[21] J. Field & F. Tip (1994): *Dynamic Dependence in Term rewriting Systems and its Application to Program Slicing*. In: *Proc. of the 6th Int'l Symposium on Programming Language Implementation and Logic Programming (PLILP'94)*, Springer-Verlag, London, UK, pp. 415–431.

[22] Roy Thomas Fielding (2000): *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine, Irvine, California. Available at `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

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

[24] Edmund M. Clarke Jr., Orna Grumberg & Doron A. Peled (1999): *Model Checking*. The MIT Press. Available at `http://www.amazon.com/Model-Checking-Edmund-Clarke-Jr/dp/0262032708%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032708`.

[25] Z. Manna & A. Pnueli (1992): *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA.

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

[27] J. Meseguer (1992): *Conditional Rewriting Logic as a Unified Model of Concurrency*. *Theoretical Computer Science* 96(1), pp. 73–155.

[28] J. Meseguer (2008): *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*, 5065, Springer-Verlag, Berlin, Heidelberg, pp. 354–382.

[29] R. Message & A. Mycroft (2008): *Controlling Control Flow in Web Applications*. In: *4th Int'l Workshop on Automated Specification and Verification of Web Sites (WWV'08)*, *Electronic Notes in Theoretical Computer Science* 200, pp. 119–131.

[30] Peter Csaba Ölveczky, editor (2010): *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*. Lecture

Notes in Computer Science 6381, Springer. Available at http://dx.doi.org/10.1007/978-3-642-16310-4.

[31] P. Réty (1987): *Improving basic narrowing techniques.* In: *Conference on Rewriting techniques and applications*, *Lecture Notes in Computer Science* 256, Springer-Verlag, London, UK, pp. 228–241.

[32] A. Riesco, A. Verdejo, R. Caballero & N. Martí-Oliet (2009): *Declarative Debugging of Rewriting Logic Specifications.* In: *Recent Trends in Algebraic Development Techniques, 19th Int'l Workshop (WADT 2008)*, *Lecture Notes in Computer Science* 5486, Springer, pp. 308–325.

[33] A. Riesco, A. Verdejo & N. Martí-Oliet (2010): *Declarative Debugging of Missing Answers for Maude.* In: *21st Int'l Conference on Rewriting Techniques and Applications (RTA 2010)*, *LIPIcs* 6, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 277–294.

[34] Daniel Omar Romero (2011): *Rewriting-based Verification and Debugging of Web Systems.* Ph.D. thesis, Universidad Politécnica de Valencia.

[35] Graphviz Graph Visualization Software: Available at: http://www.graphviz.org/.

[36] The Maude System: Available at: http://maude.cs.uiuc.edu/.

[37] C. Talcott (2008): *Pathway Logic. Formal Methods for Computational Systems Biology* 5016, pp. 21–53.

[38] TeReSe, editor (2003): *Term Rewriting Systems.* Cambridge University Press, Cambridge, UK.

[39] HAT The Haskell tracer: Available at: http://www.cs.york.ac.uk/fp/ART/.

[40] P. Viry (1994): *Rewriting: An Effective Model of Concurrency.* In: *Proceedings of the 6th Int'l PARLE Conference on Parallel Architectures and Languages Europe*, Springer-Verlag, London, UK, pp. 648–660.

[41] F. Weitl, S. Nakajima & B. Freitag (2010): *From Counterexamples to Incremental Interactive Tracing of Errors (Schrittweise Fehleranalyse auf der Grundlage von Model-Checking). it - Information Technology* 52(5), pp. 295–297.