UNIVERSIDAD POLITECNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Slicing-based debugging of Web applications in Rewriting Logic

Master's Thesis

Presented by:
Francisco Frechina Navarro

Supervisors:
María Alpuente Frasnedo
Daniel Omar Romero

Valencia - July, 14th 2011

Abstract

The pervasiveness of computing on the Internet has led to an explosive growth of Web applications that, together with their ever-increasing complexity, have turned their design and development in a major challenge.

Unfortunately, the huge expansion of development and utilization of Web computation has not been paired by the development of methods, models and debugging tools to help the developer diagnose, quickly and easily, potential problems in a Web application. There is an urgent demand of analysis and verification facilities capable to prevent insecure software that could cause unavailability of systems or services, or provide access to private data or internal resources of a given organization.

The main goal of this MSc thesis is to improve the debugging of Web applications by embedding novel analysis and verification techniques that rely on the program semantics. As a practical realization of the ideas, we use Web-TLR that is a verification engine for dynamic Web applications based on Rewrite Logic. We extend Web-TLR with a novel functionality that supports effective Web debugging for realistic Web applications involving complex execution traces. This functionality is based on a backward trace slicing technique that is based on dynamic labeling.

In order to extend the class of programs covered by the debugging methodology we formalize a generalization of the slicer to Conditional Rewriting Logic theories, greatly simplifying the debugging task by providing a novel and sophisticated form of pattern matching

Resumen

La omnipresencia de la informática en Internet ha llevado al crecimiento explosivo de aplicaciones para la Web que, unido a su cada vez más creciente complejidad, han convertido su diseño y desarrollo en un desafío importante.

Lamentablemente, la enorme expansión del desarrollo y utilización de la computación Web no ha venido acompañada por el desarrollo de métodos, modelos y herramientas de depuración que ayuden al desarrollador a diagnosticar, de manera fácil y rápida, potenciales problemas en la aplicación Web. Existe una demanda urgente de herramientas para el análisis y verificación capaces de evitar software inseguro que pueda provocar la indisponibilidad de sistemas o servicios, o que permita el acceso a datos privados o recursos internos de una organización.

El objetivo principal de este trabajo final de master es perfeccionar la depuración de aplicaciones Web mediante la incorporación de nuevos análisis y técnicas de verificación basadas en la semántica del programa. Como realización práctica de estas ideas, utilizamos Web-TLR que es un motor de verificación de aplicaciones Web dinámicas basado en la Lógica de Reescritura. Extendemos Web-TLR con una novedosa funcionalidad de depuración Web para aplicaciones Web reales con trazas de ejecución complejas. Esta funcionalidad está basada en una técnica de rebanado de trazas hacia atrás (*backward trace slicing*) basada en un etiquetado dinámico de los pasos.

Con el fin de extender los programas cubiertos por nuestra metodología de depuración, formalizamos una generalización del rebanador para teorías de lógica de reescritura condicionales, simplificando considerablemente la tarea de depuración y apostando por una novedosa y sofisticada forma de ajuste de patrones (*pattern matching*).

Contents

Introduction **1**
Summary of Contributions of this Master's Thesis 1

Preliminaries **5**
Conditional Term Rewriting Systems 6
Conditional Rewrite Theories 7

I Debugging of Web Applications **9**

1 Specification and Verification of Web Applications in RL **11**
1.1 A Navigation Model for Web Applications 13
1.1.1 Graphical Navigation Model 14
1.2 Formalizing the Navigation Model as a Rewrite Theory . 15
1.2.1 The Web Scripting Language 16
1.2.2 The Web Application Structure 17
1.2.3 The Communication Protocol 18
1.3 Modeling Multiple Web Interactions and Browser Features 23
1.3.1 The Extended Equational Theory $(\mathbf{\Sigma_{ext}}, \mathbf{E_{ext}})$ . . 23
1.3.2 The Extended Rewrite Rule Set $\mathbf{R_{ext}}$ 24
1.4 Model Checking Web Applications Using LTLR 28
1.4.1 The Linear Temporal Logic of Rewriting 28
1.4.2 LTLR properties for Web Applications 30

2 Debugging of Web Applications with Web-TLR **35**
2.1 Extending the Web-TLR System 36
2.1.1 Filtering Notation 39
2.2 Implementation of the extended Web-TLR system in RWL 40
2.3 A Case Study in Web Verification 42
2.4 A Debugging Session with Web-TLR 45

II Conditional Slicing for Rewriting Logic **53**

3 Conditions in Rewrite Theories **55**
3.1 Conditions in Maude 55
3.2 Conditional Rewriting Inference Process 57

4 Backward Trace Slicing for Conditional Rewrite Theories **61**
4.1 Term Slices 61
4.2 Extended Pattern Matching for Term Slices 63
4.3 Backward Conditional Slicing 65
4.3.1 Backward Slicing for a Rewrite Step 66
4.3.2 Backward Slicing for Execution Traces 70
4.4 Soundness of the Slicing Technique 73
4.5 Slicing of a Maude Example Trace 76

Conclusions **87**
Future Work 88

Bibliography **89**

A Operational Semantics of the Web Scripting Language **93**

B Specification of the Evaluation Protocol Function **97**

C Maude trace **99**

List of Figures

1.1 The navigation model of a Webmail application. 15

2.1 One Web state of the counter-example trace of Section 2.4. 37
2.2 The navigation model of an Electronic Forum 43
2.3 Specification of the electronic forum application in WEB-TLR 44
2.4 Trace slice $\mathcal{T}^{\bullet}$. 47
2.5 Snapshot of the WEB-TLR System. 49
2.6 Snapshot of the WEB-TLR System for the case of no counter-examples. 51

4.1 A term slice. 63
4.2 A term slice concretization. 75

Introduction

A Web application is an application hosted in a Web server that obeys a client-server architecture. It is accessed over a network (such as the internet) by means of a Web browser. In the last years, Web applications have been playing a crucial role to support financial and e-commerce transactions, fast and secure information interchange, social interactions, *etc*. In this scenario, Web applications are subject to an ever-increasing complexity with regard to their design and development, which demands highly sophisticated verification, debugging and repairing tools to assist developers in the construction process. Obviously, the specification and debugging of Web applications require the development of specific techniques that address the specific challenges of the World Wide Web. In recent years, much effort has been invested towards this endeavour although there is still much zoom for improving the current techniques and tools for verifying and debugging Web systems.

Summary of Contributions of this Master's Thesis

As the major contribution, this master's thesis formalizes a novel slicing-based technique for rewriting logic computation that can be applied to the debugging of dynamic Web applications that are specified as conditional theories in rewriting logic [MOM02].

The MSc thesis is organized in two main parts. In the following, we briefly summarize the main contributions of each part.

Part I: Debugging of Web Applications

In this part, a tool for the debugging of dynamic Web applications is developed. This tool is an extension of the rewriting logic framework proposed in [ABER10; Rom11], which allows one to naturally simulate the navigation of a user when running/executing the Web application, check

important related properties (e.g., mutual exclusion problems [MM08]), and evaluate the eventually included Web scripts. Our extension integrates the system Web-TLR of [ABER10] with the backward trace slicing technique of [ABER11a] in order to empower the tool with a new facility that supports the debugging of Web applications.

Part II: Conditional Slicing for Rewriting Logic

In this part, this MSc thesis formalizes a conditional slicing technique for Rewriting Logic Theories. Our backward conditional slicing facility allows us to systematically trace back conditional rewrite sequences modulo algebraic axioms (such as associativity and commutativity) by means of an algorithm that simplifies the traces by detecting control and data dependencies, automatically 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. We greatly simplify the slicing process described in [ABER11a; Rom11], by replacing the dynamic labeling by a novel and more convenient extended pattern matching algorithm.

To conclude this introduction, let us detail how we organized the manuscript. Each part of the thesis consists of two chapters whose contents are as follows.

Chapter 1. Specification and Verification of Web Applications in Rewriting Logic

This chapter contains the preliminaries of this work. It summarizes the Rewriting Logic framework for the formal specification of the operational semantics of Web applications first proposed in [ABR09]. The main idea is to define a rewrite theory that precisely formalizes the interactions among Web servers and Web browsers through a communicating protocol abstracting the main features of the HyperText Transfer Protocol (HTTP). The model also supports a scripting language encompassing the main features of the principal Web scripting languages (e.g., PHP, ASP, Java servlets), which is powerful enough to model complex Web application dynamics as well as advanced navigation capabilities such as

adaptive navigation (that is, a form of navigation through a Web application that can be dynamically customized according to both user and session information). A detailed characterization of browser actions (e.g., forward/backward navigation, refresh, and new window/tab openings) via rewrite rules completes the proposed specification.

The rewriting logic formalization is particularly suitable for verification purposes, since it allows one to carry out in-depth analyses of several subtle aspects of Web interactions. In particular, the Web models can be naturally model-checked by using the Linear Temporal Logic of Rewriting (LTLR) [Mes08], which is a Linear Temporal Logic [MP92] supporting model-checking of rewrite theories.

The results in this chapter are original contribution of [ABR09; Rom11] and only included in this thesis for the sake of making the manuscript self-contained.

Chapter 2. Debugging of Web Applications with Web-TLR

Web-TLR [ABER10] 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, a counterexample trace is delivered 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 endow Web-TLR with a new Web debugging facility that supports the efficient manipulation of counterexample traces. This facility is based on a backward trace-slicing technique for rewriting logic theories proposed in [ABER11a; Esp11] that allows the pieces of information that we are interested in to be 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.

The original contribution of this chapter is the parameterization and integration of the generic slicer of [ABER11a] into Web-TLR, which was

proposed in [ABE+11]. We would like to emphasize that the coupling of the slicing technique within Web-TLR is non-trivial and required to exploit the metaprogramming capabilities of Maude in order to provide the system within the backward-tracing slicing tool. In addition, we have exploited the AC pattern matching of Maude to implement both the filtering language and the slicing process itself.

Chapter 3. Conditions in Rewrite Theories

This chapter investigates the way in witch conditional rewriting steps are performed in the high-performance reflective language Maude. Maude can be seen as an effective mechanization of RWL that is particularly suitable for developing domain-specific applications [EMS03; MEM06].

The precise way in which Maude proves the conditional rewriting steps motivates our conditional slicing technique given in the last chapter of the MSc thesis. Then, a motivating explanation of conditional rewriting is given in order to understand and appreciate our conditional slicing for rewrite theories.

Chapter 4. Backward Trace Slicing for Conditional Rewrite Theories

In this chapter, we formalize a conditional slicing technique for rewriting logic that is particularly suitable for analyzing complex, textually-large system computations. Our technique can be mechanized by means a backward trace slicing process for conditional rewrite theories witch relies on a novel extended pattern matching algorithm also defined in this work. We analyze the execution trace, and inductively compute the origins of the observed relevant information at each rewrite step by applying a backward slicing for rewrite steps that takes into account the conditions of the applied rules. After getting rid of the irrelevant information finally a simplified trace is finally obtained.

The original contribution of this chapter is, firstly, the consideration of conditional rules in the slicing algorithm, and even most important, the simplification and improvement of the backward trace slicing of [ABER11a] by avoiding the complex labeling process in favour of a novel and more convenient extended pattern matching algorithm.

Preliminaries

In this section we recall some basic notions regarding conditional term rewriting systems and rewriting logic that will be used in the rest of the master's thesis. For details, we refer to [TeR03].

By $Variables$ we denote a countably infinite set of variables and Σ denotes a set of function symbols, or *signature*. We consider varyadic signatures as in [DP01] (i.e., signatures in that symbols have an unbounded arity, that is, they may be followed by an arbitrary number of arguments). Given a term t, we say that t is *ground* if no variables occur in t. $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ denote the *non-ground term algebra* and the *term algebra* built on $\Sigma \cup \mathcal{V}$ and Σ, respectively.

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

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. By $root(t)$, we denote the symbol occurring 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 $w_1.x = w_2$. $\leq_{Lex}$ denoted the lexicographic ordering between positions, that is, $\Lambda \leq_{Lex} w$ for every position w, and given the positions $w_1 = i.w_1'$ and $w_2 = j.w_2'$, then $w_1 \leq_{Lex} w_2$ iff $i < j$ or ($i = j$ and $w_1' \leq_{Lex} w_2'$). Given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t that are rooted by symbols in S. Moreover, for any position x, $\{x\}.O_S(t) = \{x.w \mid w \in O_S(t)\}$. $t_{|u}$ 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. By $path_w(t)$, we denote the set of symbols in t that occur in the path from its root to the position w of t, e.g., $path_{(2.1)}(f(a, g(b), c)) = \{f, g, b\}$. By $Var(t)$ (resp. $FSymbols(t)$), we denote the set of variables (resp. function symbols) occurring in the term t.

Syntactic equality between objects is represented by $\equiv$. Given a set S, sequences of elements of S are built with constructors $\epsilon :: S^*$ (empty sequence) and $. :: S \times S^* \to S^*$.

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 all other variables x. By ε, we denote the *empty* substitution. Given a substitution σ, the *domain* of σ is the set $Dom(\sigma) = \{x | x\sigma \neq x\}$. Given the substitutions σ_1 and σ_2, such that $Dom(\sigma_2) \subseteq Dom(\sigma_1)$, by σ_1/σ_2 we define the substitution $\{X/t \in \sigma_1 \mid X \in Dom(\sigma_1) \setminus Dom(\sigma_2)\} \cup \{X/t \in \sigma_2 \mid X \in Dom(\sigma_1) \cap Dom(\sigma_2)\} \cup \{X/X | X \notin Dom(\sigma_1)\}$. An *instance* of a term t is defined as $t\sigma$, where σ is a substitution.

A *context* is a term $\gamma \in \tau(\Sigma \cup \Box, \mathcal{V})$ with zero or more holes $\Box$, and $\Box \notin \Sigma$. 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 $t^\Box$, we denote the context obtained by applying the substitution $\sigma = \{x_1/\Box, \ldots, x_n/\Box\}$ to t, where $Var(t) = \{x_1 \ldots, x_n\}$ (i.e., $t^\Box = t\sigma$).

Conditional Term Rewriting Systems

A *conditional term-rewriting system* (CTRS 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 $(\lambda \to \rho \Leftarrow C)$, λ, $\rho \in \tau(\Sigma, \mathcal{V})$, $\lambda \notin \mathcal{V}$. The conditions C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations. Variables in C or ρ that do not occur in λ are called *extra variables*. We will often write just R instead of (Σ, R). If a rewrite rule has no condition, we write $\lambda \to \rho$. Note that function symbols are allowed to be arbitrarily nested in left-hand sides. Following Middeldorp and Hamoen [MH94], a CTRS

whose rules $(\lambda \to \rho \Leftarrow C)$ do not contain extra variables, i.e., $Var(\rho) \cup Var(C) \subseteq Var(\lambda)$), is called a 1-CTRS. A CTRS with extra variables only in the conditions of their rewrite rules, i.e., $Var(\rho) \subseteq Var(\lambda)$, is called a 2-CTRS, while a 3-CTRS may also have extra variables in the rhs's provided these occur in the corresponding conditions, i.e., $Var(\rho) \subseteq (Var(\lambda) \cup Var(C))$.

A rewrite step is the application of a rewrite rule to an expression. A term s *conditionally rewrites* to a term t via $r \in R$, $s \stackrel{r}{\to}_R t$ (or $s \xrightarrow{r,\sigma,w}_R t$), if there exists a position w in s such that λ *matches* $s_{|w}$ via a substitution σ (in symbols, $s_{|w} \equiv \lambda\sigma$) and the condition C holds; then t is obtained from s by replacing the subterm $s_{|w} \equiv \lambda\sigma$ with the term $\rho\sigma$, in symbols $t \equiv s[\rho\sigma]_w$. When no confusion can arise, we will omit any subscript (i.e., $s \to t$). We denote the transitive and reflexive closure of $\to$ by $\to^*$. t is the irreducible form of s w.r.t. R (in symbols $s \to^!_R t$) if $s \to^*_R t$ and t is irreducible.

The rule $\lambda \to \rho \Leftarrow C$ (or equation $\lambda = \rho \Leftarrow C$) is *collapsing* if $\rho \in \mathcal{V}$; it is *left-linear* if no variable occurs in λ more than once. We say that a CTRS R is *terminating*, if there exists no infinite rewrite sequence $t_1 \to_R t_2 \to_R \ldots$. A CTRS R is *confluent* if, for all terms s, t_1, t_2, such that $s \to^*_R t_1$ and $s \to^*_R t_2$, there exists a term t s.t. $t_1 \to^*_R t$ and $t_2 \to^*_R t$. When R is terminating and confluent, it is called *canonical*. In canonical CTRSs, each input term t can be univocally reduced to a unique *irreducible form*.

The conditional equation $s = t \Leftarrow C$ *holds* in a canonical CTRS R, if there exists an irreducible form $z \in \tau(\Sigma, \mathcal{V})$ w.r.t. R such that $s \to^!_R z$ and $t \to^!_R z$ and the condition C holds.

Conditional Rewrite Theories

The static state structure as well as the dynamic behavior of a concurrent system can be described by meas of a Rewriting Logic (RWL) specification encoding a *conditional rewrite theory* [Mes92]. A *conditional rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$, where:

(i) (Σ, E) is an order-sorted *conditional equational theory* equipped with a partial order $<$ modeling the usual subsort relation. The

signature Σ specifies the operators and sorts defining the type structure of $\mathcal{R}$, while $E = \Delta \cup B$ consists of a set of (oriented and possibly conditional) equations Δ together with a collection B of equational axioms (e.g., associativity, commutativity, and unity) that are associated with some operator of Σ. The equational theory (Σ, E) induces a congruence relation on the term algebra $\tau(\Sigma, \mathcal{V})$, which is usually denoted by $=_E$. Intuitively, the sorts and operators contained in the signature Σ allow one to formalize system states as ground terms of the term algebra $\tau(\Sigma, E)$ that is built upon Σ and E.

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

Variables may appear in both equational axioms and rules. By notation $x : S$, we denote that variable x has sort S.

The system evolves by applying the rules of the conditional 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 a conditional 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 conditional 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 $s_1 \stackrel{r}{\to} s_2$), 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 $s_0 \stackrel{r_1}{\to} s_1 \ldots \stackrel{r_k}{\to} s_k$, with $r_1, \ldots, r_k \in R$, $s_0, \ldots, s_k \in \tau(\Sigma, E)$.

Part I

Debugging of Web Applications

Chapter 1

Specification and Verification of Web Applications in RWL

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

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

Actually, Web browsers typically support backward and forward navigation through Web application stages, and allow the user to open distinct (instances of) Web scripts in distinct windows/tabs which are run in parallel. Such browser actions may be potentially dangerous, since they can change the browser state without notifying the server, and may easily lead to errors or undesired responses. For instance, [MM08] reports on a frequent error, called the *multiple windows problem*, which typically happens when a user opens the windows for two items in an online store, and after clicking to buy on the one that was opened first, he frequently gets the second one being bought. Moreover, clicking refresh/forward/backward browser buttons may sometimes produce error messages, since such buttons were designed for navigating stateless Web pages, while navigation through Web applications may require multiple

state changes. These problems have occurred frequently in many popular Web sites (e.g., Orbitz, Apple, Continental Airlines, Hertz car rentals, Microsoft, and Register.com) [GFKF03]. Finally, naïvely written Web scripts may allow security holes (e.g., unvalidated input errors, access control flaws, *etc.* [Pro07]) producing undesired results that are difficult to debug.

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

First we summarize a fine-grained, operational semantics of Web applications that is based on a formal navigational model which is suitable for the verification of real, dynamic Web sites. This model is formalized within the Rewriting Logic (RWL) framework [MOM02], a rule-based, logical formalism particularly appropriate to modeling concurrent systems [Mes92]. Specifically, we describe a rigorous rewrite theory which:

i) completely formalizes the interactions between multiple browsers and a Web server through a request/response protocol that supports the main features of the HyperText Transfer Protocol (HTTP);

ii) models browsers actions such as refresh, forward/backward navigation, and window/tab openings;

iii) supports a scripting language which abstracts the main common features (e.g., session data manipulation, data base interactions) of the most popular Web scripting languages.

iv) formalizes *adaptive navigation* [HH06], that is, a navigational model in which page transitions may depend on user's data or previous computation states of the Web application.

Also we show how rewrite theories specifying Web application models can be model-checked using the *Linear Temporal Logic of Rewriting* (LTLR) [BM08; Mes08]. The LTLR allows us to specify properties at a very high level using RWL rules and hence can be smoothly integrated into our RWL framework.

Finally, we overview an implementation of the verification framework in Maude [CDE+07], using a built-in model-checker for LTLR. To the best of our knowledge, this tool represents the first attempt to provide a formal RWL verification environment for Web applications which allows one to verify several important classes of properties (e.g., reachability, security, authentication constraints, mutual exclusion, liveness, *etc.*) w.r.t. a *realistic* model of a Web application which includes detailed browser-server protocol interactions, browser navigation capabilities, and Web script evaluations.

This chapter is organized as follows. Section 1.1 summarizes a general model for Web interactions which informally describes the navigation through Web applications using HTTP. The model supports both Web script evaluations and adaptive navigation. In Section 1.2, we show a rewrite theory formalizing a simplified version of the navigation model of Section 1.1. In this preliminary model, we assume that a Web server interacts with a single browser which is not equipped with the usual navigation buttons. Section 1.3 provides an extended rewrite theory which generalizes the rewrite theory formalized in Section 1.2 in order to deal with multiple Web browsers which fully support the most common navigation features of modern browsers. In Section 1.4, we introduce LTLR, and we show how we can use it to formally verify Web applications. Formal Maude specifications encoding the operational semantics of the Web scripting language and the protocol evaluation mechanism can be respectively found in Appendix A and Appendix B.

1.1 A Navigation Model for Web Applications

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

Communication between the browser and the server is given through the HTTP protocol, which works following a *request-response* scheme. Basically, in the *request* phase, the browser submits a URL to the server containing the Web page P to be accessed together with a string of input

parameters (called the *query* string). Then, the server retrieves P and, if P contains a Web script α, it executes α w.r.t. the input data specified by the query string. According to the execution of α, the server defines the Web application *continuation* (that is, the next page P' to be sent to the browser), and *enables* the links in P' dynamically (*adaptive navigation*). Finally, in the *response* phase, the server delivers P' to the browser.

Since HTTP is a stateless protocol, we assume that HTTP is coupled with some session management technique, implemented by the Web server, which allows us to define Web application states via the notion of *session*, that is, global stores that can be accessed and updated by Web scripts during an established connection between a browser and the server. Web application continuations as well as adaptive navigations are dynamically computed w.r.t. the current session (i.e., the current application state).

1.1.1 Graphical Navigation Model

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

Example 1.1.1

Consider the graphical navigation model given in Figure 1.1, which represents a generic Webmail application that provides some typical functions such as login/logout features, email management, system administration capabilities, *etc.* The Web pages of the application are pairwise connected by either navigation links (i.e., solid arrows) or continuations (i.e., dashed

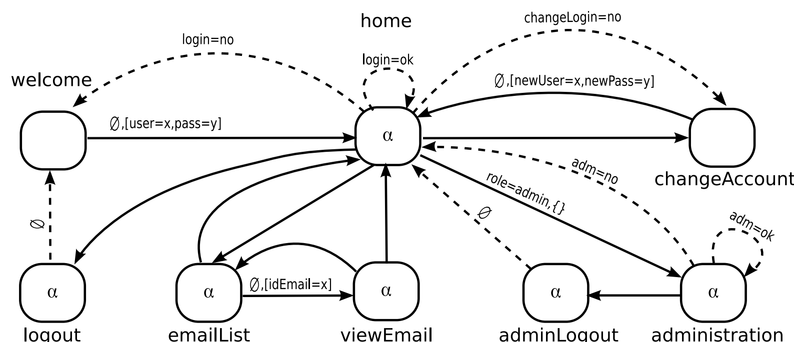

Figure 1.1: The navigation model of a Webmail application.

arrows). For example, the solid arrow between the welcome page and the home page, whose label is decorated with the string "∅,{user=x,pass=y}", defines a navigation link which is always enabled and requires two input parameters. The home page has got two possible continuations (dashed arrows) login=ok and login=no. According to the user and pass values provided in the previous transition, only continuation one is chosen. In the former case, the login succeeds and the home page is delivered to the browser, while in the latter case the login fails and the welcome page is sent back to the browser.

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

1.2 Formalizing the Navigation Model as a Rewrite Theory

In this section, we present a rewrite theory which specifies a navigation model that allows us to formalize the navigation through a Web application via a communicating protocol abstracting HTTP. Initially and to keep the model simple, we assume that the server interacts with a single browser which does not support browser actions (e.g., windows/tabs openings, refresh actions, *etc.*). Section 1.3 generalizes the model to

a more realistic scenario by defining server interactions with multiple browsers, and by equipping browsers with some standard navigation facilities.

This formalization of a Web application, first proposed in [ABR09], consists of the specification of the following three components: the Web scripting language, the Web application structure, and the communication protocol.

1.2.1 The Web Scripting Language

In [Rom11] a scripting language is considered which includes the main features of the most popular Web programming languages. Basically, it extends an imperative programming language with some built-in primitives for reading/writing session data ($\mathsf{getSession}$, $\mathsf{setSession}$), accessing and updating a data base ($\mathsf{selectDB}$, $\mathsf{updateDB}$), and capturing values contained in a query string sent by a browser ($\mathsf{getQuery}$). The language is defined by means of an equational theory (Σ_s, E_s), whose signature Σ_s specifies the syntax as well as the type structure of the language, while E_s is a set of equations modeling the operational semantics of the language through the definition of an *evaluation* operator $[\![_]\!] : \mathsf{ScriptState} \rightarrow \mathsf{ScriptState}$, where $\mathsf{ScriptState}$ is defined by the operator

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

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

1.2.2 The Web Application Structure

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

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

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

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

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

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

Example 1.2.1

Consider again the Web application of Example 1.1.1. Its Web application structure can be defined as a soup of Web pages

$$\mathsf{wapp} = \langle \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_4, \mathsf{p}_5, \mathsf{p}_6, \mathsf{p}_7, \mathsf{p}_8 \rangle$$

as follows:

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

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

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

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

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

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

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

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

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

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

$\alpha_{\mathsf{adminLogout}} =$

```
updateDB("admPage", "free")
```

$\alpha_{\mathsf{logout}} =$

```
clearSession
```

1.2.3 The Communication Protocol

The communication protocol can be defined by means of a rewrite theory (Σ_p, E_p, R_p), where (Σ_p, E_p) is an equational theory that formalizes the

Web application states, and R_p is a set of rewrite rules that specify Web script evaluations as well as request/response protocol actions.

The equational theory (Σ_p, E_p)

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

$$
\begin{array}{ll}
\mathsf{B}(_, \{_\}, \{_\}) & : (\mathsf{PageName} \times \mathsf{Url} \times \mathsf{Session}) \rightarrow \mathsf{Browser} \\
\mathsf{S}(_, \{_\}, \{_\}) & : (\mathsf{WebApplication} \times \mathsf{Session} \times \mathsf{DB}) \rightarrow \mathsf{Server} \\
\mathsf{B2S}(_, [_]) & : (\mathsf{PageName} \times \mathsf{Query}) \rightarrow \mathsf{Message} \\
\mathsf{S2B}(_, \{_\}, \{_\}) & : (\mathsf{PageName} \times \mathsf{Url} \times \mathsf{Session}) \rightarrow \mathsf{Message} \\
\mathsf{empty} & : \ \rightarrow \mathsf{Message} \\
_\|_\|_ & : \mathsf{Browser} \times \mathsf{Message} \times \mathsf{Server} \rightarrow \mathsf{WebState}
\end{array}
$$

We model a browser as a term $\mathsf{B}(\mathsf{n}, \{\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{l}\}, \{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}\})$, where n is the name of the Web page which is currently displayed on the Web browser, while $\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{l}$ is a soup of sort Url that represents the navigation links which appear in the Web page n, and $\{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}\}$ is the last session the server has sent to the browser. The sever is formalized by using a term of the form $\mathsf{S}(\langle \mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{l} \rangle, \{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}\}, \{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{k} = \mathsf{val}_\mathsf{k}\})$, where $\langle \mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{l} \rangle$ defines the Web application currently in execution, $\{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}\}$ is the session which is needed to keep track of the Web application state, and $\{\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{k} = \mathsf{val}_\mathsf{k}\}$ specifies the data base hosted by the Web server.

We assume the existence of a bidirectional channel that supports the communication between the server and browser by message passing. In this context, terms of the form $\mathsf{B2S}(\mathsf{n}, [\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}])$ model *request* messages, that is, messages sent from the browser to the server asking for the Web page n with query parameters $[\mathsf{id}_1 = \mathsf{val}_1, \ldots, \mathsf{id}_\mathsf{m} = \mathsf{val}_\mathsf{m}]$. Instead, terms of the form $\mathsf{S2B}(\mathsf{n}, \{\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{l}\}, \{\mathsf{id}'_1 = \mathsf{val}'_1, \ldots, \mathsf{id}'_\mathsf{m} = \mathsf{val}'_\mathsf{m}\})$ model *response* messages, that is, messages sent from the server to the browser including the computed Web page n together with

the navigation links $\{\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{l}\}$ occurring in n, and the current session information[1]. We denote the empty channel by the constant empty. Using the operators so far described, we can precisely formalize the notion of Web application state as a term of the form $\mathsf{br}\|\mathsf{m}\|\mathsf{sv}$, where $\mathsf{br} \in \mathsf{Browser}$, $\mathsf{m} \in \mathsf{Message}$, and $\mathsf{sv} \in \mathsf{Server}$. Intuitively, a Web application state can be interpreted as a snapshot of the system with captures the current configurations of the browser, the server and the channel.

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

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

whose semantics is specified by means of E_p (see Appendix B for the precise formalization of eval). Given a Web application w, a session s, a data base db, and a request message $\mathsf{B2S}(\mathsf{n}, [\mathsf{q}])$, $\mathsf{eval}(\mathsf{w}, \mathsf{s}, \mathsf{db}, \mathsf{B2S}(\mathsf{n}, [\mathsf{q}]))$ generates a triple $(\mathsf{s}', \mathsf{db}', \mathsf{m}')$ that consists of the updated session s', the updated data base db', and the response message $\mathsf{m}' = \mathsf{S2B}(\mathsf{n}', \{\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{m}\}, \mathsf{s}')$. Intuitively, the generation of such a triple proceeds as follows. Let α_n be the Web script occurring in the Web page n of w.

1. The server evaluates α_n by applying the evaluation function $[\![_]\!]$ to the script state $(\alpha_\mathsf{n}, \emptyset, \mathsf{s}, \mathsf{q}, \mathsf{db})$. This delivers a new script state $(\mathsf{skip}, \mathsf{m}', \mathsf{s}', \mathsf{q}, \mathsf{db}')$ in which the script's private memory, the session and the data base have been updated.

2. Then, eval returns the new session s', the new database db', and a response message $\mathsf{S2B}(\mathsf{n}', \{\mathsf{url}_1, \ldots, \mathsf{url}_\mathsf{m}\}, \mathsf{s}')$ which is built by gluing together a Web page name n' corresponding to a continuation $(\mathsf{cond}', \mathsf{n}')$ enabled w.r.t. s', the navigation links of n' enabled w.r.t. s', and the session s'.

Roughly speaking, the operator eval allows us to execute a Web script and dynamically determine (i) which Web page n' is generated by computing an enabled continuation, and (ii) which links of n' are enabled w.r.t. the current session.

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

The rewrite rule set R_p

Following [Rom11], R_p is defined by means of a collection of rewrite rules of the form

$$\mathsf{label} : \mathsf{WebState} \Rightarrow \mathsf{WebState}$$

representing the standard request-response behavior of the HTTP protocol. More specifically, R_p specifies browser requests, script evaluations, and server responses by means of the following three rules:

$$\begin{aligned}\mathsf{Req} : \mathsf{B}(\mathsf{n}, \{(\mathsf{n}_1, [\mathsf{qs}_1]), \mathsf{urls}\}, \{\mathsf{s}\}) \parallel \mathsf{empty} \parallel \mathsf{sv} \Rightarrow \\ \mathsf{B}(\mathsf{emptyPage}, \emptyset, \{\mathsf{s}\}) \parallel \mathsf{B2S}(\mathsf{n}_1, [\mathsf{qs}_1]) \parallel \mathsf{sv}\end{aligned}$$

$$\begin{aligned}\mathsf{Evl} : \mathsf{B}(\mathsf{emptyPage}, \emptyset, \{\mathsf{s}\}) \parallel \mathsf{m} \parallel \mathsf{S}(\langle \mathsf{w} \rangle, \{\mathsf{s}\}, \{\mathsf{db}\}) \Rightarrow \\ \mathsf{B}(\mathsf{emptyPage}, \emptyset, \{\mathsf{s}\}) \parallel \mathsf{m}' \parallel \mathsf{S}(\langle \mathsf{w} \rangle, \{\mathsf{s}'\}, \{\mathsf{db}'\}) \\ \text{where } \mathsf{m} = \mathsf{B2S}(\mathsf{n}_1, [\mathsf{qs}_1]) \text{ and } (\mathsf{s}', \mathsf{db}', \mathsf{m}') = \mathsf{eval}(\mathsf{w}, \mathsf{s}, \mathsf{db}, \mathsf{m})\end{aligned}$$

$$\begin{aligned}\mathsf{Res} : \mathsf{B}(\mathsf{emptyPage}, \emptyset, \{\mathsf{s}\}) \parallel \mathsf{S2B}(\mathsf{n}', \{\mathsf{urls}'\}, \{\mathsf{s}'\}) \parallel \mathsf{sv} \Rightarrow \\ \mathsf{B}(\mathsf{n}', \{\mathsf{urls}'\}, \{\mathsf{s}'\}) \parallel \mathsf{empty} \parallel \mathsf{sv}\end{aligned}$$

where $\mathsf{emptyPage} \colon \to \mathsf{PageName}$ is a constant representing a Web page without content, and $\mathsf{n}, \mathsf{n}_1, \mathsf{n}' : \mathsf{PageName}$, $\mathsf{urls}, \mathsf{urls}' : \mathsf{URL}$, $\mathsf{sv} : \mathsf{Server}$, $\mathsf{qs}_1 : \mathsf{Query}$, $\mathsf{m}, \mathsf{m}' : \mathsf{Message}$, $\mathsf{s}, \mathsf{s}' : \mathsf{Session}$, $\mathsf{db}, \mathsf{db}' : \mathsf{DB}$, $\mathsf{w} : \mathsf{WepApplication}$ are variables.

Basically, by means of rule Req, the browser requests the navigation link $(\mathsf{n}_1, [\mathsf{qs}_1])$ appearing in the current Web page n by sending a request message $\mathsf{B2S}(\mathsf{n}_1, [\mathsf{qs}_1])$ to the channel. When this happens, the emptyPage is loaded into the browser in order to avoid further browser requests until a response is obtained from the server. Rule Evl retrieves a given request message m from the channel and evaluates it. Such an evaluation updates the session and the data base on the server side with values s' and db', and generates the response message m' which is sent to the channel. Finally, through rule Res, the response message $\mathsf{S2B}(\mathsf{n}', \{\mathsf{urls}'\}, \{\mathsf{s}'\})$ is withdrawn from the channel and sent to the browser, which is then updated by using the information received.

It is worth noting that the whole protocol semantics is elegantly defined by means of only three, high-level rewrite rules without making any implementation detail explicit. Implementation details are automatically managed by the rewriting logic engine (i.e., rewrite modulo equational

theories). For instance, in the rule Req, no tricky function is needed to select an arbitrary navigation link $(\mathsf{n}_1, [\mathsf{qs}_1])$ from the URLs available in a Web page, since they are modeled as associative and commutative soups of elements (i.e., $\mathsf{Url} < \mathsf{Soup}$) and hence a single URL can be extracted from the soup by simply applying pattern matching modulo associativity and commutativity.

Example 1.2.2

Consider the Web application structure wapp specified in Example 1.2.1 together with the following two Web application states

$$\mathsf{was}_1 = \mathsf{B}(\mathsf{welcome}, \{(\mathsf{home}, [\mathsf{user} = \mathsf{Alice}, \mathsf{pass} = \mathsf{pA}])\}, \emptyset) \parallel \mathsf{empty} \parallel \mathsf{S}(\mathsf{wapp}, \emptyset, \{\mathsf{data}\})$$
$$\mathsf{was}_2 = \mathsf{B}(\mathsf{welcome}, \{(\mathsf{home}, [\mathsf{user} = \mathsf{Bob}, \mathsf{pass} = \mathsf{wrong_pB}])\}, \emptyset) \parallel \mathsf{empty} \parallel \mathsf{S}(\mathsf{wapp}, \emptyset, \{\mathsf{data}\})$$

where $\{\mathsf{data}\}$ is the data base $\{\mathsf{pwd}_{\mathsf{Alice}} = \mathsf{pA}, \mathsf{pwd}_{\mathsf{Bob}} = \mathsf{pB}, \mathsf{role}_{\mathsf{Alice}} = \mathsf{user}\}$. Then, by applying the rewrite rules of R_p to was_1, we obtain a computation trace modeling a successful login.

$$\begin{aligned}
\mathsf{was}_1 &\overset{\mathsf{Req}}{\to} \mathsf{B}(\mathsf{emptyPage}, \emptyset, \emptyset) \parallel \mathsf{B2S}(\mathsf{home}, [\mathsf{user} = \mathsf{Alice}, \mathsf{pass} = \mathsf{pA}]) \parallel \mathsf{S}(\mathsf{wapp}, \emptyset, \{\mathsf{data}\}) \\
&\overset{\mathsf{Evl}}{\to} \mathsf{B}(\mathsf{emptyPage}, \emptyset, \emptyset) \parallel \mathsf{S2B}(\mathsf{home}, \{\mathsf{urls}\}, \{\mathsf{login} = \mathsf{ok}\}) \parallel \mathsf{S}(\mathsf{wapp}, \{\mathsf{login} = \mathsf{ok}\}, \{\mathsf{data}\}) \\
&\overset{\mathsf{Res}}{\to} \mathsf{B}(\mathsf{home}, \{\mathsf{urls}\}, \{\mathsf{login} = \mathsf{ok}\}) \parallel \mathsf{empty} \parallel \mathsf{S}(\mathsf{wapp}, \{\mathsf{login} = \mathsf{ok}\}, \{\mathsf{data}\})
\end{aligned}$$

where $\mathsf{urls} = (\mathsf{changeAccount}, [\emptyset]), (\mathsf{emailList}, [\emptyset]), (\mathsf{logout}, [\emptyset])$

Note that, since the role of Alice is user, the link to the administration page is not enabled. On the other hand, by applying rules of R_p to was_2, we get a computation modeling a login failure.

$$\mathsf{was}_2 \overset{\mathsf{Req}}{\to} \mathsf{was}_3 \overset{\mathsf{Evl}}{\to} \mathsf{was}_4 \overset{\mathsf{Res}}{\to} \mathsf{B}(\mathsf{welcome}, \{(\mathsf{home}, [\mathsf{user} = \mathsf{Bob}, \mathsf{pass} = \mathsf{wrong_pB}])\}, \{\mathsf{login} = \mathsf{no}\}) \parallel \mathsf{empty} \parallel \mathsf{S}(\mathsf{wapp}, \{\mathsf{login} = \mathsf{no}\}, \{\mathsf{data}\})$$

1.3 Modeling Multiple Web Interactions and Browser Features

In a real scenario, a Web server concurrently interacts with multiple browsers through distinct connections. Besides that, the browser structure is in general more complex than the one presented in Section 1.2 —in fact, browsers are equipped with *browser navigation features* which may produce unexpected Web application behaviors as explained at the beginning of this chapter (see also [MM08]).

In the rest of the section, following [Rom11], we define a rewrite theory $(\Sigma_{ext}, E_{ext}, R_{ext})$ extending the rewrite theory (Σ_p, E_p, R_p) presented in Section 1.2 in order to manage such aspects. The augmented model generalizes the communication protocol in order to support multiple browser connections as well as the following browser navigation features: forward/backward/refresh actions, new tab/windows openings.

1.3.1 The Extended Equational Theory $(\mathbf{\Sigma_{ext}}, \mathbf{E_{ext}})$

First of all, we assume that (Σ_{ext}, E_{ext}) includes two new sorts Queue and List for modeling queues and bidirectional lists, respectively. The former data structure allows us to model the communication channel as well as the response/request messages which have to be processed by the server; while the latter is used to specify the browser history list that is needed to implement browser navigation through forward and backward buttons. Moreover, (Σ_{ext}, E_{ext}) contains the sort Nat defining natural numbers and the sort Id modeling univocal identifiers.

Extended definitions of Browser, Server, and Message are then defined by means of the following operators:

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

where we enforce the following subsort relations History $<$ List, BrowserSession $<$ Soup, Message $<$ Queue, and Browser $<$ Soup.

An extended browser is a term of the form

$$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{n}, \{\mathsf{url}\}, \{\mathsf{s}\}, \mathsf{m}, \mathsf{h}, \mathsf{i})$$

where $\mathsf{id_b}$ is an identifier representing the browser; $\mathsf{id_t}$ is an identifier modeling an open windows or tab which refers to browser $\mathsf{id_b}$; n and url are respectively the current page displayed in the window/tab $\mathsf{id_t}$ and the enabled navigation links appearing in Web page n; s is the last session received from the server; m is the last message sent to the server (this piece information is used to implement the refresh action); h is a bidirectional list recording the history of the visited Web pages; i is an internal counter used to distinguish among several response messages due to refresh actions (e.g., if a user pressed twice the refresh button, only the second refresh is displayed in the browser window).

An extended server is a term

$$\mathsf{S}(\mathsf{w}, \{\mathsf{BS}(\mathsf{id_{b1}}, \{\mathsf{s_1}\}), \ldots, \mathsf{BS}(\mathsf{id_{bn}}, \{\mathsf{s_n}\})\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}})$$

which extends the previous server definition of Section 1.2 by adding a soup of browser sessions in order to manage distinct connections, and two queues of messages $\mathsf{fifo_{req}}$,$\mathsf{fifo_{res}}$, which respectively model the request messages which still have to be processed by the server and the pending response messages that the server has still to send to the browsers.

In an analogous way, both request and response messages are augmented with information regarding the browser internal counter, and the browser and window/tab identifiers.

It is worth noting that the considered extension keep unmodified both the scripting language specification and the Web application structure which are indeed completely independent of the communicating protocol chosen.

1.3.2 The Extended Rewrite Rule Set $\mathbf{R_{ext}}$

Both the extended communication protocol supporting multiple browser connections, and the browser navigation features, are formalized by means of the rewrite rules included in R_{ext}.

The extended communication protocol

The protocol is specified via rewrite rules of the form $\mathsf{label}\colon \mathsf{Webstate} \Rightarrow \mathsf{Webstate}$, where the notion of Web application state has been adapted according to the equational theory (Σ_{ext}, E_{ext}). More specifically, a web application state is a term $\mathsf{br} \| \mathsf{m} \| \mathsf{sv}$, where br is a soup of extended browsers, m is a channel modeled as a queue of messages, and sv is an extended server. The protocol specification is as follows:

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

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

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

$$\begin{aligned}
\mathsf{ResIni}\colon\ & \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, (\underline{\mathsf{m_{idb,idt}}}, \mathsf{fifo_{res}})) \Rightarrow \\
& \quad \mathsf{br} \parallel (\mathsf{m}, \underline{\mathsf{m_{idb,idt}}}) \parallel \mathsf{S}(\mathsf{w}, \{\mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}})
\end{aligned}$$

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

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

Roughly speaking, the request phase is split into two parts, which are respectively formalized by rules ReqIni and ReqFin. Initially, when a browser with identifier $\mathsf{id_b}$ requests the navigation link $(\mathsf{np}, [\mathsf{q}])$ appearing in a Web page $\mathsf{p_c}$ of the window/tab identified by $\mathsf{id_t}$, rule ReqIni is fired. The execution of ReqIni generates a request message $\mathsf{m_{idb,idt}}$, which is enqueued in the channel and saved in the browser as the last message sent.

The history list is updated as well. Rule ReqFin simply dequeues the first request message $\mathsf{m}_{\mathsf{idb,idt}}$ of the channel and enqueues it to $\mathsf{fifo}_{\mathsf{req}}$, which is the server queue containing pending requests. Rule Evl consumes the first request message $\mathsf{m}_{\mathsf{idb,idt}}$ of the queue $\mathsf{fifo}_{\mathsf{req}}$, evaluates the message w.r.t. the corresponding browser session $(\mathsf{id_b}, \{\mathsf{s}\})$, and generates the response message which is enqeued in $\mathsf{fifo}_{\mathsf{res}}$; that is, the server queues containing the responses to be sent to the browsers. Finally, rules ResIni and ResFin implement the response phase. First, rule ResIni dequeues a response message from $\mathsf{fifo}_{\mathsf{res}}$ and sends it to the channel m. Then, rule ResFin takes the first response message from the channel queue and sends it to the window/tab of the corresponding browser.

Example 1.3.1

Consider the scenarios given in Example 1.2.2 that represent Alice's successful login and Bob's login failure. Let A be Alice's browser identifier, and let B be Bob's browser identifier. Assume that the two browsers interact simultaneously with the same server, starting from an initial state s_0.

Then, a possible computation between the browsers and the server is as follows.

$$s_0 \overset{\mathsf{ReqIni(A)}}{\rightarrow} s_1 \overset{\mathsf{ReqFin(A)}}{\rightarrow} s_2 \overset{\mathsf{ReqIni(B)}}{\rightarrow} s_3 \overset{\mathsf{ReqFin(B)}}{\rightarrow} s_4 \overset{\mathsf{Evl(B)}}{\rightarrow} s_5 \overset{\mathsf{Evl(A)}}{\rightarrow} s_6 \overset{\mathsf{ResIni(B)}}{\rightarrow} s_7 \ldots$$

where, by abuse of notation, we write $\mathsf{r(A)}$ (resp. $\mathsf{r(B)}$) to represent the fact that the variable representing the browser identifier in the rule r is instantiated with A (resp. B).

Browser navigation features

The browser navigation features are formalized in [ABR09] as follows.

$$\begin{aligned}
\mathsf{Refresh}: \; & \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \underline{\mathsf{lm}}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \Rightarrow \\
& \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{emptyPage}, \emptyset, \{\mathsf{s}\}, \underline{\mathsf{m_{idb,idt}}}, \mathsf{h}, \underline{\mathsf{i}+1}), \mathsf{br} \parallel (\mathsf{m}, \underline{\mathsf{m_{idb,idt}}}) \parallel \mathsf{sv} \\
& \text{where } \mathsf{lm} = \mathsf{B2S}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{np}, \mathsf{q}, \mathsf{i}) \text{ and } \mathsf{m_{idb,idt}} = \mathsf{B2S}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{np}, \mathsf{q}, \mathsf{i}+1)
\end{aligned}$$

$$\begin{aligned}
\mathsf{OldMsg}: \; & \mathsf{B}(\underline{\mathsf{id_b}, \mathsf{id_t}}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \underline{\mathsf{i})}, \mathsf{br} \parallel (\underline{\mathsf{S2B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p}', \mathsf{urls}', \{\mathsf{s}'\}, \mathsf{k})}, \\
& \mathsf{m}) \parallel \mathsf{sv} \Rightarrow \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \qquad \text{if } \underline{\mathsf{i} \neq \mathsf{k}}
\end{aligned}$$

$\mathsf{NewTab}\colon \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \Rightarrow$
$\mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \underline{\mathsf{B}(\mathsf{id_b}, \underline{\mathsf{id_{nt}}}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \emptyset, \emptyset, 0)}, \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv}$
where $\mathsf{id_{nt}}$ is a new fresh value of the sort Id.

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

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

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

Rules Refresh and OldMsg model the behavior of the refresh button of a Web browser. Rule Refresh applies when a Web page refresh is invoked. Basically, it increments the browser internal counter i by one unit and a new version of the last request message lm, containing the updated internal counter, is inserted into the channel queue. Note that the browser internal counter keeps track of the number of repeated refresh button clicks. Rule OldMsg is used to consume all the response messages in the channel, which might have been generated by repeated clicks of the refresh button, with the exception of the last one. This allows us to deliver just the response message corresponding to the last click of the refresh button (by using the rules ResIni and ResFin).

Finally, rules NewTab, Backward and Forward specify the behaviors of the browser buttons with regard to the generation of new tabs/windows, and the forward and backward navigation through the browser history list. The rules are quite intuitive: an application of NewTab simply generates a new Web application state containing a new fresh tab in the soup of browsers, while Backward (resp. Forward)) extracts the previous (resp. next) Web page from the history list and sets it as the current browser Web page.

It is worth noting that applications of rules in $\mathsf{R_{ext}}$ might produce and infinite number of (reachable) Web application states. For instance, infinite applications of the rule newTab generate an infinite number of

states each of which represents a distinct finite number of open tabs. Therefore, in order to make the analysis and verification feasible on our framework, we set some restrictions that limit the number of reachable states (e.g., we fixed upper bounds on the length of the history list, and on the number of windows/tabs the user can open).

An alternative approach we plan to pursue in future work, is to define a state abstraction through an equational theory, following the approach of [MPMO08], which will allow us to deal with infinite-state systems in an effective way.

1.4 Model Checking Web Applications Using LTLR

The formal specification framework presented so far is particularly suitable for verification purposes, since its fine-grained structure allows us to specify a number of subtle aspects of the Web application semantics which can be naturally verified by using model-checking techniques. To this respect, the Linear Temporal Logic of Rewriting (LTLR)[Mes08] can be fruitfully employed to model-check Web applications that are formalized via the extended rewrite theory $(\Sigma_{ext}, E_{ext}, R_{ext})$ of Section 1.3. In particular, the chosen "tandem" $LTLR/(\Sigma_{ext}, E_{ext}, R_{ext})$ allows us to formalize properties which are either not expressible or difficult to express by using other verification frameworks.

1.4.1 The Linear Temporal Logic of Rewriting

LTLR is a sublogic of the family of the Temporal Logics of Rewriting TLR* [Mes08], which allows one to specify properties of a given rewrite theory in a simple and natural way. In the following, we provide an intuitive explanation of the main features of LTRL; for a thorough discussion, we refer to [Mes08].

LTLR extends the standard Linear Temporal Logic (LTL) with *state predicates* and *spatial action patterns*. Given a system modeled as a rewrite theory $\mathcal{R}$, a state predicate is an equation of a specific sort Prop whose form is $\mathsf{statePattern} \models \mathsf{property}(\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n}) = \mathsf{booleanValue}$. Roughly speaking, a state predicate formalizes a property

$\mathsf{property}(\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n}) = \mathsf{booleanValue}$ over all the states specified by $\mathcal{R}$ which match the statePattern.

Example 1.4.1

Let (Σ_p, E_p, R_p) be the rewrite theory specified in Section 1.2, which models the Web application states as terms $\mathsf{b} \| \mathsf{m} \| \mathsf{s}$ of sort WebState where b is a browser, m is a message, and s is a server. Then, we can define the state predicate

$$\mathsf{B}(\mathsf{page}, \{\mathsf{urls}\}, \{\mathsf{session}\}) \| \mathsf{m} \| \mathsf{s} \models \mathsf{curPage}(\mathsf{page}) = \mathsf{true}$$

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

Note that, in standard LTL propositional logic, state propositions are defined via atomic constants. Instead, LTLR supports parametric state propositions via state predicates, which allows us to define complex state propositions in a very concise and simple way.

Spatial action patterns allow us to localize rewrite rule applications w.r.t. a given context and a partial substitution. Spatial action patterns have the general form $C[l(t_1, \ldots, t_n)]$, where l is a rule label, C is a context in which the rule with label l has to be applied, and $t_1, \ldots, t_n$ are terms that constrain the substitutions which instantiate the parameters of the rule l. When the context is empty, the spatial action reduces to $[l(t_1, \ldots, t_n)]$, and specifies the applications of rule l where only the substitution constraints have to be fulfilled.

Example 1.4.2

Let $(\Sigma_{ext}, E_{ext}, R_{ext})$ be the rewrite theory introduced in Section 1.3 that specifies our extended model for Web applications. Then, the spatial action pattern $\mathsf{ReqIni}(\mathsf{id} \backslash \mathsf{A})$ asserts that the general action[2]

$$\mathsf{ReqIni}(\mathsf{id}, \mathsf{p_c}, \mathsf{np}, \mathsf{q}, \mathsf{urls}, \mathsf{br}, \mathsf{m}, \mathsf{sv})$$

corresponding to applying the ReqIni rule has taken place with the rule's variable id instantiated to A. Therefore, $\mathsf{ReqIni}(\mathsf{id} \backslash \mathsf{A})$ allows us to identify all the applications of the rule ReqIni referring to the browser with identifier A.

---

[2]Note that the variables of a given rewrite rule are listed in their textual order of appearance in the left–hand side of the rule.

The syntax of the LTLR language generalizes the one of LTL[MP92] by adding state predicates and spatial action patterns to standard constructs representing logical connectives and LTL temporal operators. More precisely, LTLR is parametrized as $LTLR(SP, \Pi)$, where SP is a set of spatial action patterns, and Π is a set of state predicates. Then, LTLR formulae w.r.t. SP and Π can be defined by means of the following BNF-like syntax.

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

where $\delta \in SP$, $p \in \Pi$, and $\varphi \in LTLR(SP, \Pi)$.

1.4.2 LTLR properties for Web Applications

This section shows the main advantages of coupling LTLR with Web applications specified via the extended rewrite theory $(\Sigma_{ext}, E_{ext}, R_{ext})$ for verification purposes.

Concise and parametric properties

As LTLR is a highly parametric logic, it allows one to define complex properties in a concise way by means of state predicates and spatial action patters.

As an example, consider the Webmail application given in Example 1.1.1 and the property *"Incorrect login info is allowed only 3 times, and then login is forbidden"*.

This property might be formalized as the following standard LTL formula:

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

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

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

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

$$\begin{array}{l}
\mathsf{B}(\underline{\mathsf{id}}, \mathsf{id_t}, \underline{\mathsf{pn}}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \ \|\mathsf{m}\| \ \mathsf{sv} \models \mathsf{curPage}(\mathsf{id}, \mathsf{pn}) = \mathsf{true} \\
\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{wapp}, \{\mathsf{BS}(\underline{(\mathsf{id}, \{\mathsf{failed} = \mathsf{n}\})}, \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \models \mathsf{failedAttempt}(\mathsf{id}, \mathsf{n}) = \mathsf{true} \\
\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{wapp}, \{\mathsf{BS}(\underline{(\mathsf{id}, \{\mathsf{forbid} = \mathsf{true}\})}, \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \models \mathsf{userForbidden}(\mathsf{id}) = \mathsf{true}
\end{array}$$

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

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

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

Unreachability properties

Unreachability properties can be specified as LTLR formulae of the form

$$\Box \neg \langle \mathsf{State} \rangle$$

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

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

where State is an unwanted state the system has not to reach. By using unreachability properties over the extended rewrite theory $(\Sigma_{ext}, E_{ext}, R_{ext})$, we can detect very subtle instances of the *multiple windows problem* mentioned in [MM08].

Example 1.4.3

Consider again the Webmail application of Example 1.1.1. Assume that the user may interact with the application by using two email accounts, MA and MB. Now, let us consider a Web application state in which the user is logged in the home page with her account MA, together with the following sequence of actions: (1) the user opens a new browser window; (2) the user changes the account in one of the two open windows and logs in by using MB credentials; (3) the user accesses the $\mathsf{emailList}$ page from both windows.

After applying the previous sequence of actions, one expects to see in the two open windows the emails corresponding to the accounts MA and MB. However, the Webmail application of Example 1.1.1 shows the emails of MB in both windows. This is basically caused by action (2), which makes the server override the browser session with MB data without notifying the state change to the windows associated with the MA account.

This unexpected behavior can be recognized by using the following LTLR unreachability formula

$$\Box \neg\, \mathsf{inconsistentState}$$

where $\mathsf{inconsistentState}$ is a state predicate defined as:

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

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

Finally, it is worth nothing that by means of LTLR formulae expressing unreachability statements, we can formalize an entire family of interesting properties such as:

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

Liveness through spatial actions

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

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

Chapter 2

Debugging of Web Applications with Web-TLR

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 [WNF10].

Web-TLR [ABER10] is a software tool designed for model-checking Web applications that is based on rewriting logic [MOM02]. Web applications are expressed as rewrite theories that can be formally verified by using the Maude built-in LTLR model-checker [BM08]. 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 [ABER10], 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.

In this chapter we aim 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 [ABER11a] that allows the pieces of information that we are interested in 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.

This chapter is organized as follows. In Section 2.1 we present an extended implementation of the Web-TLR system for debugging Web applications by using backward slicing. In Section 2.3, we introduce a Case Study in Web Verification. In Section 2.4, we illustrate our methodology for interactive analysis of counterexample traces and debugging of Web Applications.

2.1 Extending the Web-TLR System

Web-TLR is a model-checking tool that implements the theoretical framework of [ABR09]. The Web-TLR system is available online via its friendly Web interface at `http://www.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

{[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 2.1: One Web state of the counter-example trace of Section 2.4.

check the property by just clicking the button Check, which invokes the Maude built-in operator tlr check[BM08] 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 2.1). Therefore, in several cases both representations may result in limited use.

In order to assist Web engineers in the debugging task, in [ABE+11] 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 the counterexamples. Roughly speaking, this technique (originally described in [ABER11a]) 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 [ABER11b].

The backward trace-slicing technique was originally implemented 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). In this master's thesis, 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. Interestedly to note 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.

2.1.1 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. 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.

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

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

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

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

2.2 Implementation of the extended Web-TLR system in RWL

The enhanced verification methodology described in this chapter has been implemented in the Web-TLR system using the high-performance, rewriting logic language Maude [CDE+07]. In this section, we discuss some of the most important features of the Maude language that we have been conveniently exploited for the optimized extension 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 [EMS03; EMM06]. 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 extended framework are as follows.

Metaprogramming

Maude is based on rewriting logic [MOM02], 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.

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 $t\sigma =_E u$, that is, $t\sigma$ 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.

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.

The overloaded operator _:_ of our implementation 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.

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" $f(b, f(f(b, a), c)) \rightarrow^*_{flat_{AC}} f(a, b, b, c) \rightarrow^*_{unflat_{AC}} f(f(b, c), f(a, b))$, 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.

2.3 A Case Study in Web Verification

We tested our tool on several complex case studies that are available at the WEB-TLR Web page and within the 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.

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

[2]See [CDE+09] (Section 4.8) for an in-depth discussion on matching and simplification modulo AC in Maude.

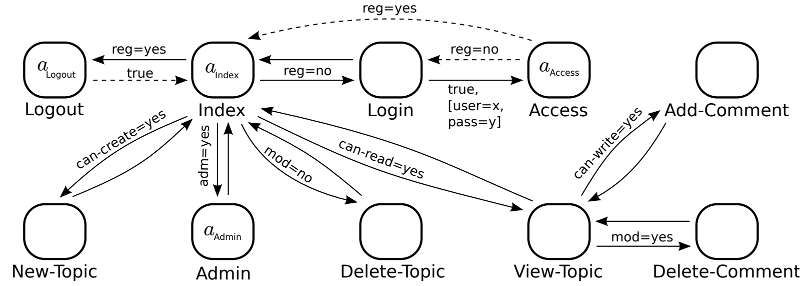

Figure 2.2: The navigation model of an Electronic Forum

The navigation model of such an application, that includes both the navigation links and the Web continuations, is given in Figure 2.2. This shows the navigation links and the Web application continuations. For example, the navigation link (solid arrow) that connects the Login and Access Web pages is always enabled and requires two input parameters (user and pass). Moreover, 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. Figure 2.3 details the formal description of the navigation model of the electronic forum application and the Web scripts involved.

In LTLR, we can define the state predicate $\mathsf{curPage}(\mathsf{id_b}, \mathsf{page})$ by means of a boolean-value function as follows,

$$[\mathsf{B}(\underline{\mathsf{id_b}}, \mathsf{id_t}, \underline{\mathsf{page}}, \mathsf{urls}, \mathsf{session}, \mathsf{sigma}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br}][\mathsf{m}][\mathsf{sv}] \models \mathsf{curPage}(\mathsf{id_b}, \mathsf{page})$$

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 $\mathsf{id_b}$.

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 difficult to express in other temporal logic frameworks (see Section 1.4.2). For example, consider the administration Web page Admin of the electronic forum application. Let us consider two administrator users whose identifiers are bidAlfred and bidAnna, respectively. Then, the

Formal description of the navigation model of the electronic forum:

$$
\begin{aligned}
\mathsf{P}_{\mathsf{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}_{\mathsf{Login}} =\ & (\mathsf{Login}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow (\mathsf{Index?}[\emptyset])) : (\emptyset \rightarrow (\mathsf{Access?}[\mathsf{user}, \mathsf{pass}]))\})\\
\mathsf{P}_{\mathsf{Access}} =\ & (\mathsf{Access}, \alpha_{\mathsf{accessScript}}, \{((\mathsf{reg} = \mathsf{yes}) \Rightarrow \mathsf{Index}) : ((\mathsf{reg} = \mathsf{no}) \Rightarrow \mathsf{Login})\}, \{\emptyset\})\\
\mathsf{P}_{\mathsf{Logout}} =\ & (\mathsf{Logout}, \alpha_{\mathsf{logout}}, \{(\emptyset \Rightarrow \mathsf{Index})\}, \{\emptyset\})\\
\mathsf{P}_{\mathsf{Admin}} =\ & (\mathsf{Admin}, \alpha_{\mathsf{admin}}, \{\emptyset\}, \{(\emptyset \rightarrow (\mathsf{Index?}[\emptyset]))\})\\
\mathsf{P}_{\mathsf{AddComment}} =\ & (\mathsf{AddComment}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{ViewTopic?}[\emptyset])\})\\
\mathsf{P}_{\mathsf{DelComment}} =\ & (\mathsf{DelComment}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{ViewTopic?}[\emptyset])\})\\
\mathsf{P}_{\mathsf{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}_{\mathsf{NewTopic}} =\ & (\mathsf{NewTopic}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{ViewTopic?}[\emptyset])\})\\
\mathsf{P}_{\mathsf{DelTopic}} =\ & (\mathsf{DelTopic}, \mathtt{skip}, \{\emptyset\}, \{(\emptyset \rightarrow \mathsf{Index?}[\emptyset])\})
\end{aligned}
$$

Electronic forum Web scripts:

α_{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
```

α_{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
```

α_{logout}

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

α_{admin}

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

Figure 2.3: Specification of the electronic forum application in Web-TLR

mutual exclusion property *"no two administrators can access the administration page simultaneously"* can be defined as follows.

$$\Box\neg(\mathsf{curPage}(\mathsf{bidAlfred}, \mathsf{Admin}) \wedge \mathsf{curPage}(\mathsf{bidAnna}, \mathsf{Admin})) \qquad (2.1)$$

Any given LTLR property can be automatically checked by using the built-in LTLR model-checker [BM08]. 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.

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

2.4 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 bidAlfred and bidAnna, respectively. Let us also recall the mutual exclusion Property 2.1 given in Section 2.3

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

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 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 2.1.1. 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 2.1. 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

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

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 $\mathcal{T} = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$, where $s_n = s$. The slicing technique proceeds backwards,

$$\mathcal{T}^\bullet = s_0^\bullet \ldots \rightarrow s_{n-6}^\bullet \overset{ScriptEval}{\rightarrow} s_{n-5}^\bullet \overset{flat/unflat}{\rightarrow} s_{n-4}^\bullet \overset{ResIni}{\rightarrow} s_{n-3}^\bullet$$
$$\overset{flat/unflat}{\rightarrow} s_{n-2}^\bullet \overset{ResFin}{\rightarrow} s_{n-1}^\bullet \overset{flat/unflat}{\rightarrow} s_n^\bullet$$

where

$s_n^\bullet =$ [B(bidAlfred, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗) : B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)] ∗ [∗][∗]

$s_{n-1}^\bullet =$ [B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗) : B(bidAlfred, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)] ∗ [∗][∗]

$s_{n-2}^\bullet =$ [B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗) : B(bidAlfred, tidAlfred, ∗, ∗, ∗, ∗, ∗, ∗, 1)]∗
[S2B(bidAlfred, tidAlfred, Admin, ∗, ∗, 1) : ∗][∗]

$s_{n-3}^\bullet =$ [B(bidAlfred, tidAlfred, ∗, ∗, ∗, ∗, ∗, ∗, 1) : B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)]∗
[∗ : S2B(bidAlfred, tidAlfred, Admin, ∗, ∗, 1)][∗]

$s_{n-4}^\bullet =$ [B(bidAlfred, tidAlfred, ∗, ∗, ∗, ∗, ∗, ∗, 1) : B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)] ∗ [∗]
[S(∗, ∗ : us(bidAlfred, ∗), ∗, (rm(S2B(bidAlfred, tidAlfred, Admin, ∗, ∗, 1), ∗, ∗) : ∗), ∗]

$s_{n-5}^\bullet =$ [B(bidAlfred, tidAlfred, ∗, ∗, ∗, ∗, ∗, ∗, 1) : B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)] ∗ [∗]
[S(∗, us(bidAlfred, ∗) : ∗, ∗, (∗ : evalScript(WEB-APP, SESSION,
B2S(bidAlfred, tidAlfred, Admin?query-empty, 1), DB)), ∗)]

$s_{n-6}^\bullet =$ [B(bidAlfred, tidAlfred, ∗, ∗, ∗, ∗, ∗, ∗, 1) : B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)] ∗ [∗]
[S(WEB-APP, (∗ : us(bidAlfred, SESSION)),
B2S(bidAlfred, tidAlfred, Admin?query-empty, 1) : ∗, ∗, DB]

Figure 2.4: Trace slice $\mathcal{T}^\bullet$.

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 2.4 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 $\mathcal{T}_{[s_{(n-6)} \ldots s_n]}$

is:

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

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 2.4.

- The sliced state $s^{\bullet}_{n}$ is the observable state that records only the relevant information defined by the slicing criterion.

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

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

- In the sliced state $s^{\bullet}_{n-4}$, 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 $s^{\bullet}_{n-5}$, 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 $s^{\bullet}_{n-4}$ is the one given as the outcome of the evaluation of the operator `evalScript` in the sliced state $s^{\bullet}_{n-5}$.

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

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(?, _, ?, _, _, _, _, _, _) Check

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 2.5: Snapshot of the WEB-TLR System.

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 2.3). A snapshot of WEB-TLR that shows the slicing process is given in Figure 2.5.

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:

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

where the new α_{Admin} is:

α_{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:

α_{index}

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

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

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 2.1. Figure 2.6 shows a snapshot of WEB-TLR for the case when a property is fulfilled.

Part II

Conditional Slicing for Rewriting Logic

Chapter 3

Conditions in Rewrite Theories

In order to understand our conditional slicing technique for rewrite theories, in this chapter we first recall some preliminary notions about the evaluation of conditional rules and equations in Maude.

Conditional rewrite rules and equations can have in Maude very general conditions involving equations, memberships, and other rewrites; that is, in their mathematical notation rules can be of the form (the form of the conditions is similar for equations):

$$l: \ t \to t' \text{ if } (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \to q_k)$$

with no restriction on which new variables may appear in the right-hand side or the condition of the equation or rule. There is no need for the condition listing first equations, then memberships, and then rewrites: this is just a notational abbreviation, since they can be listed in any order.

In Section 3.1 we explain how conditions are evaluated in Maude and the detailed syntax they follow. An example is given in Section 3.2.

3.1 Conditions in Maude

Conditions in Maude can occur in rules and equations and are evaluated from left to right, and therefore the order in which they appear, although mathematically inessential, is very important operationally.

Conditional equations are declared with this syntax:

```
ceq <Term-1> = <Term-2>
    if <EqCondition-1> /\ ... /\ <EqCondition-k>
    [<StatementAttributes>] .
```

And the syntax of conditional rules is this:

```
crl [<Label>] : <Term-1> => <Term-2>
    if <Condition-1> /\ ... /\ <Condition-k>
    [<StatementAttributes>] .
```

Let us give an illustrative example of a conditional specification in Maude.

Example 3.1.1

```
mod M is
    inc QID .

    vars X Y Z : Nat .

    ops f g : Nat Nat -> Nat .
    op h : Nat -> Nat .

    eq h(X) = X + 2 .

    crl [r1] : f(X,Y) => g(X,Z)
               if X < 5 /\ Z := h(Y) /\ Y > 1 .
endm
```

Note that we can treat equations in the same way that rules since equations are implicitly oriented from left to right.

Conditions can consist of a single statement or can be a conjunction formed with the conjunction connective /\ which is assumed to be associative.

The satisfaction of the conditions is attempted sequentially from left to right. Furthermore, the concrete syntax of conditions has two variants, namely:

- Matching Conditions $t := t'$, and
- Boolean Conditions of the form t in the kind [Bool], abbreviating the equation $t = true$.

In Example 3.1.1 there is a rule, labelled by `r1`, with a set of matching conditions $\{\mathtt{Z} := \mathtt{h}(\mathtt{Y})\}$ and a set of boolean conditions $\{\mathtt{X} < \mathtt{5}, \mathtt{Y} > \mathtt{1}\}$

Matching conditions, MC in the following, have to be handled operationally in a special way and they must satisfy special requirements. Let us illustrate it by using the matching condition $\mathtt{Z} := \mathtt{h}(\mathtt{Y})$ given in Example 3.1.1. The variable `Z` in `r1` does not appear in the left-hand side of `r1`. In the execution of `r1`, this variable becomes instantiated by matching the subject term bound to the variable `Z` against the canonical form of $\mathtt{h}(\mathtt{Y})$. Roughly speaking, we can treat matching equations as an assignment of values to fresh variables. Note that in order to obtain the canonical form of $\mathtt{h}(\mathtt{Y})$, the matching condition $\mathtt{Z} := \mathtt{h}(\mathtt{Y})$ can be seen as an *internal execution trace*. This is, let $t := s$ be a matching condition. There will be an internal execution trace such that $s \rightarrow^* t$.

Boolean conditions, BC in the following, with the form $t\ cmp\ t'$ where $cmp \in \{<, <=, >, >=, =\}$, have the usual operational interpretation, that is, for the given substitution σ, $\sigma(t)$ and $\sigma(t')$ are both reduced to canonical form and are compared for equality, modulo the algebraic laws specified as equational attributes in the module's operator declarations such as associativity, commutativity, and identity.

3.2 Conditional Rewriting Inference Process

The conditional rewriting inference process can be controlled with great flexibility in Maude by means of strategies. The Maude interpreter provides a default strategy for executing expressions in system modules that applies the rules, until no more rules can be applied, in a top-down fair way, and is provided by the *rewrite* command, in abbreviated form, *rew*. Since we assume that the equations E in a module are decomposed as the union $E = \Delta \cup B$ with B being a set of equational axioms (e.g., associativity, commutativity, and unity) declared as attributes of some operators, and Δ a set of (oriented and possibly conditional) equations modulo B. If conditions are evaluated to *true* and before the application of each rewrite rule, the expression is simplified to its canonical form by using the equations; that is, it is simplified by applying the equations Δ modulo B. Then, a rule is applied to such a simplified expression modulo

the axioms B according to the default strategy.

Roughly speaking, given two terms S_0 and S_1 and a conditional rule r, S_0 rewrites to S_1 via r if:

(*i*) S_0 can be matched within the left-hand side of r in a position w for some substitution σ,

(*ii*) the conditions in r are evaluated to $true$, and

(*iii*) S_1 only changes from S_0 at the position w by replacing $S_{0|w}$ by the instance of the right-hand side of r using σ.

Moreover, when the conditions are evaluated, fresh variables could be created and they could be used in the right-hand side of r.

The following example illustrates the conditional rewriting notions.

Example 3.2.1

Consider the following piece of Maude code:

```
mod M is
     inc QID .
     vars X Y Z : Nat .
     ops f g : Nat Nat -> Nat .
     op h : Nat -> Nat .
     op m : Nat Nat Nat -> Nat .
     eq h(X) = X + 2 .
     crl [r1] : f(X,Y) => g(X,Z)
                   if X < 5 /\ Z := h(Y) /\ Y > 1 .
     crl [r2] : m(X,Y,Z) => f(Z + Y,h(X))
                   if X > 0 /\ Z < 10 .
endm
```

and the execution command:

```
rew in M : m(2,1,3) .
```

We obtain the execution trace[1] $\mathcal{T} = \texttt{m}(2,1,3) \rightarrow \texttt{f}(4,4) \rightarrow \texttt{g}(4,6)$

Let us explain step by step how the rewriting inference process is performed along the trace $\mathcal{T}$ given in Example 3.2.1.

[1]In Appendix C we give the detailed trace delivered by Maude for this example.

As we have seen before, there exist three steps in the rewriting inference process. In the trace $\mathcal{T}$ given in Example 3.2.1 there are two rewrite steps, and for each of them we have to perform the rewriting inference process as follows:

For the initial term $S_0 = \mathtt{m}(\mathtt{2},\mathtt{1},\mathtt{3})$ and by following the default strategy for executing expressions, Maude looks in a top-down fair way for a rule whose left-hand side matches with S_0. Then, following step (i), Maude found a matching between S_0 and the left-hand side of r2 for a substitution $\sigma = \{\mathtt{x}/\mathtt{2},\mathtt{y}/\mathtt{1},\mathtt{z}/\mathtt{3}\}$. Since r2 has a set of boolean conditions $\{\mathtt{x} > \mathtt{0},\mathtt{z} < \mathtt{10}\}$, by (ii), Maude checks from left to right if both of them are evaluated to *true*. Note that σ must be applied to the set of conditions in order to evaluate them. So finally, $\mathtt{2} > \mathtt{0}$ and $\mathtt{3} < \mathtt{10}$ are evaluated to true and the rule can be applied, following (iii), witch delivers the outcome $\mathtt{f}(+(\mathtt{3},\mathtt{1}),\mathtt{h}(\mathtt{2}))$. Note that the rewriting inference process has not yet finished because equational simplification is still needed. In order to accomplish this simplification, the equation $\mathtt{h}(\mathtt{x}) = \mathtt{x}+\mathtt{2}$ has to be applied, and the resulting simplification is $\mathtt{2}+\mathtt{2}$, hence the simplified term is $S_1 = \mathtt{f}(\mathtt{4},\mathtt{4})$. Note that if the equation would have had conditions, we would have proceed recursively in the same way.

In order to obtain the normal form of S_1, Maude continues looking for a rule whose left-hand side matches with S_1. Following step (i), Maude finds a matching between S_1 and the left-hand side of r1 for a substitution $\sigma = \{\mathtt{x}/\mathtt{4},\mathtt{y}/\mathtt{4}\}$. Note that z is unbound because it is a fresh variable. In r1 there exists a set of boolean conditions $\{\mathtt{x} < \mathtt{5},\mathtt{y} > \mathtt{1}\}$, and a set of matching conditions $\{\mathtt{z} := \mathtt{h}(\mathtt{y})\}$. By (ii), Maude has to check from left to right if all boolean conditions are evaluated to *true*. In order to do this, Maude applies σ and evaluates the set of boolean conditions, so that $\mathtt{4} < \mathtt{5}$ and $\mathtt{4} < \mathtt{5}$ are evaluated to true. With respect to the set of boolean conditions, Maude has to rewrite the right-hand side of each matching condition, to its canonical form. In our case, $\mathtt{h}(\mathtt{y})$ is simplified by applying the equation $\mathtt{h}(\mathtt{x}) = \mathtt{x} + \mathtt{2}$. The resulting simplification is 6 and has to match the left-hand side of the matching condition. This matching is given by a substitution $\sigma' = \{z/6\}$ and Maude takes it to update the unbound variables in $\sigma = \{\mathtt{x}/\mathtt{4},\mathtt{y}/\mathtt{4},\mathtt{z}/\mathtt{6}\}\}$, and the condition is evaluated to true hence the rule can be applied, following (iii), which yields as outcome $\mathtt{g}(\mathtt{4},\mathtt{6})$ that can not be further simplified.

Chapter 4

Backward Trace Slicing for Conditional Rewrite Theories

In this chapter, we formalize a backward trace slicing technique for conditional rewrite theories. Our formulation is purely based on conditional rules, since (conditional) equations in Δ are treated as (conditional) rewrite rules that are used to simplify terms.

In Section 4.1, we formalize the notion of *term slice*. Section 4.2 introduces an extension of the classical pattern matching algorithm that is sensible to the slicing information. In Section 4.3, the backward conditional slicing technique is formalized and Section 4.4 demonstrates the soundness of the conditional slicing technique. Finally, Section 4.5 deploys a detailed example that illustrates how the conditional slicing technique performs in Maude.

4.1 Term Slices

A term slice is the portion of a term that contains relevant information. That is, given a term t and a set of relevant positions P, a *term slice* of t with respect to P is the portion of t that contains the information specified in P while the irrelevant information is removed.

Let us show how we can slice a term with respect to a set of relevant positions.

Definition 4.1.1 (term slice) *Let $t \in \tau(\Sigma, \mathcal{V})$ be a term, and let P be a set of relevant positions s.t. $P \subseteq \mathcal{P}os(t)$. Let $\bullet \notin \Sigma$ be a fresh variable symbol appearing nowhere else. A* term slice *of t with respect to P is defined as follows:*

$$slice(t, P) = sl_rec(t, P, \Lambda), \ \ where$$

$$sl_rec(t, P, p) = \begin{cases} t & if\ ((t \in \Sigma)\ or\ (t \in \mathcal{V}))\ and\ p \in P \\ f(sl_rec(t_1, P, p.1), \ldots, sl_rec(t_n, P, p.n)) & \\ & if\ t = f(t_1, \ldots, t_n)\ and \\ & \ \ there\ exists\ w\ s.t.\ (p.w) \in P \\ \bullet & otherwise \end{cases}$$

Note that each irrelevant subterm of t is replaced by a fresh variable symbol $\bullet$. Roughly speaking, in Definition 4.1.1 all symbols that occur within the path from the root to any relevant position in t are preserved.

Let us provide an illustrative example.

Example 4.1.2

Let $s = d(f(g(x, h(b)), y), a)$ be a term, and let $P = \{1.1.2,\ 1.2\}$ be a set of relevant positions. By applying the Definition 4.1.1, we have:

$$slice(s, P) = d(f(g(\bullet, h(\bullet)), y), \bullet)$$

In the following, we use the notation $t^\bullet$ to denote a term slice of the term t. Given a term slice $t^\bullet$, we can get the relevant positions from it, by considering the positions of $t^\bullet$ where there is not a $\bullet$. Formally.

Definition 4.1.3 (relevant positions) *Let $t^\bullet$ be a term slice. The set of relevant positions of $t^\bullet$ is defined as follows:*

$$relevant\text{-}positions(t^\bullet) = \{p \in \mathcal{P}os(t^\bullet) \mid t^\bullet_{|p} \neq \bullet\}$$

Note that the set of relevant positions of a term slice given by Definition 4.1.3 considers all the positions from the root symbol down to the leaves of t, whereas for computing a term slice (Definition 4.1.1), only the positions of the leaves are considered, because Definition 4.1.1 implicitly uses all positions on the path from the root to the position of interest.

Let us show how we can retrieve the set of relevant positions of a term slice by means of one example.

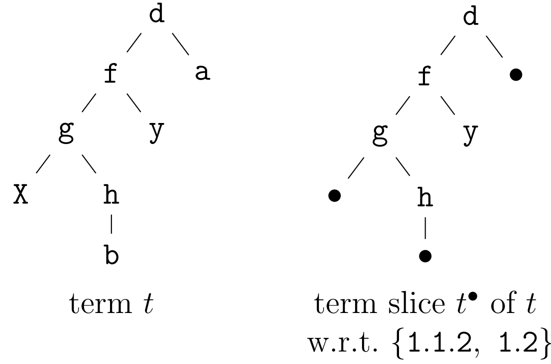

Figure 4.1: A term slice.

Example 4.1.4

Consider the term slice $t^\bullet = d(f(g(\bullet, h(\bullet)), y), \bullet)$ from Example 4.1.2. Then, by using Definition 4.1.3 we obtain the following set of relevant positions:

$$\mathit{relevant\text{-}positions}(t^\bullet) = \{\Lambda,\ 1,\ 1.1,\ 1.1.2,\ 1.2\}$$

Figure 4.1 illustrates the notion of term slice for the terms t and $t^\bullet$ of Example 4.1.4.

4.2 Extended Pattern Matching for Term Slices

Pattern matching is a technique that binds variables of a pattern to different parts of any term that fits in it. In the following, we formalize an *extended pattern matching algorithm* that allows us to deal with the fresh variable symbols $\bullet$ that stand for undetermined information in terms.

Let $F^\bullet = \{\bullet_1, \bullet_2, \ldots, \bullet_n\}$ be a set of fresh variable symbols appearing nowhere else such that $\bullet_i \notin \Sigma$ for $i = 1 \ldots, n$. We use the symbols $\bullet_i$ to

mark the irrelevant information in a term while being able to keep track of distinct irrelevant pieces of a term. When no confusion can arise, we will omit any subscript i, and we simply use $\bullet$. By notation $\Sigma^\bullet$, we denote the set $\Sigma \cup F^\bullet$.

Now we are ready to define the extended pattern matching that computes the matching of a term slice t with relation to a pattern s. More formally:

For $f/n \in \Sigma$, we define $f(\overline{\bullet}) = f(\bullet_1, \ldots, \bullet_n)$.

Definition 4.2.1 (extended pattern matching) *Given $s, t \in \tau(\Sigma \cup \mathcal{V} \cup F^\bullet)$,*

if $\exists \sigma_1 = \{\bullet_1/t_1, \ldots, \bullet_n/t_n\}$, *with* $t_i = f(\overline{\bullet})$, $i = 1, \ldots, n$,
  and $(t\sigma_1)$ *matches* s *with* $\sigma^\bullet$,
then
  $\sigma^\bullet$ *is a* $\bullet$*-matcher of* t *in* s.
else
  if $\exists \sigma_2 = \{\bullet_1/t_1, \ldots, \bullet_n/t_n\}$, *with* $t_i \in \tau(\Sigma \cup \mathcal{V} \cup F^\bullet)$, $i = 1, \ldots, n$,
    and $(t\sigma_2)$ *matches* s *with* $\sigma^\bullet$,
  then
    $\sigma^\bullet$ *is a* $\bullet$*-matcher of* t *in* s.
  fi
fi

We denote the $\bullet$-*matcher* $\sigma^\bullet$ of t in s as $match^\bullet(s, t)$.

Our extended matching algorithm has been specially devised to serve the conditional slicing technique, where a *particular value* represents information that is relevant to analyze. In other words, if there is a term with a particular value, this value should be preserved.

The following examples illustrate the notions of the extended pattern matching.

Example 4.2.2

Given terms $s = f(x, y)$ and $t = g(f(a, b))$, there is no $\bullet$-*matcher* for s and t, since $f \neq g$.

Example 4.2.3

Given terms $s = f(g(x, y))$ and $t = f(\bullet)$, we have:

$$\sigma_1 = \{\bullet/g(\bullet_1, \bullet_2)\}, \text{ and then } match^{\bullet}(s, t) = \{x/\bullet_1, y/\bullet_2\}$$

Example 4.2.4

Given terms $s = f(g(x), x)$ and $t = f(\bullet, y)$, we have:

$$\sigma_2 = \{\bullet/g(y)\}, \text{ and then } match^{\bullet}(s, t) = \{x/y\}$$

Example 4.2.5

Given terms $s = f(g(x), x, x)$ and $t = f(\bullet_1, h(a), \bullet_2)$, we have:

$$\sigma_2 = \{\bullet_1/g(h(a)), \bullet_2/h(a)\}, \text{ and then } match^{\bullet}(s, t) = \{x/h(a)\}$$

4.3 Backward Conditional Slicing

In this section, we formalize the backward conditional slicing for RWL computations.

First, we formalize the slicing criterion, which essentially represents the information we want to trace back along the execution trace in order to find out the "origins" of the data we want to observe. Given a term t, we denote by $\mathcal{O}_t$ the set of *observed* positions of t. Formally.

Definition 4.3.1 *(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 conditional slicing can be performed by means a backward conditional slicing technique for *conditional rewrite theories* by systematically applying the *match*$^{\bullet}$ function that was introduced in Definition 4.2.1. Informally, given a slicing criterion $\mathcal{O}_{S_n}$ for

an execution trace $\mathcal{T} = S_0 \to S_1 \to \ldots \to S_n$, at each rewrite step $S_{i-1} \to S_i$, $i = 1, \ldots, n$, our technique inductively computes the association between the relevant information of S_i and that in S_{i-1}. For each such rewrite step, the conditions of the rule applied are evaluated by considering the relevant and irrelevant information in S_i to be propagated back to S_{i-1}. The technique proceeds backwards, from the final term S_n to the initial term S_0. A simplified trace is obtained where each S_i is replaced by the corresponding term slice $S_i^\bullet$. Finally, we ascertain the conditions that guarantee the soundness of our conditional slicing technique (given in Section 4.4).

4.3.1 Backward Slicing for a Rewrite Step

Let us briefly recall how a conditional rewrite rule is executed. Given two terms (states) S_0 and S_1 and a conditional rule r, S_0 rewrites to S_1 via r if (i) A subterm of S_0 occurring at position w matches the left-hand side of r with substitution σ, (ii) the uninstantiated conditions in r using σ are evaluated to *true*, and (iii) S_1 consists of S_0 with the subterm of S_0 occurring at the position w replaced with the corresponding instance of the right-hand side of r by using σ. Moreover, when the conditions are evaluated, fresh variables could be considered and transfered to the right-hand side of r. Thus, our *backward rewrite step slicing* take it in account and reproduce precisely those steps in a backward special way. Namely, we slice S_1 by matching it within the right-hand side of the rule in order to obtain the relevant information into S_1, then we consider the conditions of the rule looking for information that should be tracked due to it is relevant, and finally, we slice S_0 by matching it within left-hand side of the rule preserving the relevant information.

The idea behind our slicing technique consists in considering that all the information is irrelevant, then, through the analysis of the rewrite step with respect to the slicing criterion, we will particularize the values that depend on the slicing criterion.

Given a rewrite step $S_0 \xrightarrow{r,\sigma} S_1$, the backward rewrite step slicing is defined by applying the following steps. (*Step* 1) Initialize $\sigma^\bullet$ and Slice S_1; (*Step* 2) Particularize $\sigma^\bullet$ by evaluating the MC and the BC of the rule; and (*Step* 3) Slice S_0 by using the particularized $\sigma^\bullet$.

In the following we describe the backward rewrite step slicing for

conditional rewrite rule r with respect to the slicing criterion $\mathcal{O}$. Let us consider a rewrite step $S_0 \xrightarrow{r,\sigma} S_1$ where $r = \lambda \to \rho \;\; if \; (MC, BC)$ and σ a substitution. Let w be a position in S_0 such that $S_{0|w} = \lambda\sigma$ and $S_{1|w} = \rho\sigma$.

Step 1. **Initialize $\sigma^\bullet$ and Slice S_1**

The substitution $\sigma^\bullet$ is the way that we use to transport relevant information among different rewrite steps. First, an initial $\sigma^\bullet$ is given. Then, S_1 is sliced and used for particularize $\sigma^\bullet$.

Initialize $\sigma^\bullet$. An initial $\sigma^\bullet$ is given by considering the value $\bullet$ for each variables of the rule. That is:

$$\sigma^\bullet = \{x/\bullet \mid x \in Var(r)\}$$

Slice the term S_1. Since the relevant information in S_1 is defined by the positions given by the slicing criterion $\mathcal{O}$, the slice of the term S_1 is given as follows:

$$S_1^\bullet = slice(S_1, \mathcal{O})$$

where *slice* function was formalized in Definition 4.1.1.

Particularization of $\sigma^\bullet$. The particularization of $\sigma^\bullet$ consists in updating with a relevant value (information) bound to the variables in $\sigma^\bullet$.

The extended matching algorithm allows us to get the information from $S^\bullet_{1|w}$ that is considered relevant and we will use to particularize $\sigma^\bullet$. That is

$$\begin{array}{lcl} \sigma^{\bullet\prime} & = & match^\bullet(\rho, S^\bullet_{1|w}) \\ \sigma^\bullet & = & \{(x/t) \in (\sigma^\bullet \cup \sigma^{\bullet\prime}) \mid \nexists (x/t') \in (\sigma^\bullet \cup \sigma^{\bullet\prime}) \\ & & \;\text{such that } (t' \neq \bullet) \wedge (t \neq t')\} \end{array}$$

Step 2. **Particularize $\sigma^\bullet$ by evaluating the MC and the BC of the rule**

First, we are going to consider the evaluation of MC and then BC. This is because the evaluation of MC is handled as an internal execution that could inject relevant information to be considered for evaluating BC. In order to evaluate MC, we recursively apply our backward execution trace slicing given in Definition 4.3.4.

Matching Conditions. Let $t := s$ be a matching condition from r with internal execution trace $\mathcal{T}_i = s \to^* t$ (see Section 3.1). Let $\sigma^\bullet$ be the substitution particularized in *Step* 1.

In order to continue particularizing $\sigma^\bullet$ with relevant information by evaluating the MC, we only have to analyze t because if there exists any variable candidate to be particularized in $\sigma^\bullet$, it has to be there (see Chapter 3, Section 3.2). For this, we should obtain the relevant positions of $t\sigma^\bullet$ in order to apply recursively the *backward-slicing* function given in Definition 4.3.4 with the internal execution trace and the relevant positions as arguments. The outcome of the *backward-slicing* function is the sliced trace $s^\bullet \to^* t^\bullet$. Then, we have to investigate if there exist any relevant information in $s^\bullet$ by considering the extended pattern matching from $s^\bullet$ in s. Finally the resulting $\sigma^\bullet_{\mathcal{T}_i}$ is used to particularize $\sigma^\bullet$. The statements that particularize $\sigma^\bullet$ and evaluate the MC are as follows:

$$
\begin{array}{rcl}
\mathcal{O}_{\mathcal{T}_i} & = & \textit{relevant-position}(t\sigma^\bullet) \\
(s^\bullet \to^* t^\bullet, C^\bullet) & = & \textit{backward-slicing}(s \to^* t, \mathcal{O}_{\mathcal{T}_i}) \\
\sigma^\bullet_{\mathcal{T}_i} & = & \textit{match}^\bullet(s, s^\bullet) \\
\sigma^\bullet & = & \{(x/t) \in (\sigma^\bullet \cup \sigma^\bullet_{\mathcal{T}_i}) \mid \nexists (x/t') \in (\sigma^\bullet \cup \sigma^\bullet_{\mathcal{T}_i}) \\
 & & \text{ such that } (t' \neq \bullet) \wedge (t \neq t')\}
\end{array}
$$

Note that the set of conditions $C^\bullet$ of an internal trace is not considered in the particularization, this is because this conditions are local to the internal trace and do not affect to the considered rewrite step.

Since a rule has a set of matching conditions, this step is iteratively executed for each of them until the whole set of conditions is treated.

Boolean Condition. By using the above $\sigma^\bullet$, the boolean conditions are evaluated as follows:

$$C^\bullet = \{c\sigma^\bullet \mid c \in BC\}$$

The set $C^\bullet$ constrains the domain for the concretization (see Definition 4.4.1) of the term slices.

Step 3. **Slice the term S_0 by using the particularized $\sigma^\bullet$**

Along the above steps, the substitution $\sigma^\bullet$ have been particularized with respect to the slicing criterion. In this step, we use $\sigma^\bullet$ to derive the term

slice $S_0^\bullet$ as follows:

$$S_0^\bullet = S^\bullet_{1|w}[\lambda\sigma^\bullet]$$

Roughly speaking, the term slice $S_0^\bullet$ is formed by instantiating the left-hand side λ of the rule r via the substitution $\sigma^\bullet$ at position w. Note that the redex pattern of the rule is preserved, which is the basis for ensuring the soundness, given in Section 4.4, of our conditional slicing technique.

Finally, note that if $w \notin \mathcal{P}os(S_1^\bullet)$, then $S_0^\bullet = S_1^\bullet$. That means that the information carried out along this rewrite step does not affect to the given relevant information.

In the following, we collect the above steps to formulate the backward slicing for a rewrite step.

Definition 4.3.2 (Backward rewrite step slicing) *Let $S_0 \xrightarrow{r,\sigma} S_1$ be a rewrite step where $r = \lambda \to \rho$ if (MC, BC) and σ a substitution. Let w be a position in S_0 such that $S_{0|w} = \lambda\sigma$ and $S_{1|w} = \rho\sigma$. Let $\mathcal{O}$ be a slicing criterion. Let $t := s$ be a matching condition of r with internal execution trace $\mathcal{T}_i = s \to^* t$. The backward rewrite step slicing, slice-step for abbreviation, is defined as follows:*

$$(S_0^\bullet \to S_1^\bullet, C^\bullet) = \mathit{slice\text{-}step}(S_0 \to S_1, \mathcal{O})$$

where slice-step is given by performing the following statements:

$\sigma^\bullet = \{x/\bullet \mid x \in Var(r)\}$
$S_1^\bullet = \mathit{slice}(S_1, \mathcal{O})$
If $w \notin \mathcal{P}os(S_1^\bullet)$
  then $S_0^\bullet = S_1^\bullet;\ C^\bullet = \emptyset$
  else
    $\sigma^{\bullet\prime} = \mathit{match}^\bullet(\rho, S^\bullet_{1|w})$
    $\sigma^\bullet = \{(x/t) \in (\sigma^\bullet \cup \sigma^{\bullet\prime}) \mid \nexists(x/t') \in (\sigma^\bullet \cup \sigma^{\bullet\prime})$ $\mathit{s.t.}\ (t' \neq \bullet) \wedge (t \neq t')\}$
    $\mathcal{O}_{\mathcal{T}_i} = \mathit{relevant\text{-}position}(t\sigma^\bullet)$
    $(s^\bullet \to^* t^\bullet, C^\bullet) = \mathit{backward\text{-}slicing}(s \to^* t, \mathcal{O}_{\mathcal{T}_i})$
    $\sigma^\bullet_{\mathcal{T}_i} = \mathit{match}^\bullet(s, s^\bullet)$
    $\sigma^\bullet = \{(x/t) \in (\sigma^\bullet \cup \sigma^\bullet_{\mathcal{T}_i}) \mid \nexists(x/t') \in (\sigma^\bullet \cup \sigma^\bullet_{\mathcal{T}_i})$ $\mathit{s.t.}\ (t' \neq \bullet) \wedge (t \neq t')\}$
    $C^\bullet = \{c\sigma^\bullet \mid c \in BC\}$
    $S_0^\bullet = S^\bullet_{1|w}[\lambda\sigma^\bullet]$
fi

Note that $S_0^\bullet$ (resp. $S_1^\bullet$) is the term slice from S_0 (resp. S_1) with respect to $\mathcal{O}$. Moreover, $C^\bullet$ is the set of conditions that ensures the soundness of the backward execution trace slicing given in Definition 4.4. Finally, the *backward-slicing* function is proposed within Definition 4.3.4 as well.

4.3.2 Backward Slicing for Execution Traces

In the above section, we have investigated how a rewrite step can be sliced. In this section, we extend this process to execution traces.

Given an execution trace $\mathcal{T} = S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ and a slicing criterion $\mathcal{O}_{S_n}$ that represents the information that we want to trace back along the trace, we have to process each rewrite step by applying the backward rewrite step slicing of Section 4.3.1. Roughly speaking, the *slice-step* function given in Definition 4.3.2, is applied to the rewrite step $S_{i-1} \rightarrow S_i$, and $S_{i-1}^\bullet$ and $S_i^\bullet$ are obtained. Then, the term slice $S_{n-1}^\bullet$ is used to get the relevant positions for recursively applying the *slice-step* to the precedent rewrite step $S_{i-2} \rightarrow S_{i-1}$, and so on. Moreover, in order to ensure the soundness of the technique, the sets $C^\bullet$ of conditions are carried on through the rewrite steps.

First, let us describe how the conditions from the subsequent (i.e., previously sliced) step are considered for slicing the current step.

Renaming of Variables. The set of conditions $C^\bullet$ passed back from the previously sliced rewrite step involves information that is local to that rewrite step. On the other hand, the name of the variables used in the rule applied in a rewrite step can be different for distinct rewrite steps. Therefore, in order to consider the set of conditions transferred from the previous rewrite step and avoid incorrect binding, we need to recognize when two variables with diferent names are the same and properly rename them to make them coincide.

Since the symbols $\bullet$ are handled as variables, the extended pattern matching algorithm is used to make the renaming of variables as follows.

Let us consider a rewrite step $S_0 \xrightarrow{r,\sigma} S_1$ where $r = \lambda \rightarrow \rho \ \mathit{if} \ (MC, BC)$ and σ is a substitution. Let w be a position in S_0 such that $S_{0|w} = \lambda\sigma$ and $S_{1|w} = \rho\sigma$. Let $S_p^\bullet$ and $C_p^\bullet$ be the term slice and the set of conditions from the previous rewrite step, respectively. Then, the backward rewrite

step slicing is extended with the following computations.

$$\begin{array}{rcl} \sigma_r^\bullet & = & match^\bullet(S^\bullet_{p|w}, \rho) \\ C^{\bullet\prime} & = & \{c\sigma_r^\bullet \mid c \in C_p^\bullet\} \\ C^\bullet & = & C^\bullet \cup C^{\bullet\prime} \end{array}$$

Roughly speaking, first a substitution is obtained by matching $S^\bullet_{p|w}$ within the left-hand side ρ of the applied rule r. It allows us to maintain a binding among variables of the previous rewrite step those local variables of the rule applied in the current rewrite step. Then, the carried conditions $C_p^\bullet$ are evaluated and joined to the evaluated conditions of the current rewrite step. Note that in this case, the extended matching algorithm defined in Definition 4.2.1 is not used to particularize any substitution. It is only used to maintain a binding among variables of different rewrite steps. Finally, we extend the Definition 4.3.2 as follows:

Definition 4.3.3 (Extended backward rewrite step slicing) *Let $S_0 \stackrel{r,\sigma}{\longrightarrow} S_1$ be a rewrite step where $r = \lambda \to \rho$ if (MC, BC) and σ is a substitution. Let w be a position in S_0 such that $S_{0|w} = \lambda\sigma$ and $S_{1|w} = \rho\sigma$. Let $\mathcal{O}$ be a slicing criterion. Let $t := s$ be a matching condition from r with internal execution trace $\mathcal{T}_i = s \to^* t$. Let $S_p^\bullet$ and $C_p^\bullet$ be the term slice and the set of conditions from the previously considered rewrite step, respectively. The extended backward rewrite step slicing, ex-slice-step for abbreviation, is defined as follows:*

$$(S_0^\bullet \to S_1^\bullet, C^\bullet) = \textit{ex-slice-step}(S_0 \to S_1, \mathcal{O}, S_p^\bullet, C_p^\bullet)$$

where ex-slice-step is given by performing the following statements:

$$
\begin{array}{l}
\sigma^\bullet = \{x/\bullet \mid x \in Var(r)\} \\
S_1^\bullet = slice(S_1, \mathcal{O}) \\
\textit{If } w \notin \mathcal{P}os(S_1^\bullet) \\
\quad \textit{then } S_0^\bullet = S_1^\bullet;\ C^\bullet = \emptyset \\
\quad \textit{else} \\
\quad\quad \sigma^{\bullet\prime} = match^\bullet(\rho, S^\bullet_{1|w}) \\
\quad\quad \sigma^\bullet = \{(x/t) \in (\sigma^\bullet \cup \sigma^{\bullet\prime}) \mid \nexists (x/t') \in (\sigma^\bullet \cup \sigma^{\bullet\prime}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textit{s.t. } (t' \neq \bullet) \wedge (t \neq t')\} \\
\quad\quad \mathcal{O}_{\mathcal{T}_i} = relevant\text{-}position(t\sigma^\bullet) \\
\quad\quad (s^\bullet \to^* t^\bullet, C^\bullet) = backward\text{-}slicing(s \to^* t, \mathcal{O}_{\mathcal{T}_i}) \\
\quad\quad \sigma^\bullet_{\mathcal{T}_i} = match^\bullet(s, s^\bullet) \\
\quad\quad \sigma^\bullet = \{(x/t) \in (\sigma^\bullet \cup \sigma^\bullet_{\mathcal{T}_i}) \mid \nexists (x/t') \in (\sigma^\bullet \cup \sigma^\bullet_{\mathcal{T}_i}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textit{s.t. } (t' \neq \bullet) \wedge (t \neq t')\} \\
\quad\quad C^\bullet = \{c\sigma^\bullet \mid c \in BC\} \\
\quad\quad S_0^\bullet = S^\bullet_{1|w}[\lambda\sigma^\bullet] \\
\quad\quad \sigma_r^\bullet = match^\bullet((S_p^\bullet)_{|w}, \rho) \\
\quad\quad C^{\bullet\prime} = \{c\sigma_r^\bullet \mid c \in C_p^\bullet\} \\
\quad\quad C^\bullet = C^\bullet \cup C^{\bullet\prime} \\
\textit{fi}
\end{array}
$$

If $w \notin \mathcal{P}os(S_1^\bullet)$*, then* $S_0^\bullet = S_1^\bullet$*.*

Note that $S_0^\bullet$ (resp. $S_1^\bullet$) is the term slice for S_0 (resp. S_1) with respect to $\mathcal{O}$, and $C^\bullet$ is a set of conditions that ensure the soundness of the backward execution trace slicing given in Definition 4.3.4. Note also that *backward-slicing* function is given in Definition 4.3.4.

Algorithm 1 Trace slicing.

1: **function** backward-slicing $(S_0 \to S_1 \to \ldots \to S_n,\, \mathcal{O})$
2: Let $C^\bullet_n = \{\}$
3: Let $\mathcal{O}_n = \mathcal{O}$
4: **for** $i = n$ **to** 1 **do**
5: $(S^\bullet_{i-1} \to S^\bullet_i, C^\bullet_{i-1}) \leftarrow \textit{step-slicing}(S_{i-1} \xrightarrow{r_i,\sigma_i,w_i} S_i, \mathcal{O}_i, C^\bullet_i)$
6: $\mathcal{O}_{i-1} \leftarrow \textit{relevant-positions}(S^\bullet_{i-1})$
7: **end for**
8: **return** $(S^\bullet_0 \to S^\bullet_1 \to \ldots \to S^\bullet_n, C^\bullet_0)$
9: **end function**

Now, we are ready to define the backward execution trace slicing.

Definition 4.3.4 (backward execution trace slicing) *Let $\mathcal{T} = S_0 \to S_1 \to \ldots \to S_n$ be an execution trace, and let $\mathcal{O}_{S_n}$ be a slicing criterion for $\mathcal{T}$. Then, the backward execution trace slicing, backward-slicing for abbreviation, is defined as follows:*

$$(S^\bullet_0 \to^* S^\bullet_n, C^\bullet_0) = \textit{backward-slicing}(S_0 \to^* S_n, \mathcal{O}_{S_n})$$

where

$$(S^\bullet_{i-1} \to S^\bullet_i, C^\bullet_{i-1}) = \begin{cases} \textit{ex-slice-step}(S_{n-1} \to S_n, \mathcal{O}_{S_n}, \textit{slice}(S_n, \mathcal{O}_{S_n}), \emptyset) \\ \qquad i = n \\ \textit{ex-slice-step}(S_{i-1} \to S_i, \mathcal{O}_i, S^\bullet_i, C^\bullet_i) \\ \qquad i = 1, .., n-1 \\ \qquad s.t.\ O_i = \textit{relevant-positions}(S^\bullet_i) \end{cases}$$

In Algorithms 1, 2, and 3 we provide the optimized pseudocode of our conditional slicing technique.

4.4 Soundness of the Slicing Technique

A fundamental property of our conditional slicing technique is that, given a sliced trace $\mathcal{T}^\bullet = t^\bullet_0 \to^* t^\bullet_n$, for any concretization of the term slice $t^\bullet_0$ with respect to a set of conditions $C^\bullet$, the trace slice $\mathcal{T}^\bullet$ can be reproduced. We say that a term t' is a concretization of a term slice $t^\bullet$ with respect to a set of conditions $C^\bullet$, if there exits an extended matching of t' in $t^\bullet$ such that $C^\bullet$ holds. More formally,

Algorithm 2 Step slicing.

1: **function** step-slicing ($S_{i-1} \xrightarrow{r,\sigma,w} S_i$, $\mathcal{O}_i$, $C_i^\bullet$)
2: Let $r_i = \lambda_i \to \rho_i$ if (MC_i, BC_i) be the rule applied in the rewrite step i.
3: $S_i^\bullet \leftarrow \mathit{slice}(S_i, \mathcal{O}_i)$
4: **if** $w \notin \mathcal{P}os(S_i^\bullet)$ **then**
5: **return** $S_i^\bullet \to S_i^\bullet$ if $C_i^\bullet$
6: **end if**
7: $\sigma_i^\bullet \leftarrow \mathit{match}^\bullet((S_i^\bullet)_{|w}, \rho_i)$
8: $C_i^{\bullet\prime} \leftarrow \sigma_i^\bullet(C_i^\bullet)$
9: $\sigma_i^{\bullet\prime} \leftarrow \mathit{match}^\bullet(\rho_i, (S_i^\bullet)_{|w})$
10: $\sigma_i^{\bullet\prime\prime} \leftarrow \{x/t \mid x \in Var(r_i) \wedge (t = \bullet \wedge \nexists t' \text{ s.t. } (x/t') \in \sigma_i^{\bullet\prime}) \vee ((x/t) \in \sigma_i^{\bullet\prime})\}$
11: $\sigma_i^{\bullet\prime\prime\prime} \leftarrow \mathit{mc\text{-}slicing}(\sigma_i^{\bullet\prime\prime}, \mathrm{MC}_i)$
12: $C_i^\bullet \leftarrow \sigma_i^{\bullet\prime\prime\prime}(C_i^{\bullet\prime} \wedge \mathrm{BC}_i)$
13: **return** $(S_i^\bullet)_{|w}[\lambda_i \sigma_i^{\bullet\prime\prime\prime}] \to S_i^\bullet$ if $C_i^\bullet$
14: **end function**

Algorithm 3 Matching condition slicing.

1: **function** mc-slicing ($\sigma^\bullet, \{t_1 := s_1, \ldots, t_n := s_n\}$)
2: **for** $i = n$ **to** 1 **do**
3: Let $s \to^* t$ be the internal trace $\mathcal{T}_i$ of a matching condition $t := s$
4: $\mathcal{O}_{\mathcal{T}_i} \leftarrow \mathit{relevant\text{-}positions}(\sigma^\bullet(t))$
5: $(s^\bullet \to^* t^\bullet, C^\bullet) \leftarrow \mathit{backward\text{-}slicing}(s \to^* t, \mathcal{O}_{\mathcal{T}_i})$
6: $\sigma^\bullet_{\mathcal{T}_i} \leftarrow \mathit{match}^\bullet(s, s^\bullet)$
7: $\sigma^\bullet \leftarrow \sigma^\bullet_{\mathcal{T}_i} \cup \{(x/t) \mid (x/t) \in \sigma^\bullet \wedge \nexists t' \text{ s.t. } (x/t') \in \sigma^\bullet_{\mathcal{T}_i}\}$
8: **end for**
9: **return** $\sigma^\bullet$
10: **end function**

Definition 4.4.1 (term slice concretization) *Let $t' \in \tau(\Sigma)$ be a term. Let $t^\bullet$ be a term slice and $C^\bullet$ be a set of conditions for $t^\bullet$. We say that t' is a concretization of $t^\bullet$ w.r.t. $C^\bullet$ (in symbols $t^\bullet \propto^{C^\bullet} t'$) if the following conditions hold:*

$$1)\ \exists \sigma^\bullet \text{ such that } \sigma^\bullet = match^\bullet(t^\bullet, t')$$
$$2)\ \forall c \in C^\bullet, c\sigma^\bullet \equiv true$$

Figure 4.2 illustrates the notions of term slice concretization for a given term t w.r.t. the set of positions $\{1.1.2, 1.2\}$.

Example 4.4.2

Let $t^\bullet = f(\bullet_1, g(4, \bullet_2))$ be a term slice and let $C^\bullet = \{\bullet_1 > 5 \wedge \bullet_2 = 7\}$ be a set of conditions for $t^\bullet$. Let $t'_1 = f(3, g(4, 7))$ and $t'_2 = f(6, g(4, 7))$ be two terms. By considering Definition 4.4.1, we have:

- t'_1 is not a concretization of $t^\bullet$ w.r.t. $C^\bullet$ since there exists a condition

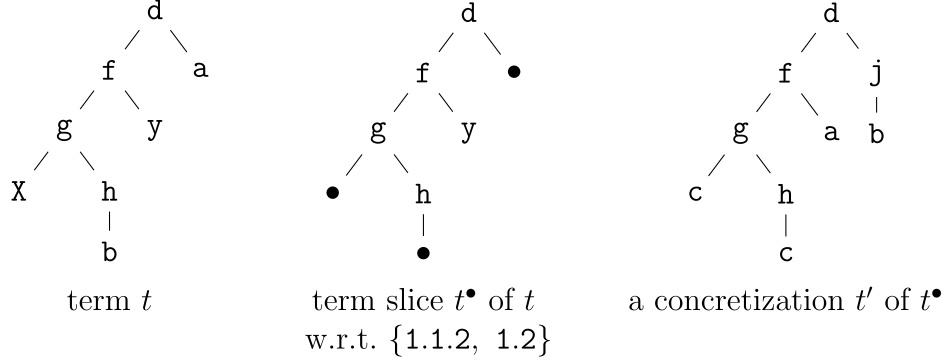

Figure 4.2: A term slice concretization.

in $C^\bullet$ that is not satisfied. Namely,

$$\sigma_1^\bullet = \mathit{match}^\bullet(t^\bullet, t'_1) = \{\bullet_1/3,\ \ \bullet_2/7\}$$
$$(\bullet_1 > 5)\sigma_1^\bullet \equiv (3 > 5) \equiv \textbf{\textit{false}} \ \ \wedge\ (\bullet_2 = 7)\sigma_1^\bullet \equiv (7 = 7) \equiv \mathit{true}$$

- t'_2 is a concretization of $t^\bullet$ w.r.t. $C^\bullet$.

$$\sigma_2^\bullet = \mathit{match}^\bullet(t^\bullet, t'_2) = \{\bullet_1/6,\ \ \bullet_2/7\}$$
$$(\bullet_1 > 5)\sigma_2^\bullet \equiv (6 > 5) \equiv \mathit{true}\ \wedge\ (\bullet_2 = 7)\sigma_2^\bullet \equiv (7 = 7) \equiv \mathit{true}$$

Now we are ready to demonstrate the soundness of our conditional slicing technique. Roughly speaking, the soundness property ensures that the rules involved in the sliced step of $\mathcal{T}^\bullet$ can be applied again, at the corresponding positions, to every concrete trace $\mathcal{T}'$ that we can obtain by instantiating all the fresh symbols in $t_0^\bullet$ with arbitrary terms.

Theorem 4.4.3 *(soundness) Let $\mathcal{R}$ be a 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 $\mathcal{T}^\bullet : t_0^\bullet \stackrel{r_1}{\to} t_1^\bullet \ldots \stackrel{r_n}{\to} t_n^\bullet$ and $C_0^\bullet$ be the corresponding trace slice and the set of conditions w.r.t. $\mathcal{O}$. Then, for any concretization t'_0 of $t_0^\bullet$ w.r.t. $C_0^\bullet$, it holds that $\mathcal{T}' : t'_0 \stackrel{r_1}{\to} t'_1 \ldots \stackrel{r_n}{\to} t'_n$ is an execution trace in $\mathcal{R}$, and $t_i^\bullet \propto^{C_i^\bullet} t'_i$, for $i = 1, \ldots, n$.*

The proof of Theorem 4.4.3 is similar to the corresponding proof for unconditional theories of [Rom11]. It relies on the fact that redex patterns are preserved by backward trace slicing (see Step 3 in Section 4.3.1). Therefore, for $i = 1, \ldots, n$, the rule r_i can be applied to any concretization t'_{i-1} of the 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}.

Note that our basic framework enjoys neededness of the extracted information (in the sense of [TeR03]) as well, since the information captured by every sliced rewrite step in a trace slice is all and the only information that is needed to produce the relevant information in the reduced term for the subsequent step.

4.5 Slicing of a Maude Example Trace

In this section, we introduce a Maude example that allows us to explain how our conditional slicing performs.

Example 4.5.1

We consider the following Maude module:

```
mod M is
    inc QID .
    vars X Y Z : Nat .
    ops f g n k : Nat Nat -> Nat .
    ops h p : Nat -> Nat .
    op  m : Nat Nat Nat -> Nat .
    eq h(X) = X + 2 .
    eq k(X,Y) = p(sd(X,2)) .
    crl [r1] : f(X,Y) => g(X,Z)
               if X < 5 /\ Z := h(Y) /\ Y > 1 .
    crl [r2] : m(X,Y,Z) => f(X,Y)
               if X < 8 /\ Z < 10 .
    crl [r3] : n(X,Y) => m(X,Z,h(Y))
               if p(Z) := k(X + 2,Y) .
endm
```

Then an execution trace in this program is:

$\mathcal{T} = S_0 \xrightarrow{r3} S_1 \xrightarrow{eq.simpl.} S_2 \xrightarrow{r2} S_3 \xrightarrow{r1} S_4$

this is,

$\mathcal{T} = \mathtt{n}(4,1) \xrightarrow{\mathtt{r3}} \mathtt{m}(4,4,\mathtt{h}(1)) \xrightarrow{\mathtt{eq.simpl.}} \mathtt{m}(4,4,3) \xrightarrow{\mathtt{r2}} \mathtt{f}(4,4) \xrightarrow{\mathtt{r1}} \mathtt{g}(4,6)$

Assume we want to simplify the trace by using our conditional slicing technique.

Our backward execution trace slicing accepts an execution trace and a slicing criterion. Then, by using Algorithms 1 and 2, we slice the trace by slicing each rewrite step. If the rule applied in a rewrite step has matching conditions, we use Algorithm 3 in order to take into account possible relevant information within them.

Our technique starts by using the *backward-slicing* function defined in Algorithm 1. We are going to slice the trace $\mathcal{T}$ with relation to the slicing criterion $\mathcal{O}_{S_4} = \{2\}$.

As mentioned before, in order to slice a trace, Algorithm 1 applies, for each of these rewrite steps, the *step-slicing* function defined in Algorithm 2. This function gives us the mechanism to slice a rewrite step $S_{i-1} \xrightarrow{r,\sigma,w} S_i$ where $r = \lambda \to \rho$ if (MC, BC). See lines 1-2 of Algorithm 2.

Slicing the Rewrite Steps

For the **last rewrite step** $f(4,4) \xrightarrow{r1} g(4,6)$, in the sequence, and by using the *step-slicing* function defined in Algorithm 2, we undertake the function call:

$$\textit{step-slicing}(f(4,4) \xrightarrow{r1,\sigma,w} g(4,6), \{2\}, \{\}) \quad \text{where } w = \lambda$$

Since this is the first considered rewrite step, the input set of conditions is empty.

In Algorithm 2, line 3 computes the slice $S_i^\bullet$ of the term S_i in the rewrite step $S_{i-1} \to S_i$ with relation to $\mathcal{O}_i$. This is, we know that there is in $\mathcal{O}_i$ all the relevant information that we want to trace back, so we can apply the slice function, defined in Definition 4.1.1, to the term S_i to discard all the positions that we do not want to trace.

$$S_4^\bullet \leftarrow slice(g(4,6),\{2\}) \quad \text{that is, } S_4^\bullet = g(\bullet,6)$$

In order to know if the information carried out along this rewrite step does not affect to the relevant information of the trace, lines 4-6 check if w is in $\mathcal{P}os(S_i^\bullet)$, in that case the process continues, otherwise, in line 5 the function returns $S_i^\bullet \rightarrow S_i^\bullet$ telling us that there is no relevant information in this rewrite step.

In our case, we have that $w = \lambda$ and $\mathcal{P}os(g(\bullet,6)) = \{\lambda, 2\}$. Since $\lambda \in \{\lambda, 2\}$, the algorithm continues.

Lines 7-8 deal with the renaming of variables. Throughout each rewrite step slicing, the set of conditions has been extended with boolean conditions occurring in previous sliced rewrite steps. So, when we are doing the slicing of a rewrite step, we have to rename the variables of set of conditions that come from previous rewrite steps. Actually we are using the extended matching function defined in Definition 4.2.1 to match the sliced term $(S_i)^\bullet_{|w}$ of the rewrite step within right-hand side ρ of the rule applied. Algorithm 2 uses the computed abstract substitution $\sigma_4^\bullet$ to rename the variables as follows:

$$\begin{array}{ll} \sigma_4^\bullet \leftarrow match^\bullet(g(\bullet,6), g(X,Z)) & \text{that is, } \sigma_4^\bullet = \{\bullet/X\} \\ C_4^{\bullet\prime} \leftarrow \sigma_4^\bullet(\{\}) & \text{that is, } C_4^{\bullet\prime} = \{\} \end{array}$$

In this last rewrite step we have an empty set of conditions, so no rename is needed.

Lines 9-10 build a new abstract substitution $\sigma_4^{\bullet\prime\prime}$ that is used in the *mc-slicing* function defined in Algorithm 3. We need to use again the extended matching function defined in Definition 4.2.1 to match the sliced term $(S_4)^\bullet_{|w}$ within the right-hand side ρ of the rule. This matching allows us obtain $\sigma_4^{\bullet\prime}$ where the irrelevant variables have been marked as $\bullet$. Line 10 builds the $\sigma_4^{\bullet\prime\prime}$ ensuring that all variables that exist in the rule are in $\sigma_4^{\bullet\prime}$ and marks them as irrelevant if they did not exist before.

$$\begin{array}{ll} \sigma_4^{\bullet\prime} \leftarrow match^\bullet(g(X,Z), g(\bullet,6)) & \text{that is, } \sigma_4^{\bullet\prime} = \{X/\bullet, Z/6\} \\ \text{and after applying line 10,} & \sigma_4^{\bullet\prime\prime} = \{X/\bullet, Y/\bullet, Z/6\} \end{array}$$

Note that the binding $Y/\bullet$ is added to $\sigma_4^{\bullet\prime\prime}$ because the variable Y exists in the rule r1 and this is the rule applied in the considered rewrite step.

Line 11 has to deal with matching conditions $MC_4 = \{Z := h(Y)\}$ of the rule r1 applied in the considered rewrite step. We need to particularize $\sigma_4^{\bullet\prime\prime}$ in order to retrieve all the relevant information from the set of matching conditions. In order to do this, we have to use the *mc-slicing* function defined in Algorithm 3 by invoking the call:

$$\textit{mc-slicing}(\sigma_4^{\bullet\prime\prime}, MC_4)$$

We explain in the next section how obtain a particularized $\sigma_i^{\bullet\prime\prime\prime}$ by applying the *mc-slicing* function to the set of matching conditions MC_i. In our example, this function returns:

$$\sigma_4^{\bullet\prime\prime\prime} = \{X/\bullet, Y/4, Z/6\}$$

Line 12 uses $\sigma_4^{\bullet\prime\prime\prime}$ to evaluate the set of conditions that are obtained by joining the current set of conditions, and the set of boolean conditions BC of the rule applied as follows:

$$\begin{array}{ll} C_4^\bullet \leftarrow \sigma_4^{\bullet\prime\prime\prime}(\{\} \wedge \{Y{>}1, X{<}5\}) & \text{that is, } C_4^\bullet = \{4{>}1, \bullet{<}5\} \\ \text{where } 4{>}1 \text{ is true,} & \text{so finally } C_4^\bullet = \{\bullet{<}5\} \end{array}$$

Finally, line 13 builds $S_{i-1}^\bullet$ by preserving the redex pattern and returns the sliced rewrite step which is as follows:

$$f(\bullet, 4) \to g(\bullet, 6) \text{ if } \{\bullet{<}5\}$$

Note that we use $\sigma_4^{\bullet\prime\prime\prime}$ to build $S_3^\bullet$. Also, we return the evaluated set of conditions $C_4^\bullet$ in order to be used in the subsequent rewrite step.

Following Algorithm 1, once a rewrite step is sliced, we have to get the slicing criterion that will be used in the next considered rewrite step; see line 6. In order to do it, we use $S_3^\bullet$ of the recently sliced rewrite step to get the relevant positions, defined in Definition 4.1.3, as follows:

$$\mathcal{O}_3 \leftarrow \textit{relevant-positions}(f(\bullet, 4)) \quad \text{that is, } \mathcal{O}_3 = \{2\}$$

Now we consider the precedent, **third rewrite step**:

$$\textit{step-slicing}(m(4,4,3) \xrightarrow{r2,\sigma,w} f(4,4), \{2\}, \{\bullet{<}5\}) \quad \text{where } w = \lambda$$

Note that in this rewrite step we consider the set of conditions $\{\bullet{<}5\}$ that comes from the previously considered (last) rewrite step.

We proceed in the same way described for the previously considered rewrite step. First, we apply the slice function, defined in Definition 4.1.1, by means of the slicing criterion $\mathcal{O}_3$ to the term S_3 in order to discard all positions that we do not want to trace.

$$S_3^\bullet \leftarrow slice(f(4,4),\{2\}) \quad \text{that is, } S_3^\bullet = f(\bullet,4)$$

Now, we check if w is in $\mathcal{P}os(S_3^\bullet)$. Effectively $\lambda \in \{\lambda, 2\}$, so we can continue.

Next, renaming of variables:

$$\begin{array}{ll} \sigma_3^\bullet \leftarrow match^\bullet(f(\bullet,4), f(X,Y)) & \text{that is, } \sigma_3^\bullet = \{\bullet/X\} \\ C_3^{\bullet\prime} \leftarrow \sigma_3^\bullet(\{\bullet{<}5\}) & \text{that is, } C_3^{\bullet\prime} = \{X{<}5\} \end{array}$$

Note how the renaming of variables in this sliced step is performed.

The corresponding $\sigma_3^{\bullet\prime\prime}$ that will be used in the *mc-slicing* function defined in Algorithm 3 is obtained as follows:

$$\begin{array}{ll} \sigma_3^{\bullet\prime} \leftarrow match^\bullet(f(X,Y), f(\bullet,4)) & \text{that is, } \sigma_3^{\bullet\prime} = \{X/\bullet, Y/4\} \\ \text{and after applying line 10,} & \sigma_3^{\bullet\prime\prime} = \{X/\bullet_1, Y/4, Z/\bullet_2\} \end{array}$$

Since there are no matching conditions in r2, we do not need to call the *mc-slicing* function. So, our $\sigma_3^{\bullet\prime\prime}$ is ready to be applied to the set of conditions obtained by joining the current set of conditions and the set of boolean conditions BC of the rule applied as follows:

$$\begin{array}{ll} C_3^\bullet \leftarrow \sigma_3^{\bullet\prime\prime}(\{X{<}5\} \wedge \{X{<}8, Z{<}10\}) & \text{that is, } C_3^\bullet = \{\bullet_1{<}5, \bullet_1{<}8, \bullet_2{<}10\} \\ \text{where } C_3^\bullet \text{ can be simplified,} & \text{so finally } C_3^\bullet = \{\bullet_1{<}5, \bullet_2{<}10\} \end{array}$$

Finally, the sliced rewrite step is:

$$m(\bullet_1, 4, \bullet_2) \rightarrow f(\bullet_1, 4) \text{ if } \{\bullet_1{<}5, \bullet_2{<}10\}$$

Note how the set of conditions has been increased with conditions from the previous rewrite step and the rename of variables has been correctly performed.

Note also that following Algorithm 1, we have to get the slicing criterion that will be used in the next considered rewrite step as follows:

$$\mathcal{O}_2 \leftarrow \mathit{relevant\text{-}positions}(m(\bullet_1, 4, \bullet_2)) \quad \text{that is, } \mathcal{O}_2 = \{2\}$$

Now for the **second rewrite step**:

$$\mathit{step\text{-}slicing}(m(4,4,h(1)) \xrightarrow{eq.simpl.,\sigma,w} m(4,4,3), \{2\}, \{\bullet_1{<}5, \bullet_2{<}10\}) \\ \text{where } w = 3$$

In this rewrite step we consider the set of conditions $\{\bullet_1{<}5, \bullet_2{<}10\}$ that comes from the previous sliced rewrite steps.

We proceed in the same way described before. First, we apply the slice function, defined in Definition 4.1.1, to the term S_2 in order to discard all positions that we do not want to trace.

$$S_2^\bullet \leftarrow \mathit{slice}(m(4,4,3), \{2\}) \quad \text{that is, } S_2^\bullet = m(\bullet_1, 4, \bullet_2)$$

Now, we check if w is in $\mathcal{P}os(S_2^\bullet)$. In this case, $3 \notin \{\lambda, 2\}$. Then, we finish with the sliced rewrite step as follows:

$$m(\bullet_1, 4, \bullet_2) \to m(\bullet_1, 4, \bullet_2) \text{ if } \{\bullet_1{<}5, \bullet_2{<}10\}$$

Note that in this rewrite step we just use $S_2^\bullet$ to build the resulting sliced rewrite step. This is because this rewrite steps does not have any relevant information with respect to the analized trace. Conditions are passed trough without changes.

Now again, following Algorithm 1, we have to get the slicing criterion that will be used in the next considered rewrite step as follows:

$$\mathcal{O}_1 \leftarrow \mathit{relevant\text{-}positions}(m(\bullet_1, 4, \bullet_2)) \quad \text{that is, } \mathcal{O}_1 = \{2\}$$

Finally consider the **first rewrite step** in the original trace:

$$\mathit{step\text{-}slicing}(n(4,1) \xrightarrow{r3.,\sigma,w} m(4,4,h(1), \{2\}, \{\bullet_1{<}5, \bullet_2{<}10\}) \quad \text{where } w = \lambda$$

First, we apply the slice function, defined in Definition 4.1.1, to the term S_1 in order to discard all positions that we do not want to trace.

$$S_1^\bullet \leftarrow slice(m(4,4,h(1)),\{2\}) \quad \text{that is, } S_1^\bullet = m(\bullet_1,4,\bullet_2)$$

Next, we check if w is in $\mathcal{P}os(S_1^\bullet)$. Effectively, $\lambda \in \{\lambda, 2\}$.

Then, rename of variables:

$$\begin{array}{ll} \sigma_1^\bullet \leftarrow match^\bullet(m(\bullet_1,4,\bullet_2), m(X,Z,h(Y))) & \text{that is, } \sigma_1^\bullet = \{\bullet_1/X, \bullet_2/h(Y)\} \\ C_1^{\bullet\prime} \leftarrow \sigma_2^\bullet(\{\bullet_1{<}5, \bullet_2{<}10\}) & \text{that is, } C_1^{\bullet\prime} = \{X{<}5, h(Y){<}10\} \end{array}$$

Finally, we get the substitution $\sigma_1^{\bullet\prime\prime}$ that will be used in the *mc-slicing* function defined in Algorithm 3 as follows:

$$\begin{array}{ll} \sigma_1^{\bullet\prime} \leftarrow match^\bullet(m(X,Z,h(Y)), m(\bullet_1,4,\bullet_2)) & \text{that is, } \sigma_1^{\bullet\prime} = \{X/\bullet_1, Z/4, Y/\bullet_2\} \\ \text{and after apply line 10,} & \sigma_1^{\bullet\prime\prime} = \{X/\bullet_1, Z/4, Y/\bullet_2\} \end{array}$$

In the next section we explain how obtain a particularized $\sigma_i^{\bullet\prime\prime\prime}$ by applying the *mc-slicing* function to the set of matching conditions MC_i. In our example, this function returns:

$$\sigma_1^{\bullet\prime\prime\prime} = \{X/4, Z/4, Y/\bullet_2\}$$

Now, the substitution is ready to be applied to the set of conditions that are obtained by joining the current set of conditions and the set of boolean conditions BC of the rule applied as follows:

$$\begin{array}{ll} C_1^\bullet \leftarrow \sigma^\bullet(\{X{<}5, h(Y){<}10\} \wedge \{\}) & \text{that is, } C_1^\bullet = \{4{<}5, h(\bullet_2){<}10\} \\ \text{where } 4{<}1 \text{ is true,} & \text{so finally } C_1^\bullet = \{h(\bullet_2){<}10\} \end{array}$$

Note that there is no boolean conditions in $r3$.

Finally, the sliced rewrite step is:

$$n(4,\bullet_2) \rightarrow m(\bullet_1,4,\bullet_2) \text{ if } \{h(\bullet_2){<}10\}$$

Following Algorithm 1, line 8 returns the full trace slice $\mathcal{T}^\bullet$ that is:

$$\mathcal{T}^{\bullet} = n(4, \bullet_2) \rightarrow m(\bullet_1, 4, \bullet_2) \rightarrow m(\bullet_1, 4, \bullet_2) \rightarrow f(\bullet_1, 4) \rightarrow g(\bullet, 6) \text{ if } \{h(\bullet_2){<}10\}$$

Note that the simplification is obvious. The presented example is so easy to appreciate the efficiency of our technique with relation to the reduction of information in a trace. Anyway, is easily appreciate that our technique will reduce complex traces that are out of the scope of this easy explanation algorithm.

Matching Conditions

Given the execution trace $\mathcal{T}$ of the Example 4.5.1, the last and first rewrite steps have matching conditions in the rules applied. Then, we have to use the *mc-slicing* function defined in Algorithm 3 in order to particularize the substitution $\sigma^{\bullet}$ given as a parameter. Then this function accepts a substitution $\sigma^{\bullet}$ that will be particularized, and a set of matching conditions to discover relevant information.

For the **last rewrite step** in the trace $\mathcal{T}$ given in the section above by using the *mc-slicing* function defined in Algorithm 3, we undertake, with the substitution $\sigma_4^{\bullet}$ and the MC as parameters, the function call:

$$\textit{mc-slicing}(\{X/\bullet, Y/\bullet, Z/6\}, \{Z := h(Y)\})$$

We can handle the evaluation of the matching condition as if it were an internal trace $\mathcal{T}_i$ (see Section 3.1) as follows:

$$Z := h(Y) \quad \text{that is, } \mathcal{T}_i = h(Y) \rightarrow +(X, 2) \rightarrow Z$$

Note that in this case, the internal trace has only two rewrite steps. We have to get relevant positions (see line 4 of Algorithm 3) of the last rewrite step of the internal trace with the application of the substitution $\sigma_4^{\bullet}$ as follows:

$$\mathcal{O}_{\mathcal{T}_{i4}} = \textit{relevant-position}(\sigma_4^{\bullet}(Z)) \quad \text{that is, } \mathcal{O}_{\mathcal{T}_{i4}} = \{\lambda\}$$

Now by applying recursively the Algorithm 1 (see line 5 of Algorithm 3), with the internal trace and the slicing criterion as parameters, we obtain an internal trace slice as follows:

$$\mathcal{T}_i^{\bullet} = h(4) \rightarrow +(4, 2) \rightarrow 6$$

By using the extended matching to match the first state of the internal trace slice $\mathcal{T}_i^{\bullet}$ within the first state of the original internal trace $\mathcal{T}_i$ (see line 6 of Algorithm 3), we obtain:

$$\sigma^{\bullet}_{\mathcal{T}_{i4}} \leftarrow match^{\bullet}(h(Y), h(4)) \quad \text{that is, } \sigma^{\bullet}_{\mathcal{T}_{i4}} = \{Y/4\}$$

Finally by joining $\sigma^{\bullet}_{\mathcal{T}_{i4}}$ with $\sigma^{\bullet}_4$ (see line 7 of Algorithm 3) we have:

$$\sigma^{\bullet}_4 = \{X/\bullet, Y/4, Z/6\}$$

Note that this $\sigma^{\bullet}_4$ is the substitution returned by the *mc-slicing* function.

Since the input of *mc-slicing* function defined in Algorithm 3 is a set of matching conditions, last statements must be iteratively executed for each of them until the whole set of conditions has been processed. In this case, there is only one matching condition.

Regarding the **first rewrite step** in the chain, we undertake with the substitution $\sigma^{\bullet}_1$ and the MC as parameters, the function call:

$$mc\text{-}slicing(\{X/\bullet_1, Z/4, Y/\bullet_2\}, \{p(Z) := k(X+2, Y)\})$$

The associated internal trace $\mathcal{T}_i$ of this matching condition is obtained as follows:

$$p(Z) := k(X+2, Y) \quad \text{that is, } \mathcal{T}_i = k(+(X,2), Y) \to p(sd(X,2)) \to p(Z)$$

In this case, the internal trace has three rewrite steps.

In order to get relevant positions (following line 4 of Algorithm 3) of the last rewrite step of the internal trace with the application of the substitution $\sigma^{\bullet}_1$, we use the statement:

$$\mathcal{O}_{\mathcal{T}_{i1}} = relevant\text{-}position(\sigma^{\bullet}_1(p(Z))) \quad \text{that is, } \mathcal{O}_{\mathcal{T}_{i1}} = \{1\}$$

Now by applying recursively the Algorithm 1 (see line 5 of Algorithm 3), with the internal trace and the slicing criterion as parameters, we obtain an internal trace slice as follows:

$$\mathcal{T}_i^{\bullet} = k(+(4,2), \bullet) \to k(6,2) \to p(sd(6,2)) \to p(4)$$

By using the extended matching to match the first state of the internal trace slice $\mathcal{T}_i^\bullet$ within the first state of the original internal trace $\mathcal{T}_i$ (see line 6 of Algorithm 3), we obtain:

$$\sigma^\bullet_{\mathcal{T}_{i1}} \leftarrow match^\bullet(k(+(X,2),\bullet), k(+(4,2),Y)) \quad \text{that is, } \sigma^\bullet_{\mathcal{T}_{i1}} = \{X/4, \bullet/Y\}$$

Finally by joining $\sigma^\bullet_{\mathcal{T}_{i1}}$ with $\sigma^\bullet_1$ (see line 7 of Algorithm 3) we have:

$$\sigma^\bullet_1 = \{X/4, Z/4, Y/\bullet_2\}$$

Then the substitution $\sigma^\bullet_1$ is returned by the *mc-slicing* function and there is no iteration more.

Conclusions

Web applications are subject to an ever-increasing complexity with regard to their design and development, which demands highly sophisticated debugging and repairing tools to assist developers in the construction process.

The antecedents of this work are on the one hand, a framework for the formal specification of the operational semantics of Web applications formalized in [ABR09], and on the other hand, a backward trace-slicing technique for unconditional rewriting logic theories described in [ABER11a]. This MSc thesis contributes to this line of work by first formalizing a novel, backward conditional slicing technique for rewrite theories, and second adapting the general slicer to the framework of Web-TLR.

Debugging of Web Applications

In order to debug Web applications, we endowed the rewriting logic framework Web-TLR with a backward trace-slicing facility. This integration greatly reduces the size of the counterexample traces making their analysis feasible even in the case of complex, real-size Web applications.

The proposed extension of Web-TLR, see [ABE+11], provides a friendly graphic interface where the developer 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 distribution package (e.g., a Webmail application, and a forum Web application). The results obtained are very encouraging and show impressive reduction rates in all cases, ranging from 90% to 95%. 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. The tool is available online at http://www.dsic.upv.es/grupos/elp/soft.html.

Conditional Slicing Technique

In order to consider conditional rewriting computations, we have completely redesigned the backward trace slicing technique proposed in [ABER11a] by getting rid of the complex labeling procedure in favour of a simpler, novel and more convenient extended pattern matching algorithm that is sensible to the slicing information.

Future Work

There are several interesting directions for resuming the research presented in this master's thesis. Let us briefly comment on some of them.

- We plan to extend Web-TLR to deal with more sophisticated Web systems based on Web service architectures (e.g., those conforming to the REST framework [FT02]).
- We intend to integrate in Web-TLR an efficient implementation of the conditional slicing algorithm.

Bibliography

[ABE+11] M. Alpuente, D. Ballis, J. Espert, F. Frechina, and D. Romero. Debugging of Web Applications with WEB-TLR. In *Proc. of 7th Int'l Workshop on Automated Specification and Verification of Web Systems WWV 2011*. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2011.

[ABER10] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Model-checking Web Applications with Web-TLR. In *Proc. of 8th Int'l Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*, volume 6252 of *LNCS*, pages 341–346. Springer, 2010.

[ABER11a] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Backward Trace Slicing for Rewriting Logic Theories. In *Proc. of 23rd Int'l Conference on Automated Deduction CADE 2011*, LNCS/LNAI. Springer, 2011. To appear.

[ABER11b] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Dynamic backward slicing of rewriting logic computations. *CoRR*, abs/1105.2665, 2011.

[ABR09] M. Alpuente, D. Ballis, and D. Romero. Specification and Verification of Web Applications in Rewriting Logic. In *Formal Methods, Second World Congress FM 2009*, volume 5850 of *LNCS*, pages 790–805. Springer, 2009.

[BM08] K. Bae and J. Meseguer. A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. In *Proc. of 9th Int'l Workshop on Rule-Based Programming RULE'08*, ENTCS. Elsevier, 2008.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer-Verlag, 2007.

[CDE+09] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talco. Maude Manual (Version 2.4). Technical report, SRI Int'l Computer Science Laboratory, 2009. Available at: http://maude.cs.uiuc.edu/maude2-manual/.

[DP01] N. Dershowitz and D. Plaisted. Rewriting. *Handbook of Automated Reasoning*, 1:535–610, 2001.

[EMM06] S. Escobar, C. Meadows, and J. Meseguer. A Rewriting-Based Inference System for the NRL Protocol Analyzer and its Meta-Logical Properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.

[EMS03] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc. 10 th Intl. SPIN Workshop*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.

[Esp11] J. Espert. Verificación de aplicaciones web dinámicas con WEB-TLR, 2011. Project fin de carrera, ETSInf-UPV.

[FT02] T. R. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[GFKF03] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In *Proc. of 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, pages 238–252. Springer, 2003.

[HH06] M. Han and C. Hofmeister. Modeling and verification of adaptive navigation in web applications. In *Proc. of 6th Int'l Conference on Web Engineering ICWE '06*, pages 329–336. ACM, 2006.

[MEM06] J. Meseguer, S. Escobar, and C. Meadows. A rewriting-based inference system for the nrl protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367:162–202, November 2006.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes08] J. Meseguer. The Temporal Logic of Rewriting: A Gentle Introduction. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of his 65th Birthday*, volume 5065, pages 354–382, Berlin, Heidelberg, 2008. Springer-Verlag.

[MH94] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[MM08] R. Message and A. Mycroft. Controlling control flow in web applications. *In Proc. of 4th Int'l Workshop on Automated Specification and Verification of Web Sites WWV'08, ENTCS*, 200(3):119–131, 2008.

[MOM02] N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MPMO08] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008.

[Pro07] Open Web Application Security Project. Top ten security flaws, 2007. Available at: http://www.owasp.org/index.php/OWASP_Top_Ten_Project.

[Rom11] D. Romero. *Rewriting-based Verification and Debugging of Web Systems*. PhD thesis, Universidad Politécnica de Valencia, 2011.

[TeR03] TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.

[WNF10] F. Weitl, S. Nakajima, and B. Freitag. From Counterexamples to Incremental Interactive Tracing of Errors (Schrittweise Fehleranalyse auf der Grundlage von Model-Checking). *it - Information Technology*, 52(5):295–297, 2010.

Appendix A

Formal Specification of the Operational Semantics of the Web Scripting Language

---

The equational theory (Σ_s, E_s), which we presented in Section 1.2.1, is formally defined by means of the following Maude specification that consists of two functional modules. The former module (called `EXPRESSION`) specifies the syntax as well as the semantics of the language *expressions*. The latter module (called `SCRIPT`) formalizes the syntax and semantics of the language *statements*. The evaluation function

$$[\![_]\!] : \mathsf{ScriptState} \rightarrow \mathsf{ScriptState}$$

is encoded via the operator `evlSt : ScriptState -> ScriptState` which is contained in the functional module `SCRIPT`.

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

  --- Signature of the Expression operators

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

  --- Semantics of the Expression operators

  vars ex ex1 ex2 : Expression .
```

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

  --- Exp: value
  eq evlEx ( v, m, s, q, db ) = v .

  --- Exp: boolean conditions = and !=
  ceq evlEx ( ex1 = ex2, m, s, q, db ) = TRUE
                    if ((evlEx(ex1, m, s, q, db)) ==  (evlEx(ex2, m, s, q, db)))  .
  ceq evlEx ( ex1 = ex2, m, s, q, db ) = FALSE
                    if ((evlEx(ex1, m, s, q, db)) =/= (evlEx(ex2, m, s, q, db)))  .
  ceq evlEx ( ex1 != ex2, m, s, q, db ) = FALSE
                    if ((evlEx(ex1, m, s, q, db)) ==  (evlEx(ex2, m, s, q, db)))  .
  ceq evlEx ( ex1 != ex2, m, s, q, db ) = TRUE
                    if ((evlEx(ex1, m, s, q, db)) =/= (evlEx(ex2, m, s, q, db)))  .

  --- Exp: evaluation of private memory identifiers
  eq  evlEx ( qid, ([qid, v] : ms), s, q, db ) = v .
  ceq evlEx ( qid,  m, s, q, db ) = null   if qid in m =/= true .

  --- Exp: arithmetic operators
   eq evlEx ( ex1 '+ ex2, m, s, q, db )
                              = evlEx(ex1, m, s, q, db) + evlEx(ex2, m, s, q, db) .
  eq evlEx ( ex1 '* ex2, m, s, q, db )
                              = evlEx(ex1, m, s, q, db) * evlEx(ex2, m, s, q, db) .


  --- Exp: attribute selector
  eq evlEx ( ex1 '. ex2, m, s, q, db )
                             = evlEx(ex1, m, s, q, db) v+ evlEx(ex2, m, s, q, db) .


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

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

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

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

endfm)

(fmod SCRIPT is inc EXPRESSION .
```

```
--- Signature of the Statement operators

sorts Script ScriptState .

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

--- Semantics of the Statement operators

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

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

--- Statement: assignment (:=)
eq evlSt ( [ (qid := ex ); ps, [qid, v] : ms, s, q, db ] ) =
         evlSt ( [ ps, [qid, evlEx(ex, [qid, v] : ms, s, q, db ) ] : ms, s, q, db]) .
ceq evlSt ( [ (qid := ex ); ps,  ms, s, q, db ] ) =
                evlSt ( [ ps, [qid, evlEx(ex, ms, s, q, db ) ] : ms, s, q, db ] )
                                                           if qid in ms =/= true .
--- Statement: if then else fi
ceq evlSt ( [ ( if t then p1 else p2 fi ) ; ps, m, s, q, db ] ) =
       evlSt ([ p1 ; ps, m, s, q, db ]) if (TRUE == evlEx(t, m, s, q, db)) == true  .
ceq evlSt ( [ ( if t then p1 else p2 fi ) ; ps, m, s, q, db ] ) =
      evlSt ([ p2 ; ps, m, s, q, db ]) if (TRUE == evlEx(t, m, s, q, db)) =/= true  .

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

--- Statement: setSession
eq evlSt ([ ( setSession(ex1, ex2) ); ps, m, s, q, db ]) =
```

```
                 evlSt ([ ps, m, setSessionValue (s, evlEx(ex1, m, s, q, db),
                                               evlEx(ex2, m, s, q, db) ) , q, db ])  .

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

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

endfm)
```

Appendix B

Formal Specification of the Evaluation Protocol Function

The protocol evaluation function eval, which we presented in Section 1.2.3, is formally specified by means of the following Maude functional module.

```
(fmod EVAL is inc WEB_MODEL  .

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

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

  --- Evaluation of the enabled continuations
  eq  holdContinuation(np, (cond => np) : conts, ss)
                                    = holdCont (np, (cond => np) : conts, ss) .

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

```
  eq whichQid ( np, np1 ) = np1 [owise] .


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


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

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

endfm)
```

Appendix C

A trace example of rules with conditions

The execution trace delivered by Maude for the Example 3.2.1 is as follows:

```
==========================================
rewrite in M : m(2, 1, 3) .
*********** trial #1
crl m(X, Y, Z) => f(Z + Y, h(X)) if X > 0 = true /\ Z < 10 = true [label r2] .
X --> 2
Y --> 1
Z --> 3
*********** solving condition fragment
X > 0 = true
*********** equation
(built-in equation for symbol _>_)
2 > 0
--->
true
*********** success for condition fragment
X > 0 = true
X --> 2
Y --> 1
Z --> 3
*********** solving condition fragment
Z < 10 = true
*********** equation
(built-in equation for symbol _<_)
3 < 10
--->
true
*********** success for condition fragment
Z < 10 = true
X --> 2
Y --> 1
Z --> 3
*********** success #1
*********** rule
crl m(X, Y, Z) => f(Z + Y, h(X)) if X > 0 = true /\ Z < 10 = true [label r2] .
X --> 2
Y --> 1
Z --> 3
m(2, 1, 3)
--->
f(3 + 1, h(2))
```

```
*********** equation
(built-in equation for symbol _+_)
1 + 3
--->
4
*********** equation
eq h(X) = X + 2 .
X --> 2
h(2)
--->
2 + 2
*********** equation
(built-in equation for symbol _+_)
2 + 2
--->
4
*********** trial #2
crl f(X, Y) => g(X, Z) if X < 5 = true /\ Z := h(Y) /\ Y > 1 = true [label r1]
    .
X --> 4
Y --> 4
Z --> (unbound)
*********** solving condition fragment
X < 5 = true
*********** equation
(built-in equation for symbol _<_)
4 < 5
--->
true
*********** success for condition fragment
X < 5 = true
X --> 4
Y --> 4
Z --> (unbound)
*********** solving condition fragment
Z := h(Y)
*********** equation
eq h(X) = X + 2 .
X --> 4
h(4)
--->
4 + 2
*********** equation
(built-in equation for symbol _+_)
2 + 4
--->
6
*********** success for condition fragment
Z := h(Y)
X --> 4
Y --> 4
Z --> 6
*********** solving condition fragment
Y > 1 = true
*********** equation
(built-in equation for symbol _>_)
```

```
4 > 1
--->
true
*********** success for condition fragment
Y > 1 = true
X --> 4
Y --> 4
Z --> 6
*********** success #2
*********** rule
crl f(X, Y) => g(X, Z) if X < 5 = true /\ Z := h(Y) /\ Y > 1 = true [label r1]
    .
X --> 4
Y --> 4
Z --> 6
f(4, 4)
--->
g(4, 6)
rewrites: 11 in 1ms cpu (5ms real) (10156 rewrites/second)
result Nat: g(4, 6)
```