UNIVERSIDAD POLITECNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Rewriting-based Verification and Debugging of Web Systems

Ph.D. Thesis

Presented by:
Daniel Omar Romero

Supervisors:
María Alpuente Frasnedo
Demis Ballis

Valencia, April 2011

UNIVERSIDAD POLITECNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Rewriting-based Verification and Debugging of Web Systems

Ph.D. Thesis

A dissertation submitted by Daniel Omar Romero in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science with European mention at the Universidad Politécnica de Valencia.

Valencia, April 2011

Siempre ten presente que la piel se arruga,
el pelo se vuelve blanco,
los días se convierten en años.....
Pero lo importante no cambia,
tu fuerza y tu convicción no tienen edad.
Tu espíritu es el plumero de cualquier telaraña.

Detrás de cada día de llegada, hay una partida.
Detrás de cada logro, hay otro desafío.
Mientras estés vivo, siéntete vivo.

Si extrañas lo que hacías, vuelve hacerlo.
No vivas de fotos amarillas.
Sigue aunque todos esperen que abandones.
No dejes que se oxide el hierro que hay en tí.
Haz que en vez de lástima, te tengan respeto.

Cuando por los años no puedas correr, trota.
Cuando no puedas trotar, camina.
Cuando no puedas caminar, usa el bastón.
Pero nunca te detengas.

Madre Teresa de Calcuta

Always keep in mind that your skin will wrinkle
and that your hair will go white
and that your days will become years…
But the most important thing never changes,
your strength of will and your convictions don't have an age limit.
Your spirit is like a feather duster to wipe away the cobwebs.

After every arrival there is a leaving.
After every accomplishment there is another challenge.
While you are alive, feel and know that you are alive.

When you are feeling sorry for yourself
about what you used to be able to do, do something new.
Don't live surrounded by the yellowed photos of yesterday.
Continue forward, even though you feel abandoned by others.
Don't let rust take away the steel that is in you.
Behave in a way that others respect you, not pity you.

When, due to your years, you cannot run, trot.
When you can no longer trot, walk.
When you can no longer walk, grab a cane and keep on going.
Never stop yourself.

Mother Teresa of Calcutta

.... acknowledgements or dedication?

I am a little confused as to whether I should write an *acknowledgement* or a *dedication* with regard to my thesis, since I feel that there are a lot of people who will not understand a single line of what is written here, but who have helped me by giving me the strength needed to reach this point.

However, the people with whom I have worked on this thesis deserve a dedication, because I am sure that without their help, patience, corrections, emails, long hours, etc..., this work could not have been completed. This thesis produced more friends than papers!!!

I do not believe that it is possible to reach an agreement about this and be fair to everyone.

The truth is that the process of creating this thesis is like a long road which began in Rio Cuarto as an abstract idea between two friends, and then took shape to reach the corridors of the DSIC. This pilgrimage was made with old and new friends, each of whom contributed a little of their time towards organizing all my time. They do not need to see their names written here. They know that are part of the road I followed, and how much I owe to them. If you have read this far, then you are sure to be one of them. If you stopped in the middle, it is because our paths did not cross, but no matter, you never know.

Daniel — Valencia, April 2011

Abstract

The increasing complexity of Web system has led to the development of sophisticated formal methodologies for verifying and correcting Web data and Web programs. In general, establishing whether a Web system behaves correctly with respect to the original intention of the programmer or checking its internal consistency are non-trivial tasks as witnessed by many studies in the literature.

In this dissertation, we face two challenging problems related to the verification of Web systems.

Firstly, we extend a previous Web verification framework based on partial rewriting by providing a semi-automatic technique for repairing Web systems. We propose a basic repairing methodology that is endowed with several strategies for optimizing the number of repair actions that must be executed in order to fix a given Web site. Also, we develop an improvement of the Web verification framework that is based on abstract interpretation and greatly enhances both efficiency and scalability of the original technique.

Secondly, we formalize a framework for the specification and model-checking of dynamic Web applications that is based on Rewriting Logic. Our framework allows one to simulate the user navigation and the evaluation of Web scripts within a Web application, and also check important related properties such us reachability and consistency. When a property is refuted, a counter-example with the erroneous trace is delivered. Such information can be analyzed in order to debug the Web application under examination by means of a novel backward trace slicing technique that we formulated for this purpose. This technique consists in tracing back, along an execution trace, all the relevant symbols of the term (or state) that we are interested to observe.

Resumen

El incremento de la complejidad de los sistemas Web ha dado lugar al desarrollo de sofisticadas metodologías formales para verificar y corregir la información y los programas en la Web. En general, establecer si un sistema Web se comporta correctamente con respecto a la intención original del programador o demostrar su consistencia no son tareas triviales. La prueba de esto es la cantidad de estudios al respecto que existen en la literatura.

En esta tesis abordamos dos problemas interesantes relacionados a la verificación de sistemas Web.

En primer lugar, extendemos un marco previo de verificación Web, basado en reescritura parcial, agregando una técnica para la reparación de sistemas Web. Proponemos una metodología de reparación básica que está dotada con distintas estrategias para obtimizar el número de acciones que deben ser ejecutadas para reparar un sitio Web dado. Además, desarrollamos una mejora del marco de verificación Web que está basada en interpretación abstracta y mejora en gran medida la eficiencia y la escalabilidad de la técnica original.

En segundo lugar, formalizamos un marco para la especificación y compilación de modelos (verificación) de aplicaciones Web dinámicas basado en la lógica de reescritura. Nuestro marco nos permite simular la navegación de un usuario y evaluar los scripts dentro de la aplicación Web, así como verificar importantes propiedades tales como la alcanzabilidad y la consistencia. Cuando una propiedad es refutada, el verificador entrega un contraejemplo que consiste en la traza errónea. Esta información puede ser analizada con el fin de depurar la aplicación Web que se está examinando. Para este propósito, formulamos una nueva técnica de slicing que analiza la traza en sentido opuesto a la ejecución. Esta técnica consiste en rastrear hacia atrás, sobre dicha traza de ejecución, los símbolos relevantes del término (o estado) que estamos interesados.

Resum

L'increment de la complexitat dels sistemes Web ha donat lloc al desenvolupament de sofisticades metodologies formals per a verificar i corregir la informació i els programes a la Web. En general, comprovar si un sistema Web es comporta correctament en respecte a la intenció original del programador així com verificar la seua consistència no son tasques trivials. La prova d'açò es la quantitat d'estudis que existixen a la lliteratura al respecte d'estes comprovacions.

A la tesis, abordem dos problemes interessants relacionats a la verificació de sistemes Web.

En primer lloc, ampliem un marc previ de verificació Web, basat en re-escritura parcial, afegint una tècnica per a la reparació de sistemes Web. Proposem una metodologia de reparació bàsica dotada amb distintes estratègies per optimitzar el nombre d'accions que deuen ser executades per reparar una web donada. A més, desenvolupem un millora del marc de verificació Web que està basada en interpretació abstracta i millora en gran mida la eficiència i la escalabilitat de la tècnica original.

En segon lloc, formalizem un marc lògic de re-escritura per l'especificació i 'model-checking' de aplicacions Web dinàmiques. El nostre marc, mos permet simular la navegació d'un usuari i avaluar els scripts Web dins de la aplicació Web, així com verificar importants propietats com per eixample d'abastabilitat i consistència. Quan una propietat es refutada, es mostra un contra-eixample amb la traça errònea. Aquesta informació pot ser analitzada amb la finalitat de depurar l'aplicació Web que esta sent analitzada. Per aquest propòsit, formulem una nova tècnica d'slicing cap enrere sobre les traces donades. Aquesta tècnica consisteix en buscar cap enrere, sobre la traça d'execució, els símbols rellevants del terme (o estat) en el qual estem interessats a observar.

Contents

Introduction **1**
What is a Web system? 1
Contributions of the Thesis 2
Part I – Static Web Verification 3
Part II – Dynamic Web Verification 6

Preliminaries **9**
Term Rewriting Systems 10
Rewrite Theories 11

I Static Web Verification **13**

1 Rewriting-based Web Verification **15**
1.1 Web Site Description 15
1.2 Web Specification Language 17
1.3 Homeomorphic Embedding and Partial Rewriting 20
1.4 Error Diagnoses 22

2 Semi-Automatic Repairing of Web Sites **27**
2.1 Repairing Faulty Web Sites 27
2.1.1 Fixing Correctness Errors 28
2.1.2 Fixing Completeness Errors 32
2.2 Automatic Error Repair 37
2.2.1 Incompatibility of Conditions 40
2.3 Related Work 40

3 Optimization Strategies for Repairing Web Sites **43**
3.1 Fixing Web Sites by Using Correction Strategies 43
3.1.1 Correctness Error Dependencies 44
3.1.2 Correction Strategies 48
3.2 Fixing Web Sites by Using Completeness Strategies . . . 55
3.2.1 Completeness Error Dependencies 56

3.2.2 Completion Strategies 59

4 The Web Verification Service WebVerdi-M 67
4.1 Web Site Verification Using Maude 67
4.1.1 Homeomorphic Embedding Implementation . . . 69
4.2 Prototype Implementation 70
4.2.1 The WebVerdiService 71
4.2.2 The XML API 72
4.2.3 The Verdi-M Engine 72
4.2.4 The WebVerdiClient 73
4.2.5 The Database 73
4.3 The API 73
4.3.1 Data Representation 74
4.3.2 Methods Exported 79
4.4 Experimental Evaluation 81

5 An Abstract Generic Framework for Web Verification 83
5.1 Introduction 83
5.1.1 Web Site Description 84
5.2 Abstract Web Site Verification 85
5.2.1 Abstract Web Specification 88
5.2.2 Abstract Web Site 89
5.2.3 Abstract Verification Soundness 92
5.3 Implementation 101
5.4 Related Work 102

II Dynamic Web Verification 103

6 Specification and Verification of Web Applications in RL 105
6.1 A Navigation Model for Web Applications 108
6.1.1 Graphical Navigation Model 108
6.2 Formalizing the Navigation Model as a Rewrite Theory . 110
6.2.1 The Web Scripting Language 110
6.2.2 The Web Application Structure 111
6.2.3 The Communication Protocol 113
6.3 Modeling Multiple Web Interactions and Browser Features 117
6.3.1 The Extended Equational Theory $(\mathbf{\Sigma_{ext}}, \mathbf{E_{ext}})$ . . 117

6.3.2 The Extended Rewrite Rule Set $\mathbf{R_{ext}}$ 119
6.4 Model Checking Web Applications Using LTLR 122
6.4.1 The Linear Temporal Logic of Rewriting 123
6.4.2 LTLR properties for Web Applications 124
6.5 Related Work 128

7 Backward Trace Slicing for Rewriting Logic Theories 129
7.1 Introduction 129
7.2 Rewriting Modulo Equational Theories 131
7.3 Backward Trace Slicing for Elementary Rewrite Theories 132
7.3.1 Labeling Procedure for Rewriting Theories 133
7.3.2 The Backward Trace Slicing Algorithm 136
7.4 Backward Trace Slicing for Extended Rewrite Theories . 143
7.4.1 Dealing with Collapsing and Nonleft-linear Rules 143
7.4.2 Built-in Operators 146
7.4.3 Associative-Commutative Axioms 146
7.4.4 Extended Soundness 149
7.5 Experimental Evaluation 153
7.6 Related Work 155
7.7 Conclusions 157

8 Model-checking Web Applications with Web-TLR 159
8.1 The Web-TLR System 160
8.2 A Case Study in Web Verification 161
8.3 Web-TLR Graphical Web interface 163
8.4 Debugging of Web Applications 168
8.4.1 Debugging: A Case Study 168

Conclusions 173
Static Web Verification 173
Dynamic Web Verification 175
Future Work 176

Bibliography 179

A Operational Semantics of the Web Scripting Language 191

B Specification of the Evaluation Protocol Function 195

C Example: The War of Souls **197**
The Backward Trace Slicing Technique in Action 198
Trace Slice Analysis 202

List of Figures

1.1 A Web page and its corresponding encoding as a ground term . . . 15
1.2 An example of a Web site W for a research group 16

3.1 Taxonomy of error dependencies. 46
3.2 The $\mathbb{MNO}$ strategy 54

4.1 Components of WebVerdi-M 71
4.2 WebVerdiClient Snapshot 74
4.3 XML encoding of a Web site 75
4.4 XML representation for a Web specification 77
4.5 XML error representation 78
4.6 Completeness error representation 78
4.7 XML action representation 79

5.1 Web site and Web specification for an on-line auction system. 85
5.2 Abstract embedding 95

6.1 The navigation model of a Webmail application. 109

7.1 A term slice and one possible concretization. 138

8.1 The navigation model of an Electronic Forum 161
8.2 Maude specification of the navigation model 164
8.3 Maude specification of the curPage state predicate 164
8.4 Electronic Forum Application in Web-TLR 166
8.5 Slideshow of the Web-TLR execution 167
8.6 One state of the counter-example trace of Section 8.2. . 169
8.7 Trace slice $\mathcal{T}^{\bullet}$. 171

C.1 Maude specification of the *WoS* game. 198
C.2 Trace given by the `frew` command of Maude. 199

C.3 Labeling and sequence of relevant position sets. In order to facilitate the understanding, the terms involved in each step are underlined. 201
C.4 The execution trace $\mathcal{T}$ and its corresponding trace slice $\mathcal{T}^\bullet$. 202

Introduction

In the last decade, Web environments have evolved into very sophisticated systems that play a crucial role in the modern information society. Nowadays, Web systems pervade our digital life: as a matter of fact, almost all Web scenarios rely on some kind of Web systems in order to perform tasks such as financial and e-commerce transactions, fast and secure information interchange, social interactions, *etc.*

This evolution comes together with a rise in the complexity of Web script languages and communication protocols that makes it necessary to assist developers and Web administrators in the analysis, verification and repairing of such complex systems. As a consequence, the specification and debugging of Web systems require the development of specific techniques that address the specific challenges of the World Wide Web.

In order to achieve this goal, it is essential to develop of formal methods, models, and automated tools, which should be able not only to detect errors in the syntactic structure, but also in the semantics of Web systems. Web system failures must be precisely diagnosed in order to apply (semi)automatic repair strategies that allow one to obtain a correct and complete Web system with respect to a reference specification. Systematic, formal approaches can bring many benefits to Web system development and maintenance, giving support for automated Web verification and repairing.

Our work is certainly neither the first nor the only proposal for verifying Web systems, but it can be distinguished from many others for advocating the use of term rewriting technology. An updated and completed description of the state of the art can be found in [ACD09].

What is a Web system?

There is no general agreement on what a Web system is. Actually, its definition may vary depending on the specific scientific community. In this dissertation, we focus on server-side Web systems, that is, systems

hosted in a server that are accessed over a network (such as the Internet) by means of a Web browser, which interprets and displays the system's outcome.

Roughly speaking, it is possible to classify the Web systems into two groups: Web systems contain *static content*, also called Web sites in this thesis, and *dynamic Web systems* (often called Web application). The former class represents those Web systems that consist in a collection of interconnected static Web pages, where the content presented to the user does not change dynamically, e.g., blog and forum repositories, home pages, news Webs, and digital libraries. Such Web systems are called *static* because their content only changes if it is explicitly modified by the user or the system administrator. The latter class represents those Web systems where the content is the result of processing the system's state along with the (possible) input provided by the user, e.g., user level privileges, visited pages history, and parameters furnished to the server. These Web systems are called *dynamic* because their content is generated on-the-fly each time a Web page is requested by the user. Examples of Web systems in this group are: Webmailers, online auction Web sites, Web database managers, and Web-based conference management software systems. As we will see, the difference in behavior between the static and dynamic Web systems has a great impact on the kind of analyses and techniques we need to use for their verification.

Contributions of the Thesis

Verification of Web systems is a nontrivial task because of their complex and distributed nature. Although in recent years much effort has been invested into this problem, there is still a generalized lack of techniques and tools for verifying and debugging Web systems. We do believe that systematic, formal approaches can bring many benefits to Web system development, thus giving support to automated verification and repairing.

This dissertation develops a series of novel, rewriting-based techniques for the verification of static as well as dynamic Web systems with a particular focus on the formal verification of semantic properties, as opposed to many current tools that mainly support syntax check (e.g., [Osk05;

Sol10; Gmb]).

The thesis is organized in two parts:

- Part I – Static Web verification. This part extends a rewriting-based, Web verification framework first presented in [ABF06] by adding a semi-automatic technique for repairing Web sites together with several strategies that optimize the number of repair actions that must be executed in order to fix a given Web site. Also, we present an improvement of the verification methodology of [ABF06], based on abstract interpretation [CC77; CC79], that greatly enhances both efficiency and scalability of the original technique.

- Part II – Dynamic Web verification. In this part, a rewriting logic framework for the specification and model-checking of dynamic Web applications is proposed. The framework allows one to simulate the navigation of a user when using the Web application, check important related properties such as the open windows and mutual exclusion problems [MM08], and evaluate the possibly included Web scripts.

In the following, we briefly summarize the main contributions in the two parts of this thesis.

Part I – Static Web Verification

[ABF06] presents a rewriting-based approach for Web site specification and verification. This methodology allows one to specify integrity conditions for a given Web site and then diagnose the errors by computing the requirements not fulfilled in the considered Web site, that is, by finding out incorrect/forbidden patterns and missing/incomplete Web pages. This approach is particularly suitable for checking large static Web sites, e.g., digital libraries, which contain a number of deeply interconnected XML documents; or collaborative Web sites, that is, sites where several users may freely change/remove data. In these scenarios, keeping the static Web contents correct and complete is obviously not trivial and requires advanced verification capabilities that are naturally supported by this methodology, which was implemented in the prototype Verdi

[Bal05; BV05] written in Haskell. The main foundations of this verification methodology that serve as a basic for the original contributions of this thesis are described in Chapter 1.

Semi-Automatic Repairing for Web Sites (Chapter 2)

This chapter describes a repairing methodology for fixing Web sites. It is based on our work [ABFR06], and complements the verification methodology presented in [ABF06] for detecting Web site errors. Our aim is to complement the verification methodology with a tool-independent technique that gives support for semi-automatically repairing the errors found during that verification phase.

First, we formalize the different kinds of errors that can be found in a Web site with respect to a Web site specification. Then, we classify the *repair actions* that can be performed to repair each kind of error. Since different repair actions can be executed in order to repair a given error, our method is tuned to deliver a set of correct and complete repair actions to choose between. Our repair methodology is formulated in two phases. First, all the necessary actions to make the Web site *correct* are performed. Once correctness of the Web site has been achieved, the user is given the option to execute all the necessary actions to make it *complete* (while preserving correctness) with respect to a given formal specification. Also, this methodology allows us to manage the interference issues that might arise from the interaction among multiple repair actions.

Optimization Strategies for Repairing Web Sites (Chapter 3)

The repair framework of [ABFR06] does not investigate the relations/dependencies among the errors. Such analysis can be a potential source of optimization and can increase the level of automation of the repair system. As a matter of fact, errors in a given Web site are often deeply interrelated. This fact suggests us that correcting a given bug may lead to an automatic fix of a "related" bug without executing any other repair action. In this chapter, we extend the repair methodology of Chapter 2 in order to optimize the repair process by considering how the number of repair actions can be minimized and reduce the amount of information that needs to be changed/removed in order to fix the Web site [BR07b; ABF+07c].

The Web Verification Service WebVerdi-M (Chapter 4)

In order to make the verification and repair techniques available to any interested user easier to use by hiding the technical details to the user, a new prototype WebVerdi-M that extends the Verdi system [BV05] is described in Chapter 4. The prototype is based on the implementation infrastructure presented in [ABF+07a].

WebVerdi-M relies on a strictly more powerful Web verification engine written in the Rewriting Logic language Maude [CDE+07] that automatically derives the error symptoms of a given Web site. Thanks to the AC pattern matching supported by Maude and its metalevel features, we have significantly improved both the performance and the usability of the original Verdi system.

A Java Web client that is publicly available interacts with a Web verification service by using SOAP messages [GHM+07] and other Web-related standards.

Finally, we report on some benchmarks gathered from an experimental evaluation of our system by using several correctness and completeness rules of different complexity for a number of randomly generated XML documents.

An Abstract Generic Framework for Web Site Verification (Chapter 5)

For correctness checking, WebVerdi-M shows impressive performance thanks to the Associativity-Commutativity (AC) pattern matching and the metalevel features supported by Maude (for instance, verifying correctness over a 10Mb XML document with 302000 nodes takes less than 13 seconds). Both resource allocation and elapsed time scale linearly. Unfortunately, for the verification of completeness, a (finite) fixpoint computation is typically needed that leads to unsatisfactory performance. Indeed, the verification tool is only able to efficiently process XML documents smaller than 1Mb.

In this chapter, we develop an abstract approach to Web site verification that makes use of an approximation technique based on abstract interpretation [CC77; CC79] that greatly improves the previous performance. We also ascertain the conditions that ensure the correctness of the approximation, so that the resulting abstract rewrite engine safely

supports accurate Web site verification. Since the abstract framework is parametric with respect to the considered abstraction, we precisely characterize the conditions that allow us to ensure the correctness of the abstraction, which is implemented as a source-to-source transformation of concrete Web sites and Web specifications into abstract ones. Thanks to this source-to-source approximation scheme, all facilities supported by our previous verification system are straightforwardly adapted and reused with very little effort. The abstract methodology presented in this chapter was presented in [ABF+07b; ABF+08].

Part II – Dynamic Web Verification

A Web application runs in a server and is shown in a browser, which acts as the interface between the user and the Web application. Browsers were initially intended to access to static content, and their main features (e.g., back, forward, and reload buttons) have not been properly adapted to the Web application evolution. This mismatch contributes to many errors present in the Web [GFKF03; MM08].

The goal of this part of the thesis is to explore the application of the formal methods to formal modeling and automatic verification of complex, real-size dynamic Web applications.

Specification and Verification of Web Applications in Rewriting Logic (Chapter 6)

This chapter describes a Rewriting Logic framework for the formal specification of the operational semantics of Web applications first proposed in [ABR09]. In particular, we 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). Our 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 ac-

tions (e.g., forward/backward navigation, refresh, and new window/tab openings) via rewrite rules completes the proposed specification.

Our formalization is particularly suitable for verification purposes, since it allows one to carry out in-depth analyses of several subtle aspects of Web interactions. To this respect, we show how our 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.

Backward Trace Slicing for Rewriting Logic Theories (Chapter 7)

Trace slicing is a widely used technique for execution trace analysis that is effectively used in program debugging, analysis and comprehension. In this chapter, we present a backward trace slicing technique [ABER11] that can be used for the analysis of Rewriting Logic theories. In Rewriting Logic, system computations are modeled by means of rewrite rules that describe transitions between states. System states are represented as elements of an algebraic data type that is defined by means of an equational theory E that may include sorts, functions and algebraic laws (such as commutativity and associativity). Our trace slicing technique allows us to systematically trace back rewrite sequences *modulo* E (i.e., system computations) by means of a backward algorithm that dynamically simplifies the traces by detecting control and data dependencies, 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. In particular, we use this slicing methodology to simplify the counter-examples that are delivered by Web-TLR executions.

Model-checking Web Applications with Web-TLR (Chapter 8)

Web-TLR [ABER10] is a software tool designed for model-checking Web applications that is based on rewriting logic. Web applications are expressed as rewrite theories that can be formally verified by using the Maude built-in LTLR model-checker. Web-TLR is equipped with a user-friendly, graphical Web interface that shields the user from

unnecessary information. Whenever a property is refuted, an interactive slideshow is generated, which allows one to reproduce visually, step by step, the erroneous navigation trace that underlies the failing model checking computation. This provides deep insight into the system behavior that helps to debug Web applications. This chapter describes the main features of our tool and presents several examples that demonstrate the feasibility of our approach.

Preliminaries

We recall in this section some basic notions that will be used in the rest of the thesis.

By $\mathcal{V}$ 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 sym-

bols 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., $t^{\Box} = t\sigma$).

Term Rewriting Systems

Term rewriting systems provide an adequate computational model for functional languages. In the sequel, we follow the standard framework of term rewriting (see [BN98; Klo92]). A *term rewriting system* (TRS for short) is a pair (Σ, R), where Σ is a signature and R is a finite set of reduction (or rewrite) rules of the form $\lambda \to \rho$, $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, $\lambda \notin \mathcal{V}$ and $Var(\rho) \subseteq Var(\lambda)$. We will often write just R instead of (Σ, R). Sometimes, we denote the signature of a TRS (Σ, R) by Σ_R.

A rewrite step is the application of a rewrite rule to an expression. A term s *rewrites* to a term t via $r \in R$, $s \xrightarrow{r}_R t$ (or $s \xrightarrow{r,\sigma}_R t$), if there exists

a position q in s such that λ *matches* $s_{|q}$ via a substitution σ (in symbols, $s_{|q} \equiv \lambda\sigma$), and t is obtained from s by replacing the subterm $s_{|q} \equiv \lambda\sigma$ with the term $\rho\sigma$, in symbols $t \equiv s[\rho\sigma]_q$. 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 $\lambda \to \rho$ (or equation $\lambda = \rho$) is *collapsing* if $\rho \in \mathcal{V}$; it is *left-linear* if no variable occurs in λ more than once. We say that a TRS R is *terminating*, if there exists no infinite rewrite sequence $t_1 \to_R t_2 \to_R \ldots$. A TRS 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 TRSs, each input term t can be univocally reduced to a unique *irreducible form*.

Let $s = t$ be an equation, we say that the equation $s = t$ *holds* in a canonical TRS R, if there exists an irreducible form $z \in \tau(\Sigma, \mathcal{V})$ w.r.t. R such that $s \to^!_R z$ and $t \to^!_R z$. Let $s \neq t$ be an inequation. We say that $s \neq t$ *holds* in a canonical TRS R, when $s = t$ does not hold in $\mathcal{R}$. A *condition* is a finite set of equations and inequations. We say that a condition C *holds* in a canonical TRS R, if for each equation (inequation) $e \in C$, e holds in R. The *empty* condition $\emptyset$ trivially holds in any canonical TRS R.

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 *rewrite theory* [MOM02]. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$, where:

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

Part I

Static Web Verification

Chapter 1

Rewriting-based Web Verification

In this chapter, we briefly recall the formal verification methodology proposed in [ABF06], which is able to detect erroneous as well as missing information in a Web site. By executing a Web specification on a given Web site, this methodology is able to recognize and exactly locate the source of a possible discrepancy between the Web site and the properties stated in the Web specification.

1.1 Web Site Description

In [ABF06], a *Web page* is either an XML [BPM+08] or an XHTML [Pem00] document, which is assumed to be well-formed. This is justified by the plenty of programs and online services that are able to validate XHTML/XML syntax and perform link checking (e.g., [Sol10; Osk05]). As Web pages are provided with a tree-like structure, they can be straightforwardly encoded into ordinary terms of a suitable term algebra $T_{\mathcal{T}ext \cup \mathcal{T}ag}$, where $\mathcal{T}ext \cup \mathcal{T}ag$ is a signature containing the text and the tags on which the Web pages are built, as shown in Figure 1.1.

```
<people>                    people(
 <person>                    person(
   <id>per0</id>               id(per0),
   <name>Conte</name>          name(Conte)
 </person>                    )
</people>                   )
```

Figure 1.1: A Web page and its corresponding encoding as a ground term

p_1) members(member(name(mario), surname(rossi), status(professor)),
member(name(franca), surname(bianchi), status(technician)),
member(name(giulio), surname(verdi), status(student)),
member(name(ugo), surname(blu), status(professor)))

p_2) hpage(fullname(mariorossi), phone(3333), status(professor),
hobbies(hobby(reading), hobby(gardening))),

p_3) hpage(fullname(francabianchi), status(technician), phone(5555),
links(link(url(www.google.com), urlname(google)),
link(url(www.sexycalculus.com), urlname(FormalMethods))),

p_4) hpage(fullname(annagialli), status(professor), phone(4444),
teaching(course(algebra))),

p_5) pubs(pub(name(ugo), surname(blu), title(blah1), blink(year(2003))),
pub(name(anna), surname(gialli), title(blah2), year(2002))),

p_6) projects(project(pname(A1), grant1(1000), grant2(200),
total(1100), coordinator(fullname(mariorossi))),
project(pname(B1), grant1(2000), grant2(1000),
projectleader(surname(gialli), name(anna)),
total(3000)))

$$W = \{p_1,\ p_2,\ p_3,\ p_4,\ p_5,\ p_6\}$$

Figure 1.2: An example of a Web site W for a research group

Note that XML/XHTML tag attributes can be considered as common tagged elements, and hence translated in the same way. Therefore, *Web sites* can be represented as finite sets of (ground) terms.

In the following, we will also consider terms of the non-ground term algebra $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$, which may contain variables. An element $\mathtt{s} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ is called *Web page template*. In our methodology, Web page templates are used for specifying properties on Web sites as described in the following section.

Example 1.1.1

In Figure 1.2, we represent a Web site W of a research group, which contains information about group members affiliation, scientific publications, research projects, teaching and personal data.

1.2 Web Specification Language

A Web specification is a triple (I_N, I_M, R), where R, I_N, and I_M are finite set of rules. The set R contains the definition of some auxiliary functions which the user would like to provide, such as string processing, arithmetic, boolean operators, *etc.* It is formalized as a canonical term rewriting system which is handled by standard rewriting [Klo92]. This implies that each input term t can be univocally reduced to an irreducible form.

The set I_N describes constraints for detecting erroneous Web pages (*correctNess rules*). As the amount of faulty information is typically a small portion of the whole content of a Web site, the correctness rules model erroneous patterns rather than correct/safe patterns, which facilitates both the specification and the verification of correctness properties. Formally, a correctness rule has the following form:

$$\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{C}, \text{ with } Var(\mathtt{C}) \subseteq Var(\mathtt{l})$$

where `l` is a term, `error` is a reserved constant, and `C` is a (possibly empty) finite sequence containing membership tests (e.g., $\mathtt{X} \in \mathtt{rexp}$) w.r.t. a given regular language[1], and/or equations over terms. For the sake of expressiveness, we also allow to write inequalities of the form $\mathtt{s} \neq \mathtt{t}$ in `C`, which hold whenever the corresponding equation $\mathtt{s} = \mathtt{t}$ does not hold. When `C` is empty, we simply write $\mathtt{l} \rightharpoonup \mathtt{error}$.

The meaning of a correctness rule $\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{C}$, where $\mathtt{C} \equiv (\mathtt{X_1}$ in $\mathtt{rexp_1}, \ldots, \mathtt{X_n}$ in $\mathtt{rexp_n}, \ \mathtt{s_1} = \mathtt{t_1} \ldots \mathtt{s_m} = \mathtt{t_m})$, is the following. We say that `C` *holds* for substitution σ, if (i) each structured text $\mathtt{X_i}\sigma$, $\mathtt{i} = 1, \ldots, \mathtt{n}$, is contained in the language of the corresponding regular expression $\mathtt{rexp_i}$; (ii) each instantiated equation (resp. inequality) $(\mathtt{s_i} = \mathtt{t_i})\sigma$ (resp. $(\mathtt{s_i} \neq \mathtt{t_i})\sigma$) , $\mathtt{i} = 1, \ldots, \mathtt{m}$, holds in R.

The Web page `p` is considered incorrect if an instance $\mathtt{l}\sigma$ of `l` is *recognized* within `p`, and `C` holds for σ.

The third set of rules I_M specifies some properties for detecting incomplete/missing Web pages (*coMpleteness rules*). A completeness rule

[1] Regular languages are represented by means of the usual Unix-like regular expressions syntax.

is defined as

$$\mathtt{l} \rightharpoonup \mathtt{r}\ \langle \mathtt{q} \rangle$$

where $\mathtt{l}$ and $\mathtt{r}$ are terms and $\mathtt{q} \in \{\mathtt{E}, \mathtt{A}\}$. Completeness rules of a Web specification formalize the requirement that some information must be included in all or some pages of the Web site. The attributes $\langle \mathtt{A} \rangle$ and $\langle \mathtt{E} \rangle$ distinguish the "universal" rules from the "existential" rules. Right-hand sides of completeness rules can contain functions, which are defined in R. Intuitively, the interpretation of a universal rule $\mathtt{l} \rightharpoonup \mathtt{r}\ \langle \mathtt{A} \rangle$ (respectively, an existential rule $\mathtt{l} \rightharpoonup \mathtt{r}\ \langle \mathtt{E} \rangle$) w.r.t. a Web site W is as follows: if (an instance of) $\mathtt{l}$ is recognized in W, also (an instance of) the irreducible form of $\mathtt{r}$ must be recognized in *all* (respectively, *some*) of the Web pages which embed (an instance of) $\mathtt{r}$.

Sometimes, we may be interested in checking a given completeness property only on a subset of the whole Web site. For this purpose, some symbols in the right-hand sides of the rules are marked by means of the constant symbol $\sharp$. Marking information of a given rule r is used to select the subset of the Web site in which we want to check the condition formalized by r. More specifically, rule r is executed on all and only the Web pages embedding the marking information. A detailed example follows.

Example 1.2.1

Consider the Web specification that consists of the following completeness and correctness rules along with a term rewriting system defining the string concatenation function $+\!+$, the arithmetic operators $+$ and $*$ on natural numbers and the relational operator $\leq$. That is:

$r_1)$ $\mathtt{member(name(X), surname(Y))} \rightharpoonup \sharp\mathtt{hpage(fullname(X +\!+ Y), status)}\ \langle \mathtt{E} \rangle$
$r_2)$ $\mathtt{hpage(status(professor))} \rightharpoonup \sharp\mathtt{hpage(\sharp status(\sharp professor), teaching)}\ \langle \mathtt{A} \rangle$
$r_3)$ $\mathtt{pubs(pub(name(X), surname(Y)))} \rightharpoonup \sharp\mathtt{members(member(name(X), surname(Y)))}\ \langle \mathtt{E} \rangle$
$r_4)$ $\mathtt{courselink(url(X), urlname(Y))} \rightharpoonup \sharp\mathtt{cpage(title(Y))}\ \langle \mathtt{E} \rangle$
$r_5)$ $\mathtt{hpage(X)} \rightharpoonup \mathtt{error} \mid \mathtt{X\ in[:TextTag:]^{*}sex[:TextTag:]^{*}}$
$r_6)$ $\mathtt{blink(X)} \rightharpoonup \mathtt{error}$
$r_7)$ $\mathtt{project(grant1(X), grant2(Y), total(Z))} \rightharpoonup \mathtt{error} \mid \mathtt{X} + \mathtt{Y} \neq \mathtt{Z}$
$r_8)$ $\mathtt{project(grant1(X), grant2(Y))} \rightharpoonup \mathtt{error} \mid \mathtt{X} \neq \mathtt{Y} * 2$
$r_9)$ $\mathtt{total(Z)} \rightharpoonup \mathtt{error} \mid \mathtt{Z} \geq 500000 = \mathtt{true}$

This Web specification models some required properties for the Web site of Figure 1.2. First rule formalizes the following property: if there is a Web page containing a member list, then for each member, a home page should exist which contains (at least) the full name and the status of this member. The full name is computed by concatenating the name and the surname strings by means of the `++` function. The marking information establishes that the property must be checked only on home pages (i.e., pages containing the tag "hpage"). Second rule states that, whenever a home page of a professor is recognized, that page must also include some teaching information. The rule is universal, since it must hold for each professor home page. Such home pages are selected by exploiting the marks which identify professor home pages. Third rule specifies that, whenever there exists a Web page containing information about scientific publications, each author of a publication should be a member of the research group. In this case, we must check the property only in the Web page containing the group member list. The fourth rule formalizes that, for each link to a course, a page describing that course must exist. The fifth rule forbids sexual contents from being published in the home pages of the group members. This is enforced by requiring that the word `sex` does not occur in any home page by using the regular expression [: `TextTag` :]*`sex`[: `TextTag` :]*, which identifies the regular language of all the strings built over $(\mathcal{T}ext \cup \mathcal{T}ag)$ containing word `sex`. The sixth rule is provided with the aim of improving accessibility for people with disabilities. It simply states that blinking text is forbidden in the whole Web site. The last three rules respectively state that, for each research project, the total project budget must be equal to the sum of the grants, the first grant should be the double of the second one, and the total budget is less than 500000 euros.

The error diagnoses are carried out by running Web specifications on Web sites. This is mechanized by means of *partial rewriting*, a novel rewriting technique which is obtained by replacing the traditional pattern-matching of term rewriting with a new mechanism based on *page* (tree) *embedding* (cf. [ABF06]).

1.3 Homeomorphic Embedding and Partial Rewriting

Partial rewriting extracts "some pieces of information" from a page, pieces them together, and then rewrites the glued term. The assembling is done by means of the *homeomorphic embedding* relation, which recognizes the structure and the labeling of a given term (Web page template) inside a particular page of the Web site.

The notion of homeomorphic embedding, $\trianglelefteq$, is an adaptation of Kruskal's *embedding* (or "syntactically simpler") relation [Bez03] where the usual *diving* rule[2] [Leu02] is ignored.

Definition 1.3.1 (homeomorphic embedding) *The* homeomorphic embedding *relation*

$$\trianglelefteq \subseteq \tau(\mathcal{T}ext \cup \mathcal{T}ag) \times \tau(\mathcal{T}ext \cup \mathcal{T}ag)$$

on Web pages is the least relation satisfying the rule:

$$\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_m) \trianglelefteq \mathtt{g}(\mathtt{s}_1, \ldots, \mathtt{s}_n)$$
$$\textit{iff } f \equiv g \textit{ and } t_i \trianglelefteq s_{\pi(i)}, \textit{ for } i = 1, \ldots, m,$$
$$\textit{and some injective function } \pi : \{1, \ldots, m\} \to \{1, \ldots, n\}.$$

Given two Web pages $\mathtt{s}_1$ and $\mathtt{s}_2$, if $\mathtt{s}_1 \trianglelefteq \mathtt{s}_2$ we say that $\mathtt{s}_1$ *simulates* (or *is embedded* or *recognized* into) $\mathtt{s}_2$. We also say that $\mathtt{s}_2$ *embeds* $\mathtt{s}_1$. Note that, in Definition 1.3.1, for the case when $\mathtt{m}$ is 0 we have $\mathtt{c} \trianglelefteq \mathtt{c}$ for each constant symbol $\mathtt{c}$. Note also that $\mathtt{s}_1 \not\trianglelefteq \mathtt{s}_2$ if either $\mathtt{s}_1$ or $\mathtt{s}_2$ contain variables.

Regarding to the positions involved in the homeomorphic embedding relation, we give the following auxiliary definition that will be needed later.

Definition 1.3.2 ($Emb_{\mathtt{s}}(\mathtt{t})$) *Let* $\mathtt{s}, \mathtt{t} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *such that* $\mathtt{s} \trianglelefteq \mathtt{t}$. *We define the set* $Emb_{\mathtt{s}}(\mathtt{t})$ *as the set of all the positions in* $\mathtt{t}$ *which embed some subterm of* $\mathtt{s}$.

[2] The diving rule allows one to "strike out" a part of the term at the right-hand side of the relation $\trianglelefteq$. Formally, $s \trianglelefteq f(t_1, \ldots t_n)$, if $s \trianglelefteq t_i$, for some i.

For instance, consider the terms $f(k, g(c))$, and $f(b, g(c), k)$. Then, $f(k, g(c)) \trianglelefteq f(b, g(c), k)$, and

$$Emb_{f(k,g(c))}(f(b, g(c), k)) = \{\Lambda, 2, 2.1, 3\}$$

Now we are ready to introduce the *partial rewrite* relation between Web page templates. Without loss of generality, conditions and/or quantifiers from the Web specification rules are disregarded.

Definition 1.3.3 (partial rewriting) *Let* $\mathtt{s}, \mathtt{t} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. *Then,* $\mathtt{s}$ partially rewrites *to* $\mathtt{t}$ *via rule* $\mathtt{l} \rightharpoonup \mathtt{r}$ *and substitution* σ *iff there exists a position* $u \in O_{\mathcal{T}ag}(\mathtt{s})$ *such that:*

(i) $\mathtt{l}\sigma \trianglelefteq \mathtt{s}_{|u}$, *and*
(ii) $\mathtt{t}$ = *Reduce*($\mathtt{r}\sigma$,*R), where function Reduce*(x, R) *computes, by standard term rewriting, the irreducible form of* x *in* R.

Roughly speaking, given a Web specification rule $\mathtt{l} \rightharpoonup \mathtt{r}$, partial rewriting allows us to extract from, a given Web page $\mathtt{s}$, a subpart of $\mathtt{s}$ which is embedded by a ground instance of $\mathtt{l}$, and to replace $\mathtt{s}$ by a reduced, ground instance of $\mathtt{r}$. Note that the context of the selected reducible expression $\mathtt{s}_{|u}$ is disregarded after each rewrite step. By notation $\mathtt{s} \rightharpoonup_I \mathtt{t}$, we denote that $\mathtt{s}$ is partially rewritten to $\mathtt{t}$ using some rule belonging to the set I. A partial rewrite sequence is of the form $\mathtt{s}_0 \rightharpoonup \mathtt{s}_1 \rightharpoonup \ldots \rightharpoonup \mathtt{s}_n$. Moreover, we denote the transitive closure (resp., the transitive and reflexive closure) of $\rightharpoonup$ by $\rightharpoonup^+$ (resp., $\rightharpoonup *$).

Example 1.3.4

Let $p = \mathtt{h}(\mathtt{f}(\mathtt{a}), \mathtt{f}(\mathtt{b}))$ be a Web page. Let $I = \{r_1, r_2\}$ be a Web specification where $r_1 = \mathtt{f}(\mathtt{x}) \rightharpoonup \mathtt{g}(\mathtt{h}(\mathtt{x}), \mathtt{b})$ and $r_2 = \mathtt{h}(\mathtt{a}) \rightharpoonup \mathtt{m}(\mathtt{a}, \mathtt{b})$. Then, we get the following partial rewrite sequences:

$$s_1 = \mathtt{h}(\mathtt{f}(\mathtt{a}), \mathtt{f}(\mathtt{b})) \rightharpoonup \mathtt{g}(\mathtt{h}(\mathtt{a}), \mathtt{b}) \rightharpoonup \mathtt{m}(\mathtt{a}, \mathtt{b})$$
$$s_2 = \mathtt{h}(\mathtt{f}(\mathtt{a}), \mathtt{f}(\mathtt{b})) \rightharpoonup \mathtt{g}(\mathtt{h}(\mathtt{b}), \mathtt{b})$$

1.4 Error Diagnoses

In order to diagnose correctness as well as completeness errors, we follow the method presented in [ABF06].

We classify the kind of errors which can be found in a Web site in terms of the different outputs delivered by our verification technique, when is fed with a Web site specification. In Chapter 2, we will exploit this information to develop our repairing/correction methodology. Let us start by characterizing correctness errors.

Applying the correctness rules to Web pages. If a Web page is partially rewritten to the constant `error`, then a correctness error for that page is signaled, since a piece of erroneous/forbidden information has been recognized.

Definition 1.4.1 (correctness error) *Let W be a Web site and (I_M, I_N, R) be a Web specification. Then, the quadruple $(\mathtt{p}, w, \mathtt{l}, \sigma)$ is a* correctness error evidence *iff $\mathtt{p} \in W$, $w \in O_{\mathcal{T}ag}(\mathtt{p})$, and $\mathtt{l}\sigma$ is an instance of the left-hand side $\mathtt{l}$ of a correctness rule belonging to I_N such that $\mathtt{l}\sigma \trianglelefteq \mathtt{p}_{|w}$.*

Given a correctness error evidence $(\mathtt{p}, w, \mathtt{l}, \sigma)$, $\mathtt{l}\sigma$ represents the erroneous information which is embedded in a subterm of the Web page $\mathtt{p}$, namely $\mathtt{p}_{|w}$.

We denote the set of all correctness error evidences of a Web site W w.r.t. a set of correctness rules I_N by $E_N(W)$. When no confusion can arise, we just write E_N.

Example 1.4.2

Consider the correctness rule r_5 in the Web specification of Example 1.2.1 and the Web site in Figure 1.2. Then, our verification methodology outputs $(p_3, \Lambda, \mathtt{l}, \sigma)$, where

$$
\begin{aligned}
l &\equiv \mathtt{hpage(X)} \\
\sigma &\equiv \{\mathtt{X/links(link(url(www.sexycalculus.com),} \\
&\qquad\qquad \mathtt{urlname(FormalMethods)))}\ \}
\end{aligned}
$$

Applying the completeness rules to Web pages. First, a set of requirements (i.e., pieces of information which must be contained in the site) is generated by partially rewriting the Web pages via the completeness rules; then, is used a homeomorphic embedding algorithm to check whether the requirements are fulfilled, that is, the required information is not missing. When a requirement is not satisfied, it witnesses the lack of some data and the system outputs the incomplete Web page $\mathtt{p}$ together with the information which should be added to $\mathtt{p}$ in order to fulfill the requirement.

As for completeness errors, we can distinguish three classes of errors: (*i*) *Missing Web pages*, (*ii*) *Universal completeness errors*, (*iii*) *Existential completeness errors*. These completeness errors can be detected by partially rewriting Web pages to some expression $\mathtt{r}$ by means of the rules of I_M, and then checking whether $\mathtt{r}$ does not occur in a suitable subset of the Web site.

Definition 1.4.3 (Missing Web page) *Let W be a Web site, and let (I_M, I_N, R) be a Web specification. Then the pair $(\mathtt{r}, W)$ is a* missing *Web page error evidence if there exists $\mathtt{p} \in W$ s.t. $\mathtt{p} \rightharpoonup^{+}_{I_M} \mathtt{r}$ and $\mathtt{r} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ does not belong to W.*

When a missing Web page error is detected, the evidence $(\mathtt{r}, W)$ signals that the expression $\mathtt{r}$ does not appear in the whole Web site W.

In order to formalize existential as well as universal completeness errors, the following auxiliary definition is introduced.

Definition 1.4.4 *Let $\mathtt{P}$ be a set of terms in $\tau(\mathcal{T}ext \cup \mathcal{T}ag)$, and let $\mathtt{r} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ be a term. We say that $\mathtt{P}$ is* universally *(resp.* existentially*)* complete *w.r.t. $\mathtt{r}$ iff for each $\mathtt{p} \in \mathtt{P}$ (resp. for some $\mathtt{p} \in \mathtt{P}$), there exists $w \in O_{\mathcal{T}ag}(\mathtt{p})$ s.t. $\mathtt{r} \trianglelefteq \mathtt{p}_{|w}$.*

Definition 1.4.5 (Universal completeness error) *Let W be a Web site, and let (I_M, I_N, R) be a Web specification. Then the triple $(\mathtt{r}, \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}, \mathtt{A})$ is a* universal completeness error evidence*, if there exists $\mathtt{p} \in W$ s.t. $\mathtt{p} \rightharpoonup^{+}_{I_M} \mathtt{r}$ and $\{\mathtt{p}_1, \ldots, \mathtt{p}_n\}$ is not universally complete w.r.t. $\mathtt{r}$, $\mathtt{p}_i \in W$, $i = 1, \ldots, n$.*

Definition 1.4.6 (Existential completeness error) *Let W be a Web site, and let (I_M, I_N, R) be a Web specification. Then the triple* $(\mathtt{r}, \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}, \mathtt{E})$ *is an* existential completeness error evidence*, if there exists* $\mathtt{p} \in W$ *s.t.* $\mathtt{p} \rightharpoonup^{+}_{I_M} \mathtt{r}$ *and* $\{\mathtt{p}_1, \ldots, \mathtt{p}_n\}$ *is not existentially complete w.r.t.* $\mathtt{r}$*,* $\mathtt{p}_i \in W$*,* $i = 1, \ldots, n$*.*

Note that Definition 1.4.5 (resp. Definition 1.4.6) formalizes the fact that the Web site W fails to fulfil the requirement that a piece of information must occur in *all* (resp. *some*) Web pages of a given subset of W. We denote by $E_M(W)$ the set containing all the completeness error evidences w.r.t. I_M for a Web site W (missing Web pages as well as universal/existential completeness errors evidences). When no confusion can arise, we just write E_M.

Example 1.4.7

Consider the Web specification of Example 1.2.1 and the Web site in Figure 1.2. The following completeness error evidences are delivered:

$$e_{M_1} = (\ \mathtt{hp}(\mathtt{fullname}(\mathtt{giulioverdi}), \mathtt{status}),\ W)$$
$$e_{M_2} = (\ \mathtt{hp}(\mathtt{status}(\mathtt{professor}), \mathtt{teaching}),\ p_2, p_4,\ A)$$
$$e_{M_3} = (\ \mathtt{members}(\mathtt{member}(\mathtt{name}(\mathtt{anna}), \mathtt{surname}(\mathtt{gialli}))),\ p_1,\ E)$$

$$E_M = \{e_{M_1},\ e_{M_2},\ e_{M_3}\}$$

e_{M_1} is a missing Web page error showing the missing page's `Giulio Verdi`; e_{M_2} represents a universal completeness error which formalizes the fact that there are some professors' home pages without any teaching information; and e_{M_3} identifies an existential completeness error telling us that the members' Web page should contain the group member `Anna Gialli`.

Definition 1.4.8 (Web site correctness) *Given a Web specification (I_M, I_N, R), a Web site W is* correct *w.r.t. (I_M, I_N, R) iff the set E_N of correctness error evidences w.r.t. I_N is empty.*

Definition 1.4.9 (Web site completeness) *Given a Web specification (I_M, I_N, R), a Web site W is* complete *w.r.t. (I_M, I_N, R) iff the set E_M of completeness error evidences w.r.t. I_M is empty.*

Corollary 1.4.10 (Correct and complete) *Given a Web site W and a Web specification (I_M, I_N, R). We say that W is correct and complete w.r.t. (I_M, I_N, R) iff $E_N = \emptyset$ and $E_M = \emptyset$.*

The verification methodology of [ABF06] generates the sets of correctness and completeness error evidences E_N and E_M mentioned above for a given Web site w.r.t. the input Web specification. Starting from these sets, in the following chapter we formulate a method for fixing the errors and delivering a Web site which is *correct* and *complete* w.r.t. the intended Web specification.

Chapter 2

Semi-Automatic Repairing of Web Sites

This chapter describes the semi-automatic methodology for repairing faulty Web sites which complements the verification methodology given in Chapter 1.

Given a faulty Web site W and the sets of errors $\mathtt{E}_N$ and $\mathtt{E}_M$, our goal is to modify the given Web site by adding, changing, and removing information in order to produce a Web site that is correct and complete with respect to the considered Web specification. For this purpose, in correspondence with the error categories distinguished in the previous chapter, we introduce a catalog of *repair actions* that can be applied to the faulty Web site. Therefore, in our framework, fixing a Web site consists in selecting a set of suitable repair actions that are automatically generated, and executing them in order to remove inconsistencies and wrong data from the Web site.

2.1 Repairing Faulty Web Sites

In order to repair a faulty Web site, we introduce four repair actions that will be used as primitives into the repair strategies. The primitive repair actions that we consider are the following:

- **change**$(\mathtt{p}, w, \mathtt{t})$, which replaces the subterm $\mathtt{p}|_w$ in $\mathtt{p}$ with the term $\mathtt{t}$;
- **insert**$(\mathtt{p}, w, \mathtt{t})$, which modifies the term $\mathtt{p}$ by adding the term $\mathtt{t}$ into $\mathtt{p}_{|w}$;
- **add**$(\mathtt{p}, W)$, which adds the Web page $\mathtt{p}$ to the Web site W;

- **delete**$(\mathtt{p}, t)$, which deletes all the occurrences of the term `t` in the Web page `p`.

Each repair action returns the modified/added Web page after its execution. Note that it is possible that a particular error could be repaired by means of different actions. For instance, a correctness error can be fixed by deleting the incorrect/forbidden information, or by changing the data which rise that error. Similarly, a completeness error can be fixed by either (i) inserting the missing information, or (ii) deleting all the data in the Web site that caused that error. Moreover, modifying or inserting arbitrary information may cause the appearance of new correctness errors. In order to avoid this, we have to ensure that the data considered for insertion are *safe* w.r.t. the Web specification, i.e., they cannot fire any correctness rule. For this purpose, we introduce the following definition.

Definition 2.1.1 *Let* (I_M, I_N, R) *be a Web specification, and let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page. Then,* `p` *is* safe *w.r.t.* I_N*, iff for each* $w \in O_{\mathcal{T}ag}(\mathtt{p})$ *and* $(\mathtt{l} \rightharpoonup \mathtt{r} \mid \mathtt{C}) \in I_N$*, either (i) there is no* σ *s.t.* $\mathtt{l}\sigma \trianglelefteq \mathtt{p}_{|w}$*; or (ii)* $\mathtt{l}\sigma \trianglelefteq \mathtt{p}_{|w}$*, but* $\mathtt{C}\sigma$ *does not hold.*

In the following, we develop a repairing methodology which gets rid of both, correctness and completeness errors. We proceed in two main phases. First, we deal with correctness errors. Some repair actions are automatically inferred and run in order to remove the wrong information from the Web site. After this phase, we will end up with a correct Web site which still can be incomplete. At this point, other repair actions are synthesized and executed in order to provide Web site completeness.

2.1.1 Fixing Correctness Errors

Throughout this section, we will consider a given Web site W, a Web specification (I_M, I_N, R) and the set $E_N \neq \emptyset$ of the correctness error evidences w.r.t. I_N for W. Our goal is to modify W in order to generate a new Web site which is correct w.r.t. (I_M, I_N, R). We proceed as follows: whenever a correctness error is found, we choose a possible repair action (among the different actions described below) and we execute it in order to remove the erroneous information, provided that it does not introduce any new bug.

Given $\mathtt{e} = (\mathtt{p}, w, \mathtt{l}, \sigma) \in E_N$, $\mathtt{e}$ can be repaired in two distinct ways: we can decide either (i) to remove the wrong content $\mathtt{l}\sigma$ from the Web page $\mathtt{p}$ (specifically, from $\mathtt{p}_{|w}$), or (ii) to change $\mathtt{l}\sigma$ into a piece of correct information. Hence, it is possible to choose between the following repair strategies.

"Correctness through Deletion" Strategy

In this case, we simply remove all the occurrences of the subterm $\mathtt{p}_{|w}$ of the Web page $\mathtt{p}$ containing the wrong information $\mathtt{l}\sigma$ by applying the repair action $\mathbf{delete}(\mathtt{p}, \mathtt{p}_{|w})$.[1]

Example 2.1.2

Consider the Web site in Figure 1.2 and the Web specification in Example 1.2.1. The term $\mathtt{l}\sigma \equiv \mathtt{p}_{|1.4} \equiv \mathtt{blink}(\mathtt{year}(2003))$ embedded in the Web page `(5)` of W (which is also called $\mathtt{p}$ in this example) generates a correctness error evidence $(\mathtt{p}, 1.4, \mathtt{l}, \sigma)$ w.r.t. the rule $\mathtt{blink}(\mathtt{x}) \rightharpoonup \mathtt{error}$ and hence a **delete** action will remove from $\mathtt{p}$ the subterm $\mathtt{blink}(\mathtt{year}(2003))$.

"Correctness through Change" Strategy

Given a correctness error $\mathtt{e} = (\mathtt{p}, w, \mathtt{l}, \sigma) \in E_N$, we replace the subterm $\mathtt{p}_{|w}$ of the Web page $\mathtt{p}$ with a new term t introduced by the user. The new term t must fulfill some conditions which are automatically provided and checked in order to guarantee the correctness of the inserted information. In the following we show how to compute such constraints.

Roughly speaking, whenever we fix some wrong data by executing a repair action $\mathbf{change}(\mathtt{p}, w, t)$, it is not enough to ensure that the term t to be introduced has no errors, we also need to consider t within the context that surrounds it in $\mathtt{p}$. If we don't pay attention to such a global condition, some subtle correctness errors might arise as witnessed by the following example.

[1] Note that, instead of removing the whole subterm $\mathtt{p}_{|w}$, it would be also possible to provide a more precise though also time-expensive implementation of the **delete** action which only gets rid of the part $\mathtt{l}\sigma$ of $\mathtt{p}_{|w}$ which is responsible for the correctness error.

Example 2.1.3

Consider the Web page $\mathtt{p} \equiv \mathtt{f}(\mathtt{g}(\mathtt{a}), \mathtt{b}, \mathtt{h}(\mathtt{c}))$, and the following correctness rule set

$$I_N \equiv \{(\mathtt{r1})\, \mathtt{f}(\mathtt{g}(\mathtt{b})) \rightharpoonup \mathtt{error}, \quad (\mathtt{r2})\, \mathtt{g}(\mathtt{a}) \rightharpoonup \mathtt{error}\}.$$

The Web page $\mathtt{p}$ contains a correctness error according to rule ($\mathtt{r2}$). The Web page $\mathtt{f}(\mathtt{g}(\mathtt{b}), \mathtt{b}, \mathtt{h}(\mathtt{c}))$ is obtained from $\mathtt{p}$ by executing, for instance, the repair action

$$\mathbf{change}(\mathtt{f}(\mathtt{g}(\mathtt{a}), \mathtt{b}, \mathtt{h}(\mathtt{c})), 1, \mathtt{g}(\mathtt{b})).$$

Although the term $\mathtt{g}(\mathtt{b})$ is safe w.r.t. I_N (i.e., it does not introduce any new correctness error), the replacement of $\mathtt{g}(\mathtt{a})$ with $\mathtt{g}(\mathtt{b})$ in $\mathtt{p}$ produces a new correctness error which is recognizable by rule ($\mathtt{r1}$).

In order to avoid such kinds of undesirable repairs, we define the following global correctness property, which simply prevents a new term t from firing any correctness rule when inserted in the Web page to be fixed.

Definition 2.1.4 *Let* (I_M, I_N, R) *be a Web specification,* $\mathtt{p}' \equiv \mathbf{change}(\mathtt{p}, w, t)$ *be a repair action producing the Web page* $\mathtt{p}'$*. Then,* $\mathbf{change}(\mathtt{p}, w, t)$ obeys the global correctness property *if, for each correctness error evidence* $\mathtt{e} = (\mathtt{p}', w', \mathtt{l}, \sigma)$ *w.r.t.* I_N *such that* $w' \leq w$*,*

$$\{w\}.O_{\mathcal{T}ag}(t) \cap \{w'\}.Emb_{\mathtt{l}\sigma}(\mathtt{p}'_{|w'}) = \emptyset$$

The idea behind Definition 2.1.4 is that any error $\mathtt{e}$ in the new page $\mathtt{p}' \equiv \mathbf{change}(\mathtt{p}, w, t)$, obtained by inserting term t within $\mathtt{p}$, is not a consequence of this change but already present in a different sub-term of $\mathtt{p}$. For this purpose, we require that (the set of positions of) the wrong information $\mathtt{l}\sigma$ does not "overlap" the considered term t.

Example 2.1.5

Consider again Example 2.1.3. The repair action

$$\mathbf{change}(\mathtt{f}(\mathtt{g}(\mathtt{a}), \mathtt{b}, \mathtt{h}(\mathtt{c})), 1, \mathtt{g}(\mathtt{b}))$$

does not obey the global correctness property. Indeed, it generates a Web page $\mathtt{f}(\mathtt{g}(\mathtt{b}), \mathtt{b}, \mathtt{h}(\mathtt{c}))$ containing a correctness error.

The use of the "Correctness through Deletion" and "Correctness through Change" strategies decreases the number of correctness errors of the original Web site as stated by the following proposition.

Proposition 2.1.6 *Let (I_M, I_N, R) be a Web specification, and let W be a Web site. Let $E_N(W)$ be the set of correctness error evidences w.r.t. I_N of W, and let $(\mathtt{p}, w, \mathtt{l}, \sigma) \in E_N(W)$ be a correctness error. By executing a repair action* **delete**$(\mathtt{p}, \mathtt{p}_w)$ *(resp.* **change**$(\mathtt{p}, w, t)$*, which obeys the global correctness property), we have that*

$$|E_N(W')| < |E_N(W)|$$

where

$$W' \equiv W \setminus \{\mathtt{p}\} \cup \{\mathbf{delete}(\mathtt{p}, \mathtt{p}_w)\}$$
$$(\textit{resp. } W' \equiv W \setminus \{\mathtt{p}\} \cup \{\mathbf{change}(\mathtt{p}, w, t)\})$$

Proof. We prove the two cases separately.

Case (i). Assume that the repair action **delete**$(\mathtt{p}, \mathtt{p}_w)$ is executed. In this case, the proof is immediate, since no new information is added to the Web site, hence, no extra correctness errors can be introduced.

Case (ii). Assume that the repair action **change**$(\mathtt{p}, w, t)$, which obeys the global correctness property, is executed. The proof for this case is also immediate. It suffices to observe that Definition 2.1.4 (global correctness property) prevents new errors from being introduced by any application of a repair action **change**.

■

We say that performing a repair action **delete** (resp. **change**) is *safe*, if it does not introduce any new correctness error, i.e., $|E_N(W')| < |E_N(W)|$. In Algorithm 1 we provide the pseudocode of the correction algorithm we implemented.

Algorithm 1 An algorithm for repairing correctness errors in a Web site.

Require:
W be a Web site, I_N be a set of correctness rules.
E_N be a set of correctness error in W w.r.t. I_N

```
1: procedure Correctness-Repair (W, I_N)
2:   while a correctness error evidence (p, w, l, σ) ∈ E_N exists do
3:     option ← AskUser()
4:     if option = delete then
5:       W ← W \ {p} ∪ {delete(p, lσ)}  // Delete action
6:     else
7:       t ← AskUser()  // Change action
8:       if change(p, w, t) obeys the local and the global correctness
         properties then
9:         W ← W \ {p} ∪ {change(p, w, t)}
10:      else
11:        Error("incorrect term t")
12:      end if
13:    end if
14:  end while
15: end procedure
```

2.1.2 Fixing Completeness Errors

In this section, we address the problem of repairing an incomplete Web site W. Without loss of generality, we assume that W is an incomplete but correct Web site w.r.t. a given Web specification (I_M, I_N, R). Such an assumption will allow us to design a repair methodology which "completes" the Web site and does not introduce any incorrect information.

Let $E_M(W)$ be the set of completeness error evidences risen by I_M for the Web site W. Any completeness error evidence belonging to $E_M(W)$ can be repaired following distinct strategies and thus by applying distinct repair actions. On the one hand, we can think of adding the needed data, whenever a Web page or a piece of information in a Web page is missing. On the other hand, all the information that caused the error might be removed to get rid of the bug. In both cases, we must ensure that the execution of the chosen repair action does not introduce any

new correctness/completeness error to guarantee the termination and the soundness of our methodology. In the following, we distinguish and argue about the two possible repair strategies mentioned above.

"Completeness through Insertion" strategy

We consider two distinct kinds of repair actions, namely $\mathbf{add}(\mathtt{p}, W)$ and $\mathbf{insert}(\mathtt{p}, w, t)$, according to the kind of completeness error we have to fix. The former action adds a new Web page $\mathtt{p}$ to a Web site W and thus will be employed whenever the system has to fix a given missing Web page error. The latter allows us to add a new piece of information t to (a subterm of) an incomplete Web page $\mathtt{p}$, and therefore is suitable to repair universal as well as existential completeness errors. More specifically, the insertion repair strategy works as follows.

Missing Web page errors. Given a missing Web page error evidence $(\mathtt{r}, W)$, we fix the bug by adding a Web page $\mathtt{p}$, which embeds the missing expression $\mathtt{r}$, to the Web site W. Hence, the Web site W will be "enlarged" by effect of the following **add** action: $W = W \cup \{\mathbf{add}(\mathtt{p}, W)\}$, where $\mathtt{r} \trianglelefteq \mathtt{p}_{|w}$ for some $w \in O_{\mathcal{T}ag}(\mathtt{p})$.

Existential completeness errors. Given an existential completeness error evidence $(\mathtt{r}, \{\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n\}, \mathtt{E})$, we fix the bug by inserting a term t, that embeds the missing expression $\mathtt{r}$, into an arbitrary page $\mathtt{p}_i$, $i = 1, \ldots, n$. The position of the new piece of information t in $\mathtt{p}_i$ is typically provided by the user, who must supply a position in $\mathtt{p}_i$ where t must be attached. The **insert** action will transform the Web site W in the following way: $W = W \setminus \{\mathtt{p}_i\} \cup \{\mathbf{insert}(\mathtt{p}_i, w, t)\}$, where $\mathtt{r} \trianglelefteq \mathtt{p}_{i|w}$ for some $w \in O_{\mathcal{T}ag}(\mathtt{p})$.

Universal completeness errors. Given a universal completeness error evidence $(\mathtt{r}, \{\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n\}, \mathtt{A})$, we fix the bug by inserting a term t_i, that embeds the missing expression $\mathtt{r}$, into every Web page $\mathtt{p}_i$, $i = 1, \ldots, n$ not embedding $\mathtt{r}$. The position of the new piece of information t_i in each $\mathtt{p}_i$ is typically provided by the user, who must supply a postition w_i in $\mathtt{p}_i$ where t_i must be attached. In this case, we will execute a sequence of **insert** actions, exactly one for each incomplete Web page $\mathtt{p}_i$. Therefore, the Web site W will be transformed in the following way. For each

$\mathtt{p}_i \in \{\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n\}$ such that $\mathtt{r} \not\trianglelefteq \mathtt{p}$, $W = W \setminus \{\mathtt{p}_i\} \cup \{\mathbf{insert}(\mathtt{p}_i, w_i, t_i)\}$, where $\mathtt{r} \trianglelefteq \mathtt{p}_{|w_i}$ for some $w_i \in O_{\mathcal{T}ag}(\mathtt{p}_i)$.

Both the **add** action and the **insert** action introduce new information in the Web site which might be potentially dangerous, since it may contain erroneous as well as incomplete data. It is therefore important to constrain the kind of information a user can add. In order to preserve correctness, we compel the user to only insert safe information in the sense of Definition 2.1.1. Hence, the new data being added by the execution of some repair action cannot fire a correctness rule subsequently.

Additionally, we want to prevent the execution of the repair actions from introducing new completeness errors, that is, we just want to fix all and only the initial set of completeness error evidences of the Web site W, namely $E_M(W)$. Given a completeness error evidence $\mathtt{e}$, we use notation $\mathtt{e}(\mathtt{r})$ to make evident the unsatisfied requirement $\mathtt{r}$ signaled by $\mathtt{e}$.

Definition 2.1.7 *Let (I_M, I_N, R) be a Web specification, and let W be a Web site w.r.t. (I_M, I_N, R). Let $E_M(W)$ be the set of completeness error evidences of W w.r.t. I_M.*

- *the repair action $\mathtt{p}_1 \equiv \mathbf{insert}(\mathtt{p}, w, t)$ is* acceptable *w.r.t. (I_M, I_N, R) and W iff*
  1. *$\mathtt{p}_1$ is safe w.r.t. (I_M, I_N, R);*
  2. *$\mathtt{r} \trianglelefteq t_{|w}$, $w \in O_{\mathcal{T}ag}(t)$, for some $\mathtt{e}(\mathtt{r}) \in E_M(W)$;*
  3. *if $W' \equiv W \setminus \{\mathtt{p}\} \cup \{\mathtt{p}_1\}$, then $E_M(W') \subset E_M(W)$.*
- *the repair action $\mathtt{p}_2 \equiv \mathbf{add}(\mathtt{p}_2, W)$ is* acceptable *w.r.t. (I_M, I_N, R) and W iff*
  1. *$\mathtt{p}_2$ is safe w.r.t. (I_M, I_N, R);*
  2. *$\mathtt{r} \trianglelefteq \mathtt{p}_{2|w}$, $w \in O_{\mathcal{T}ag}(\mathtt{p}_2)$, for some $\mathtt{e}(\mathtt{r}) \in E_M(W)$;*
  3. *if $W' \equiv W \cup \{\mathtt{p}_2\}$, then $E_M(W') \subset E_M(W)$.*

Definition 2.1.7 guarantees that the information which is added by **insert** and **add** actions is correct and does not yield any new completeness error. More precisely, the number of completeness errors decreases by effect of the execution of such repair actions.

As an immediate consequence of Definition 2.1.7, we have the following.

Corollary 2.1.8 *Let (I_M, I_N, R) be a Web specification, and let W be a Web site w.r.t. (I_M, I_N, R). Let $E_M(W)$ be the set of completeness error evidences of W w.r.t. I_M. By execution a repair action $\mathtt{p}_1 \equiv \mathbf{insert}(\mathtt{p}, w, t)$ (resp. $\mathtt{p}_2 \equiv \mathbf{add}(\mathtt{p}_2, W)$) , which obeys the acceptable property, we have that*

$$|E_N(W')| < |E_N(W)|$$

where

$$W' \equiv W \setminus \{\mathtt{p}\} \cup \{\mathtt{p}_1\}, \text{ where } \mathtt{p}_1 \equiv \mathbf{insert}(\mathtt{p}, w, t)$$
$$(\text{resp. } W' \equiv W \cup \{\mathtt{p}_2\}, \text{ where } \mathtt{p}_2 \equiv \mathbf{add}(\mathtt{p}_2, W))$$

Example 2.1.9

Consider the Web specification of Example 1.2.1 and the universal completeness error evidence $(\mathtt{hp}(\mathtt{status}(\mathtt{professor}), \mathtt{teaching}), \mathtt{p}_2, \mathtt{p}_4, \mathtt{A})$ where $\mathtt{p}_1$ and $\mathtt{p}_2$ are the home pages of `Mario Rossi` and `Anna Gialli` in the Web site of Figure 1.2. To fix the error, we should add some information to Web page $\mathtt{p}_2$, while Web page $\mathtt{p}_4$ is complete w.r.t. the requirement $\mathtt{hp}(\mathtt{status}(\mathtt{professor}), \mathtt{teaching})$. Consider the pieces of information

$$t_1 \equiv \mathtt{teaching}(\mathtt{course}(\mathtt{title}(\mathtt{logic}), \mathtt{syllabus}(\mathtt{blah})))$$
$$t_2 \equiv \mathtt{teaching}(\mathtt{courselink}(\mathtt{url}(\mathtt{www.mycourse.com}), \mathtt{urlname}(\mathtt{Logic}))).$$

If we introduce term t_1, the corresponding **insert** action is acceptable. However, inserting term t_2 would produce a new completeness error (i.e., a broken link error).

"Completeness through Deletion" strategy.

When dealing with completeness errors, sometimes it is more convenient to delete incomplete data instead of completing them. In particular, this option can be very useful, whenever we want to get rid of out-of-date

information as illustrated in Example 2.1.11 below. The main idea of the deletion strategy is to remove all the information in the Web site that caused a given completeness error. The strategy is independent of the kind of completeness error we are handling, since the missing information is computed in the same way for all the three kinds of errors by partially rewriting the original Web pages of the Web site. In other words, given the missing expression $\mathtt{r}$ of a completeness error evidence $\mathtt{e}(\mathtt{r})$ (that is, a missing page error evidence $(\mathtt{r}, W)$, or an existential completeness error evidence $(\mathtt{r}, \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}, \mathtt{E})$, or a universal completeness error evidence $(\mathtt{r}, \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}, \mathtt{A})$), there exists a Web page $\mathtt{p} \in W$ such that $\mathtt{p} \rightharpoonup^+ \mathtt{r}$.

Therefore, we proceed by computing and eliminating from the Web pages the term that started the partial rewrite sequence that lead to a missing expression $\mathtt{r}$.

Definition 2.1.10 (repairByDelete) *Let W be a Web site, and let (I_M, I_N, R) be a Web specification. Let $\mathtt{e}(\mathtt{r})$ be a completeness error. Them, the Web site W will change in the following way.*

For each $t \rightharpoonup^+ \mathtt{r}$, where $t \trianglelefteq \mathtt{p}_{|w}$, $w \in O_{\mathcal{T}ag}(\mathtt{p})$, $\mathtt{p} \in W$

$$\begin{aligned} W \equiv &\{\mathtt{p} \in W \mid t \not\trianglelefteq \mathtt{p}_{|w}, \forall\, w \in O_{\mathcal{T}ag}(\mathtt{p})\} \cup \\ &\{\mathbf{delete}(\mathtt{p}, t) \mid \mathtt{p} \in W, t \trianglelefteq \mathtt{p}_{|w}, w \in O_{\mathcal{T}ag}(\mathtt{p})\} \end{aligned}$$

Example 2.1.11

Consider the Web specification of Example 1.2.1, the Web site W of Figure 1.2 and the missing Web page error evidence

$$(\mathtt{hpage}(\mathtt{fullname}(\mathtt{ugoblu}), \mathtt{status}), \mathtt{W})$$

which can be detected in W by using the completeness rules in I_M. The missing information is obtained by means of the following partial rewrite sequence:

$$\begin{aligned} &\mathtt{pub}(\mathtt{name}(\mathtt{ugo}), \mathtt{surname}(\mathtt{blu}), \mathtt{title}(\mathtt{blah1}), \\ &\mathtt{blink}(\mathtt{year}(2003))) \rightharpoonup \\ &\mathtt{member}(\mathtt{name}(\mathtt{ugo}), \mathtt{surname}(\mathtt{blu})) \rightharpoonup \\ &\mathtt{hpage}(\mathtt{fullname}(\mathtt{ugoblu}), \mathtt{status}) \end{aligned}$$

By choosing the deletion strategy, we would delete all the information regarding the group membership and the publications of `Ugo Blu` from the Web site.

As in the case of the insertion strategy, we have to take care about the effects of the execution of the repair actions. More precisely, we do not want the execution of any **delete** action to introduce new completeness errors. For this purpose, we consider the following notion of *acceptable* delete action.

Definition 2.1.12 *Let (I_M, I_N, R) be a Web specification, and let W be a Web site w.r.t. (I_M, I_N, R). Let $E_M(W)$ be the set of completeness error evidences of W w.r.t. $I_M(W)$. The repair action $\mathtt{p}_1 \equiv \mathbf{delete}(\mathtt{p}, t)$ is* acceptable *w.r.t. (I_M, I_N, R) and W iff $W' \equiv W \setminus \{\mathtt{p}\} \cup \{\mathtt{p}_1\}$ implies $E_M(W') \subset E_M(W)$.*

Algorithm 2 outlines a procedure for repairing completeness errors. It implements the correction strategies described in the previous sections. For any given completeness error, the user is asked to choose between deletion of wrong information and insertion of new data. In both cases the performed repair actions must be acceptable in order to ensure the termination and the correctness of the procedure.

2.2 Automatic Error Repair

The execution of the strategies seen so far allow us to guarantee the termination of our repair methodology as well as avoid to introduce new errors in the Web site w.r.t. a given Web specification. These strategies are semi-automatics, because it is necessary to involve the user either to define the new term to be added or to decide the deletion of the incorrect information. In the following, we will see a particular case where it is possible to repair a correctness error in an automatic way.

Let us consider a correctness error $e \equiv (\mathtt{p}, w, l, \sigma) \in E_N$ given by a conditional rule, i.e., a rule as follows

$$\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{C}, \text{ where } \mathtt{C} \neq \emptyset \text{ and } Var(\mathtt{C}) \subseteq Var(\mathtt{l})$$

Let us also consider the repair action $\mathbf{change}(\mathtt{p}, w, t)$. Note that to fix the error e it is enough to give a substitution σ' such that $t = l\sigma'$.

We call the *constraint satisfaction problem* associated with $\mathtt{e}$, in symbols $CS_{\mathtt{e}}$, to the set of conditions

Algorithm 2 An algorithm for repairing completeness errors.

Require:
W be a Web site, (I_M, I_N, R) be a Web specification.

procedure Completeness-errors-Repair (W, I_M, I_N, R)
  while a completeness error $\mathtt{e}(\mathtt{r}) \in E_M(W)$ is found **do**
    option ← AskUser()
    if option = delete **then**
      call DeletionStrategy$(e(r), W, I_M, I_N, R)$
    else
      switch $e(r)$ **of**

      • **case** $[e(r) \equiv (\mathtt{r}, W)]$ // Missing Web page error
      $\mathtt{p}$ ← AskUser()
      if add$(\mathtt{p}, W)$ is acceptable w.r.t. (I_M, I_N, R) and W **then**
        $W \leftarrow W \cup \{\mathbf{add}(\mathtt{p}, W)\}$
      else
        Error("incorrect page", $\mathtt{p}$)
      end if

      • **case** $[e(r) \equiv (\mathtt{r}, \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}, \mathtt{E})]$ // Existential completeness error
      (t, w) ← AskUser()
      select $\mathtt{p} \in \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}$
      call InsertTermToPage$(\mathtt{p}, w, t, W, I_M, I_N, R)$

      • **case** $[e(r) \equiv (\mathtt{r}, \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}, \mathtt{A})]$ // Universal completeness error
      for all $\mathtt{p}_i \in \{\mathtt{p}_1 \ldots \mathtt{p}_n\}$ s.t. $\mathtt{r} \not\trianglelefteq \mathtt{p}_i$ **do**
        (t_i, w_i) ← AskUser()
        call InsertTermToPage$(\mathtt{p}_i, w_i, t_i, W, I_M, I_N, R)$
      end for

      end switch
    end if
  end while
end procedure

procedure DeletionStrategy $(e(r), W, I_M, I_N, R)$
  for all $t \rightharpoonup^{+} \mathtt{r}$, where $t \trianglelefteq \mathtt{p}_{|w}$, $w \in O_{\mathcal{T}ag}(\mathtt{p})$, $\mathtt{p} \in W$ **do**
    for all $\mathtt{p} \in W$ **and** $t \trianglelefteq \mathtt{p}_{|w}$, for some $w \in O_{\mathcal{T}ag}(\mathtt{p})$ **do**
      if delete$(\mathtt{p}, t)$ is acceptable w.r.t. (I_M, I_N, R) and W **then**
        $W \leftarrow W \setminus \{\mathtt{p}\} \cup \{\mathbf{delete}(\mathtt{p}, t)\}$
      else
        Error("incorrect delete action")
      end if
    end for
  end for
end procedure

procedure InsertTermToPage $(\mathtt{p}, w, t, W, I_M, I_N, R)$
  if insert$(\mathtt{p}, w, t)$ is acceptable w.r.t. (I_M, I_N, R) and W **then**
    $W \leftarrow W \setminus \{\mathtt{p}\} \cup \{\mathbf{insert}(\mathtt{p}, w, t)\}$
  else
    Error("incorrect term",t, "in page",$\mathtt{p}$)
  end if
end procedure

$$CS_{\mathtt{e}} \equiv \{\neg \mathtt{C} \mid \exists \quad (\mathtt{l} \rightharpoonup \mathtt{r} \mid \mathtt{C}) \in I_N,\ \text{a position } w', \\ \text{a substitution } \sigma \text{ s.t. } \mathtt{l}\sigma \trianglelefteq \mathtt{p}_{|w.w'}\}$$

Roughly speaking, $CS_{\mathtt{e}}$ is obtained by collecting and negating all the conditions of those rules which detect correctness errors in $\mathtt{p}_{|w}$. Such collection of constraints, that can be solved manually or automatically by means of an appropriate constraint solver [Apt03], which can be used to provide suitable values for the substitution σ'. We say that $CS_{\mathtt{e}}$ is *satisfiable* iff there exists at least one assignment of values for the variables occurring in $CS_{\mathtt{e}}$ that satisfies all the constraints. We denote by $Sol(CS_{\mathtt{e}})$ the set of all the assignments that verify the constraints in $CS_{\mathtt{e}}$. The restriction of $Sol(CS_{\mathtt{e}})$ to the variables occurring in σ is denoted by $Sol(CS_{\mathtt{e}})_{|\sigma}$. Let us see an example.

Example 2.2.1

Consider the Web site W in Figure 1.2 and the Web specification of Example 1.2.1. The following subterm of Web page `(6)`

$$\mathtt{project}(\mathtt{pname}(\mathtt{A1}), \mathtt{grant1}(1000), \mathtt{grant2}(200), \mathtt{total}(1100), \\ \mathtt{coordinator}(\mathtt{fullname}(\mathtt{mariorossi})))$$

causes a correctness error `e` w.r.t. the rule

$$\mathtt{project}(\mathtt{grant1}(\mathtt{X}), \mathtt{grant2}(\mathtt{Y}), \mathtt{total}(\mathtt{Z})) \rightharpoonup \mathtt{error} \mid \mathtt{X} + \mathtt{Y} \neq \mathtt{Z}.$$

The error can be fixed by changing the values of the grants and the total amount, according to the solution of the constraint satisfaction problem $CS_{\mathtt{e}}$ that follows.

$$\left\{ \begin{array}{l} \mathtt{X} + \mathtt{Y} = \mathtt{Z}, \\ \mathtt{X} = \mathtt{Y} * 2, \\ Z < 500000 \end{array} \right\}$$

The constraints in $CS_{\mathtt{e}}$ come from the conditions of the last three rules. An admissible solution, which can be chosen by the user, might be

$$\{\mathtt{X}/1000,\ \mathtt{Y}/500,\ \mathtt{Z}/1500\} \in Sol(CS)$$

and the term t to be inserted might be

$$\texttt{project}(\texttt{pname}(\texttt{A1}), \texttt{grant1}(1000), \texttt{grant2}(500), \texttt{coordinator}(\texttt{fullname}(\texttt{mariorossi})), \texttt{total}(1500))$$

which does not contain incorrect data.

2.2.1 Incompatibility of Conditions

Sometimes CS_{e} might be not solvable, since there are two or more rules demanding incompatible conditions for correctness. For example, consider the following scenarios.

i) Given the following set of correctness rules

$$I_N = \{l \rightharpoonup \texttt{error} \mid c, l \rightharpoonup \texttt{error} \mid \neg c\}$$

we obtain the unsolved set $CS_{\mathsf{e}} = \{\neg c, c\}$.

ii) Given the following set of correctness rules

$$I_N = \{l_1 \rightharpoonup \texttt{error},\ l_2 \rightharpoonup \texttt{error} \mid c\} \text{ where } l_1 \trianglelefteq l_2$$

along with the correctness error $e = (\mathtt{p}, w, l_2, \sigma)$, we obtain the set $CS_{\mathsf{e}} = \{\neg c\}$. Let $\sigma' \in Sol(CS_{\mathsf{e}})$ be a substitution, then, the new term to change is $t = l_2\sigma'$. However, t is not a suitable term, because $l_1 \trianglelefteq t$ and, in this way, a new correctness error is introduced (see Definition 2.1.4).

In both scenarios the user is asked to fix the Web specification before proceeding.

2.3 Related Work

A lot of research work has been invested in consistency management and repairing of software applications and databases, whereas similar technologies are much less mature for Web systems. [CF07] proposes a framework for Web site verification which can be used at both compile-time and run-time, and is based on type verification of the rules that

can be applied to the considered Web document. The base language is XCentric [CF04], which is a logic programming language. Errors can be automatically fixed by performing actions that are executed whenever an error is found. In [NEF03], a repair framework for inconsistent distributed documents is presented that complements the tool xlinkit [CEFN02]. The main contribution is the semantics that maps xlinkit's first order logic language to a catalogue of repairing actions that can be used to interactively correct rule violations, although it does not predict whether a repair action can provoke new errors to appear. Also, it is not possible to detect whether two formulae expressing a requirement for the Web site are incompatible. Similarly, in [SRBS04b; SRBS04a] an extension of the tool CDET [SBRS03] is developed. This extension includes a mechanism to remove inconsistencies from sets of interrelated documents, which first generates direct acyclic graphs (DAGs) representing the relations among the documents and then appropriate repair actions are directly derived from such DAGs. In this case, temporal rules are supported and interference and compatibility of repairs are not completely neglected. Unfortunately, this compatibility is too much expensive to check for temporal rules. Both approaches rely on basic techniques borrowed from the field of active databases [BP99]. Current research in this field focuses on the derivation of active rules that automatically fire repair actions leading to a consistent state after each update [MT99].

Chapter 3

Optimization Strategies for Repairing Web Sites

In previous chapters, we have presented a rewriting-based approach to Web site verification and correction. Our methodology allows us to automatically recognize forbidden/incorrect patterns as well as incomplete/missing Web pages in a Web site with respect to a given formal specification, and then repair the detected bugs by running/executing a sequence of repair actions that are semi-automatically inferred by the system. Since different repair actions are able to repair the same error, in this chapter we present some optimization strategies that allow us to generate a reduced sequence of repair actions that significantly improves the basic technique.

First, we define two correction strategies that are aimed to increase the level of automation of our repair method. Then, since the Web site is correct with respect to a given Web specification, we also define two completion strategies that optimize the repairing of completeness errors. Specifically, the proposed strategies minimize both the amount of information to be changed and the number of repair actions to be executed in a faulty Web site in order to make it correct and complete.

3.1 Fixing Web Sites by Using Correction Strategies

Chapter 1 describes a specification language along with a verification technique for the definition and the validation of formal properties over Web sites. Among the distinguished features that our framework provides, it allows one to detect erroneous/forbidden information in a Web site yielding as the outcome a set of correctness error evidences which

basically represent pieces of faulty information (i.e., correctness errors, Definition 1.4.1).

In this section, we extend our basic repairing methodology in several ways. First, we carry out a systematic analysis on the relations among correctness errors that we exploit in order to formalize two possible correction strategies: the $\mathbb{M}$ strategy allows one to minimize the number of repair actions to be executed, while the $\mathbb{MNO}$ strategy reduces the amount of information to be changed/removed in order to fix the Web site. In both cases, it is worth noting that the number of errors we need to correct in order to repair the Web site W is much less than the total number of errors occurring in W. Consequently, employing such strategies guarantees a better performance of our repair methodology.

3.1.1 Correctness Error Dependencies

Typically, a given Web page can contain several correctness errors which may be somehow interrelated. Since the execution of a repair action might fix more related errors simultaneously, it is crucial to discover whether an error depends on other errors. In this section, we analyze the dependencies among error correctness evidences. Later on, we will exploit this information in order to develop two correction strategies with the aim of minimizing the amount of information the user needs to update or delete.

First of all, let us consider the order among error correctness evidences, which can be induced from the positions of the errors in the considered Web page. Such order is formalized by means of the following definition.

Definition 3.1.1 *Let* $e_1 \equiv (\mathtt{p}, w_1, \mathtt{l}_1, \sigma_1, C_1)$ *and* $e_2 \equiv (\mathtt{p}, w_2, \mathtt{l}_2, \sigma_2, C_2)$ *be two correctness error evidences in* $E_N(\mathtt{p})$*. Then,* $e_1 \preceq e_2$ *iff* $w_1 \leq w_2$*.*

We say that e_1 and e_2 are not comparable (w.r.t. $\preceq$) iff $e_1 \not\preceq e_2$ and $e_2 \not\preceq e_1$. By exploiting the order of Definition 3.1.1, we are able to prove the following result.

Proposition 3.1.2 *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page, and* $e_i = (\mathtt{p}, w_i, \mathtt{l}_i, \sigma_i, C_i) \in E_N(\mathtt{p})$*,* $i = 1, \ldots, n$*, such that* $e_1 \preceq e_2 \preceq \ldots \preceq e_n$*. The following results hold:*

- *If* $p' \equiv \textbf{change}(\mathtt{p}, w_1, t)$ *is a safe repair action, then* $p'_{|w_1} \equiv t$ *is repaired.*

- *If* $p' \equiv \textbf{delete}(\mathtt{p}, w_1, t)$ *is a repair action, then* $p'_{|w_1}$ *is repaired.*

Proof. (Sketch) The proof of this result relies on the fact that the errors $e_2, \ldots, e_n$ are located into the subterm $\mathtt{p}_{|w_1}$, which is changed or deleted by the action under consideration. Note also that the action **change** is *safe*, which implies that not new errors are introduced because of the execution of the repair action. ■

In other words, Proposition 3.1.2 states that repairing a given correctness error evidence $e_1 \equiv (\mathtt{p}, w_1, \mathtt{l}_1, \sigma_1, C_1)$ allows us to fix automatically any error which is included in the term $\mathtt{p}_{|w_1}$.

However, what happens when errors are not comparable w.r.t. $\preceq$, or we decide to fix an error which is not the smallest in the order? Is it still possible to fix more than one error at a time? In the following, we deepen our analysis about the relation among correctness error evidences in order to answer these questions. Let us start by providing an auxiliary definition.

Definition 3.1.3 *Let* $e_1 \equiv (\mathtt{p}, w_1, \mathtt{l}_1, \sigma_1, C_1)$ *and* $e_2 \equiv (\mathtt{p}, w_2, \mathtt{l}_2, \sigma_2, C_2)$ *be two correctness error evidences in* $E_N(\mathtt{p})$*. We say that* e_2 overlaps e_1 *in* w *(in symbols,* $e_2 \between_w e_1$*), iff (i)* $e_1 \preceq e_2$*, and (ii) there exists* $w \equiv min(Emb_{\mathtt{l}_1}(\mathtt{p}_{|w_1}) \cap Emb_{\mathtt{l}_2}(\mathtt{p}_{|w_2}))$*, where* $min(X) = w$ *s.t.* $w \leq w_i$ *for all* $w_i \in X$*. When position* w *is not relevant or clear from the context, we simply write* e_2 overlaps e_1 *or* $e_2 \between e_1$*.*

By notation $e_2 \not\between e_1$, we denote that e_2 does not overlap e_1. Given two correctness errors evidences e_1 and e_2 of a Web page $\mathtt{p}$, we can distinguish three possible scenarios:

1. e_1 and e_2 are not comparable w.r.t. $\preceq$ (see Figure 3.1(a));

2. $e_1 \preceq e_2$ and e_2 does not overlap e_1 (see Figure 3.1(b));

3. e_2 overlaps e_1 (see Figure 3.1(c)).

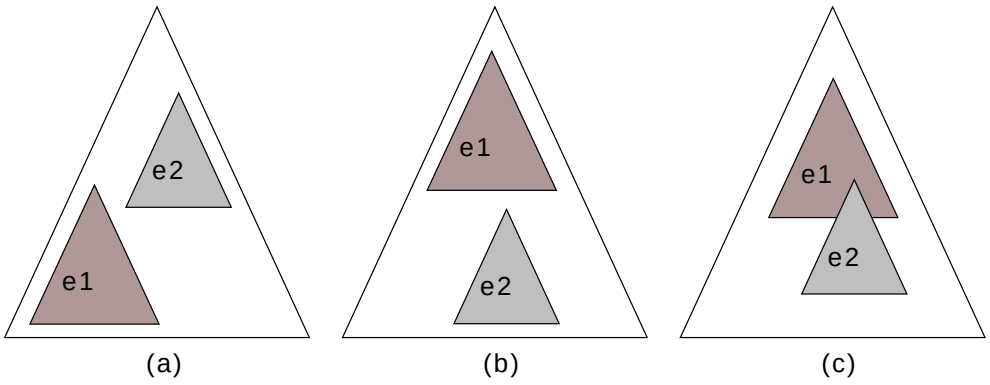


Figure 3.1: Taxonomy of error dependencies.

In the case 1, correctness error evidences e_1 and e_2 are completely independent, and hence repairing one of them does not affect the correction of the other one. This fact, together with Proposition 3.1.2, leads us to an obvious optimization of the correction framework which is formalized by the $\mathbb{M}$ strategy described in Section 3.1.2.

Let us consider now the case 2. In this case, still by Proposition 3.1.2, we are able to fix automatically e_2 by just repairing e_1. Vice versa, fixing e_2 will not help to fix e_1, as stated in the next proposition.

Proposition 3.1.4 *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page. Let* $e_1 \equiv (\mathtt{p}, w_1, \mathtt{l}_1, \sigma_1, C_1)$ *and* $e_2 \equiv (\mathtt{p}, w_2, \mathtt{l}_2, \sigma_2, C_2)$ *be two correctness error evidences in* $E_N(\mathtt{p})$ *such that* $e_1 \preceq e_2$ *and* $e_2 \not\preceq e_1$. *If* $\mathtt{p}' \equiv action(\mathtt{p}, w_2, t)$, *with action* $\in \{\mathbf{delete}, \mathbf{change}\}$, *then (i)* $p'_{|_{w_2}}$ *is repaired, (ii)* $(p', w_1, \mathtt{l}_1, \sigma', C_1) \in E_N(\mathtt{p}')$ *for some substitution* σ'.

Proof. By contradiction. Assume that by repairing the error e_2, also the error e_1 is repaired. This implies that there exists at least a position v such that $v \in Emb_{\mathtt{l}_1}(p)$ and $w_2 \leq v$. Since $w_1 \leq w_2$, then also $w_2 \in Emb_{\mathtt{l}_1}(p)$. Thus, $(Emb_{\mathtt{l}_1}(p) \cap Emb_{\mathtt{l}_2}(p)) \neq \emptyset$, i.e., $e_2 \between e_1$, which leads to a contradiction. ■

Example 3.1.5

Consider the Web page $\mathtt{p} \equiv f(g(a), h(b))$ and the following correctness error evidences

$$e_1 \equiv (\mathtt{p}, \Lambda, f(X), \{X/h(b)\}, \emptyset\}),\ e_2 \equiv (\mathtt{p}, 2, h(Y), \{Y/b\}, \emptyset)$$

Thus, $e_1 \preceq e_2$ and e_2 does not overlap e_1. Now observe that we can fix e_2 by either removing subterm $h(b)$ or by changing subterm $h(b)$ with a suitable term t. In both cases, such a repair will not fix e_1.

In the case 3, e_1 and e_2 are "connected", since $e_1 \preceq e_2$ and e_2 overlaps e_1. Roughly speaking, this fact tells us that the correctness error evidence e_2 is partly "contained" in e_1 and thus fixing e_2 might also yield a fix for e_1. Anyway, this is not always the case as the next example shows.

Example 3.1.6

Consider the Web page $\mathtt{p} \equiv f(g(a), h(b))$ and the following correctness error evidences $e_1 \equiv (\mathtt{p}, \Lambda, f(h(X)), \{X/b\}, \emptyset)$, $e_2 \equiv (\mathtt{p}, 2, h(X), \{X/b\}, \emptyset)$. Thus, $e_1 \preceq e_2$ and e_2 overlaps e_1. We can fix e_2 by changing, for instance, $h(b)$ with $h(a)$. However, such a fix would not repair e_1 automatically, while by removing $h(b)$ or by replacing $h(b)$ with term $l(c)$ we would fix both e_2 and e_1 just by executing a single repair action.

As Example 3.1.6 illustrates, some conditions are necessary in order to automatically achieve a fix for e_1 by simply correcting e_2. The next proposition clarifies the ingredients we need to this purpose.

Proposition 3.1.7 *Let $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ be a Web page. Let $e_1 \equiv (\mathtt{p}, w_1, \mathtt{l}_1, \sigma_1, C_1)$ and $e_2 \equiv (\mathtt{p}, w_2, \mathtt{l}_2, \sigma_2, C_2)$ be two correctness error evidences in $E_N(\mathtt{p})$ such that $e_1 \preceq e_2$ and e_2 overlaps e_1 in w. The following results hold:*

1) *If $\mathtt{p}' \equiv \mathbf{delete}(\mathtt{p}, w_2, t)$, then*
   (i) $p'_{|w_2}$ is repaired,
   (ii) $(p', w_1, \mathtt{l}_1, \sigma') \notin E_N(\mathtt{p}')$, for any substitution σ' ;

2) *If* $\mathtt{p}' \equiv \mathbf{change}(\mathtt{p}, w_2, t)$ *and* $\mathtt{l}_{1|w} \not\trianglelefteq t$*, then*
   (i) $p'_{|w_2}$ *is repaired,*
   (ii) $(p', w_1, \mathtt{l}_1, \sigma') \notin E_N(\mathtt{p}')$*, for any substitution* σ'*;*

3) *If* $\mathtt{p}' \equiv \mathbf{change}(\mathtt{p}, w_2, t)$ *is a safe repair action,* $\mathtt{l}_{1|w}\sigma' \trianglelefteq t$ *for some substitution* σ'*, and* $C_1(\sigma_1/\sigma')$ *does not hold, then*
   (i) $p'_{|w_2}$ *is repaired,*
   (ii) $(p', w_1, \mathtt{l}_1, \sigma_1/\sigma', C_1) \notin E_N(\mathtt{p}')$.

Proof. Claim 1 and 2 follow from Proposition 3.1.2 straightforwardly. The proof of Claim 3 exploits Proposition 3.1.2 and Proposition 2.1.6, which establishes that no new errors are introduced in the Web page by executing a change action. ■

Roughly speaking, Proposition 3.1.7 states that: (i) when a **delete** action is chosen to fix a correctness error evidence e_2, which overlaps a smaller (w.r.t. $\preceq$) correctness error evidence e_1, such an action will always fix e_1 as well; (ii) when a repair action $\mathtt{p}' \equiv \mathbf{change}(p, w_2, t)$ is performed in order to fix e_2, some extra conditions are necessary in order to ensure that the term t to be inserted will automatically fix e_1. Basically, these conditions establish that either (an instance of) the faulty term $\mathtt{l}_1$ is not recognized in $\mathtt{p}'$ or, if such an instance is detected, the associated condition does not hold. This suffices to guarantee that $(p', w_1, \mathtt{l}_1, \sigma_1/\sigma', C_1) \notin E_N(\mathtt{p}')$.

3.1.2 Correction Strategies

As we explained in Section 2.1, a given correctness error evidence e in a Web page $\mathtt{p}$ can be fixed by executing a suitable repair action a. By (e, a) we denote a pair containing a repair action a that fixes e. Moreover, by notation $p' = a(p)$ we intend the execution of the repair action a on the Web page p which returns the Web page p'.

Definition 3.1.8 *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page, and let* $\mathbb{E}(\mathtt{p}) = \{e_1, \ldots, e_n\}$ *be the set of correctness error evidences of* p*. A* correction strategy *for* $\mathtt{p}$ *is a sequence* $\langle (e_1, a_1), \ldots (e_2, a_n)\rangle$*, where* $a_1, \ldots a_n$ *are repair actions such that*

1. $\mathtt{p}_0 = \mathtt{p}$*;*

2. $\mathtt{p}_i = a_i(\mathtt{p}_{i-1})$, $0 < i \leq n$.

and $\mathtt{p}_n$ *is repaired.*

Roughly speaking, given a faulty Web page $\mathtt{p}$, a correction strategy for $\mathtt{p}$ allows one to fix all the bugs in $\mathtt{p}$ by running all the repair actions occurring in the strategy.

As we shown in Section 3.1.1, fixing a correctness error evidence may automatically repair others bugs. This fact suggests us that a correctness strategy does not necessary contain a pair (e, a) for any correctness error evidence e which appears in a faulty Web page. In the following, we describe two possible correction strategies which exploit the results of Section 3.1.1. The former aims at minimizing the number of actions which are needed in order to repair a Web page, whereas the purpose of the latter one is to reduce the amount of information to be removed/changed for correcting a Web site. In both cases, we assume that for any $e \in E_N(\mathtt{p})$, we have an *error/action* pair (e, a) at hand, and we call the set containing such pairs $\mathbb{EA}(\mathtt{p})$. In other words, we associate a repair action a with every correctness error evidence e.

The minimal strategy

First of all, we provide a partial ordering over $\mathbb{EA}(\mathtt{p})$ which is directly induced by the ordering $\preceq$ over correctness error evidences.

Let $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ be a Web page. Given $(e_1, a_1), (e_2, a_2) \in \mathbb{EA}(\mathtt{p})$, $(e_1, a_1) \sqsubseteq_T (e_2, a_2)$ iff $e_1 \preceq e_2$. We say that $(e, a) \in \mathbb{EA}(\mathtt{p})$ is *minimal* w.r.t. $\sqsubseteq_T$ iff there does no exist $(e', a') \in \mathbb{EA}(\mathtt{p})$ such that $(e', a') \sqsubseteq_T (e, a)$.

Now, let us observe the following facts.

- **Fact 1.** By Proposition 3.1.2, we note that for any $(e, a), (e', a') \in \mathbb{EA}(\mathtt{p})$ such that $(e, a) \sqsubseteq_T (e', a')$, the execution of the repair action a will fix both e and e'. Therefore, fixing a correctness error evidence e which corresponds to a minimal $(e, a) \in \mathbb{EA}(\mathtt{p})$ w.r.t. $\sqsubseteq_T$ will fix all the correctness error evidences e' which are greater than e, without running any other repair action.

- **Fact 2.** Given (e_1, a_1), $(e_2, a_2) \in \mathbb{EA}(\mathtt{p})$ both minimal w.r.t. $\sqsubseteq_T$, e_1 and e_2 are not comparable w.r.t. $\preceq$.

In the light of these facts, it should be rather clear that it suffices to fix those errors corresponding to minimal error/action pairs in order to fix the whole Web page.

Definition 3.1.9 (Minimal strategy) *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page, and let* $\mathbb{E}(\mathtt{p})$ *be the set of correctness error evidences of p. A* minimal *strategy (or* $\mathbb{M}$ strategy*) for* $\mathtt{p}$ *is a sequence* $\langle (e_1, a_1), \ldots (e_m, a_m) \rangle$, $(e_i, a_i) \in \mathbb{EA}(\mathtt{p})$, $i = 1, \ldots m$, *such that each* (e_i, a_i) *is minimal w.r.t.* $\sqsubseteq_T$.

Roughly speaking, only the repair actions associated with errors evidences that are rooted at minimal positions need to be executed in order to make the Web page correct.

Proposition 3.1.10 *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page. Then, the* $\mathbb{M}$ *strategy for* $\mathtt{p}$ *is a correction strategy for* $\mathtt{p}$.

Proof. Immediate by the definition of minimality w.r.t. $\sqsubseteq_T$ and Proposition 3.1.2. ■

Moreover, since minimal error/action pairs only refer to incomparable (w.r.t. $\preceq$), and thus independent, correctness error evidences, we may run each repair action of the $\mathbb{M}$ strategy in parallel, whenever a parallel architecture is available, speeding up the correction process.

Example 3.1.11

Consider the Web page $\mathtt{p} \equiv f(g(10), h(d), 20)$ together with the following sequence of error/action pairs:

$$\begin{array}{l}\langle((\mathtt{p}, \Lambda, f(g(X), 20), \{X/10\}, \{X < 20\}),\\ \ \mathbf{change}(p, \Lambda, f(g(20), 10)),\\ \ ((\mathtt{p}, 2, h(Y), \{Y/d\}, \emptyset), \mathbf{delete}(\mathtt{p}, 2, h(d))\rangle\end{array}$$

In this case, the $\mathbb{M}$ strategy corresponds to the unary sequence

$$\begin{array}{l}\langle((\mathtt{p}, \Lambda, f(g(X), 20), \{X/10\}, \{X < 20\}),\\ \ \mathbf{change}(p, \Lambda, f(g(20), 10))\rangle.\end{array}$$

The following result establishes that it suffices to consider minimal error/action pairs in order to define a correction strategy which minimizes the number of actions we need to perform for repairing a given Web page.

Proposition 3.1.12 *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page, and let* $\mathcal{T}$ *be the* $\mathbb{M}$ *strategy for* $\mathtt{p}$*. Then, for every correction strategy* $\mathcal{S}$ *for* $\mathtt{p}$*,* $length(\mathcal{T}) \leq length(\mathcal{S})$*, where* $length(\cdot)$ *computes the number of error/action pairs of a given correction strategy.*

Proof. By contradiction. Let us assume that $length(\mathcal{S}) < length(\mathcal{T})$ for some correction strategy $\mathcal{S}$. Then, there exists a pair $(e, a) \in \mathcal{T}$ such that $(e, a) \notin \mathcal{S}$. Moreover, since $\mathcal{S}$ is a correction strategy, there exists a pair $(e', a') \in \mathcal{S}$ such that, by performing (e', a'), the error e is also repaired. Hence, $e' \leq e$, which leads to a contradiction, since by Definition 3.1.9 if $(e, a) \in \mathcal{T}$ then (e, a) is minimal w.r.t. $\sqsubseteq_T$. ■

Minimal non-overlapping strategy

The $\mathbb{M}$ strategy typically forces the user to modify/introduce a lot of information in a Web page $\mathtt{p}$, even if only minor changes are required to fix $\mathtt{p}$. Let us see an example.

Example 3.1.13

Let us consider the Web page $p \equiv f(g(a), k(m(c)), h(a))$ and the set $E_N(\mathtt{p}) = \{(\mathtt{p}, \Lambda, f(g(X), h(Y)), \{X/a, Y/a\}, \{X = Y\})\}, (\mathtt{p}, 1, g(a), \varepsilon, \emptyset)\}$. The $\mathbb{M}$ strategy would only fix the "minimal" error at the root position. This fact might force the user to provide a quite big amount of information in case a **change** action is taken, even if a close variant of $\mathtt{p}$ would have been enough to fix the bug.

For instance, if the chosen **change** action was **change**$(\mathtt{p}, \Lambda, f(g(b), k(m(c)), h(a)))$, the user should re-enter the whole Web page $\mathtt{p}$ with just a small change at position 1.1.

Instead, if we repaired $(\mathtt{p}, 1, g(a), \varepsilon, \emptyset)$ by means of the following action **change**$(\mathtt{p}, 1, g(b))$, the user would correct both errors by introducing a smaller amount of information.

The idea behind the minimal non-overlapping strategy is thus to "push" the corrections towards the leaves of the Web page as much as possible and to automatically propagate them up to the root position.

Obviously, given two error evidences e and e' sucht that $e' \preceq e$, correcting e does not guarantee to automatically fix e' (see, for instance, Example 3.1.5). Indeed, by Proposition 3.1.4, whenever an error evidence e' does not overlap a given error evidence e, there is no possibility to automatically spread a correction for e up to e'. On the other hand, under suitable conditions, overlapping error evidences allow one to infer a repair on e' by just fixing e (see Proposition 3.1.7).

Therefore, the strategy works as follows. First of all, given a Web page $\mathtt{p}$, we partition $E_N(\mathtt{p})$ into the two following sets:

- $\mathbb{N}\text{OVL}(\mathtt{p}) = \{e \in E_N(\mathtt{p}) \mid \nexists\, e', e' \between e\}$
- $\emptyset\text{VL}(\mathtt{p}) = E_N(\mathtt{p}) \setminus \mathbb{N}\text{OVL}(\mathtt{p})$.

Clearly, $E_N(\mathtt{p}) = \mathbb{N}\text{OVL}(\mathtt{p}) \cup \emptyset\text{VL}(\mathtt{p})$. We call error evidences in $\mathbb{N}\text{OVL}(\mathtt{p})$ (resp., in $\emptyset\text{VL}(\mathtt{p})$) *non-overlapping* (resp., *overlapping*) error evidences. Note that a non-overlapping error evidence e cannot be automatically fixed by executing a repair action on an error evidence e' such that $e \preceq e'$, since correction effects cannot be propagated up. However, this is the case of the overlapping error evidences which may be implicitly affected by other repairs. Actually, the following facts hold.

- **Fact 1.** Given an overlapping error evidence e, there must exist a non-overlapping error evidence e' such that $e \preceq e'$.
- **Fact 2.** Let $e, e_0, e_1, \ldots e_n$, $n \geq 0$, be correctness error evidences. If e is an overlapping error evidence s.t. e_0 overlaps e and $e \preceq e_n \preceq e_{n-1}, \ldots \preceq e_0$, then e_i overlaps e, $i = 1, \ldots n$.

These facts, together with Proposition 3.1.2, suggest us that it suffices to fix only non-overlapping error evidences in order to get a repaired Web page. This is because: (i) all the error evidences which are greater (w.r.t. $\preceq$) than the considered non-overlapping error evidences will be repaired, as stated by Proposition 3.1.2; (ii) for each overlapping error evidence e there is always $e' \in \mathbb{N}\text{OVL}(\mathtt{p})$ which overlaps e, hence repairing e' will also fix e, whenever the following *safety* property is fulfilled:

Definition 3.1.14 *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page. Let* $e \equiv (\mathtt{p}, w, \mathtt{l}, \sigma, C) \in \mathbb{N}OVL(\mathtt{p})$ *be a correctness error, and let*

$(e, \textbf{change}(\mathtt{p}, w, t)) \in \mathbb{EA}(\mathtt{p})$ *be an error/action pair. Then, the* safety *property for* $(e, \textbf{change}(\mathtt{p}, w, t)) \in \mathbb{EA}(\mathtt{p})$ *states that, for each* $e' \equiv (\mathtt{p}, w', \mathtt{l}', \sigma', C') \in \emptyset VL(\mathtt{p})$ *such that* $e' \preceq e$*, one of the following conditions must hold:*

(*i*) $\mathtt{l}'_{|w} \not\trianglelefteq t$, *or*
(*ii*) $\mathtt{l}'_{|w}\sigma' \trianglelefteq t$ *for some substitution* σ'*, and* $C'(\sigma/\sigma')$ *does not hold.*

Note that the above safety property directly comes from Proposition 3.1.7 which guarantees the automatic propagation of the repairs. Moreover, observe that such a property only affects **change** actions, since **delete** actions always enable the correction propagation.

Now, we are ready to provide the minimal non-overlapping strategy.

Definition 3.1.15 (Minimal non-overlapping strategy) *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *be a Web page, and let* $\mathbb{N}OVL(\mathtt{p})$ *be the set of non-overlapping correctness error evidences of* p*. A* minimal non-overlapping *strategy (or* $\mathbb{MNO}$ *strategy) for* $\mathtt{p}$ *is a sequence* $\langle (e_1, a_1), \ldots (e_m, a_m) \rangle$*,* $(e_i, a_i) \in \mathbb{EA}(\mathtt{p})$*,* $i = 1, \ldots m$*, such that*

(*i*) $e_i \in \mathbb{N}OVL(\mathtt{p})$ *and each* (e_i, a_i) *is minimal w.r.t.* $\sqsubseteq_T$ *in* $\mathbb{N}OVL(\mathtt{p})$;
(*ii*) *if* a_i *is a* **change** *action, then the safety property for* (e_i, a_i) *must hold.*

Proposition 3.1.16 *Let* $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$*. Then the* $\mathbb{MNO}$ *strategy for* $\mathtt{p}$ *is a correction strategy for* $\mathtt{p}$*.*

Proof. Immediate from the notion of minimality w.r.t. $\sqsubseteq_T$, Definition 3.1.14, and Proposition 3.1.2. ■

In Figure 3.2, we show how the $\mathbb{MNO}$ strategy works. For the sake of simplicity we just label each node of the given Web page with: `ok`, if no error evidence is rooted at the considered node; `ov`, if an overlapping error evidence is rooted at the considered node; or `no`, if a non-overlapping error evidence is rooted at the considered node. The Web page contains nine errors, but we just need to fix three errors in order to end up with a repaired Web page. Precisely, these errors correspond to the minimal non-overlapping error evidences occurring in the Web page.

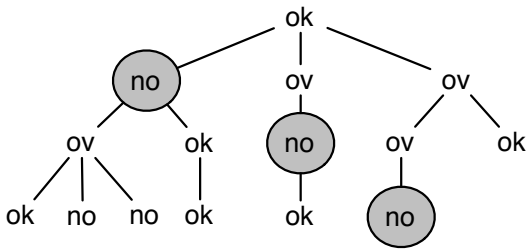


Figure 3.2: The $\mathbb{MNO}$ strategy

Example 3.1.17

Consider the Web page
$\mathtt{p} \equiv f(g_1(h_1(a_1, a_2), h_2(b_1, b_2)), g_2(h_3(c_1), h_4))$ and

$$\begin{aligned} E_N(\mathtt{p}) = \{&(\mathtt{p}, 1, g_1, \varepsilon, \emptyset), (\mathtt{p}, 1.1, h_1(a_2)\varepsilon, \emptyset), \\ &(\mathtt{p}, 1.1.2, a_2, \varepsilon, \emptyset), (\mathtt{p}, 2.1.1, c_1, \varepsilon, \emptyset), \\ &(\mathtt{p}, 2.1, h_3(c_1), \varepsilon, \emptyset), \\ &(\mathtt{p}, 2, g_2(h_3(X)), \{X/c_1\}, \emptyset)\} \end{aligned}$$

Hence,

$$\begin{aligned} \mathbb{N}\mathrm{OVL}(\mathtt{p}) = \{&(\mathtt{p}, 1, g_1, \varepsilon, \emptyset), (\mathtt{p}, 1.1.2, a_2, \varepsilon, \emptyset), \\ &(\mathtt{p}, 2.1.1, c_1, \varepsilon, \emptyset)\} \end{aligned}$$

$$\begin{aligned} \emptyset\mathrm{VL}(\mathtt{p}) = &(\mathtt{p}, 1.1, h_1(a_2), \varepsilon, \emptyset), \\ &(\mathtt{p}, 2, g_2(h_3(X)), \{X/c_1\}, \emptyset), \\ &(\mathtt{p}, 2.1, h_3(c_1), \varepsilon, \emptyset)\} \end{aligned}$$

The $\mathbb{MNO}$ strategy only corrects minimal non-overlapping error evidences. A possible $\mathbb{MNO}$ strategy for $\mathtt{p}$ might be:

$$\begin{aligned} \langle\ &((\mathtt{p}, 1, g_1, \varepsilon, \emptyset), \mathbf{delete}(\mathtt{p}, 1, g_1(h_1(a_1, a_2), h_2(b_1, b_2)), \\ &((\mathtt{p}, 2.1.1, c_1, \varepsilon, \emptyset), \mathbf{change}(\mathtt{p}, 2.1.1, c_4))\ \rangle \end{aligned}$$

And the safety property for $((\mathtt{p}, 2.1.1, c_1, \varepsilon, \emptyset), \mathbf{change}(\mathtt{p}, 2.1.1, c_4))$ is fulfilled. The execution of the correction strategy thus yields the following repaired Web page $f(g_2(h_3(c_4), h_4))$.

Finally, observe that we needed to fix only two errors out of six, and just minor fixes were necessary to make the original Web page correct.

So far, we have presented a significant extension of the preliminary Web site correction framework of [ABFR06], which improved several aspects of the repair methodology:

(i) we provided a detailed analysis of errors which clarified the relation among correctness errors in Web sites;

(ii) by exploiting the results of the analysis, we formulated two correction strategies which reduce the number of repair actions as well as the amount of information that is needed to fix a given Web site;

(iii) the considered correction strategies increase the level of automation of the repairing methodology, since the user has to fix just a small number of correctness errors in order to make the whole Web site correct.

3.2 Fixing Web Sites by Using Completeness Strategies

The specification language together with the verification technique given in Chapter 1 allow us to detect incomplete or missing Web pages, delivering as outcome a set of completeness error evidences. These errors represent the incomplete or absent information in the Web site. There are three kinds of completeness errors, namely: missing page (M), universal completeness error (A), and existential completeness error (E). These errors can be detected by means of partial rewriting on the Web site with respect to the Web specification.

In this section, first we extended the Definitions 1.4.3, 1.4.5, and 1.4.6 of completeness errors by adding, into the error structure, the rewrite sequence that led to detecting the error. Then, we define two new completeness strategies that further improve our repairing framework [ABFR06].

Let us start with an auxiliary definition. Let $s \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ be a term, and let W be a Web site. We define the set $mark(s, W)$ as the set of all pages in W that embed the marking information s. For example, consider the Web site $W = \{p_1, p_2\}$ where $p_1 = f(m(a))$ and $p_2 = h(g(b))$, then $mark(\sharp f(h(X)), W) = \{p_1\}$.

A completeness error is defined as follows.

Definition 3.2.1 (completeness error) – Extended version of Definitions 1.4.3, 1.4.5, and 1.4.6 – *Let W be a Web site, and let (I_N, I_M, R) be a Web specification. Let $c \in \{M, A, E\}$ be the kind of completeness error (Missing Web page, Universal completeness error, or Existential completeness error, respectively). Let $q \in \{A, E\}$. Then, the tuple $e \equiv (s_0 \rightharpoonup^+_{I_M} s_n, P, c)$ is a completeness error in W w.r.t. I_M if:*

(i) For some $l \rightharpoonup r\langle q \rangle \in I_M$, there exists a substitution σ such that:
$$l\sigma = s_0 \ \wedge \ s_0 \trianglelefteq p \ \wedge \ p \in W$$

(ii) For some $l \rightharpoonup r\langle q \rangle \in I_M$, there exists a substitution σ such that:
$$r\sigma = s_n \ \wedge \ P = \{p \mid p \in mark(r, W) \ \wedge \ s_n \not\trianglelefteq p\} \ \wedge$$
$$((c = M \wedge P = \emptyset) \text{ or } (c = q \wedge P \neq \emptyset))$$

Roughly speaking, the chain $s_0 \rightharpoonup^+_{I_M} s_n$ represents the sequence of partial rewriting that generates the requirement which is not fulfilled. The first condition defines the subpart of the Web page that satisfies the left-hand side of a completeness rule, whereas the second condition returns the requirement not fulfilled by the set of Web pages P. Note that the rules used in these conditions could be different, due to the transitive closure of partial rewriting relation.

We denote by $E_M(W)$ the set of all the completeness error evidences w.r.t. I_M for a Web site W. When no confusion can arise, we just write E_M.

3.2.1 Completeness Error Dependencies

As in Section 3.1.1, the completeness errors in a Web site may be somehow connected, and repair one of them might fix other errors. In this section, we analyze the dependencies among completeness errors. Later on, we will exploit this information in order to develop two novel completeness strategies that aim to minimizing the amount of information the user needs to change or delete.

First of all, we define two partial orders ($\preceq_{inf}$ and $\preceq^{sup}$) on the set of completeness error evidences. These orders are based on the sequence of partial rewriting of the completeness error.

Definition 3.2.2 ($\preceq_{inf}$) *Let $e_1, e_2 \in E_M(W)$ be two completeness error with $e_1 \equiv (s_0 \rightharpoonup^+_{I_M} s_n, P_1, q_1)$ and $e_2 \equiv (t_0 \rightharpoonup^+_{I_M} t_m, P_2, q_2)$. Then,*

$$e_1 \preceq_{inf} e_2 \quad \textit{iff} \quad s_0 \trianglelefteq t_0$$

Note that e_1 is incomparable with e_2 w.r.t. $\preceq_{inf}$ if $s_0 \not\trianglelefteq t_0$. We say that an error $e \in E_M$ is minimal *w.r.t. $\preceq_{inf}$, if and only if there does not exist $e' \in E_M$ such that $e' \preceq_{inf} e$ and $e' \not\equiv e$.*

Definition 3.2.3 ($\preceq^{sup}$) *Let $e_1, e_2 \in E_M(W)$ be two completeness error with $e_1 \equiv (s_0 \rightharpoonup^+_{I_M} s_n, P_1, q_1)$ and $e_2 \equiv (t_0 \rightharpoonup^+_{I_M} t_m, P_2, q_2)$. Then,*

$$e_1 \preceq^{sup} e_2 \quad \textit{iff} \quad s_n \trianglelefteq t_m.$$

Note that e_1 is incomparable with e_2 w.r.t. $\preceq^{sup}$ if $s_n \not\trianglelefteq t_n$. We say that an error $e \in E_M$ is maximal *w.r.t. $\preceq^{sup}$, if and only if there does not exist $e' \in E_M$ such that $e \preceq^{sup} e'$ and $e' \not\equiv e$.*

Note that the above partial orders are defined by considering the embedding relation between the first two terms of the sequences of partial rewriting ($\preceq_{inf}$), and the last two terms of the sequences of partial rewriting ($\preceq^{sup}$), respectively.

With regard to the relation $\preceq_{inf}$, the following proposition states that the action of repairing a minimal error e with respect to $\preceq_{inf}$ via the operation *repairByDelete* (Definition 2.1.10), allows us to repair all errors related with e by means of $\preceq_{inf}$ in an automatic way.

Proposition 3.2.4 *Let W be a Web site. Let $e_i \in E_M(W), i = 1 \ldots n$ be completeness errors in W, and let e_1 be a minimal error such that $e_1 \preceq_{inf} \ldots \preceq_{inf} e_n$. Then, after repairing the completeness error e_1 by using the operation repairByDelete, all errors $e_1, \ldots, e_n$ are repaired.*

Proof. First of all, let us recall from Definition 2.1.10 that a completeness error is repaired by removing the term (or any subterm of it) that begins the partial rewriting sequence.

Consider the completeness errors $e_1, \ldots, e_n \in E_M(W)$ such that $e_1 \equiv (s_{1_0} \rightharpoonup^+_{I_M} s_{1_m}, P_1, q_1), \ldots, e_n \equiv (s_{n_0} \rightharpoonup^+_{I_M} s_{n_m}, P_n, q_n)$ and $e_1 \preceq_{inf} \ldots \preceq_{inf} e_n$. By Definition 3.2.2, we have that $s_{1_0} \trianglelefteq, \ldots, \trianglelefteq s_{n_0}$. Then, by using

$repairByDelete(e_1, W)$ to repair the error e_1, the term s_{1_0} is removed from the whole Web site W. This fact implies that each subterm $t = s_{1_0}$ of $s_{2_0}, \ldots s_{n_0}$ is removed too. Finally, by Definition 2.1.10, the errors $e_2, \ldots e_n$ are repaired, which concludes the proof. ■

Note that Proposition 3.2.4 does not depend on the kind of completeness error (Missing Web page, Universal completeness error, and Existential completeness error).

As for the relation $\preceq^{sup}$, let us show how we can repair a Web site by adding the missing information required. First, we need to analyze the errors with respect to both the relation $\preceq^{sup}$ and the information needed to repair them.

Let us consider $e_1 \preceq^{sup} e_2$ with $e_1 \equiv (s_0 \rightharpoonup^+_{I_M} s_n, P_1, q_1)$ and $e_2 \equiv (t_0 \rightharpoonup^+_{I_M} t_m, P_2, q_2)$. Then, is it possible to repair more than one error in an automatic way?. The reason why it is possible to repair more than one error in an automatic way stems from the following considerations:

- Since the last term in the sequence of partial rewriting is the information to be added in order to repair the error, then, by definition of $\preceq^{sup}$, $s_n \trianglelefteq t_n$. Hence, if we add t_n, we also added s_n.
- Assume $e_1 \preceq^{sup} e_2$. Then, by the previous point, e_2 must be repaired before e_1.
- If a maximal error e is repaired, then all the errors e' such that $e' \preceq^{sup} e$ with e' being of kind *missing Web page* or *existential error*, are repaired too. This is because in a simple step, we add the information that embeds the requirements from the other missing Web pages and existential errors.

These considerations are the basis of Algorithm 3, which implements the operation $repairByInsert$ that allows us to repair completeness errors that are related by means of the order $\preceq^{sup}$.

The result given in the Proposition 3.2.4 together with the Algorithm 3 provide an obvious optimization of the repair framework that we will formalize by means of the strategies presented in the next section.

Algorithm 3 Procedure to repair a set of completeness error evidences ordered by $\preceq^{sup}$.

Require:
$E = \{e_i \mid e_i \in E_M(W), i = 1, \ldots m, \text{ and } e_1 \preceq^{sup} \ldots \preceq^{sup} e_m\}$
W be a Web site.
Ensure:
$W \mid \forall e \in E, e \notin E_M(W)$

```
1: procedure repairByInsert (E, W)
2:   P_R = {} // Set of repaired pages.
3:   for i ← m to 1 do
4:     (s_0 →+_{I_M} s_n, P, q) ← e_i
5:     if q = M and P_R = {} then
6:       W ← W ∪ {add(s_n, W)}
7:       P_R ← P_R ∪ {s_n}
8:     else if q = E and P_R = {} then
9:       p ← element(P) // Get a Web page.
10:      p' ← insert(p, w, s_n) // w is an arbitrary position in p.
11:      W ← W \ {p} ∪ {p'}
12:      P_R ← P_R ∪ {p}
13:    else if q = A then
14:      P_Aux ← P \ P_R
15:      for all p ∈ P_Aux do
16:        p' ← insert(p, w, s_n) // w is an arbitrary position in p.
17:        W ← W \ {p} ∪ {p'}
18:        P_R ← P_R ∪ {p}
19:      end for
20:    end if
21:  end for
22: end procedure
```

3.2.2 Completion Strategies

The above results suggest the idea that the application of a repair action can fix more than one completeness error. This implies we do not need to execute a different repair action for each detected error, but rather we can choose suitable subsets of errors to act upon. In the following, we present two completion strategies. The aim of the first one is to reduce

the amount of information to be removed in order to derive a Web site free of completeness errors, whereas the second strategy aims to reduce the amount of information to be added to complete the Web site.

By (e, a) we denote the pair that consists of the completeness error e, and the repair action a that we intend to use to fix e, and by notation $W' = a(e, W)$ we specify the execution of the repair action a on the Web site W, which returns the Web site W' where the error e has been fixed.

Definition 3.2.5 (Repair strategy) *Let W be a Web site, and let $\{e_1, \ldots, e_n\} \subset E_M(W)$ be a subset of the completeness errors in W. A* repair strategy *for W is the sequence $[(e_1, a_1), \ldots, (e_n, a_n)]$, where $a_1, \ldots, a_n$ are repair actions such that:*

$$\begin{array}{ll} (i) & W_0 = W; \\ (ii) & W_i = a_i(e_i, W_{i-1})\ \forall i, 1 \leq i \leq n; \\ (iii) & E_M(W_n) = \emptyset \end{array}$$

By abuse, we sometimes write $([e_0, \ldots, e_n], a)$ for a subsequence $[(e_0, a_0), \ldots, (e_n, a_n)]$ when $a_i = a$ for $i = 1, \ldots, n$. Note that, in Definition 3.2.5, only a subset of the errors is considered. Roughly speaking, a *repair strategy* is a sequence of repair actions that, once executed, allows all the completeness errors in a given Web site to be repaired.

In the following, we formalize several completion strategies for a given Web site with respect to the specification I_M.

Reduce-delete-actions Strategy

The relation $\preceq_{inf}$ defines a partial order on the set of completeness error evidences E_M. Furthermore, Proposition 3.2.4 ensures that if a minimal error e is fixed by applying $repairByDelete$, then the other errors in the set that are greater than or equal to e are repaired too. Obviously, if there are two minimal independent errors, both errors need to be repaired independently as well.

Definition 3.2.6 (*Reduce-delete-actions* Strategy - $\mathbb{RDA}$) *Let W be a Web site, and let $E_M(W)$ be a set of completeness error evidences in W. We call $\mathbb{RDA}$ strategy the repair strategy that reduces the number of* **delete** *actions for a given Web site W as follows:*

$([e_1, \ldots, e_n], repairByDelete)$,
where $\forall i, 1 \leq i \leq n, e_i \in E_M(W) \wedge e_i$ *is minimal w.r.t.* $\preceq_{inf}$

Roughly speaking, the strategy boils down to repair all minimal errors with respect to the relation $\preceq_{inf}$. The following proposition establishes that the $\mathbb{RDA}$ strategy allows us to derive a Web site free of completeness errors by repairing only a subset of the errors in $E_M(W)$.

Proposition 3.2.7 *Let W be a Web site, and let $E_M(W)$ be the set of completeness error evidences in W. Then, the $\mathbb{RDA}$ strategy transforms W into a Web site free of completeness error evidences by applying a number of repair actions that is less than or equal to the number of completeness errors $E_M(W)$.*

Proof. Given a set of completeness error evidences $E_M(W)$, let $e \in E_M(W)$ be a completeness error. When applying the $\mathbb{RDA}$ strategy on W w.r.t. $E_M(W)$, we can distinguish two cases:

Case(i). e be minimal w.r.t. $\preceq_{inf}$. This implies that there exists a pair $(e, repairByDelete)$ in the $\mathbb{RDA}$ strategy, hence e is repaired by applying one repair action.

Case(ii). Let $e' \in E_M(W)$ be minimal w.r.t. $\preceq_{inf}$, and let $e' \preceq_{inf} e$ with $e' \neq e$. This implies that there exists a pair $(e', repairByDelete)$ in the $\mathbb{RDA}$ strategy. By Proposition 3.2.4, e is repaired without applying any specific repair action on it. Consequently, the number of repair actions needed to repair W is less than (or equal to) the number of completeness errors.

■

Reduce-insertion-actions Strategy

The procedure *repairByInsert* (Algorithm 3) allows us to reduce both the amount of information to be added and the number of repair actions to be executed on a subset of completeness error evidences defined by relation $\preceq^{sup}$.

Let $\{e_1^m, \ldots, e_n^m\}$ be the set of maximal errors of E_M w.r.t. $\preceq^{sup}$. By $\mathsf{C}_{E_M}^{\preceq^{sup}}$ we denote the set of subsets of E_M that are induced by relation

$\preceq^{sup}$ as follows:

$$\begin{array}{rll} \mathsf{C}_{E_M}^{\preceq^{sup}} = & \{c_1, \ldots, c_n\} & \\ \text{where} & (\forall e_i^m \in \{e_1^m, \ldots, e_n^m\},\ 1 \leq i \leq n,\ e_i^m \in c_i) \wedge & (1) \\ & (\forall e \in E_M,\ \exists i, 1 \leq i \leq n,\ \text{s.t. } e \in c_i) \wedge & (2) \\ & (\forall i, 1 \leq i \leq n,\ \forall e_1, e_2 \in c_i,\ e_1 \preceq^{sup} e_2 \vee e_2 \preceq^{sup} e_1) & (3) \end{array}$$

Note that the number of subsets is given by the number of maximal errors. Note also that, in each subset, all the errors are related by $\preceq^{sup}$, and that all errors in E_M belong at least to one subset. Finally, observe that a completeness error e may belong to more than one subset c_i.

A naïve strategy would execute the procedure $repairByInsert$ on each subset of $\mathsf{C}_{E_M}^{\preceq^{sup}}$. However, if a particular error belong to two different subsets and one of them is repaired, this naïve strategy may lead to an inconsistency.

Let us consider $\mathsf{C}_{E_M}^{\preceq^{sup}} = \{c_1, \ldots, c_n\}$ ordered by the cardinality of the subsets c_i, i.e., $|c_i| \geq |c_{i+1}|$ for $1 \leq i < n$. By $\mathsf{C}_{E_M}^{\preceq^{sup}}(i)$ we denote $\{c_1, \ldots, c_i\}$. Then, we define a partition Γ_{E_M} of E_M as follows:

$$\begin{array}{l} \Gamma_{E_M} = \{m_i \mid m_i = dif(\mathsf{C}_{E_M}^{\preceq^{sup}}(i)),\ \forall i, 1 \leq i \leq k,\ k = |\mathsf{C}_{E_M}^{\preceq^{sup}}|\}, \\ \qquad\qquad \text{where } dif(\{x_0\}) = x_0 \\ \qquad\qquad\qquad dif(\{x_0, \ldots, x_n\}) = x_n \setminus \ldots \setminus x_0,\ \text{if } n > 0 \end{array}$$

Definition 3.2.8 (*Reduce-insertion-actions* Strategy - $\mathbb{RIA}$) *Let W be a Web site and $E_M(W)$ be the set completeness errors in W. We call $\mathbb{RIA}$ strategy the repair strategy that reduces the information to be added in order to repair a given Web site W as follows:*

$$\begin{array}{l} [(m_1, repairByInsert), \ldots, (m_n, repairByInsert)], \\ \qquad\qquad where\ \forall i, 1 \leq i \leq n,\ m_i \in \Gamma_{E_M}) \end{array}$$

Roughly speaking, the $\mathbb{RIA}$ strategy executes the procedure $repairByInsert$ on each set of the partition Γ_{E_M}.

Proposition 3.2.9 *Let W be a Web site, and let $E_M(W)$ be the set of completeness error evidences in W. Let $\Gamma(E_M)$ be a partition on E_M. Then, the $\mathbb{RIA}$ strategy transforms W in a complete Web site by applying a number of repair actions less than or equal to the number of completeness errors E_M.*

Proof. This result directly derives from Algorithm 3, which reduces the number of repair actions to be applied, since each completeness error belongs to only one set of the partition $\Gamma(E_M)$ and the errors in each set are related by means of $\preceq^{sup}$. ■

Let us illustrate these strategies by means of a rather intuitive example.

Example 3.2.10

Let W be a Web site, and let (I_N, I_M, R) be a Web specification. W and I_M are defined as follows:

Web site $W = \{p_1, p_2, p_3, p_4\}$	Completeness rules $I_M = \{r_1, r_2, r_3, r_4\}$
$p_1 = m(s(b), f(a))$	$r_1 = f(X) \rightharpoonup \sharp g(X)\langle A\rangle$
$p_2 = m(m(g(a)))$	$r_2 = g(X) \rightharpoonup \sharp h(X)\langle E\rangle$
$p_3 = m(l(b, a))$	$r_3 = h(X) \rightharpoonup \sharp p(X)\langle A\rangle$
$p_4 = h(b)$	$r_4 = l(X, Y) \rightharpoonup \sharp p(X, Y)\langle A\rangle$

Then, $E_M = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ is the set of detected completeness errors in W, where

$e_1 = ((g(a) \rightharpoonup h(a)), \{p_4\}, E)$	$e_4 = ((f(a) \rightharpoonup g(a) \rightharpoonup h(a)), \{p_4\}, A)$
$e_2 = ((h(b) \rightharpoonup p(b)), \{\}, M)$	$e_5 = ((g(a) \rightharpoonup h(a) \rightharpoonup p(a)), \{\}, M)$
$e_3 = ((l(b, a) \rightharpoonup p(b, a)), \{\}, M)$	$e_6 = ((f(a) \rightharpoonup g(a) \rightharpoonup h(a) \rightharpoonup p(a)), \{\}, M)$

Note that the errors e_2, e_3, e_5 and e_6 refer to missing Web pages, whereas e_1 is an existential error and the error e_4 is universal.

In the following we describe how to apply the strategies given in this section.

A) **Reduce-delete-actions Strategy**
The relation $\preceq_{inf}$ induces the following subsets of errors.

$$\preceq_{inf} : \{e_1 \preceq_{inf} e_5\}; \{e_2\}; \{e_3\}; \{e_4 \preceq_{inf} e_6\}$$

Then, the $\mathbb{RDA}$ strategy is applied as follows:

$$
\begin{aligned}
W' = repairByDelete(&e_4,\\
&repairByDelete(e_3,\\
&\quad repairByDelete(e_2,\\
&\quad\quad repairByDelete(e_1, W))))\\
\text{where } W' &= \{p_1, p_2, p_3\}\\
p_1 &= m(s(b))\\
p_2 &= m(m(\ ))\\
p_3 &= m(\ )\\
&- p_4 \text{ was removed } -
\end{aligned}
$$

B) **Reduce-insertion-actions Strategy**
The relation $\preceq^{sup}$ induces the following subsets of errors.

$$\preceq^{sup} : \{e_4 \preceq^{sup} e_1\}; \{e_2 \preceq^{sup} e_3\}; \{e_5 \preceq^{sup} e_6 \preceq^{sup} e_3\}$$

First of all, we define the partition Γ_{E_M} as follows:

$$
\begin{aligned}
\mathsf{C}_{E_M}^{\preceq^{sup}} &= \{c_1, c_2, c_3\}, \text{ where } c_1 = \{e_5, e_6, e_3\},\ c_2 = \{e_2, e_3\}, \text{ and}\\
&\qquad\qquad c_3 = \{e_4, e_1\}\\
\Gamma(E_M) &= \{m_1, m_2, m_3\}\\
&\quad \text{where } m_1 = c_1 = \{e_5, e_6, e_3\},\\
&\qquad m_2 = c_2 \setminus c_1 = \{e_2\} \text{ and}\\
&\qquad m_3 = c_3 \setminus c_2 \setminus c_1 = \{e_4, e_1\}
\end{aligned}
$$

Then, the $\mathbb{RIA}$ strategy is applied as follows:

$$
\begin{aligned}
W' = repairByInsert(&m_3,\\
&repairByInsert(m_2,\\
&\quad repairByInsert(m_1, W))))\\
\text{where } W' &= \{p_1, p_2, p_3, p_4, p_5, p_6\}\\
p_1 &= m(s(b), f(a))\\
p_2 &= m(m(g(a)))\\
p_3 &= m(l(b, a))\\
p_4 &= h(b, a)\\
p_5 &= p(b, a)\\
p_6 &= p(b)
\end{aligned}
$$

To conclude, by systematically applying the above strategies we are able to optimize the performance of our repair system. To our knowledge, no repair system supports such kind of optimization based on the notion of repairing strategy, which gives support for faster and simpler correction of faulty Web site.

Chapter 4

The Web Verification Service WebVerdi-M

In this chapter, we describe the rewriting-based, Web verification service WebVerdi-M [ABF+07a], which is able to recognize forbidden/incorrect patterns and incomplete/missing Web pages, and interacts with the user to (semi-)automatically repair them. WebVerdi-M relies on a powerful Web verification engine that is written in Maude, which automatically derives the error symptoms. Thanks to the AC pattern matching supported by Maude and its metalevel facilities, WebVerdi-M enjoys much better performance and usability than a previous implementation of the verification framework [BV05]. By using the XML Benchmarking tool xmlgen, we develop some scalable experiments that demonstrate the practicality of our approach.

4.1 Web Site Verification Using Maude

Maude [CDE+07] is a high-level language and high-performance system supporting both equational and rewriting logic computation, which is particularly suitable for developing domain-specific applications [EMM06; EMS03]. In addiction, the Maude language is not only intended for system prototyping, but it has to be considered as a real programming language with competitive performance. In this section, we recall some of the most important features of the Maude language which we have conveniently exploited for the optimized implementation of our Web site verification engine.

Equational attributes

Let us describe how we can model (part of) the internal representation of XML documents in our system. The chosen representation slightly mod-

ifies the data structure provided by the Haskell HXML Library [Eng02] by adding commutativity to the standard XML tree-like data representation. In other words, in our setting, the order of the children of a tree node is not relevant: e.g., $f(a, b)$ is "equivalent" to $f(b, a)$.

```
fmod TREE-XML is
sort XMLNode .
op RTNode : -> XMLNode .                    -- Root information item
op ELNode _ _ : String AttList -> XMLNode . -- Element information item
op TXNode _ : String -> XMLNode .           -- Text information items
--- ...  definitions of the other XMLNode types omitted ...
sorts XMLTreeList XMLTreeSeq XMLTree .
op Tree (_) _ : XMLNode XMLTreeList - > XMLTree .
subsort XMLTree < XMLTreeSeq .
op _,_ : XMLTreeSeq XMLTreeSeq -> XMLTreeSeq    [comm assoc id:null] .
op null : -> XMLTreeSeq .
op [_] : XMLTreeSeq -> XMLTreeList .
op [] : -> XMLTreeList .
endfm
```

In the previous module, the XMLTreeSeq constructor `_,_` is given the equational attributes `comm assoc id:null`, which allow us to get rid of parentheses and disregard the ordering among `XML` nodes within the list. The significance of this optimization will be clear when we consider rewriting XML trees with AC pattern matching.

AC pattern matching

The evaluation mechanism of Maude is based on rewriting modulo an equational theory E (i.e., a set of equational axioms), which is accomplished by performing *pattern matching modulo* the equational theory E. More precisely, given an equational theory E, a term t and a term u, we say that t *matches* u *modulo* E (or that t *E-matches* u) if there is a substitution σ such that $\sigma(t) =_E u$, that is, $\sigma(t)$ and u are equal modulo the equational theory E. When E contains axioms for expressing the associativity and commutativity of some operators, we instead use *AC pattern matching*. AC pattern matching is a powerful matching mechanism, which we employ to inspect and extract the partial structure of a term. In particular, we use it directly to implement the notion of homeomorphic embedding of Definition 1.3.1.

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 in $\mathcal{U}$ (as a data) any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself), and then mimick in $\mathcal{U}$ the behavior of $\mathcal{R}$. We have used the metaprogramming capabilities of Maude to implement the semantics of correctness as well as completeness rules (e.g., implementing the homeomorphic embedding, evaluating conditions of conditional rules, etc.). Namely, during the partial rewriting process, functional modules are dynamically created and run by using the meta-reduction facilities of the language.

Now we are ready to explain how we implemented the homeomorphic embedding relation of Section 1.3, by exploiting the aforementioned Maude high-level features.

4.1.1 Homeomorphic Embedding Implementation

Let us consider two XML document templates l and p. The critical point of our methodology is to (i) discover whether $l \trianglelefteq p$ (i.e., l is embedded into p); (ii) find the substitution σ such that $l\sigma$ is the instance of l recognized inside p, whenever $l \trianglelefteq p$.

Given l and p, our proposed solution can be summarized as follows. By using Maude metalevel feactures, we first dynamically build a module `M` that contains a single rule of the form

$$\texttt{eq l} = \texttt{sub}(\text{"}\texttt{X}_1\text{"}/\texttt{X}_1), \ldots, \texttt{sub}(\text{"}\texttt{X}_\texttt{n}\text{"}/\texttt{X}_\texttt{n}), \qquad \texttt{X}_\texttt{i} \in \texttt{Var(l)}, \texttt{i} = \texttt{1}, \ldots \texttt{n},$$

where `sub` is an associative operator used to record the substitution σ that we want to compute. Next, we try to reduce the XML template p by using such a rule. Since l and p are internally represented by means of the binary constructor `_,_` with the attributes `comm assoc id:null` (see Section 4.1), the execution of module `M` on p essentially boils down to computing an AC-matcher between l and p. Moreover, since AC pattern matching directly implements the homeomorphic embedding relation, the execution of `M` corresponds to finding all the homeomorphic embeddings of l into p (recall that the set of AC matchers of two compatible terms is not generally a singleton). Additionally, as a side effect of the execution

of M, we obtain the computed substitution σ for free as the collection of bindings for the variables $\mathtt{X_i}$, $\mathtt{i} = 1, \ldots, \mathtt{n}$ which occur in the instantiated rhs

$$\mathtt{sub}(\text{"}\mathtt{X_1}\text{"}/\mathtt{X_1})\sigma, \ldots, \mathtt{sub}(\text{"}\mathtt{X_n}\text{"}/\mathtt{X_n})\sigma, \qquad \mathtt{X_i} \in \mathtt{Var(l)}, \mathtt{i} = 1, \ldots \mathtt{n},$$

of the dynamic rule after the partial rewriting step.

Example 4.1.1

Consider the following XML document templates (called s_1 and s_2, respectively):

$$hpage(surname(Y), status(prof), name(X), teaching)$$

$$\begin{aligned} hpage(\quad & name(mario), surname(rossi), status(prof), \\ & teaching(course(logic1), course(logic2)) \\ & hobbies(hobby(reading), hobby(gardening))) \end{aligned}$$

Note that $s_1 \trianglelefteq s_2$, since the structure of s_1 can be recognized inside the structure of s_2, while $s_2 \not\trianglelefteq s_1$.

We build the dynamic module M containing the rule

$$\begin{aligned} \mathtt{op\ hpage(surname(Y), status(prof), name(X), teaching)} \\ = \mathtt{sub}(\text{"}\mathtt{Y}\text{"}/\mathtt{Y}), \mathtt{sub}(\text{"}\mathtt{X}\text{"}/\mathtt{X})\ . \end{aligned}$$

Since $s_1 \trianglelefteq s_2$, there exists an AC-match between s_1 and s_2 and, hence, the result of executing M against the (ground) XML document template s_2 is the computed substitution: $\mathtt{sub}(\text{"}\mathtt{Y}\text{"}/\mathtt{rossi}), \mathtt{sub}(\text{"}\mathtt{X}\text{"}/\mathtt{mario})$.

4.2 Prototype Implementation

The verification system has been structured as a SOAP Web Service [GHM+07]. The main reason behind this choice is the advantage of using the Service Oriented Architecture paradigm [IBM07]. In this paradigm, services are distributed, autonomous, and independent. They are realized using standard protocols, in order to build networks of collaborating applications. This type of architecture allows one the reuse of services at the macro level, rather than at the micro level (object).

As a service-oriented architecture, WebVerdi-M allows one to access the core verification engine Verdi-M as a reusable entity. This implementation is public available at http://www.dsic.upv.es/users/elp/webverdi-m.

WebVerdi-M is structured in two layers: front-end and back-end. The back-end layer provides web services that support the front-end layer. This architecture allow clients on the network to invoke the Web service functionality through the available interfaces.

The tool consists of the following components: Web service WebVerdiService, Web client WebVerdiClient, core engine Verdi-M, XML API, and database DB. Figure 4.1 illustrates the overall architecture of the system.

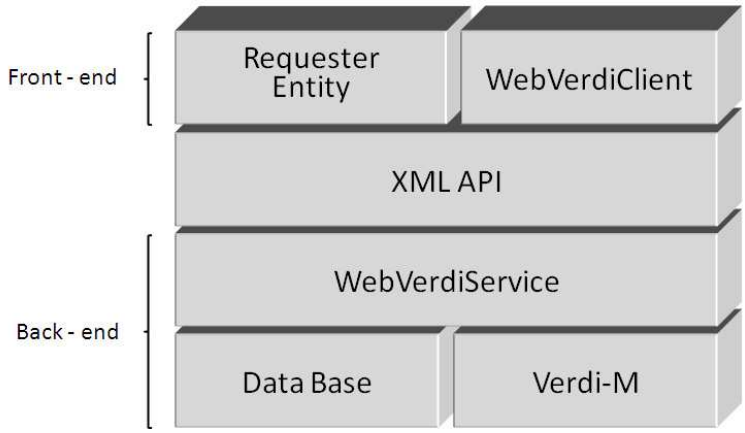


Figure 4.1: Components of WebVerdi-M

4.2.1 The WebVerdiService

The sever has been structured as a Web Service and its control parts are implemented in Java 1.4 [Mic03], which allows us to use the large number of implementations of the Web Service standards available as the TriActive JDO persistence package [Pro05]. Persistence is used to store both web pages and specification locally to the sever in a MySQL database [MyS07].

TriActive JDO is an open source implementation of Sun's JDO specification [Mic06], designed to support transparent persistence using any

JDBC-compliant database. TriActive JDO allows one to generate schemata, meaning it takes user-written Java classes and automates the tasks required to transparently allocate persistent objects into a database.

The web service exports six operations that are network-accessible through standardized XML messaging. These operations are: store a Web site, remove a Web site, retrieve a Web site, add Web page to a Web site, check correctness, and check completeness. The Web service acts as a single access point to the core engine Verdi-M which implements the Web verification methodology in Maude. Following the standards, the architecture is also platform and language independent. It is made accessible via scripting environment as well as via client applications across multiple platforms.

4.2.2 The XML API

In order for successful communications to occur, both the WebVerdiService and WebVerdiClient (or any user) must agree to a common format for the messages being delivered so that they can be properly interpreted at both ends. The WebVerdiService is endowed with an API (see Section 4.3) that encompasses the executable library of the core engine. This is achieved by making use of Apache Axis [Apab], integrated into Apache Tomcat Web server [Apaa]. The Apache Axis handles all procedures needed for the Web service deployment. Synthesized error symptoms are also encoded as XML documents in order to be transferred from the WebVerdiService Web service to client applications as an XML response by means of the SOAP protocol.

4.2.3 The Verdi-M Engine

Verdi-M is the core part of the tool, where the verification and repair methodologies are implemented. This component is implemented in Maude and kept independent of the other system components. The systems was first described in [ABE+07]. The modules of verification and repair are invoked by different Java process when they are required. Overall, the flow execution is under the Java control.

4.2.4 The WebVerdiClient

The client consist of a Java graphical user interface which allows one to use the services (functionalities) offered by the Web Server. The client uses the API specified in Section 4.3 to interact with the server through a network connection. In order to provide an easy, one-click activation of the client, the Java Web Start framework is used. The main goal was to provide a versatile and friendly user interface for the system.

The graphical interface offers three complementary views for both the specification rules and the pages of the considered Web site: the first one is based on the typical idea of accessing contents by using folders trees and is particularly useful for beginners; the second one is based on XML, and the third one is based on term algebra syntax. The tool provides all possible translations among the three views. A snapshot of the WebVerdiClient is shown in Figure 4.2. Any other client using the API of the Web Server could be employed as well.

4.2.5 The Database

The WebVerdiService Web service needs to transmit abundant XML data over the Web to and from client applications. The tool allows users to modify the default rules provided for every Web specification and then verify a particular Web site. After parameterizing the Web specification, it is necessary to send back to the service the considered specification as well as the whole Web site to verify. After the application invokes the WebVerdiService with these two inputs, the synthesized errors are progressively generated and transferred to the client application. The standard Web service architecture would require client applications to wait until all data are received and then errors are sent, which could cause significant time lags. In order to avoid this overhead and improve performance, we introduced a local MySQL data base where the Web site and Web errors are temporarily stored at the server side.

4.3 The API

This section summarizes the categories of methods and the specific message exchange patterns that are considered for interacting with WebVerdi-

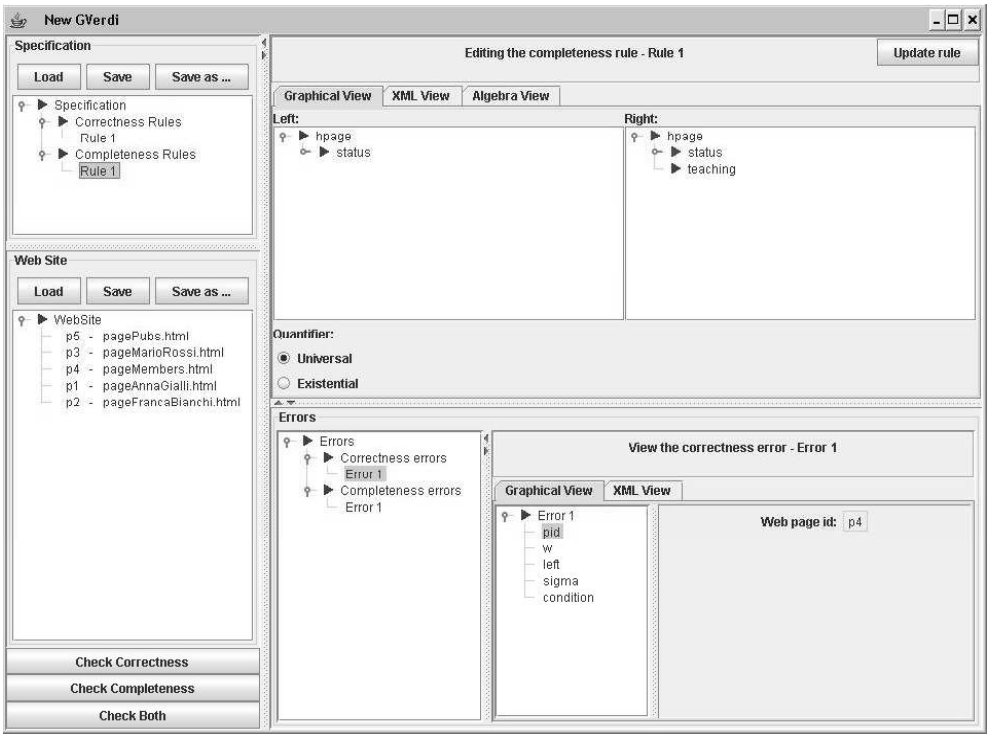


Figure 4.2: WebVerdiClient Snapshot

M.

We considerer the data and method representation separately.

4.3.1 Data Representation

Web Sites

A website consist of a set of pages, which are represented as a triple (id,name,data) where:

- **id:** is the page identifier;
- **name:** is the name of the page;
- **data:** is the XML content of the page.

An example of page is given in Example 4.3.1.

```
<webSite>
   <page> ... </page>
   <page> ... </page>
   <page> ... </page>
   ...
   <page> ... </page>
</webSite >
```

Figure 4.3: XML encoding of a Web site

Example 4.3.1
XML representation of a Web page:

```
<page>
   <id>p1</id>
   <name>biblio.htm</name>
   <data>
      <biblioteca>
         <libro ndoc="99231">
            <isbn>8437607000</isbn>
            <autor>Rojas, Fernando de</autor>
            <titulo>La Celestina</titulo>
         </libro>
         <libro ndoc="158290">
            <isbn>8403870485</isbn>
            <autor>Homero</autor>
            <titulo>Iliada</titulo>
         </libro>
         <libro ndoc="181227">
            <isbn>8466401040</isbn>
            <autor>Kafka, Franz</autor>
            <titulo>La metamorfosi</titulo>
         </libro>
      </biblioteca>
   </data>
</page>
```

A Web Site is represented by a collection of web pages. The XML encoding of a typical Web Site is shown in Figure 4.3.

Rules

In our methodologies, there are two different kinds of rules, which are handled differently. Different XML representations are therefore needed.

- **Correctness rules.** A correctness rule is defined by the triple (l,r,C), where:

  - **l:** is the left-hand side of the rule;
  - **r:** is the right-hand side of the rule;
  - **C:** is the condition (if any).

  The XML representation of a correctness rule is as follows.

```
<ruleCorrectness Name="...">
   <left> ... </left>
   <right> ... </right>
   <condition> ... </condition>
</ruleCorrectness>
```

- **Completeness rules.** A completeness rule is encoded by the triple (l,r,q), where:

  - **l:** is the left-hand side of the rule;
  - **r:** is the right-hand side of the rule;
  - **q** is a logical quantifier, which can be either E (Existential) o A (Universal).

  An attribute is added to help identify a particular rule inside the Web specification. The XML representation of completeness rules is as follows

```
<ruleCompletness Name="Rule 1">
   <left>
      <atrib>
         <f>
            <atrib>
               <g>
                  <atrib>X</atrib>
               </g>
            </atrib>
         </f>
      </atrib>
   </left>
   <right>
      <atrib Mark=true>
         <h>
            <atrib Mark=true>
               <g>
                  <atrib>X</atrib>
               </g>
            </atrib>
         </h>
      </atrib>
   </rigth>
   <quantifier>E</quantifier>
</ruleCompletness>
```

A Web specification is a collection of completeness and/or correctness rules. Its XML representation is sketched in Figure 4.4.

```
<specification>
   <ruleCorrectness Name=xxx> ... </ruleCorrectness>
   <ruleCompletness Name=xxx> ... </ruleCompletness>
   <ruleCompletness Name=xxx> ... </ruleCompletness>
   ...
   <ruleCorrectness Name=xxx> ... </ruleCorrectness>
</specification>
```

Figure 4.4: XML representation for a Web specification

Errors

Correction errors are given by the tuple (pid,w,l,sigma,C), where:

- **pid:** is the identifier of the page that contains errors;
- **w:** is the position within the page where the error is located; this position is defined by an array of integers like [1, 4, 5]; in XML this is written "1.4.5";
- **l:** the left-hand side of the rule that produces the error;
- **sigma:** the substitution(s) in `l` that produce(s) the error;
- **C:** the condition of the rule that produces the error.

The completeness error representation depends on the type of error: missing page error or universal/existential error.

- Missing page errors are defined by the triple (r,W,sigma):
  - **r:** rule which generates the error;
  - **W:** Web Site;
  - **sigma:** the substitution(s) which produce(s) the error.
- Universal/Existential errors are defined by the triple (r,P,sigma):
  - **r:** rule which generates the error;
  - **P:** the set of identifiers of pages which do not comply with the considered rule;
  - **sigma:** the substitution(s) which produce(s) the error.

```
<errorCorrectness>
   <pid>p1</pid>
   <w>1.2</w>
   <l>
      <autor>X</autor>
   </l>
   <sigma>
      <sust>
          <var>X</var>
          <value>Rojas, Fernando de</value>
      </sust>
   </sigma>
   <condition>X=Rojas, Fernando de</condition>
</errorCorrectness>
```

Figure 4.5: XML error representation

```
<errorCompletness>
   <r> ... </r>
   <pages>
      <pid>p1</pid>
      <pid> ... </pid>
      <pid>pn</pid>
   </pages>
   <sigma> ... </sigma>
   <type> ... </type>
</errorCompletness>
```

Figure 4.6: Completeness error representation

An attribute is needed to distinguish among the different types of errors. The values of this attribute can be either M (Missing Page), A (Universal), or E (Existential). Its XML representation is given in Figure 4.6.

Actions

An action is the primitive carried out in order to repair the website. There are four different primitives:

- **change (pid,w,t):** changes the subterm in the position **w** of the page **pid** with the term **t**;
- **insert(pid,w,t):** adds term **t** in the position **w** of the page **pid**;
- **delete(pid,t):** deletes the term **t** from the page **pid**;
- **add(p, idWS):** adds the page **p** to the website **idWS**.

```
<action>
   <type>change</type>
   <pid>p1</pid>
   <w>1.2</w>
   <t>
      <autor>Perez, Pepito</autor>
   </t>
</action>
```

Figure 4.7: XML action representation

4.3.2 Methods Exported

To optimize the transfer of data, some methods are provided as defined below (storeWebSite, retrievePage, retrieveWebSite and removeWebSite).

Descriptions of Methods

storeWebSite (WebSite): Stores a Web Site in the local client server.
Input: WebSite: The Web Site to be stored.
Output: The identifier of the Web Site stored in the local server.

retrievePage(idP, idWS): Loads a page of the specified Web Site.
Input: idP, the identifier of the page; idWS, the identifier of the Web Site stored in the local server.
Output: The retrieved page of the website, if the identifiers exists.

retrieveWebSite(idWS): Loads the Web Site stored in the local server.
Input: idWS, the identifier of the website as stored in the local server.
Output: The Web Site, if the identifiers exist.

removeWebSite(idWS): Deletes the Web Site stored in the local server.
Input: idWS, the Web Site Identifier stored in the local server.
Output: True if the website has been successfully deleted; False otherwise.

checkCorrectness(idWS SPEC): Returns a collection of correctness errors of a Web Site, w.r.t. the given specification.
Input: idWS, the identifier of the Web Site as stored in the local server; SPEC, the XML representation of the Web specification.
Output: An XML encoding of a collection of correctness errors.

checkCompletness(idWS SPEC): Returns a collection of completeness errors of a Web Site, w.r.t. the given specification.
Input: idWS, the identifier of the Web Site as stored in the local server; SPEC, the XML representation of the Web specification.
Output: An XML encoding collection of a completeness errors.

fixErrorCorrectnessByDelete (errorCorrecness idWS): Repairs a completeness error by a **delete** action.
Input: errorCorrecness, error to be fixed; idWS, the identifier of the Web Site as stored in the local server.
Output: True in case the error was successfully deleted; False otherwise.

changeCS (errorCorrecness SPEC idWS): Repairs automatically the correctness error by invoking a constraint solver.
Input: errorCorrecness, error to be fixed; SPEC, the XML representation of the web specification; idWS, the identifier of the Web Site as stored in the local server.
Output: True in case the error was successfully solved; False otherwise.

fixErrorCompleteness (pairErrorAction SPEC idWS): Repairs a completeness error.
Input: pairErrorAction, pair error-action; SPEC, the XML representation of the Web specification; idWS, the identifier of the Web Site as stored in the local server.
Output: True in case the error can be fixed; False otherwise.

fixErrorCompletenessByStrategyM (idWS pairEA_Set): Repairs a completeness error using a Minimal strategy.
Input: idWS, the identifier of the Web Site as stored in the local server; pairEA_Set, the set of pairs $< error, action >$.
Output: True in case the error can be fixed; False otherwise.

fixErrorCompletenessByStrategyMNO (idWS pairEA_Set): Repairs a completeness error using a Minimal non-overlapping strategy.
Input: idWS, the identifier of the Web Site as stored in the local server; pairEA_Set, the set of pairs $< error, action >$.
Output: True in case the error can be fixed; False otherwise.

4.4 Experimental Evaluation

In order to evaluate the usefulness of our approach in a realistic scenario (that is, for Web sites with a big volume of data), we have benchmarked our system by using a repository of correctness as well as completeness rules of different complexity for a number of XML documents randomly generated by using the XML documents generator xmlgen (available within the XMark project [SWK+02]). The tool xmlgen is able to produce a set of XML data, each of which is intended to challenge a particular primitive of XML processors or storage engines by using different scale factors.

Table 4.1 shows some of the results we obtained for the simulation of three different Web specifications WS_1, WS_2 and WS_3 in five different, randomly generated XML documents. Specifically, we tuned the generator for scaling factors from 0.01 to 0.1 to match an XML document whose size ranges from 1Mb –corresponding to an XML tree of about 31000 nodes– to 10Mb –corresponding to an XML tree of about 302000 nodes– (an exhaustive evaluation, including comparison with related systems can be found in `http://www.dsic.upv.es/users/elp/webverdi-m/`).

Both Web specifications WS_1 and WS_2 aim at checking the verification power of our tool regarding data correctness, and thus include only correctness rules. The specification rules of WS_2 contain more complex and more demanding constraints than the ones formalized in WS_1, with involved error patterns to match, and conditional rules with a number of membership tests and functions evaluation. The Web specification WS_3 aims at checking the completeness of the randomly generated XML documents. In this case, some critical completeness rules have been formalized which recognize a significant amount of missing information.

The results shown in Table 4.1 were obtained on a personal computer equipped with 1Gb of RAM memory, 40Gb hard disk and a Pentium Centrino CPU clocked at 1.75 GHz running Ubuntu Linux 5.10.

Let us briefly comment our results. Regarding the verification of correctness, the implementation is extremely time efficient, with elapsed times scaling lineary. Table 4.1 shows that the execution times are small even for very large documents (e.g., running the correctness rules of Web specification $WS1$ over a 10Mb XML document with 302000 nodes takes less than 13 seconds). Concerning the completeness verification, the fix-

Size	Nodes	Scale factor	Time		
			WS_1	WS_2	WS_3
1 Mb	30,985	0.01	0.930 s	0.969 s	165.578 s
3 Mb	90,528	0.03	2.604 s	2.842 s	1768.747 s
5 Mb	150,528	0.05	5.975 s	5.949 s	4712.157 s
8 Mb	241,824	0.08	8.608 s	9.422 s	12503.454 s
10 Mb	301,656	0.10	12.458 s	12.642 s	21208.494 s

Table 4.1: Verdi-M Benchmarks

point computation which is involved in the evaluation of the completeness rules typically burdens the expected performance (see [ABF06]), and we are currently able to process efficiently XML documents whose size is not bigger than 1Mb (running the completeness rules of Web specification $WS3$ over a 1Mb XML document with 31000 nodes takes less than 3 minutes).

Finally, we want to point out that the current Maude implementation of the verification system supersedes and greatly improves our preliminary system, called GVerdi[ABF06; BV05], that was only able to manage correctness for small XML repositories (of about 1Mb) within reasonable time.

Chapter 5

An Abstract Generic Framework for Web Site Verification

This chapter formalizes an abstract framework for Web site verification which improves the performance of our previous, rewriting-based Web verification methodology. The approximated framework is formalized as a source-to-source transformation which is parametric with respect to the chosen abstraction. This transformation significantly reduces the size of the Web documents by dropping or merging contents that do not influence the properties to be checked. This allows us to reuse all verification facilities of the previous system WebVerdi-M to efficiently analyze Web sites. In order to ensure that the verified properties are not affected by the abstraction, we characterize the conditions that allow us to ensure the correctness of the abstraction. An experimental implementation shows a huge speedup with respect to the previous methodology which did not use this transformation.

5.1 Introduction

The basic idea of abstract interpretation [CC77; CC79] is to infer information from programs by interpreting ("running") them using abstract data rather than concrete ones, thus obtaining safe approximations of the programs. The "concrete" data and operators are replaced by corresponding "abstract" (approximated) data and operators. The "answers" obtained by running the program in the abstract domain are proven sound by exploiting the correspondence between the abstract and concrete domains.

5.1.1 Web Site Description

In order to describe Web sites, in this chapter we use the formulation given in [Luc05], which considers the hyperlinks to surf among Web pages. This will allows us to deal later with dynamic Web pages that can be generated from a database by a Web script.

We use an alphabet $\mathcal{P}$ to give names to Web pages and to express the different transitions between pages.

Definition 5.1.1 (immediate successors) *The immediate successors relation for a given Web page p is defined by*

$$\rightarrow_p = \{(p, p') \subseteq \mathcal{P} \times \mathcal{P} \mid p' \text{ is directly accessible from } p\}$$

Definition 5.1.1 establishes a relationship between the page p and its immediate successors (i.e., the pages $p_1, \ldots, p_n$ that p points to by means of hyperlinks). We will use the associated computational relations $\rightarrow_{\mathcal{P}}$, $\rightarrow^+_{\mathcal{P}}$, etc., to describe the dynamic behavior of a Web site. For instance, the reachability of a given Web page p' from another page p can be expressed as $p \rightarrow^*_{\mathcal{P}} p'$.

Definition 5.1.2 (Web site) *A Web site is defined as a set of reachable Web pages from an initial Web page, and is denoted by*

$$W = \{p_1, \ldots, p_n\}, \ \ s.t. \ \ \exists i, 1 \leq i \leq n, \ \forall j, 1 \leq j \leq n, \ p_i \rightarrow^*_W p_j$$

Definition 5.1.2 formalizes the idea that a Web site has an initial Web page which allows one to visit the whole Web site. Note that there may exist several initial Web pages of a given Web site.

Example 5.1.3

The algebraic description of a simple Web site modeling an on-line auction system along with its Web specification are shown in Figure 5.1. The Web site contains information regarding open and closed auctions, and auctioned items. The Web specification contains three rules. The first rule is intended for auditing Web site contents and requires that, in an open auction, the reserve price (or the lower price at which a seller is willing to sell an item) is greater than the initial one. The second rule

Web site $W = \{p_1, p_2, p_3\}$, where

p_1) list-items(
item(id(ite0),
name(racket),state(sold),
description(Wilson tennis racket),
incategories(category(cat1))),
item(id(ite1),
name(shirt),state(available),
description(men's t-shirts),
incategories(category(cat1),
category(cat2))),
item(id(ite2),
name(shoes),state(available),
description(women's shoes),
incategories(category(cat0),
category(cat2))))

p_2) list-categories(
pack(category(cat0)),
unit(category(cat1),
category(cat2)))

p_3) open-auctions(
open-auction(item(ite2),
initial(48.51), reserve(64.3),
seller(Bob), bidder(Alice)))

p_4) closed-auctions(
closed-auction(
seller(Bob), buyer(John),
item(ite0), price(77.5)))

Web specification (I_N, I_M, R), where $I_N = \{r_1\}$, $I_M = \{r_2, r_3\}$, and
r_1) open-auction(initial(X),reserve(Y)) $\rightharpoonup$ error | X > Y
r_2) list-items(item(incategories(X,Y))) $\rightharpoonup$ ♯list-categories(pack(X),unit(Y)) $\langle E \rangle$
r_3) list-item(item(id(X),state(sold)) $\rightharpoonup$ ♯closed-auction(item(X)) $\langle E \rangle$

Figure 5.1: Web site and Web specification for an on-line auction system.

also states an existential property: if there is an auctioned item that is listed in two or more categories, then at least two of these categories must be "unit" and "pack". The last rule states that, for every item that is sold, a closed auction associated to the item must exist.

5.2 Abstract Web Site Verification

In this chapter, we are particularly interested in the abstraction of the completeness process, because the fixpoint computation needed for the completeness verification leads to unsatisfactory performance (see the last column of Table 4.1 in page 82).

We want to formalize the abstraction as a source–to–source transformation which translates Web documents and Web specification rules into constructions of the very same languages, hence our concrete and abstract domains do coincide. In this way, the domain of abstract terms $\mathcal{D}^\alpha$ is equal to the domain of concrete terms $\mathcal{D}$. Let us first introduce the notion of abstract domain.

Definition 5.2.1 (abstract non-ground term algebra, poset) *Let $\mathcal{D} = (\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}), \leq)$ be the standard domain of (equivalence classes of) terms, ordered by the standard partial order $\leq$ induced by the preorder on terms given by the relation of being "more general".*

Then the domain of abstract terms $\mathcal{D}^\alpha$ is equal to $\mathcal{D}$.

We define the abstraction (t^α) of a term t as: $t^\alpha = \alpha(t)$. Our framework is parametric w.r.t. the abstraction function α, which can be used to tune the accuracy of the approximation. For example, for on–line auctioning, a convenient abstraction function would be better defined as to distinguish registered bidders and sellers, and auctioned items.

Let us introduce the definition of term abstraction α.

Definition 5.2.2 (term abstraction α) *Let $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$ be a text abstraction function.*

$$\begin{aligned} &\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \\ &\alpha(t) = \hat{\alpha}(\epsilon, t) \end{aligned}$$

where the auxiliary function $\hat{\alpha}$ is given by

$$\begin{aligned} \hat{\alpha} :: \mathcal{T}ag^* \times \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) &\to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \\ \hat{\alpha}(\_, x) &= x, \ \textit{if } x \in \mathcal{V} \\ \hat{\alpha}(c, f(t_1, .., t_n)) &= f(\hat{\alpha}(c.f, t_1), .., \hat{\alpha}(c.f, t_n)), \ \textit{if } f \in \mathcal{T}ag \\ \hat{\alpha}(c, w) &= atext(c, w), \ \textit{if } w \in \mathcal{T}ext \end{aligned}$$

The reader may notice that elements of $\mathcal{T}ext$ are abstracted by taking into account the chain of *tags* under which a particular piece of text appears. This is formalized by means of the text abstraction function

$$atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$$

which is left undefined and is actually the formal parameter of the definition.

The text abstraction function should be conveniently fixed in order to tune the abstraction for each particular domain. For instance, in the case where no $\mathcal{T}ag$ distinction is needed, each element in $\mathcal{T}ext$ could be simply replaced by some (abstract) fresh, constant symbol d.

Example 5.2.3

Consider the Web page p_2 of Example 5.1.3. By fixing

$$atext(\_, w) = \mathit{first}(w), \text{ where } \mathit{first}(x.xs) = x,$$

the resulting abstraction of p_2 is

$$\alpha(p_2) = \mathit{list\text{-}categories}(\mathit{pack}(\mathit{category}(c)), \\ \mathit{unit}(\mathit{category}(c), \mathit{category}(c)))$$

Note that this abstraction function does not distinguish among some leaves in the term that do influence the properties to be verified. As a consequence of this lack of precision, by using this abstraction we would not be able to observe the constraint given by the completeness rule r_2 anymore. Specifically, the rule r_2 states that if there is an auctioned item that is listed in two or more categories, then at least two of these categories must be "unit" and "pack". Clearly, the second item in the Web page p_1 does not satisfy this constraint.

In order to achieve correctness of the abstraction, we restrict our interest to text abstraction functions *atext* which distinguish those pieces of text that are observed by the Web specification rules and then potentially affect the result of the verification.

The auxiliary function $gen\_emb\_tt_{\mathcal{I}}(c, t)$ allows us to know whether a sequence of tags c (with leaf t) is recognized within some rule of the Web specification $\mathcal{I}$. This allows us to determine whether a term within a given Web page needs to be carefully considered.

Text abstraction functions are required to obey the following correctness condition w.r.t. W.

Definition 5.2.4 (correctness condition w.r.t. W) *Let W be a Web site, and let $\mathcal{I}$ be a Web specification. Let $s, t \in \mathcal{T}ext$ be any two pieces of text in W. For every $c \in Tag^*$ such that $gen\_emb\_tt_{\mathcal{I}}(c, s) \equiv True$ and $gen\_emb\_tt_{\mathcal{I}}(c, t) \equiv True$, the text abstraction function atext satisfies*

$$s \not\equiv t \Rightarrow \ atext(c, s) \not\equiv atext(c, t)$$

The condition above formalizes the idea that, whenever two pieces of text are indistinguishable in the abstract domain, then they are also indistinguishable in the concrete domain.

5.2.1 Abstract Web Specification

The abstraction of the correctness and completeness Web specification rules is simply based on abstracting the terms occurring in the left-hand side and the right-hand side of the rules. In particular, the conditional parts of the correctness rules are not abstracted, and hence we let concrete conditions to be applied to concrete data as well as to abstract data.

Definition 5.2.5 (abstract specification rule) *Let*

$$\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$$

be a term abstraction function with text abstraction function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$. *Let* $rl_M \equiv l \rightharpoonup r\langle q \rangle$ *be a completeness rule, and let* $rl_N \equiv l \rightharpoonup error|C$ *be a correctness rule. We denote by* rl_M^α *(resp.* rl_N^α*) the abstraction of* rl_M *(resp.* rl_N*), where* $rl_M^\alpha \equiv \alpha(l) \rightharpoonup \alpha(r)\langle q \rangle$ *(resp.* $rl_N^\alpha \equiv \alpha(l) \rightharpoonup error|C$*).*

Example 5.2.6

Consider the completeness rule r_3 of Example 5.1.3. By fixing the text abstraction function $atext(c, x) = last(x)$ where $last(w)$ returns the last element of the sequence w, the computed abstract completeness rule $\alpha(r_3)$ is

$$\mathit{list\text{-}item}(\mathit{item}(\mathit{id}(X), \mathit{state}(d)) \rightharpoonup \sharp\mathit{closed\text{-}auction}(\mathit{item}(X))\langle E \rangle$$

Roughly speaking, the abstraction of a rewrite rule consists of abstracting the two terms. When no confusion can arise, we just write $rl_M^\alpha \equiv l^\alpha \rightharpoonup r^\alpha\langle q \rangle$ (resp. $rl_N^\alpha \equiv l^\alpha \rightharpoonup error|C$). The Web specification (I_N, I_M, R) is lifted to $(I_N^\alpha, I_M^\alpha, R)$ element–wise.

To ensure the soundness of the abstract framework, we need to precisely relate the satisfiability of the conditions over abstract and concrete data. Specifically, we require the fulfillment of an abstract condition to imply the fulfillment of the corresponding concrete description.

Definition 5.2.7 (correctness condition w.r.t. I_N) *Let* $\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ *be a term abstraction function with text abstraction function* $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$. *Let* $rl \equiv l \rightharpoonup error|C$ *be*

a correctness rule. Then, α *is* correct *w.r.t. rl iff, for each substitution* $\sigma \equiv \{X_1/t_1, \ldots, X_n/t_n\}$,

$$C\sigma^{\alpha} \text{ holds} \quad \Rightarrow \quad C\sigma \text{ holds}$$

where $\sigma^{\alpha} \equiv \{X_1/\alpha(t_1), \ldots, X_n/\alpha(t_n)\}$. *Moreover,* α *is* correct *w.r.t.* I_N *if it is correct w.r.t. every correctness rule of* I_N.

5.2.2 Abstract Web Site

When navigating a Web site, it is common to find a number of pages that have a similar structure but different contents. This happens very often when pages are dynamically generated by some script which extracts contents from a database (e.g., in Amazon's Web site). This can make our simple analysis impracticable unless we are able to provide a mechanism to drastically reduce the Web size. In order to ensure that the verified properties are not affected by the abstraction, in this section we develop an abstract methodology which derives an approximation of Web sites from the considered Web specifications.

Let us introduce a compression function for terms which reduces the size of each singular Web page by dropping some arguments, thus reducing the number of branches of the tree. This is used as a preprocess prior to the abstraction of a given Web site.

Web Compression pre–processing

Let (I_N, I_M, R) be a Web specification and $s, t \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$. We define two auxiliary functions $join$ and max_ar. The function $join(s,t)$ returns the term that is obtained by concatenating the arguments of terms s and t (if they exist), whenever $root(s) = root(t)$, e.g., $join(f(a,b,c), f(b,e)) = f(a,b,c,b,e)$. The function $max\_ar(f, I_M)$ returns the maximal arity of f in I_M.

Definition 5.2.8 (correctness condition w.r.t. I_M**)** *Let* (I_N, I_M, R) *be a Web specification. Then, the term* $f(t_1, ..., t_n) \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ *is compressed by using function* COMPRESS *given in Algorithm 4, which packs together those subterms which are rooted by the same root symbol while ensuring that the arity of* f *after the transformation is not smaller than* $max\_ar(f, I_M)$.

Algorithm 4 Term Compression Transformation.

Input:
Term $t = f(t_1, \ldots, t_n)$
I_M a set of completeness rules
Output:
Term $f(t'_1, \ldots, t'_m)$, with $m \leq n$

function COMPRESS (t, I_M)
  if $n = 0$ **then**
    $\leftarrow f$
  else if $max\_ar(f, I_M) < n$ **and**
      $\exists\, i, j$ s.t. $root(t_i) = root(t_j)$ **then**
    $t' \leftarrow join(t_i, t_j)$
    $\leftarrow$ COMPRESS$(f(t_1, \ldots, t_{i-1}, t', t_{i+1}, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n), I_M)$
  else
    $\leftarrow f($COMPRESS$(t_1, I_M), .., $COMPRESS$(t_n, I_M))$
  end if
end function

The idea behind Definition 5.2.8 is as follows. Roughly speaking, all arguments with the same root symbol f ocurring at level i are joined together. Then, compression recursively proceeds to level $(i + 1)$. The condition that the maximal arity of f in I_M must be respected is essential for the correctness of our method, as this condition ensures that a partial rewrite step on an abstract term is always enabled, whenever the corresponding partial rewrite step can be executed in the concrete domain. Let us see an example.

Example 5.2.9

Consider the Web page p_1 and the completeness rule r_2 of Example 5.1.3. The left–hand side of rule r_2 is embedded in p_1, in symbols

$$\textit{list-items}(\textit{item}(\textit{incategories}(X, Y))) \trianglelefteq p_1$$

If we naïvely compressed p_1 without respecting the maximal arity in I_M of function symbol "*incategories*", we would get

p'_1 = list-items(
item(id(ite0), name(racket),state(sold),
description(Wilson tennis racket),
incategories(category(cat1))),
item(id(ite0), name(shirt),state(available),
description(men's t-shirts),
incategories(category(cat1,cat2))),
item(id(ite2), name(shoes),state(sold),
description(women's shoes),
incategories(category(cat0,cat2))))

Unfortunately,

$$\textit{list-items}(\textit{item}(\textit{incategories}(X, Y))) \not\trianglelefteq p'_1$$

since in p'_1 the arity of the symbol "*incategories*" is lower than in the lhs of r_2.

Now we are ready to formalize our notion of Web site approximation.

Web site abstraction

In order to approximate a Web site, we start from an initial Web page and recursively apply the successor relation ($\rightarrow$), while implementing a simple *depth-first* search (DFS) [CLRS01].

Definition 5.2.10 (Abstract Web Site) *Let W be a Web site, let p be an initial page of W, and let (I_N, I_M, R) be a Web specification. Then, the abstraction of W is defined by:*

$$\alpha(W) = \text{DFS}(p, \emptyset, I_M)$$
where function DFS *is given in Algorithm 5.*

Note that, after applying the transformation above, the information in the Web pages as well as the number of pages in the Web site can be significantly reduced.

Algorithm 5 Web site abstraction.

Input:
p :: $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$
W^α :: $set(\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}))$
I_M a set of completeness rules
Output:
W^α = set $(\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}))$
1: **function** DFS (p, W^α, I_M)
2: $p^\alpha \leftarrow \text{COMPRESS}(\alpha(p), I_M)$
3: $W^\alpha \leftarrow W^\alpha \cup \{p^\alpha\}$
4: **for all** i s.t. $(p, p_i) \in \rightarrow_p$ **and**
$\text{COMPRESS}(\alpha(p_i), I_M) \notin W^\alpha$ **do**
5: $W^\alpha \leftarrow \text{DFS}(p_i,\ W^\alpha)$
6: **end for**
7: $\leftarrow W^\alpha$
8: **end function**

5.2.3 Abstract Verification Soundness

The abstraction function given in Definition 5.2.2 defines abstractions by a source-to-source transformation. Due to this source-to-source approximation scheme, all facilities supported by our previous verification system can be straightforwardly adapted and reused with very little effort.

Informally, our abstract verification methodology applies to the considered abstract descriptions of the Web site and Web specification. Given a Web specification (I_N, I_M, R) and a Web site W, we first generate the corresponding abstractions $(I_N^\alpha, I_M^\alpha, R)$ and W^α. Then — since we consider a source to source transformation — we apply our original verification algorithm [ABF06] to analyse W^α w.r.t. $(I_N^\alpha, I_M^\alpha, R)$. We call *abstract* error, each error which is detected in W^α using $(I_N^\alpha, I_M^\alpha, R)$ by the verification methodology.

In order to guarantee the soundness of the abstract diagnosis, we have to ensure that, when fed with the abstracted data, the partial rewriting relation, $\rightharpoonup$, correctly approximates the behavior of the partial rewriting relation over the corresponding concrete representation. In the following, we demonstrate the soundness of our abstract representation. First of all we introduce the notion of *abstract embedding*, which is used to establish a relation between concrete and abstract terms.

Definition 5.2.11 (abstract embedding) *The* abstract embedding *relation*

$$\trianglelefteq^\sharp \subseteq \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \times \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$$

w.r.t. a function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$ *on Web page templates is the least relation satisfying the rules:*

1. $X \trianglelefteq^\sharp t$, *for all* $X \in \mathcal{V}$ *and* $t \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$.
2. $f(t_1, \ldots, t_m) \trianglelefteq^\sharp g(s_1, \ldots, s_n)$ *iff* $f \equiv g$ *and*
   $t_i \trianglelefteq^\sharp s_{\pi(i)}$, *for* $i = 1, \ldots, m$, *and*
   some total function $\pi :: \{1, \ldots, m\} \to \{1, \ldots, n\}$.
3. $c \trianglelefteq^\sharp c'$ *iff* $c' \equiv atext(x, c)$ *for some* $x \in \mathcal{T}ag^*$.

Given $t_1, t_2 \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ such that $t_1 \trianglelefteq^\sharp t_2$ w.r.t. $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$, we say that t_2 safely approximates t_1.

Basically, Definition 5.2.11 slightly modifies Definition 1.3.1 (homeomorphic embedding) by allowing the detection of noninjective embeddings and the renaming of some constants. In other words, two distinct paths in t_1 may be mimicked (i.e., simulated) by a single path appearing in t_2 modulo (a possible) renaming of some leaves of t_1.

Example 5.2.12

Consider the terms $t_1 \equiv f(g(a), g(b))$ and $t_2 \equiv f(g(d, e))$ which are depicted in Figure 5.2. Then, $t_1 \not\trianglelefteq t_2$ and $t_2 \not\trianglelefteq t_1$, but we have $t_1 \trianglelefteq^\sharp t_2$ w.r.t. the function $\{(f.g,\ a) \mapsto c, (f.g\ , b) \mapsto d\}$ as shown in Figure 5.2 by means of dashed arrows. Moreover, note that the two distinct t_1's edges from f to g are represented in t_2 by a single edge from f to g.

By using Definition 5.2.11, we are able to map any concrete path within a concrete term into a path in an abstract term: the structure and the labeling of t are represented in a compressed and suitable relabeled version of t such that many paths of t are mapped into one shared path of the abstract description, as stated by the following proposition.

Proposition 5.2.13 *Let* $t \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. *Let* $\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ *be a term abstraction function using the text abstraction function* $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$. *Then,* $t \trianglelefteq^\sharp \alpha(t)$ *w.r.t.* $atext$.

Proof. (sketch) We proceed by induction on the structure of t.

Case $t \equiv X, X \in \mathcal{V}$**.** By Definition 5.2.11, $X \trianglelefteq^\sharp t'$ for any $t' \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ w.r.t. $atext$. Hence, $X \trianglelefteq^\sharp \alpha(X)$ w.r.t. $atext$.

Case $t \equiv w, w \in \mathcal{T}ext$**.** By Definition 5.2.2, $atext(seq, w) \equiv c$, for some $seq \in \mathcal{T}ag^*$. Hence, $w \trianglelefteq^\sharp \alpha(w)$ w.r.t. $atext$.

Case $t \equiv f(t_1, \ldots, t_n), n > 0$**.** By Definition 5.2.2, we have

$$root(\alpha(t)) \equiv root(t).$$

Hence, $\alpha(t) \equiv f(t'_1, \ldots, t'_m)$. Observe that, whenever $m < n$, some root symbols of the terms $t_1, \ldots, t_n$ have been compressed with the aim of reducing the arity of f. (Poner la definicion de preserva afuera) Note that compression preserves the paths of t. That is, if there exists a path from x to y in t, then there exists a corresponding path in $\alpha(t)$. Thus, each t'_j of $\alpha(t)$ may correspond to many t_i's which are included in t. Therefore, we can define the total (possibly, non-injective) function $\pi :: \{1, \ldots, n\} \to \{1, \ldots, m\}$ in the following way: $\pi(i) = j$, where $root(t_i) \equiv root(t_j)$ and t'_j in $f(t'_1, \ldots, t'_m)$ corresponds to t_i in $f(t_1, \ldots, t_n)$. By using such a definition of π and the inductive hypothesis, we get $t_i \trianglelefteq^\sharp t'_{\pi(i)}$ w.r.t $atext$. Consequently, $t \trianglelefteq^\sharp \alpha(t)$ w.r.t. $atext$.

■

Roughly speaking, Proposition 5.2.13 establishes that $\alpha(t)$ safely approximates the (concrete) term t.

Example 5.2.14

Consider the terms t_1 and t_2 of Figure 5.2. Let $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$ be defined as $\{(f.g, a) \mapsto c, (f.g, b) \mapsto d\}$. Assume that there exists a Web specification in which the maximal arity of g equals to 1, so that the compression of the t_1's nodes labeled with g is enabled. Then, $\alpha(t_1) \equiv t_2$ and $t_1 \trianglelefteq^\sharp \alpha(t_1)$ w.r.t. function $atext$.

Proposition 5.2.13 states that abstract terms simulate the concrete terms through an abstract embedding (i.e., the $\trianglelefteq^\sharp$ relation w.r.t. a suitable renaming function). In the following, we demonstrate that such a

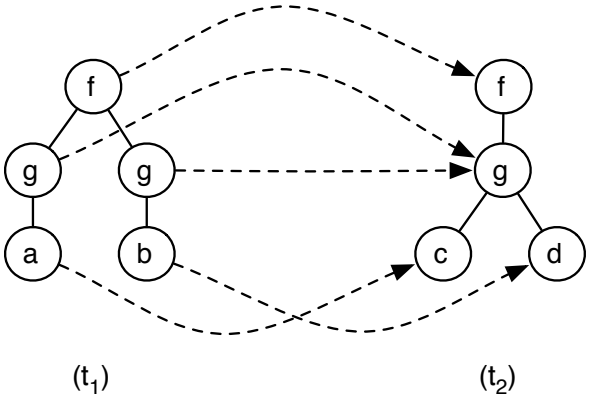


Figure 5.2: Abstract embedding

property is preserved by partial rewriting. In other words, whenever a partial rewrite step $t_1 \rightharpoonup t_2$ is executed in the concrete domain, a partial rewrite step over the abstract counterpart is enabled, in symbols $\alpha(t_1) \rightharpoonup t_2'$, such that t_2 still simulates the obtained abstract term t_2' w.r.t. $\trianglelefteq^\sharp$. The following property holds.

Proposition 5.2.15 *Let $s, t \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. Let $\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ be a term abstraction function with text abstraction function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$. Let $\mathcal{I} \equiv (I_N, I_M, R)$ be a Web specification, and let $\mathcal{I}^\alpha \equiv (I_N^\alpha, I_M^\alpha, R)$ be the abstract version of $\mathcal{I}$. If $s \rightharpoonup_{I_M} t$, then*

- $\alpha(s) \rightharpoonup_{I_M^\alpha} t'$
- $t \trianglelefteq^\sharp t'$ *w.r.t. atext*

Proof. (sketch) Consider $s, t \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ such that $s \rightharpoonup_{I_M} t$ via the rule $l \rightharpoonup r \langle q \rangle \in I_M$, $q \in \{\mathtt{A}, \mathtt{E}\}$. Therefore, $t \equiv r\sigma$ for some substitution $\sigma = \{X_1/s_1, \ldots, X_n/s_n\}$. Now, observe the following two facts.

(i) Given the abstract rule $\alpha(l) \rightharpoonup \alpha(r) \langle q \rangle \in I_M^\alpha$, by applying Definition 5.2.5, we have $\alpha(l) \equiv l$ (resp., $\alpha(r) \equiv r$) modulo renaming of some constants in l (resp. r) via the function *atext*. In particular, by Proposition 5.2.13, $l \trianglelefteq^\sharp \alpha(l)$ w.r.t. *atext* (resp., $r \trianglelefteq^\sharp \alpha(r)$ w.r.t. *atext*).

(ii) Given a term $t \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$, the arity of each symbol f appearing in $\alpha(t)$ is reduced as far as it overcomes the maximal arity of f appearing in I_M (see Definition 5.2.8).

Fact (i) and Fact (ii) imply that if $l \trianglelefteq s_{|w}$, then $\alpha(l) \trianglelefteq \alpha(s)_{|w'}$, $w \in O_{\mathcal{T}ag}(s)$, $w' \in O_{\mathcal{T}ag}(\alpha(s))$. Therefore, $\alpha(s) \rightharpoonup_{I_M^\alpha} t'$ via $\alpha(l) \rightharpoonup \alpha(r)\ \langle q \rangle \in I_M^\alpha$, and $t' \equiv \alpha(r)\sigma^\alpha$, where $\sigma^\alpha = \{X_1/\alpha(s_1), \ldots, X_2/\alpha(s_n)\}$. Now, by Proposition 5.2.13, $s \trianglelefteq^\sharp \alpha(s)$ w.r.t. *atext*, and consequently $s_i \trianglelefteq^\sharp \alpha(s_i)$ w.r.t. *atext*, $i = 1, \ldots, n$. Moreover by Fact (ii), we have $r \trianglelefteq^\sharp \alpha(r)$ w.r.t. *atext*. Hence, $r\sigma \trianglelefteq^\sharp \alpha(r)\sigma^\alpha$ w.r.t. *atext*. Finally,

$$t \equiv r\sigma \trianglelefteq^\sharp \alpha(r)\sigma^\alpha \equiv t' \text{ w.r.t. } atext$$

■

Proposition 5.2.15 can be generalized to partial rewrite sequences using a simple inductive argument. More formally,

Proposition 5.2.16 *Let $\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ be a term abstraction function with text abstraction function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$. Let $\mathcal{I} \equiv (I_N, I_M, R)$ be a Web specification, and let $\mathcal{I}^\alpha \equiv (I_N^\alpha, I_M^\alpha, R)$ be the abstract version of $\mathcal{I}$.*

If $t_0 \rightharpoonup_{I_M} t_1 \rightharpoonup_{I_M} \ldots \rightharpoonup_{I_M} t_n$, $n \geq 0$, then

1. *$\alpha(t_0) \rightharpoonup_{I_M^\alpha} t'_1 \rightharpoonup_{I_M^\alpha} \ldots \rightharpoonup_{I_M^\alpha} t'_n$;*
2. *$t_n \trianglelefteq^\sharp t'_n$ w.r.t. atext*

Proof. We proceed by induction on the length n of the concrete partial rewrite sequence.

Case $n = 0$**.** In this case, we trivially have $t_n \equiv t_0$, hence $t'_n \equiv \alpha(t_n)$ which directly implies claim 1. To prove claim 2, observe that, by Proposition 5.2.13, $t_n \trianglelefteq^\sharp \alpha(t_n) \equiv t'_n$ w.r.t. *atext*.

Case $n > 0$**.** We consider the following concrete partial rewrite sequence

$$t_0 \rightharpoonup_{I_M} t_1 \rightharpoonup_{I_M} \ldots \rightharpoonup_{I_M} t_n, n > 0.$$

By inductive hypothesis, there exists $\alpha(t_0) \rightharpoonup_{I_M^\alpha} t'_1 \rightharpoonup_{I_M^\alpha} \ldots \rightharpoonup_{I_M^\alpha} t'_{n-1}$ such that $t_{n-1} \trianglelefteq^\sharp t'_{n-1}$ w.r.t. *atext*. Since $t_{n-1} \trianglelefteq^\sharp t'_{n-1}$, $t'_{n-1} \equiv \alpha(t_{n-1})$. By Proposition 5.2.15, we obtain $t'_{n-1} \equiv \alpha(t_{n-1}) \rightharpoonup_{I_M^\alpha} t'_n$

such that $t_n \trianglelefteq^\sharp t'_n$. Therefore, by composing the computed abstract partial rewrite sequences, we obtain

$$\alpha(t_0) \rightharpoonup_{I_M^\alpha} t'_1 \rightharpoonup_{I_M^\alpha} \ldots \rightharpoonup_{I_M^\alpha} t'_n$$

with $t_n \trianglelefteq^\sharp t'_n$ w.r.t. *atext*.

■

Given an (abstract) partial rewrite sequence $\mathcal{S}^\alpha \equiv \alpha(t_1) \rightharpoonup_{I_M^\alpha} t'_2 \rightharpoonup_{I_M^\alpha} \ldots \rightharpoonup_{I_M^\alpha} t'_n$, we call abstract completeness requirement any term appearing in $\mathcal{S}^\alpha$. Now, the concrete verification methodology works as follows: first, we compute the concrete completeness requirements and, then, we check whether such requirements are fulfilled in the considered Web site. As explained in Section 1.4, a completeness requirement r is a term which occurs in a partial rewrite sequence of the form $p \equiv t_0 \rightharpoonup t_1 \rightharpoonup t_2 \ldots \rightharpoonup t_n \equiv r$, where $p \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ is a Web page of the Web site W and $r \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ is the computed (completeness) requirement. Our novel abstract methodology exploits Proposition 5.2.16 in order to avoid the computation of concrete requirements, since they are safely approximated by their abstract descriptions.

Therefore, by Proposition 5.2.16, we can directly conclude that each concrete requirement is safely approximated by its abstract description. More formally, the following corollary holds.

Corollary 5.2.17 *Let $\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ be a term abstraction function with text abstraction function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$. Let W be a Web site, and let W^α be the abstract version of W. Let (I_N, I_M, R) be a Web specification, and let $(I_N^\alpha, I_M^\alpha, R)$ be the abstract version of (I_N, I_M, R). If r is a concrete completeness requirement for W computed by (I_N, I_M, R), then there exists an abstract completeness requirement r^α for W^α computed by $(I_N^\alpha, I_M^\alpha, R)$ such that $r \trianglelefteq^\sharp r^\alpha$ w.r.t. atext.*

The fact that any concrete completeness requirement is safely approximated by an abstract completeness requirement ensures that the abstract verification is safe, that is, whenever an abstract requirement is fulfilled in the abstract Web site, each concrete representation is fulfilled in the concrete domain. This allows us to conclude the absence of concrete errors in the case when no abstract errors are detected.

Formally, the following theorem holds.

Theorem 5.2.18 *Let $\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \rightarrow \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ be a term abstraction function with text abstraction function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \rightarrow \mathcal{T}ext$. Let W be a Web site, and let W^α be the abstract version of W. Let $\mathcal{I} \equiv (I_N, I_M, R)$ be a Web specification, and let $\mathcal{I}^\alpha \equiv (I_N^\alpha, I_M^\alpha, R)$ be the abstract version of $\mathcal{I}$. Then, W^α does not contain any abstract (universal/existential) completeness error w.r.t. $\mathcal{I}^\alpha$, then W does not contain any concrete (universal/existential) completeness error w.r.t. $\mathcal{I}$.*

Proof. (sketch) First of all, we show that if $\alpha(t) \trianglelefteq \alpha(p)$, for $t \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ and $p \in W$, then $t \trianglelefteq p$ (*requirement safeness property*). By contradiction, we assume that $t \not\trianglelefteq p$. This amounts to saying that there is a path in t which is not recognized in p. On the other hand, by Proposition 5.2.13, we have $t \trianglelefteq^\sharp \alpha(t)$ w.r.t. *atext*, and hence all the paths in t are simulated (recognized) in $\alpha(t)$. Since $\alpha(t) \trianglelefteq \alpha(p)$, all the paths in t are recognized in $\alpha(p)$. Finally, by Proposition 5.2.13, $p \trianglelefteq^\sharp \alpha(p)$ w.r.t. *atext*, hence all the paths in t are recognized in p, which leads to a contradiction.

Now, we use this result to prove the main theorem in this section. In the following, we will distinguish three cases according to the kind of completeness errors we want to detect.

Existential completeness error. By contradiction, we assume there exists a concrete existential completeness error

$$(r, \{p_1, p_2, \ldots, p_n\}, \mathtt{E})$$

in W w.r.t. $\mathcal{I}$. That is, there exists $p \in W$ such that $p \rightharpoonup_{I_M}^+ r$ and $\{p_1, p_2, \ldots, p_n\}$ is not existentially complete w.r.t. r (i.e. $\forall\ i = 1, \ldots, n, w \in O_{\mathcal{T}ag}(p_i),\ r \not\trianglelefteq p_{i|w}$). By Corollary 5.2.17, $r \trianglelefteq^\sharp \alpha(r)$ w.r.t. *atext*. On the other hand, W^α does not contain any abstract existential completeness error w.r.t. $\mathcal{I}^\alpha$. This implies that any computed abstract requirement r^α is embedded in some $p^\alpha \in W^\alpha$. In particular, the abstract requirement $\alpha(r)$ is embedded in some $\alpha(p_i) \in \{\alpha(p_1), \ldots, \alpha(p_n)\} \subseteq W^\alpha$ (in symbols, $\exists\ i = 1, \ldots, n, w \in O_{\mathcal{T}ag}(\alpha(p_i)),\ \alpha(r) \trianglelefteq \alpha(p)_{i|w}$). By the requirement safeness property, we then derive $r \trianglelefteq p_{i|w'}$, which leads to a contradiction, as we supposed that $\{p_1, p_2, \ldots, p_n\}$ is not existentially complete w.r.t. r.

Missing Web page error. Analogous to the first case.

Universal completeness error. Analogous to the first case.

■

Note that —whenever we detect an abstract completeness error— we are not able to guarantee the presence of a concrete completeness error. This is mainly due to the fact that the abstraction can enable partial rewriting steps over abstract descriptions that are not feasible in the concrete domain. Thus, there might be an abstract requirement that does not correspond to any concrete requirement, as illustrated by the following example.

Example 5.2.19

Consider the following set I_M of completeness rules of a Web specification

$$\begin{array}{ll} r_1) & f(X,Y) \rightharpoonup m \ \langle \mathtt{E} \rangle \\ r_2) & f(a,b) \rightharpoonup m' \ \langle \mathtt{E} \rangle \end{array}$$

and the Web site $W = \{f(f(a), f(b), h(c)), m\}$. Assume that the text abstract function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \to \mathcal{T}ext$ is defined as $atext(\_, t) = t$ for each $t \in \mathcal{T}ext$. Then,

$$W^\alpha = \{f(f(a,b), h(c)), m\}.$$

Moreover, the abstract description of I_M is equal to I_M (i.e., $I_M \equiv I_M^\alpha$). The set of concrete requirements of W w.r.t. I_M is $\{m\}$; while the set of abstract requirements of W^α w.r.t. I_M^α is $\{m, m'\}$. Note that m' is not fulfilled in W^α, as it is not embedded in any abstract page of W^α, that is, m' represents an abstract completeness error. On the other hand, requirement m' cannot be computed in the concrete domain, since rule r_2 cannot be applied to Web pages in W. Consequently, m' is not responsible for any concrete completeness error in W.

The approximation we considered allows us to establish a safe connection between abstract and concrete correctness errors as well. In particular, we are able to ensure that, whenever an abstract correctness error is detected, a corresponding correctness error must exist in the concrete counterpart.

Given a concrete correctness error $e \equiv (p, w, l, \sigma)$, we define $\alpha(e) \equiv (\alpha(p), w^\alpha, \alpha(l), \alpha(\sigma))$, $w^\alpha \in O_{\mathcal{T}ag(\alpha(p))}$.

Theorem 5.2.20 *Let $\alpha :: \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \rightarrow \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ be a term abstraction function with text abstraction function $atext :: \mathcal{T}ag^* \times \mathcal{T}ext \rightarrow \mathcal{T}ext$. Let W be a Web site, and let W^α be the abstract version of W. Let $\mathcal{I} \equiv (I_N, I_M, R)$ be a Web specification such that α is correct w.r.t. I_N, and let $\mathcal{I}^\alpha \equiv (I_N^\alpha, I_M^\alpha, R)$ be the abstract version of $\mathcal{I}$. If W^α contains an abstract correctness error*

$$e^\alpha \equiv (p^\alpha, w^\alpha, l^\alpha, \sigma^\alpha)$$

w.r.t. $\mathcal{I}^\alpha$, then W contains a concrete correctness error $e \equiv (p, w, l, \sigma)$ w.r.t. $\mathcal{I}$ such that $\alpha(e) \equiv e^\alpha$.

Proof. (sketch) By contradiction, we assume there exists no concrete correctness error in the Web site W such that $e^\alpha \equiv \alpha(e)$.

The fact that there exists an abstract correctness error

$$(p^\alpha, w^\alpha, l^\alpha, \sigma^\alpha)$$

in an abstract Web page p^α w.r.t. the abstract rule $rl^\alpha \equiv l^\alpha \rightharpoonup error | C \in I_N^\alpha$ implies that $l^\alpha \trianglelefteq p^\alpha_{|w^\alpha}$, for some $w^\alpha \in O_{\mathcal{T}ag}(p^\alpha)$, and $C\sigma^\alpha$ holds.

Now, by Proposition 5.2.13, we derive (i) $p \trianglelefteq^\sharp p^\alpha \equiv \alpha(p)$ w.r.t. *atext*, and (ii) $l \trianglelefteq^\sharp l^\alpha \equiv \alpha(l)$ w.r.t. *atext*. By (i), (ii), and $l^\alpha \trianglelefteq p^\alpha_{|w^\alpha}$, we conclude that $l \trianglelefteq p_{|w}$, $w \in O_{\mathcal{T}ag}(p)$. Moreover, α is correct w.r.t. I_N , and $C\sigma^\alpha$ holds. Thus, $C\sigma$ holds.

Summing up, there exists $l \rightharpoonup error | \overline{C}$ which detects a concrete correctness error (p, w, l, σ) w.r.t. $\mathcal{I}$, which contradicts the initial hypothesis. ■

To conclude, by the approximation scheme formalized so far, we are able to apply the original verification framework to abstract data, providing an extremely efficient analysis which is able to locate correctness errors as well as to ensure the absence of completeness errors in the concrete descriptions quickly, saving time to the user.

5.3 Implementation

An experimental implementation αVerdi of the abstract framework proposed in this chapter has been developed and compared to the previous Verdi implementation for the realistic test cases given in [ABF+07a]. Table 5.1 shows some of the results we obtained for the simulation of the Web specification rules for an on–line auction system in five different, randomly generated XML documents. Specifically, we tuned the generator xmlgen (available within the XMark project [SWK+02]), for scaling factors from 0.01 to 0.1 to produce XML documents whose size ranges from 1Mb (corresponding to an XML tree of about 30 thousand nodes) to 10Mb (corresponding to an XML tree of about 301 thousand nodes).

Nodes	Mb	Time		
		Verdi	Abstraction	
			App	αVerdi
30 th	1	165.34 s	11 s	0.92 s
90 th	3	1,768.65 s	154 s	3.01 s
150 th	5	4,712.39 s	732 s	52.45 s
241 th	8	12,503.85 s	5,330 s	186.22 s
301 th	10	21,208.28 s	8,132 s	285.51 s

Table 5.1: Verdi and αVerdi Benchmarks

The results shown in Table 5.1 were obtained on a personal computer equipped with 1Gb of RAM memory, 40Gb hard disk and a Pentium Centrino CPU clocked at 1.75 GHz running Ubuntu Linux 7.04.

Column Verdi shows the runtime of the original Verdi tool. Column *App* shows the time used for the approximation of the Web site w.r.t. the corresponding abstract Web specification. Finally, column αVerdi shows the execution time of the abstract verification tool αVerdi.

The preliminary results that we have obtained demonstrate a huge speedup w.r.t. our previous methodology. At the same time, the abstraction times are affordable in view of the complexity and size of the involved data sets: less than 5 minutes for the largest benchmark (10 Mb), with a very reduced space budget. We note that the original Verdi implementation was only able to process efficiently XML documents whose size was not bigger than 1Mb.

5.4 Related Work

In the literature, abstract interpretation frameworks have been scarcely applied to analyze Web sites. Actually, we have found very few works addressing this issue, and all of them focus on the dynamic aspects of the distributed system underlying the Web site. For instance, in [GJJ06] an abstract approach is developed which allows one to analyse the communication protocols of a particular distributed system with the aim of enforcing a correct global behavior of the system. [KEG06] uses abstract interpretation for secret property verification: the methodology applies Input/Output abstract set descriptions to finite state machines in order to validate cryptographic protocols implementing secure Web transactions.

To the best of our knowledge, this work develops the first methodology based on abstract interpretation techniques which is general enough to support the verification of Web sites.

Our inspiration comes from the area of approximating (XML) query answering [BGK03; PGI04], where XML queries are executed on compressed versions of XML data (i.e., document synopses) in order to obtain fast, albeit approximate, answers. Roughly speaking, document synopses represent *abstractions* of the original data on which abstract computations (i.e., queries) are performed.

In our methodology, both the XML documents (Web pages) and the constraints (Web specification rules) are approximated via an abstraction function. Then, the verification process is carried out using the abstract descriptions of the considered XML data. This approach results in a powerful abstract verification methodology which pays off in practice.

Part II

Dynamic Web Verification

Chapter 6

Specification and Verification of Web Applications in Rewriting Logics

---

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.

Our contribution. This chapter presents the following original contributions.

- We define 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. Our model is formalized within the Rewriting Logic (RWL) framework [MOM02], a rule-based, logical formalism particularly appropriate to modeling concurrent systems [Mes92]. Specifically, we provide a rigorous rewrite theory which:

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

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

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

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

- We also show how rewrite theories specifying Web application models can be model-checked using the *Linear Temporal Logic of Rewriting* (LTLR) [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 report an implementation of the verification framework in Maude [CDE+07], using a built-in model-checker for LTLR. By running our prototype, we conducted an experimental evaluation which demonstrates the usefulness of our approach.

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

Plan of the chapter. The rest of the chapter is organized as follows. Section 6.1 illustrates 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 6.2, we specify a rewrite theory formalizing a simplified version of the navigation model of Section 6.1. 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 6.3 provides an extended rewrite theory which generalizes the rewrite theory formalized in Section 6.2 in order to deal with multiple Web browsers which fully support the most common navigation features of modern browsers. In Section 6.4, we introduce LTLR, and we show how we can use it to formally verify Web applications. In Section 6.5, we discuss some related work. Formal Maude specifications encoding the operational semantics of our Web scripting language and the protocol evaluation mechanism can be respectively found in Appendix A and Appendix B.

6.1 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 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).

6.1.1 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

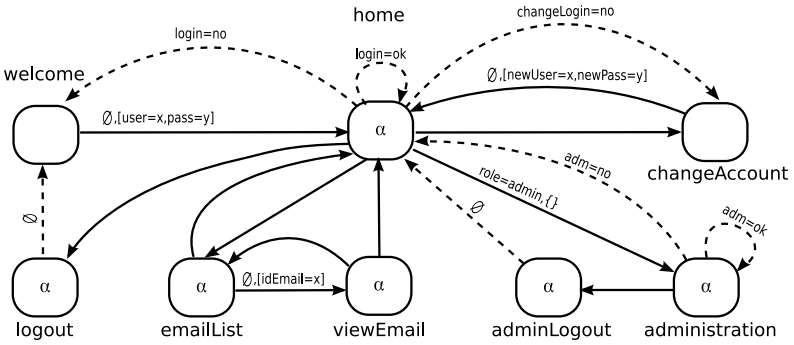


Figure 6.1: The navigation model of a Webmail application.

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 6.1.1

Consider the graphical navigation model given in Figure 6.1, 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 arrows). For example, the solid arrow between the welcome page and the home page, whose label is decorated with the string "$\emptyset$,{user=x,pass=y}", defines a navigation link which is always enabled and requires two input parameters. The home page has got two possible continuations (dashed arrows) login=ok and login=no. According to the user and pass values provided in the previous transition, only 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.

6.2 Formalizing the Navigation Model as a Rewrite Theory

In this section, we define 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, 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 6.3 generalizes the model into a more realistic scenario by defining server interactions with multiple browsers, and by equipping browsers with some standard navigation facilities.

Our formalization of a Web application consists of the specification of the following three components: the Web scripting language, the Web application structure, and the communication protocol.

6.2.1 The Web Scripting Language

We consider a scripting language which includes the main features of the most popular Web programming languages. Basically, it extends an imperative programming language with some built-in primitives for reading/writing session data (getSession, setSession), accessing and updating a data base (selectDB, updateDB), and capturing values contained in a query string sent by a browser (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 ScriptState is defined by the operator

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

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 our

framework, sessions, private memories, query strings and data bases are modeled by sets of pairs $\mathsf{id} = \mathsf{val}$, where id is an identifier whose value is represented by val. The full formalization of the operations semantics of our scripting language as a system theory in Maude can be found in Appendix A.

6.2.2 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:

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

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

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

where we enforce the following subsort relations $\mathsf{Page} < \mathsf{Soup}$, $\mathsf{Query} < \mathsf{Soup}$, $\mathsf{Continuation} < \mathsf{Soup}$, $\mathsf{Navigation} < \mathsf{Soup}$, $\mathsf{Condition} < \mathsf{Soup}$. Each subsort 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 $\langle \_ \rangle : \mathsf{Page} \rightarrow \mathsf{WebApplication}$.

Example 6.2.1

Consider again the Web application of Example 6.1.1. 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:

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

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

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

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

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

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

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

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

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

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

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

6.2.3 The Communication Protocol

We define the communication protocol by means of a rewrite theory (Σ_p, E_p, R_p), where (Σ_p, E_p) is an equational theory that formalizes the Web application states, and R_p is a set of rewrite rules that specifies 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

[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.

$(\mathsf{cond}', \mathsf{n}')$ 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.

The rewrite rule set R_p

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:

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

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

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

where $\mathsf{emptyPage} : \rightarrow \mathsf{PageName}$ 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 $\mathsf{emptyPage}$ 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 6.2.2

Consider the Web application structure wapp specified in Example 6.2.1 together with the following two Web application states

$$\begin{aligned}\mathsf{was}_1 &= \mathsf{B}(\mathsf{welcome}, \{(\mathsf{home}, [\mathsf{user} = \mathsf{Alice}, \mathsf{pass} = \mathsf{pA}])\}, \emptyset) \parallel \mathsf{empty} \parallel \\ &\qquad\qquad \mathsf{S}(\mathsf{wapp}, \emptyset, \{\mathsf{data}\}) \\ \mathsf{was}_2 &= \mathsf{B}(\mathsf{welcome}, \{(\mathsf{home}, [\mathsf{user} = \mathsf{Bob}, \mathsf{pass} = \mathsf{wrong\_pB}])\}, \emptyset) \parallel \mathsf{empty} \parallel \\ &\qquad\qquad \mathsf{S}(\mathsf{wapp}, \emptyset, \{\mathsf{data}\})\end{aligned}$$

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.

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

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

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.

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

6.3 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 6.2 —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, we define a rewrite theory $(\Sigma_{ext}, E_{ext}, R_{ext})$ extending the rewrite theory (Σ_p, E_p, R_p) presented in Section 6.2 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.

6.3.1 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:

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

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

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 6.2 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.

6.3.2 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:

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

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

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

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

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

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 $\mathsf{m_{idb,idt}}$ of the channel and enqueues it to $\mathsf{fifo_{req}}$, which is the server queue containing pending requests. Rule Evl consumes the first request message $\mathsf{m_{idb,idt}}$ of the queue $\mathsf{fifo_{req}}$, evaluates the message w.r.t. the corresponding browser session $(\mathsf{id_b}, \{\mathsf{s}\})$, and generates the response message which is enqeued in $\mathsf{fifo_{res}}$; 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 $\mathsf{fifo_{res}}$ 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 6.3.1

Consider the scenarios given in Example 6.2.2 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.

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

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

We formalize browser navigation features as follows.

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

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

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

$$\mathsf{Backward}\colon \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \mathsf{p_c}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \Rightarrow \mathsf{B}(\mathsf{id_b}, \mathsf{id_t}, \underline{\mathsf{p_h}, \{\mathsf{urls_h}\}}, \{\mathsf{s}\}, \underline{\mathsf{lm_h}}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv}$$
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 $\mathsf{R}_{\mathsf{ext}}$ 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 the future, 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.

6.4 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 6.3. 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.

6.4.1 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 $\mathsf{statePattern} \models \mathsf{property}(\mathsf{a}_1, \ldots, \mathsf{a}_n) = \mathsf{booleanValue}$. Roughly speaking, a state predicate formalizes a property $\mathsf{property}(\mathsf{a}_1, \ldots, \mathsf{a}_n) = \mathsf{booleanValue}$ over all the states specified by $\mathcal{R}$ which match the $\mathsf{statePattern}$.

Example 6.4.1

Let (Σ_p, E_p, R_p) be the rewrite theory specified in Section 6.2, which models the Web application states as terms $\mathsf{b}\|\mathsf{m}\|\mathsf{s}$ of sort $\mathsf{WebState}$ 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 6.4.2

Let $(\Sigma_{ext}, E_{ext}, R_{ext})$ be the rewrite theory introduced in Section 6.3 that specifies our extended model for Web applications. Then, the spatial action pattern $\mathsf{ReqIni(id\backslash A)}$ asserts that the general action[2]

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

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

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 $\varphi \in LTLR(SP, \Pi)$.

6.4.2 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.

[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.

As an example, consider the Webmail application given in Example 6.1.1 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,

$$\mathsf{B}(\underline{\mathsf{id}}, \mathsf{id_t}, \underline{\mathsf{pn}}, \{\mathsf{urls}\}, \{\mathsf{s}\}, \mathsf{lm}, \mathsf{h}, \mathsf{i}), \mathsf{br} \parallel \mathsf{m} \parallel \mathsf{sv} \models \mathsf{curPage}(\mathsf{id}, \mathsf{pn}) = \mathsf{true}$$

$$\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{wapp}, \{\mathsf{BS}(\underline{(\mathsf{id}, \{\mathsf{failed} = \mathsf{n}\})}, \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}}) \models \mathsf{failedAttempt}(\mathsf{id}, \mathsf{n}) = \mathsf{true}$$

$$\mathsf{br} \parallel \mathsf{m} \parallel \mathsf{S}(\mathsf{wapp}, \{\mathsf{BS}(\underline{(\mathsf{id}, \{\mathsf{forbid} = \mathsf{true}\})}, \mathsf{bs}\}, \{\mathsf{db}\}, \mathsf{fifo_{req}}, \mathsf{fifo_{res}}) \models \mathsf{userForbidden}(\mathsf{id}) = \mathsf{true}$$

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

[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.

change the parameter counting the login attempts in the state predicate $\mathsf{failedAttempt}(\mathsf{A}, 3)$. Besides, note that we can define state predicates (and more in general LTLR formulae) which depend on Web script evaluations. For instance, the predicate 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

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

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 6.4.3

Consider again the Webmail application of Example 6.1.1. 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 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 6.1.1 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 inconsistentState is a state predicate defined as:

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

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:

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

6.5 Related Work

Web applications are complex software systems playing a primary role of primary importance nowadays. Not surprisingly that a significant work has been invested in the modeling and verification of such systems. A variant of the μ-calculus (called constructive μ-calculus) is proposed in [Alf01] which allows one to model-check connectivity properties over the *static* graph-structure of a Web system. However, this methodology does not support the verification of dynamic properties— e.g., reachability over Web pages generated by means of Web script executions.

Both Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) have been used for the verification of dynamic Web applications. For instance, [FLV08] and [HSP08] support model-checking of LTL properties w.r.t. Web application models represented as Kripke structures. Similar methodologies have been developed in [MZ07] and [DMRT06] to verify Web applications by using CTL formulae. All these model-checking approaches are based on coarse Web application models which are concerned neither with the communication protocol underlying the Web interactions nor the browser navigation features. Moreover, as shown in Section 6.4, CTL and LTL property specifications are very often textually large and hence difficult to formulate and understand. [HH06] presents a modeling and verification methodology that uses CTL and considers some basic adaptative navigation features. In constrast, our framework provides a complete formalization which supports more advanced adaptive navigation capabilities.

Finally, both [GFKF03] and [Que04] do provide accurate analyses of Web interactions which point out typical unexpected application behaviors which are essentially caused by the uncontrolled use of the browser navigation buttons as well as the shortcomings of HTTP. Their approach however is different from ours since it is based on defining a novel Web programming language which allows one to write safe Web applications: [GFKF03] exploits type checking techniques to ensure application correctness, whereas [Que04] adopts a semantic approach which is based on program continuations. None of these provide a full-equipped verification framework comparable to ours.

Chapter 7

Backward Trace Slicing for Rewriting Logic Theories

Trace slicing is a widely used technique for execution trace analysis that is effectively used in program debugging, analysis and comprehension. In this chapter, we present a backward trace slicing technique [ABER11] that can be used for the analysis of Rewriting Logic theories. Our trace slicing technique allows us to systematically trace back rewrite sequences modulo equational axioms (such as associativity and commutativity) by means of an algorithm that dynamically simplifies the traces by detecting control and data dependencies, and dropping useless data that do not influence the final result. Our methodology is particularly suitable for analyzing complex, textually-large system computations such as those delivered as counter-example traces by Maude model-checkers.

7.1 Introduction

The analysis of execution traces plays a fundamental role in many program manipulation techniques. Trace slicing is a technique for reducing the size of traces by focusing on selected aspects of program execution, which makes it suitable for trace analysis and monitoring [CR09].

Rewriting Logic (RWL) is a very general *logical* and *semantic framework*, which is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [BBF09; Tal08] and Web systems [ABER10; ABR09]). RWL is efficiently implemented in the high-performance system Maude [CDE+07]. Roughly speaking, a *rewriting logic theory* seamlessly combines a *term rewriting system* (TRS) together with an *equational theory* that may include sorts, functions, and algebraic laws (such as commutativity and associativity) so that rewrite steps are applied *modulo* the equations. Within this framework, the system states

are typically represented as elements of an algebraic data type that is specified by the equational theory, while the system computations are modeled via the rewrite rules, which describe transitions between states.

Due to the many important applications of RWL, in recent years, the debugging and optimization of RWL theories have received growing attention [ABBF10; MOM02; RVCMO09; RVMO10]. However, the existing tools provide hardly support for execution trace analysis. The original motivation for our work was to reduce the size of the counter-example traces delivered by Web-TLR, which is a RWL-based model-checking tool for Web applications proposed in [ABER10; ABR09]. As a matter of fact, the analysis (or even the simple inspection) of such traces may be unfeasible because of the size and complexity of the traces under examination. Typical counter-example traces in Web-TLR are 75 Kb long for a model size of 1.5 Kb, that is, the trace is in a ratio of 5.000% w.r.t. the model.

To the best of our knowledge, this chapter presents the first trace slicing technique for RWL theories. The basic idea is to take a trace produced by the RWL engine and traverse and analyze it backwards to filter out events that are irrelevant for the rewritten task. The trace slicing technique that we propose is fully general and can be applied to optimizing any RWL-based tool that manipulates rewrite logic traces. Our technique relies on a suitable mechanism of backward tracing that is formalized by means of a procedure that labels the calls (terms) involved in the rewrite steps. The backward traversal is preferred to a forward one because a causal relation is computed. This allows us to infer, from a term t and positions of interest on it, positions of interest of the term that was rewritten to t. Our labeling procedure extends the technique in [BKdV00], which allows descendants and origins to be traced in orthogonal (i.e., left-linear and overlap-free) term rewriting systems in order to deal with rewrite theories that may contain commutativity/associativity axioms, as well as nonleft-linear, collapsing equations and rules.

Plan of the chapter. In Section 7.3, we formalize our backward trace slicing technique for rewriting logic theories, which computes the reverse dependence among the symbols involved in a rewrite step and removes all data that are irrelevant with respect to a given slicing criterion. Section 7.4 extends the trace slicing technique of Section 7.3 by considering

extended rewrite theories, i.e., rewrite theories that may include collapsing, nonleft-linear rules, associative/commutative equational axioms, and built-in operators. Section 7.5 describes a software tool that implements the proposed backward slicing technique and presents an experimental evaluation of the tool that allows us to assess the practical advantages of the trace slicing technique. In Section 7.6, we discuss some related work, and Section 7.7 concludes. Appendix C illustrates our trace slicing technique by means of a practical example that allows one to assess the advantages of our approach.

7.2 Rewriting Modulo Equational Theories

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

A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory, and R is a TRS. Examples of rewrite theories can be found in [CDE+07].

Rewriting modulo equational theories [MOM02] can be defined by lifting the standard rewrite relation $\rightarrow_R$ on terms to the E-congruence classes induced by $=_E$. More precisely, the rewrite relation $\rightarrow_{R/E}$ for rewriting modulo E is defined as $=_E \circ \rightarrow_R \circ =_E$. A computation in $\mathcal{R}$ using $\rightarrow_{R\cup\Delta,B}$ is a *rewriting logic deduction*, in which the *equational simplification* with Δ (i.e., applying the oriented equations in Δ to a term t until a canonical form $t\downarrow_E$ is reached where no further equations can be applied) is intermixed with the rewriting computation with the rules of R, using an *algorithm of matching modulo*[1] B in both cases.

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

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

(*ii*) rewrite (modulo B) ($s_0 \downarrow_E$) to term v by using $r \in R$; and (*iii*), apply (modulo B) the equations of Δ on v again to reach a canonical form for v, $s_1 = v \downarrow_E$.

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

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

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

The computability of $\rightarrow_{R\cup\Delta,B}$ as well as its equivalence w.r.t. $\rightarrow_{R/E}$ are assured by enforcing some conditions on the considered rewrite theories [MOM02; Vir94], specifically, *coherence* between the rules and the equations as well as the assumption of *Church–Rosser* and *termination* properties of Δ modulo the equational axioms B[2].

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

7.3 Backward Trace Slicing for Elementary Rewrite Theories

In this section, we formalize a backward trace slicing technique for *elementary rewrite theories* that is based on a term labeling procedure that is inspired by [BKdV00]. Since equations in Δ are treated as rewrite rules

[2]These conditions are quite natural in practical rewriting logic specifications, and can generally be checked by using the Maude Church–Rosser, Termination, and Coherence tools [CDE+07].

that are used to simplify terms, our formulation for the trace slicing technique is purely based on standard rewriting. In Section 7.4, we will drop all these restrictions in order to consider more expressive rewrite theories.

7.3.1 Labeling Procedure for Rewriting Theories

Let us define a labeling procedure for rules similar to [BKdV00] that allows us to trace symbols involved in a rewrite step. First, we provide the notion of labeling for terms, and then we show how it can be naturally lifted to rules and rewrite steps.

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

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

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

Labeling of Rules

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

Example 7.3.1

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

Definition 7.3.2 *(rule labeling) [BKdV00] Given a rule $r : \lambda \rightarrow \rho$, a labeling L_r for r is defined by means of the following procedure.*

r_1. *The redex pattern $\lambda^{\Box}$ is labeled by means of an initial labeling L.*

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

Example 7.3.3

Consider the rule r of Example 7.3.1. The labeled version of rule r using the initial labeling $L = \{(\Lambda \mapsto \alpha, 1 \mapsto \beta, 2 \mapsto \gamma\}$ is as follows: $f^{\alpha}(g^{\beta}(x, y), a^{\gamma}) \rightarrow d^{\alpha\beta\gamma}(s^{\alpha\beta\gamma}(y), y)$.

The labeled version of r w.r.t. L_r is denoted by r^{L_r}. Note that the labeling procedure shown in Definition 7.3.2 does not assign labels to variables but only to the function symbols occurring in the rule.

Labeling of Rewrite Steps

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

Definition 7.3.4 *(substitution labeling) Let $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ be a substitution. A labeling L_σ for the substitution σ is defined by a set of initial labelings $L_\sigma = \{L_{x_1/t_1}, \ldots, L_{x_n/t_n}\}$ such that (i) for each binding (x_i/t_i) in the substitution σ, t_i is labeled using the corresponding initial labeling L_{x_i/t_i}, and (ii) the sets $Cod(L_{x_1/t_1}), \ldots, Cod(L_{x_n/t_n})$ are pairwise disjoint.*

By using Definition 7.3.4, we can formulate a labeling procedure for rewrite steps as follows.

Definition 7.3.5 *(rewrite step labeling) Let $r : \lambda \rightarrow \rho$ be a rule, and let $\mu : t \xrightarrow{r,\sigma} s$ be a rewrite step using r such that $t = C[\lambda\sigma]_q$ and $s = C[\rho\sigma]_q$, for a context C and position q. Let $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. Let L_r be a labeling for the rule r, let L_C be an initial labeling for the context C,*

and let $L_\sigma = \{L_{x_1/t_1}, \ldots, L_{x_n/t_n}\}$ *be a labeling for the substitution* σ *such that the sets* $Cod(L_C)$, $Cod(L_r)$, *and* $Cod(\sigma)$ *are pairwise disjoint, where* $Cod(\sigma) = \bigcup_{i=1}^{n} Cod(L_{x_i/t_i})$.

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

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

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

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

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

Example 7.3.6

Consider again the rule $r : f(g(x, y), a)) \to d(s(y), y)$ of Example 7.3.1, and let $\mu : C[\lambda\sigma] \xrightarrow{r} C[\rho\sigma]$ be a rewrite step using r, where $C[\lambda\sigma] = d(f(g(a, h(b)), a), a)$, $C[\rho\sigma] = d(d(s(h(b)), h(b)), a)$, and $\sigma = \{x/a, y/h(b)\}$.

Assume that r is labeled by means of the rule labeling of Example 7.3.3, that is

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

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

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

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

Definition 7.3.7 *(origin positions) Let $\mu : t \xrightarrow{r} s$ be a rewrite step, and let L be a labeling for μ where L_t (resp. L_s) is the labeling of t (resp. s). Given a position w of s, the set of origin positions of w in t w.r.t. μ and L (in symbols, $\lhd^L_\mu w$) is defined as follows:*

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

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

Example 7.3.8

Consider again the rewrite step $\mu^L : t^L \rightarrow s^L$ of Example 7.3.6, and let w be the position 1.2 of s^L. The set of labeled symbols occurring in s^L in the path from its root to position w is the set $\mathtt{z} = \{\mathtt{h}^\eta, \mathtt{d}^{\alpha\beta\gamma}, \mathtt{d}^\delta\}$. Now, the labeled symbols occurring in t^L whose label is contained in the label of one element of $\mathtt{z}$ is the set $\{\mathtt{h}^\eta, \mathtt{f}^\alpha, \mathtt{g}^\beta, \mathtt{a}^\gamma, \mathtt{d}^\delta\}$. By Definition 7.3.7, the set of origin positions of w in μ^L is $\lhd^L_\mu \mathtt{w} = \{\mathtt{1.1.2},\ \mathtt{1},\ \mathtt{1.1},\ \mathtt{1.2},\ \Lambda\}$.

Note that the origin positions of w in the rewrite step $\mu : t \xrightarrow{r} s$ are not the antecedent positions of w in μ [Rét87]; one main difference is the fact that we consider all positions of s in the path from its root to w for computing the origins, and we use the labeling to trace back every relevant piece of information involved in the step μ.

7.3.2 The Backward Trace Slicing Algorithm

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

Definition 7.3.9 *(slicing criterion) Given a rewrite theory $\mathcal{R} = (\Sigma, \Delta, R)$ and an execution trace $\mathcal{T} : s \rightarrow^* t$ in $\mathcal{R}$, a slicing criterion for $\mathcal{T}$ is any set $\mathcal{O}_t$ of positions of the term t.*

In the following, we show how backward trace slicing can be performed by exploiting the backward tracing relation $\lhd_\mu^L$ that was introduced in Definition 7.3.7. Informally, given a slicing criterion $\mathcal{O}_{t_n}$ for $\mathcal{T} : t_0 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$, at each rewrite step $t_{i-1} \rightarrow t_i$, $i = 1, \ldots, n$, our technique inductively computes the backward tracing relation between the relevant positions of t_i and those in t_{i-1}. The algorithm proceeds backwards, from the final term t_n to the initial term t_0, and recursively generates at step i the corresponding set of relevant positions, $P_{t_{n-i}}$. Finally, by means of a removal function, a simplified trace is obtained where each t_j is replaced by the corresponding *term slice* that contains only the relevant information w.r.t. P_{t_j}.

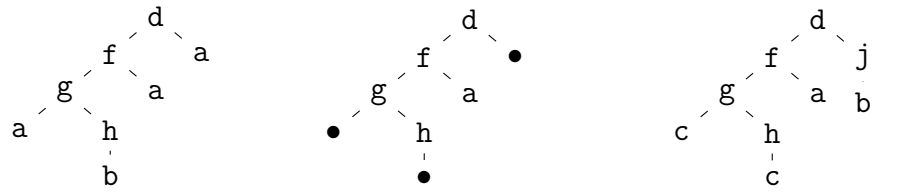
Definition 7.3.10 *(sequence of relevant position sets) Let $\mathcal{R} = (\Sigma, \Delta, R)$ be a rewrite theory, and let $\mathcal{T} : t_0 \xrightarrow{r_1} t_1 \ldots \xrightarrow{r_n} t_n$ be an execution trace in $\mathcal{R}$. Let L_i be the labeling for the rewrite step $t_i \rightarrow t_{i+1}$ with $0 \leq i < n$. The sequence of relevant position sets in $\mathcal{T}$ w.r.t. the slicing criterion $\mathcal{O}_{t_n}$ is defined as follows:*

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

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

Definition 7.3.11 *(term slice) Let $t \in T_\Sigma$ be a term, and let P be a set of positions of t. A* term slice *of t with respect to P is defined as follows:*

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



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

Figure 7.1: A term slice and one possible concretization.

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

Definition 7.3.12 *(term slice concretization) Given $t' \in T_\Sigma$ and a term slice $t^\bullet$, we define $t^\bullet \propto t'$ if $[t^\bullet]$ is (syntactically) more general than t' (i.e. $[t^\bullet]\sigma = t'$, for some substitution σ). We also say that t' is a concretization of $t^\bullet$.*

Figure 7.1 illustrates the notions of term slice and term slice concretization for a given term t w.r.t. the set of positions $\{1.1.2, 1.2\}$.

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

Definition 7.3.13 *(sliced rewrite step) Let $\mathcal{R} = (\Sigma, \Delta, R)$ be a rewrite theory, and let r be a rule of $\mathcal{R}$. The term slice $s^\bullet$ rewrites to the term slice $t^\bullet$ via r (in symbols, $s^\bullet \xrightarrow{r} t^\bullet$) if there exist two terms s and t such that $s^\bullet$ is a term slice of s, $t^\bullet$ is a term slice of t, and $s \xrightarrow{r} t$.*

Using Definition 7.3.13, backward trace slicing is formalized as follows.

Definition 7.3.14 *(backward trace slicing) Let $\mathcal{R} = (\Sigma, \Delta, R)$ be a rewrite theory, and let $\mathcal{T} : t_0 \xrightarrow{r_1} t_1 \ldots \xrightarrow{r_n} t_n$ be an execution trace in $\mathcal{R}$. Let $\mathcal{O}_{t_n}$*

be a slicing criterion for $\mathcal{T}$*, and let* $[P_0, \ldots, P_n]$ *be the sequence of the relevant position sets of* $\mathcal{T}$ *w.r.t.* $\mathcal{O}_{t_n}$*. A trace slice* $\mathcal{T}^\bullet$ *of* $\mathcal{T}$ *w.r.t.* $\mathcal{O}_{t_n}$ *is defined as the sliced rewrite sequence of term slices* $t_i^\bullet = slice(t_i, P_i)$ *which is obtained by glueing together the sliced rewrite steps in the set*

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

Note that in Definition 7.3.14, the sliced rewrite steps that do not affect the relevant positions (i.e., $t_{k-1}^\bullet \xrightarrow{r_k} t_k^\bullet$ with $t_{k-1}^\bullet = t_k^\bullet$) are discarded, which further reduces the size of the trace.

A desirable property of a slicing technique is to ensure that, for any concretization of the term slice $t_0^\bullet$, the trace slice $\mathcal{T}^\bullet$ can be reproduced. This property ensures that the rules involved in $\mathcal{T}^\bullet$ can be applied again to every concrete trace $\mathcal{T}'$ that we can derive by instantiating all the variables in $[t_0^\bullet]$ with arbitrary terms.

Theorem 7.3.15 *(soundness) Let* $\mathcal{R}$ *be an elementary rewrite theory. Let* $\mathcal{T}$ *be an execution trace in the rewrite theory* $\mathcal{R}$*, and let* $\mathcal{O}$ *be a slicing criterion for* $\mathcal{T}$*. Let* $\mathcal{T}^\bullet : t_0^\bullet \xrightarrow{r_1} t_1^\bullet \ldots \xrightarrow{r_n} t_n^\bullet$ *be the corresponding trace slice w.r.t.* $\mathcal{O}$*. Then, for any concretization* t'_0 *of* $t_0^\bullet$*, it holds that* $\mathcal{T}' : t'_0 \xrightarrow{r_1} t'_1 \ldots \xrightarrow{r_n} t'_n$ *is an execution trace in* $\mathcal{R}$*, and* $t_i^\bullet \propto t'_i$*, for* $i = 1, \ldots, n$*.*

The proof of Theorem 7.3.15 relies on the fact that redex patterns are preserved by backward trace slicing. Therefore, for $i = 1, \ldots, n$, the rule r_i can be applied to any concretization t'_{i-1} of term $t_{i-1}^\bullet$ since the redex pattern of r_i does appear in $t_{i-1}^\bullet$, and hence in t'_{i-1}. A detailed proof of Theorem 7.3.15 is given in following section.

Proof of Theorem 7.3.15

We first demonstrate some auxiliary results which facilitate the proof of Theorem 7.3.15. The following auxiliary result is straightforward.

Lemma 7.3.16 *Let* $t^\bullet$ *be a term slice, and let* t' *be a term such that* $t^\bullet \propto t'$*. For every position* $w \in \mathcal{P}os(t')$*, it holds that, either* $root(t'_{|w}) = root(t_{|w}^\bullet)$*, or there exists a position* u *of* $t^\bullet$ *such that* $u \leq w$ *and* $root(t_{|u}^\bullet) = \bullet$*.*

Proof. Immediate by Definition 7.3.12. ∎

The following definitions are auxiliary. Let C be a context. We define the set of positions of C as the set $\mathcal{P}os(C) = \{v \mid root(C_{|v}) \neq \Box\}$. Given a term t, 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., $path_{(2.1)}(f(a, g(b), c)) = \{f, g, b\}$.

Definition 7.3.17 *Let $r : \lambda \rightarrow \rho$ be a rule of $\mathcal{R}$. Let $\mu : s \overset{r,\sigma}{\rightarrow} t$ be a rewrite step such that $s = C[\lambda\sigma]$ and $t = C[\rho\sigma]$. Given a position w, we say that w* is involved in *μ, if there exist w' and w'' such that $w = w'.w''$, $C_{|w'} = \Box$ and $w'' \in \mathcal{P}os(\rho\sigma)$.*

The following lemma establishes that, if a relevant position is involved in a rewrite step, then the origin position relation preserves the redex pattern of the rule.

Lemma 7.3.18 *Let $r : \lambda \rightarrow \rho$ be a rule of an elementary rewrite theory $\mathcal{R}$. Let $\mu : s \overset{r,\sigma}{\rightarrow} t$ be a rewrite step such that $s = C[\lambda\sigma]$ and $t = C[\rho\sigma]$, where σ is a substitution and C is a context. Let L be a labeling for the rewrite step μ, and $w \in \mathcal{P}os(t)$.*

1. *if $w \in \mathcal{P}os(C)$, then $\lhd^L_\mu w = \{v \in \mathcal{P}os(C) \mid w = v.v'\}$*
2. *if $w = w'.w''$, $C_{|w'} = \Box$, and $w'' \in \mathcal{P}os(\rho\sigma)$, then $\lhd^L_\mu w \supseteq \{w'.v' \in \mathcal{P}os(s) \mid v' \in \mathcal{P}os(\lambda)\}$*

Proof. Given the rule $r : \lambda \rightarrow \rho$ and the labeling L for the rewrite step $\mu : s \overset{r,\sigma}{\rightarrow} t$, let us consider the labeled rewrite step $\mu^L : s^L \overset{r^L,\sigma^L}{\rightarrow} t^L$. By Definition 7.3.5, we can decompose the labeling L into three labelings L_C, L_r, and L_σ that respectively label the context C, the redex and the contractum patterns appearing in μ, and the terms in μ introduced by the substitution σ. In other words, we have $s^L = C^{L_C}[\lambda^{L_r}\sigma^{L_\sigma}]$ and $t^L = C^{L_C}[\rho^{L_r}\sigma^{L_\sigma}]$.

Let us prove the two claims independently.

Claim 1. We assume that $w \in \mathcal{P}os(t)$ and $w \in \mathcal{P}os(C)$. Since the context C has the same initial labeling C^{L_C} in both s and t, and the sets $Cod(L_C)$, $Cod(L_r)$, and $Cod(L_\sigma)$ are pairwise disjoint, the set of origin

positions $\lhd^L_{s \to t} w$ in s is the set of positions lying on the path from the root position of s to w. Hence, $\lhd^L_\mu w = \{v \in \mathcal{P}os(C) \mid w = v.v'\}$.
Claim 2. We assume that $w = w'.w''$, $C_{|w'} = \Box$, and $w'' \in \mathcal{P}os(\rho\sigma)$. Then, since r belongs to an elementary rewrite theory $\mathcal{R}$, r is non-collapsing. This implies that there exists a labeled symbol $f^{l'} \in path_w(t^L)$ belonging to the contractum pattern of the rule r. By Definition 7.3.2, for each labeled symbol g^l in the redex pattern of r, we have that $l \subseteq l'$. Now, since the redex pattern of r is embedded into s and the contractum pattern of r is embedded into t, the inclusion $\lhd^L_\mu w \supseteq \{v.v' \in \mathcal{P}os(s) \mid v' \in \mathcal{P}os(\lambda)\}$ trivially holds by Definition 7.3.7. ■

The following lemma establishes that, given the rewrite step $\mu : t_0 \xrightarrow{r} t_1$ and a term slice $t_0^\bullet$ of t_0, any concretization of $t_0^\bullet$ is reduced by the rule r to the corresponding term slice concretization of t_1.

Lemma 7.3.19 *Let $r : \lambda \to \rho$ be a rule of an elementary rewrite theory $\mathcal{R}$. Let $\mu : t_0 \xrightarrow{r,\sigma} t_1$ be a rewrite step such that $t_0 = C[\lambda\sigma]$ and $t_1 = C[\rho\sigma]$, where σ is a substitution and C is a context. Let L be a labeling for the rewrite step μ, and let $[P_0, P_1]$ be the sequence of the relevant position sets for $\mu : t_0 \xrightarrow{r,\sigma} t_1$ w.r.t. the slicing criterion $\mathcal{O}$. Let $t_0^\bullet = slice(t_0, P_0)$, and $t_1^\bullet = slice(t_1, P_1)$.*

1. *if $P_1 \subseteq \mathcal{P}os(C)$ then $t_0^\bullet = t_1^\bullet$.*
2. *if $P_1 \cap \{w | w = v.v', C_{|v} = \Box, \text{ and } v' \in \mathcal{P}os(\rho\sigma)\} \neq \emptyset$, then for any concretization t'_0 of $t_0^\bullet$, we have that $t'_0 \xrightarrow{r,\sigma'} t'_1$ where $t_1^\bullet \propto t'_1$.*

Proof. We prove the two claims separately.
Claim 1. Let $P_1 \subseteq \mathcal{P}os(C)$. Then, by Lemma 7.3.18 (Claim 1), for any $w \in P_1$, $\lhd^L_\mu w = \{v \in \mathcal{P}os(C) \mid w = v.v'\}$. Additionally, by Definition 7.3.10, $P_0 = \bigcup_{w \in P_1} (\lhd^L_\mu w)$, and hence $P_0 = \bigcup_{w \in P_1} \{v \in \mathcal{P}os(C) \mid w = v.v'\}$. Therefore, it holds that (i) $P_1 \subseteq P_0 \subseteq \mathcal{P}os(C)$, and for any $v \in P_0 \setminus P_1$, there exists a position v' such that $w = v.v'$ for some $w \in P_1$; (ii) by Definition 7.3.11, the function $slice(t, P)$ delivers a term slice $t^\bullet$ where all the symbols of t that do not occur in the path connecting the root position of t with some position $w \in P$ are abstracted by the $\bullet$ symbol. Now, since $t_0^\bullet = slice(t_0, P_0)$ and $t_1^\bullet = slice(t_1, P_1)$, by (i) and (ii), we can conclude that $\lambda\sigma$ and $\rho\sigma$ are abstracted by $\bullet$, and the

context C is abstracted by the term slice $C^\bullet$ in both t_0 and t_1. Hence, $t_0^\bullet = C^\bullet[\bullet] = t_1^\bullet$.

Claim 2. We assume $P_1 \cap \{w | w = v.v', C_{|v} = \Box, \text{ and } v' \in \mathcal{P}os(\rho\sigma)\} \neq \emptyset$. Then, there exists a position $w \in P_1$ such that $w \in \{w | w = v.v', C_{|v} = \Box, \text{ and } v' \in \mathcal{P}os(\rho)\}$. By Lemma 7.3.18 (Claim 2), it follows that $\lhd_\mu^L w \supseteq \{v.v' \in \mathcal{P}os(t_0) \mid v' \in \mathcal{P}os(\lambda)\}$. By Definition 7.3.10, $P_0 = \bigcup_{w \in P_1} (\lhd_\mu^L w)$, and hence $P_0 \supseteq \{v.v' \in \mathcal{P}os(t_0) \mid v' \in \mathcal{P}os(\lambda)\}$. Now, by Definition 7.3.11 and the fact that $P_0 \supseteq \{v.v' \in \mathcal{P}os(t_0) \mid v' \in \mathcal{P}os(\lambda)\}$, the redex pattern of the rule r is embedded into $t_0^\bullet = \mathit{slice}(t_0, P_0)$. In other words, $t_0^\bullet = C^\bullet[\lambda\sigma^\bullet]$, where $C^\bullet$ is a term slice for the context C, and $\sigma^\bullet$ represents the term slices for the terms introduced by the substitution σ. Thus, by Lemma 7.3.16, any concretization t_0' of $t_0^\bullet$ has the form $t_0' = C'[\lambda\sigma']$, where $C^\bullet \propto C'$ and for each $x/t \in \sigma'$, there exists $x/t^\bullet \in \sigma^\bullet$ such that $t^\bullet \propto t$. Note also that $t_0^\bullet$ embeds the redex pattern $\lambda^\Box$ of r. Furthermore, since r belongs to the elementary rewrite theory $\mathcal{R}$, r is left-linear. Thus, the following rewrite step $t_0' \overset{r,\sigma'}{\to} t_1'$ can be executed for any substitution σ'. The rewrite step $t_0' \overset{r,\sigma'}{\to} t_1'$ can be decomposed as follows: $t_0' = C'[\lambda\sigma'] \overset{r,\sigma'}{\to} C'[\rho\sigma']$, for some context C' and substitution σ'. Moreover, by definition of rewrite step, t_1' embeds the contractum pattern of r. Finally, $t_1^\bullet = C^\bullet[\rho^\bullet\sigma^\bullet]$, and thus t_1' is a concretization of $t_1^\bullet$. ■

The following proposition allows the soundness of our methodology to be proved for one-step traces on an elementary rewrite theory.

Proposition 7.3.20 *Let $\mathcal{R}$ be an elementary rewrite theory. Let $\mathcal{T}$ be an execution trace in $\mathcal{R}$, and let $\mathcal{O}$ be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^\bullet : t_0^\bullet \overset{r_1}{\to} t_1^\bullet$ be the trace slice w.r.t. $\mathcal{O}$ of $\mathcal{T}$. Then, for any concretization t_0' of $t_0^\bullet$, it holds that $\mathcal{T}' : t_0' \overset{r_1}{\to} t_1'$ is an execution trace in $\mathcal{R}$ such that $t_1^\bullet \propto t_1'$.*

Proof. Given the trace slice $\mathcal{T}^\bullet : t_0^\bullet \overset{r_1}{\to} t_1^\bullet$ w.r.t. $\mathcal{O}$ of $\mathcal{T}$, let $[P_0, P_1]$ be the sequence of the relevant position sets of $\mathcal{T}$ w.r.t. $\mathcal{O}$. We have (i) $t_0^\bullet = \mathit{slice}(s_0, P_0)$ and $t_1^\bullet = \mathit{slice}(s_1, P_1)$, where $s_0 \overset{r_1}{\to} s_1$ is a rewrite step occurring in $\mathcal{T}$; (ii) $t_0^\bullet \neq t_1^\bullet$. Let r_1 be the rule $\lambda \to \rho$. The rewrite step $s_0 \overset{r_1}{\to} s_1$ can be decomposed as follows: $s_0 = C[\lambda\sigma] \overset{r_1}{\to} C[\rho\sigma] = s_1$, for some context C and substitution σ.

Since $\mathcal{R}$ is elementary and $t_0^\bullet \neq t_1^\bullet$, by Claim 1 of Lemma 7.3.19, $P_1 \not\subseteq \mathcal{P}os(C)$. Hence, there exists a position $w \in P_1$ such that $w = v.v'$ and $v' \in \mathcal{P}os(\rho\sigma)$. Also, because $\mathcal{R}$ is elementary, we can apply Claim 2 of Lemma 7.3.19, and for any concretization t'_0 of $t_0^\bullet$, we get $t'_0 \xrightarrow{r_1} t'_1$ such that t'_1 is a concretization of $t_1^\bullet$. ■

Theorem 7.3.15. *(soundness) Let $\mathcal{R}$ be an elementary rewrite theory. Let $\mathcal{T}$ be an execution trace in $\mathcal{R}$, and let $\mathcal{O}$ be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^\bullet : t_0^\bullet \xrightarrow{r_1} t_1^\bullet \ldots \xrightarrow{r_n} t_n^\bullet$ be the corresponding trace slice w.r.t. $\mathcal{O}$. Then, for any concretization t'_0 of $t_0^\bullet$, it holds that $\mathcal{T}' : t'_0 \xrightarrow{r_1} t'_1 \ldots \xrightarrow{r_n} t'_n$ is an execution trace in $\mathcal{R}$, and $t_i^\bullet \propto t'_i$, for $i = 1, \ldots, n$.*

Proof. The proof proceeds by induction on the length of the trace slice $\mathcal{T}^\bullet$ and exploits Proposition 7.3.20 to prove the inductive case. Routine. ■

7.4 Backward Trace Slicing for Extended Rewrite Theories

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

It is worth noting that all the proposed extensions are restricted to the labeling procedure of Section 7.3.1, leaving the backbone of our slicing technique unchanged.

7.4.1 Dealing with Collapsing and Nonleft-linear Rules

Collapsing Rules. The main difficulty with collapsing rules is that they have a trivial contractum pattern, which consists in the empty context $\Box$;

hence, it is not possible to propagate labels from the left-hand side of the rule to its right-hand side. This makes the rule labeling procedure of Definition 7.3.2 completely unproductive for trace slicing.

In order to overcome this problem, we keep track of the labels in the left-hand side of the collapsing rule r, whenever a rewrite step involving r takes place. This amounts to extending the labeling procedure of Definition 7.3.5 as follows.

Definition 7.4.1 *(rewrite step labeling for collapsing rules) Let $\mu : t \overset{r,\sigma}{\rightarrow} s$ be a rewrite step s.t. $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. Let L_r be a labeling for the rule r. For the case of a rewrite step given by using a collapsing rule $r : \lambda \rightarrow x_i$, the labeling procedure formalized in Definition 7.3.5 is extended as follows:*

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

Example 7.4.2

Consider again the labeled collapsing rule $f^{\alpha}(a^{\beta}, x) \rightarrow x$, together with the rewrite step $\mu : f(a, h(b)) \rightarrow h(b)$ and matching substitution $\sigma = \{x/h(b)\}$. Let $L_\sigma = \{\{\Lambda \mapsto \gamma,\ 1 \mapsto \delta\}\}$ be the labeling for σ. Then, by applying Definition 7.4.1, the labeling of μ is

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

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

Note that if we had merely applied Definition 7.3.5 instead of Definition 7.4.1, we would have got the following labeling for μ: $f^{\alpha}(a^{\beta}, h^{\gamma}(b^{\delta})) \rightarrow h^{\gamma}(b^{\delta})$, which is undesirable since it does not correctly record the redex pattern information that we need for backward trace slicing: e.g., if we slice the rewriting step μ w.r.t. $\{\Lambda\}$ using this wrong labeling, we would get $f(\bullet, h(\bullet)) \rightarrow h(\bullet)$.

Nonleft-linear Rules. The trace slicing technique we described in Section 7.3 does not work for nonleft-linear TRS. Consider the rule: r :

$f(x,y,x) \to g(x,y)$ and the one-step trace $\mathcal{T} : f(a,b,a) \to g(a,b)$. If we are interested in tracing back the symbol g that occurs in the final state $g(a,b)$, we would get the following trace slice $\mathcal{T}^\bullet : f(\bullet,\bullet,\bullet) \to g(\bullet,\bullet)$. However, $f(a,b,b)$ is a concretization of $f(\bullet,\bullet,\bullet)$ that cannot be rewritten by using r. In the following, we augment Definition 7.4.1 in order to also deal with nonleft-linear rules.

Definition 7.4.3 *(rewrite step labeling procedure for nonleft-linear rules) Let $\mu : t \overset{r,\sigma}{\to} s$ be a rewrite step s.t. $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. Let $L_\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ be a labeling for the substitution σ. For the case of a rewrite step given by using a nonleft-linear rule r, the labeling procedure formalized in Definition 7.4.1 is extended as follows:*

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

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

Example 7.4.4

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

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

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

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

7.4.2 Built-in Operators

In practical implementations of RWL (e.g., Maude [CDE+07]), several commonly used operators are pre-defined (e.g., arithmetic and boolean operators, if-then-else constructs). Obviously, backward trace slicing of function calls involving built-in operators is not supported by our basic technique. This would require an explicit (rule-based or equational) specification of every single operator involved in the execution trace. To overcome this limitation, we further extend the labeling procedure of Definition 7.4.3 in order to deal with built-in operators.

Definition 7.4.5 *(rewrite step labeling procedure for built-in operators) For the case of a rewrite step $\mu : C[op(t_1, \ldots, t_n)] \rightarrow C[t']$ involving a call to a built-in, n-ary operator op, we extend Definition 7.4.3 by introducing the following additional case:*

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

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

For example, by applying Definition 7.4.5, the addition of two natural numbers implemented through the built-in operator + might be labeled as $+^{\alpha}(7^{\beta}, 8^{\gamma}) \rightarrow 15^{\beta\gamma}$.

7.4.3 Associative-Commutative Axioms

Let us finally consider an extended rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where B is a set of associativity (A) and commutativity (C) axioms that hold for some function symbols in Σ. As described in Section 7.2, an execution trace in $\mathcal{R}$ may contain rewrite steps modulo B that have the form $t =_B t' \rightarrow t''$, where $=_B$ is the congruence relation induced by the set of axioms B. Now, since B only contains associativity/commutativity (AC) axioms, terms can be represented by means of a single representative of

their AC congruence class, called *AC canonical form* [Eke03]. This representative is obtained by replacing nested occurrences of the same AC operator by a flattened argument list under a variadic symbol, whose elements are sorted by means of some linear ordering. In other words, if a function symbol f is declared to be associative, then the subterms rooted by f of any term t are flattened; and if f is also commutative, the subterms are sorted with respect to a fixed (internal) ordering [3].

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

For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the B-equivalence $f(b, f(f(b, a), c)) =_B f(f(b, c), f(a, b))$, we can represent it by using the "internal sequence"

$$f(b, f(f(b, a), c)) \rightarrow^*_{flat_B} f(a, b, b, c) \rightarrow^*_{unflat_B} f(f(b, c), f(a, b))$$

where the first one corresponds to the *flattening* transformation sequence that obtains the AC canonical form, while the second one corresponds to the inverse, unflattening one.

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

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

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

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

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

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

Definition 7.4.6 *(AC Labeling.) Let f be an associative-commutative operator, and let B be the AC axioms for f. Consider the B-equivalence $t_1 =_B t_2$ and the corresponding (internal) flat/unflat transformation $\mathcal{T}$: $t_1 \rightarrow^*_{flat_B} s \rightarrow^*_{unflat_B} t_2$. Let L be an initial labeling for t_1. The labeling procedure for $t_1 =_B t_2$ is as follows.*

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

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

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

Example 7.4.7

Consider the transformation sequence

$$f(b, f(b, f(a, c))) \rightarrow^*_{flat_B} f(a, b, b, c) \rightarrow^*_{unflat_B} f(f(b, c), f(a, b))$$

by using Definition 7.4.6, the associated transformation sequence can be labeled as follows:

$$f^{\alpha}(b^{\beta}, f^{\gamma}(b^{\delta}, f^{\epsilon}(a^{\zeta}, c^{\eta}))) \rightarrow^*_{flat_B} \quad f^{\alpha\gamma\epsilon}(a^{\zeta}, b^{\beta}, b^{\delta}, c^{\eta}) \rightarrow^*_{unflat_B} f^{\alpha\gamma\epsilon}(f^{\alpha\gamma\epsilon}(b^{\beta}, c^{\eta}), f^{\alpha\gamma\epsilon}(a^{\zeta}, b^{\delta}))$$

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

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

7.4.4 Extended Soundness

Soundness of the backward trace slicing algorithm for the extended rewrite theories is established by the following theorem which properly extends Theorem 7.3.15.

Theorem 7.4.8 *(extended soundness) Let $\mathcal{R} = (\Sigma, E, R)$ be an extended rewrite theory. Let $\mathcal{T}$ be an execution trace in the rewrite theory $\mathcal{R}$, and let $\mathcal{O}$ be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^\bullet : t_0^\bullet \xrightarrow{r_1} t_1^\bullet \ldots \xrightarrow{r_n} t_n^\bullet$ be the corresponding trace slice w.r.t. $\mathcal{O}$. Then, for any concretization t'_0 of $t_0^\bullet$, it holds that $\mathcal{T}' : t'_0 \xrightarrow{r_1} t'_1 \ldots \xrightarrow{r_n} t'_n$ is an execution trace in $\mathcal{R}$, and $t_i^\bullet \propto t'_i$, for $i = 1, \ldots, n$.*

Proof of Theorem 7.4.8

In oder to prove Theorem 7.4.8, we use the same proof scheme as for elementary rewrite theories, since the extended technique described in Section 7.4 is only concerned with suitable extensions of the labeling procedure given in Definition 7.3.5, which do not affect the overall backward trace slicing methodology.

Let us start by proving an extension of Lemma 7.3.18 (Claim 2), which holds for nonleft-linear as well as collapsing rules.

Lemma 7.4.9 *Let $r : \lambda \to \rho$ be a rule that is either nonleft-linear or collapsing. Let $\mu : s \xrightarrow{r,\sigma} t$ be a rewrite step such that $s = C[\lambda\sigma]$ and $t = C[\rho\sigma]$, where σ is a substitution and C is a context. Let L be a labeling for the rewrite step μ, and $w \in \mathcal{P}os(t)$. Then,*

1. *if $w \in \mathcal{P}os(C)$, then $\lhd_\mu^L w = \{v \in \mathcal{P}os(C) \mid w = v.v'\}$*
2. *if $w = w'.w''$, $C_{|w'} = \Box$, and $w'' \in \mathcal{P}os(\rho\sigma)$, then $\lhd_\mu^L w \supseteq \{w'.v' \in \mathcal{P}os(s) \mid v' \in \mathcal{P}os(\lambda)\}$*

Proof. We prove the two claims separately.
Claim 1. The proof is identical to the proof of Claim 1 of Lemma 7.3.18.
Claim 2. To prove the lemma, we distinguish three cases.

Case 1: Rule r is collapsing. Given the collapsing rule $r = \lambda \rightarrow \rho$ where $\rho = x$ with $x \in Var(\lambda)$, let us consider the term t_i introduced by the substitution σ via the binding x/t_i, and we have $\mu = C[\lambda\sigma] \xrightarrow{r} C[t_i]$. Let us also consider the labeled rewrite step $\mu^L : s^L \overset{r^{L_r},\sigma^{L_\sigma}}{\rightarrow} t^L$ via the labeling L. By Definition 7.3.5, we have $s^L = C^{L_C}[\lambda^{L_r}\sigma^{L_\sigma}]$ and $t^L = C^{L_C}[t_i^{L_\sigma}]$.

Let $f^{l'}$ be the labeled root symbol of $t_i^{L_\sigma}$. By Definition 7.4.1 (Step s_4), we have that $l' = l_\lambda l_i$, where l_λ is formed by joining all the labels appearing in the redex pattern λ^{L_r} and l_i is the label of the root of the labeled term $t_i^{L_\sigma}$. This implies that, for each labeled symbol g^l in the redex pattern of r, we have that $l \subseteq l'$. Furthermore, by hypothesis, we have that $w \in C[t_i]$ and $w'' \in Pos(t_i)$. Hence, by Definition 7.3.7, the inclusion $\lhd_\mu^L w \supseteq \{v.v' \in \mathcal{P}os(s) \mid v' \in \mathcal{P}os(\lambda)\}$ trivially holds.

Case 2: rule r is nonleft-linear. Given the nonleft-linear rule r, the proof is perfectly analogous to the proof of Lemma 7.3.18 since, by Definition 7.4.3 (Step s_5), the label of each symbol in the contractum pattern of the rule r includes all the labels appearing in the redex pattern of r.

Case 3: rule r is collapsing and nonleft-linear. Since r is both collapsing and nonleft-linear, μ is labeled according to Definition 7.4.1 (Step s_4) and Definition 7.4.3 (Step s_5). Therefore, we can prove the claim by simply combining the arguments used to prove Case 1 ad Case 2.

■

The following Lemma extends Lemma 7.3.19 to deal with collapsing and nonleft-linear rules.

Lemma 7.4.10 *Let $r : \lambda \rightarrow \rho$ be a rule which is either left-linear or collapsing. Let $\mu : t_0 \overset{r,\sigma}{\rightarrow} t_1$ be a rewrite step such that $t_0 = C[\lambda\sigma]$*

and $t_1 = C[\rho\sigma]$*, where* σ *is a substitution and* C *is a context. Let* L *be a labeling for the rewrite step* μ*, and* $[P_0, P_1]$ *be the sequence of the relevant position sets for* $\mu : t_0 \overset{r,\sigma}{\rightarrow} t_1$ *w.r.t. the slicing criterion* $\mathcal{O}$*. Let* $t_0^\bullet = slice(t_0, P_0)$*, and* $t_1^\bullet = slice(t_1, P_1)$*. Then,*

1. *if* $P_1 \subseteq \mathcal{P}os(C)$ *then* $t_0^\bullet = t_1^\bullet$*.*
2. *if* $P_1 \cap \{w | w = v.v', C_{|v} = \Box, \text{ and } v' \in \mathcal{P}os(\rho\sigma)\} \neq \emptyset$*, then for any concretization* t_0' *of* $t_0^\bullet$*, we have that* $t_0' \overset{r,\sigma'}{\rightarrow} t_1'$ *where* $t_1^\bullet \propto t_1'$*.*

Proof. We prove the two claims separately.

Claim 1. The proof is identical to the proof of Claim 1 of Lemma 7.3.19.
Claim 2. To prove the lemma, we distinguish three cases.

Case 1: rule r is collapsing. Given the collapsing rule r, the proof is perfectly analogous to the one of Lemma 7.3.19 Claim 2. By using Lemma 7.4.9 instead of Lemma 7.3.18, we are still able to prove that the redex pattern of r embedded in t_0 is also embedded in $t_0^\bullet$, and hence for any concretization t_0' of $t_0^\bullet$, the rewrite step $t_0' \overset{r,\sigma'}{\rightarrow} t_1'$ can be proved. Finally, by using the same argument of Lemma 7.3.19 Claim 2, we conclude that $t_1^\bullet \propto t_1'$.

Case 2: rule r is nonleft-linear. Given the nonleft-linear rule r, the proof is similar to the one of Lemma 7.3.19. By exploiting Lemma 7.4.9 and Definition 7.4.3 (Step s_5), we can show that (i) the redex pattern of r embedded in t_0 is also embedded in $t_0^\bullet$, and (ii) for each term t introduced in t_0 by a binding $x/t \in \sigma$ such that x occurs multiple times in λ, t is preserved in $t_0^\bullet$ (i.e., t is not abstracted by $\bullet$ in $t_0^\bullet$). By (i) and (ii), it is immediate to prove that, for any concretization t_0' of $t_0^\bullet$, the rewrite step $t_0' \overset{r,\sigma'}{\rightarrow} t_1'$ can be proved. Finally, by using the same argument of Lemma 7.3.19 Claim 2, we can show that $t_1^\bullet \propto t_1'$.

Case 3: rule r is collapsing and nonleft-linear. Firstly we observe that, as the rule r is collapsing, by Lemma 7.4.9 the redex pattern of r embedded in t_0 is also embedded in $t_0^\bullet$, and hence for any concretization t_0' of $t_0^\bullet$, the redex pattern of r is embedded in t_0' as

well. Secondly, since r is nonleft-linear, by Lemma 7.4.9 and Definition 7.4.3 (Step s_5), for each term t introduced in t_0 by a binding $x/t \in \sigma$ such that x occurs multiple times in λ, t is preserved in $t_0^\bullet$. Hence, t is also embedded in t'_0, for any concretization t'_0 of $t_0^\bullet$. From the two facts above, it directly follows that for any t'_0 such that $t_0^\bullet \propto t'_0$, the rewrite step $t'_0 \overset{r,\sigma'}{\rightarrow} t'_1$ can be proved. Finally, by using the same argument of Lemma 7.3.19 Claim 2, we can show that $t_1^\bullet \propto t'_1$.

■

The following proposition allows us to prove the soundness of our methodology for one-step traces on an extended rewrite theory.

Proposition 7.4.11 *Let $\mathcal{R}$ be an extended rewrite theory. Let $\mathcal{T} : t_0 \overset{r_1}{\rightarrow} t_1$ be an execution trace in $\mathcal{R}$, and let $\mathcal{O}$ be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^\bullet : t_0^\bullet \overset{r_1}{\rightarrow} t_1^\bullet$ be the trace slice w.r.t. $\mathcal{O}$ of $\mathcal{T}$. Then, for any concretization t'_0 of $t_0^\bullet$, it holds that $\mathcal{T}' : t'_0 \overset{r_1}{\rightarrow} t'_1$ is an execution trace in $\mathcal{R}$ such that $t_1^\bullet \propto t'_1$.*

Proof. Consider the rewrite step $\mu : t_0 \overset{r_1}{\rightarrow} t_1$. In the case when r_1 is left-linear and non-collapsing (i.e., a rule belonging to an elementary rewrite theory), the proof is identical to the proof of Proposition 7.4.11. Hence w.l.o.g. we assume that r corresponds to a collapsing or nonleft-linear rule, built-in operator evaluation, or AC axiom.

Nonleft-linear/collapsing rules. In this case, the proof of Proposition 7.4.11 is analogous to the proof of Proposition 7.3.20, by using Lemma 7.4.10 in the place of Lemma 7.3.19.

Built-in Operators. Let $t_0 = C[op(t_1, \ldots, t_m)]$ and $t_1 = C[t']$. Hence, $\mu : C[op(t_1, \ldots, t_m)] \rightarrow C[t']$ is a rewrite step mimicking the evaluation of the built-in operator call $op(t_1, \ldots, t_m)$. By Definition 7.4.5 and Definition 7.3.7, it is immediate to show that $op(t_1, \ldots, t_m)$ is embedded in $t_0^\bullet$, and thus for any concretization $t_0^\bullet \propto t'_0$, $t'_0 \overset{r_1}{\rightarrow} t'_1$ and $t_1^\bullet \propto t'_1$.

Associative-Commutative Axioms. Flat/unflat transformations are interpreted as rewrite steps that reduce AC symbols. Let us first consider the flat transformation $t \rightarrow_{flat_B} t'$ that reduces the AC symbol f. By Definition 7.4.6, the label of the occurrence of f in t' contains all the labels of the different occurrences of f appearing in t that have been reduced by the transformation. In other words, the label of f in t' keeps track of all the occurrences of f that have been reduced in t, and therefore the claim holds directly. The claim for unflat transformations can be proved in a similar way.

■

Finally, we exploit Proposition 7.4.11 in order to prove the extended soundness of our methodology on extended rewrite theories.

Theorem 7.4.8. *(extended soundness) Let $\mathcal{R} = (\Sigma, E, R)$ be an extended rewrite theory. Let $\mathcal{T}$ be an execution trace in the rewrite theory $\mathcal{R}$, and let $\mathcal{O}$ be a slicing criterion for $\mathcal{T}$. Let $\mathcal{T}^\bullet : t_0^\bullet \stackrel{r_1}{\rightarrow} t_1^\bullet \ldots \stackrel{r_n}{\rightarrow} t_n^\bullet$ be the corresponding trace slice w.r.t. $\mathcal{O}$. Then, for any concretization t'_0 of $t_0^\bullet$, it holds that $\mathcal{T}' : t'_0 \stackrel{r_1}{\rightarrow} t'_1 \ldots \stackrel{r_n}{\rightarrow} t'_n$ is an execution trace in $\mathcal{R}$ and $t_i^\bullet \propto t'_i$, for $i = 1, \ldots, n$.*

Proof. The proof proceeds by induction on the length of the trace slice $\mathcal{T}^\bullet$ and exploits Proposition 7.4.11 in order to prove the inductive case. Routine. ■

7.5 Experimental Evaluation

We have developed a prototype implementation of our slicing methodology which is publicly available at `http://www.dsic.upv.es/~dromero/slicing.html`. The implementation is written in Maude and consists of approximately 800 lines of code. Maude is a high-performance, reflective language that supports both equational and rewriting logic programming, which is particularly suitable for developing domain-specific applications [EMS03; EMM06]. The reflection capabilities of Maude allow metalevel computations in RWL to be handled at the object-level. This facility allows us to easily manipulate computation traces of Maude itself and eliminate the irrelevant contents by implementing the backward slicing

procedures that we have defined in this chapter. Using reflection to implement the slicing tool has one important additional advantage, namely, the ease to rapidly integrate the tool within the Maude formal tool environment [CDH+07], which is also developed using reflection.

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

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

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

Fault-Tolerant Communication Protocol (FTCP)
: FTCP is a Maude specification borrowed from [Mes08] that models a fault-tolerant, client-server communication protocol. There can be many clients and many servers, where a server can serve many clients; however, each client communicates with a single server. Also, the communication environment might be faulty —that is, messages can arrive out of order, can be duplicated, or can be lost.

Web-TLR: The Web application verifier
: Web-TLR [ABR09; ABER10] is a software tool designed for model-checking real-size Web applications (Web-mailers, Electronic forums, *etc.*) which is based on rewriting logic. Web applications are expressed as rewrite theories which can be formally verified by using the Maude built-in LTL(R) model checker [BM08].

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

We have tested our tool on some execution traces which were generated by the Maude applications described above by imposing different slicing criteria. For each application, we considered two execution traces

that were sliced using two different criteria. Table 7.1 summarizes the results we achieved.

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

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

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

For each criterion, Table 7.1 shows the size of the original trace and that of the computed trace slice, both measured as the length of the corresponding string. The *%reduction* column shows the percentage of reduction achieved. These results are very encouraging, and show an impressive reduction rate (up to $\sim 95\%$). Actually, sometimes the trace slices are small enough to be easily inspected by the user, who can restrict her attention to the part of the computation she wants to observe getting rid of those data which are useless or even noisy w.r.t. the considered slicing criterion.

7.6 Related Work

Our backward tracing relation (Definition 7.3.7) extends a previous tracing relation that was formalized in [BKdV00] for orthogonal TRSs. In [BKdV00], a label is formed from atomic labels by using the operations of sequence concatenation and underlining, which are used to record every rule application (e.g., a, b, ab, $\underline{\underline{abcd}}$, are labels). Collapsing rules are

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

Table 7.1: Summary of the reductions achieved.

simply avoided by coding them away. This is done by replacing each collapsing rule $\lambda \to x$ with the rule $\lambda \to \varepsilon(x)$, where ε is a unary dummy symbol. Then, in order to lift the rewrite relation to terms containing ϵ occurrences, infinitely many new extra-rules are added that are built by saturating all left-hand sides with $\varepsilon(x)$. In contrast to [BKdV00], we use a more sophisticated notion of labeling, where composite labels are interpreted as sets of atomic labels, which allows us to deal with collapsing as well as nonleft-linear rules in an effective way.

Two works are closely related to our approach [FT94; TeR03]. [FT94] formalizes a notion of dynamic dependence among symbols by means of contexts and studies its application to program slicing of TRSs that may include collapsing as well as nonleft-linear rules. Both the *creating* and the *created* contexts associated with a reduction (i.e., the minimal subcontext that is needed to match the left-hand side of a rule and the minimal context that is "constructed" by the right-hand side of the rule, respectively) are tracked. Intuitively, these concepts are similar to our notions of redex and contractum patterns. The main differences with respect to our work are as follows. First, in [FT94] the slicing is given as a context, while we consider term slices. Second, the slice is obtained only on the first term of the sequence by the transitive and reflexive closure of the dependence relation, while we slice the whole execution trace, step by step. Obviously, their notion of slice is smaller, but we think that our approach can be more useful for trace analysis and program debugging.

[TeR03] carries out an in-depth analysis of reduction equivalence by

adopting labeling and tracing techniques. Specifically, [TeR03] describes a methodology of static and dynamic tracing which is mainly based on the notion of *sample of a traced proof term* —i.e., a pair (μ, P) that records a rewrite step $\mu = s \to t$, and a set P of reachable positions in t from a set of observed positions in s. The tracing proceeds forward, while ours employs a backward strategy which is particularly convenient for trace analysis.

Finally, [FT94] and [TeR03] apply to TRSs whereas we deal with the richer framework of RWL that considers equations and equational axioms as well as a more sophisticated rewrite relation, namely rewriting modulo equational theories.

7.7 Conclusions

Trace slicing has been widely studied in imperative languages, where the dependence among program statements is generally determined by a program dependency graph (e.g., see [RH05] and the references therein). However, the notion of "dependence" in term rewriting languages, and particularly in RWL, is much more involved due to the combination of equations and rules. To the best of our knowledge, no trace slicing methodology for rewriting logic theories has yet been proposed.

The key idea behind our backward trace slicing technique consists in tracing back —through the rewrite sequence— all the relevant symbols of the final state that we are interested in. Given a slicing criterion $\mathcal{O}$ for a trace $\mathcal{T}$, our algorithm computes a trace slice $\mathcal{T}^\bullet$ that contains only the relevant information of $\mathcal{T}$ with respect to $\mathcal{O}$. The trace slicing technique can be applied to execution trace analysis of sophisticated rewrite theories, which can include equations and equational axioms as well as nonleft-linear and collapsing rules.

The proposed slicing technique has been implemented in a prototype system. Preliminary experiments demonstrate that the system works very satisfactorily on our benchmarks; e.g., we obtained trace slices that achieved a reduction of up to almost 95% in reasonable time (max. 0.5s on a Linux box equipped with an Intel Core 2 Duo 2.26GHz and 4Gb of RAM memory). Naturally, there is still much room for improvement, and we are currently working on increasing the efficiency of the tool. Also, as

future work, we plan to deal with the execution traces of more sophisticated theories that may include membership and conditional equations.

Chapter 8

Model-checking Web Applications with Web-TLR

This chapter describes Web-TLR [ABER10], which is a model-checking tool that implements the theoretical framework of Chapter 6. Web-TLR is written in Maude and is equipped with a freely accessible graphical Web interface written in Java, which allows users to introduce and check their own specification of a Web application, together with the properties to be verified. In order to check the properties against the specifications, the Maude built-in LTLR model-checker is used. In the case when the property is proven to be false (refuted), an online facility can be invoked that dynamically generates the navigation trace (from the counter-example trace given by the model-checker), which is ultimately responsible for the erroneous Web application behavior. In order to improve the understandability and usability of the system and since the textual information associated with counter-example traces is usually rather large and poorly readable, the checker has been endowed with the capability to generate and display on-the-fly slideshows that allow the erroneous navigation trace to be visually reproduced step by step. This graphical facility provides deep insight into Web application behavior and is extremely effective for debugging purposes. Web-TLR focuses on the Web application tier (business logic), and thus handles server-side scripts.

To provide better support for the analysis of counter-example traces and since the counter-examples are expressed as a sequence of rewrite steps, Web-TLR is also coupled with a slicing tool that implements the backward trace slicing technique presented in Chapter 7 that allows one to accurately identify those parts of the trace that influence a given slicing criterion.

8.1 The Web-TLR System

Our verification methodology for dynamic Web applications has been implemented in the Web-TLR system by using the high-performance, rewriting logic language Maude [CDE+07] (around 750 lines of code without including third-party components). Web-TLR is available on-line via its friendly Web interface at `http://www.dsic.upv.es/~dromero/web-tlr.html`. The Web interface frees users from having to install any application on their local computer and hides the hardest technical details of the tool operation. After introducing the Maude specification of a Web application (which can be done by customizing a default one), together with an initial Web state st_0 and the LTLR formula φ to be verified, φ can be automatically checked at state st_0. Once all inputs have been entered into the system, we can automatically check the considered 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.

In the case when φ is refuted by the model-checker, a counter-example is provided, which is expressed as a model-checking computation trace starting from st_0. The counter-example is graphically displayed by means of an interactive slideshow that supports forward and backward navigation through the computation's Web states. Each slide contains a graph that models the structure of (a part of) the Web application. The nodes of the graph represent the Web pages, and the edges that connect the Web pages specify Web links or Web script continuations[1]. The graph also shows the current Web page of each active Web browser. The graphical representation is combined with a detailed textual description of the current configurations of the Web server and the active Web browsers. A snapshot of Web-TLR showing a slide that displays part of a counter-example is given in Figure 8.5.

[1] To obey the stateless nature of the Web, the structure of Web applications has traditionally been "inverted", resembling programs written in a continuation–passing style [GFKF03].

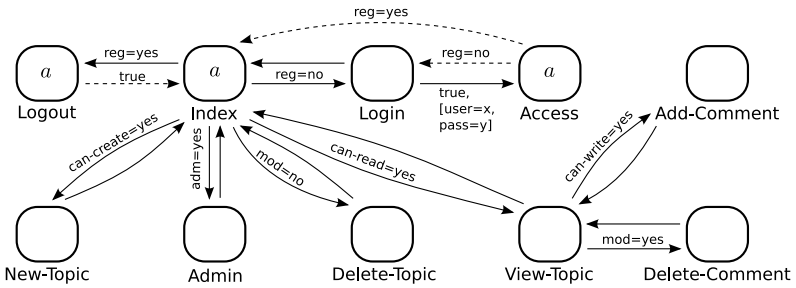


Figure 8.1: The navigation model of an Electronic Forum

8.2 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.

The graphical navigation model (see Section 6.1.1) of such an application is given in Figure 8.1.

In LTLR, we can define the following state predicates as boolean functions:

- failedAttempt(bid,n), which holds when browser bid has performed n failed login attempts (this is achieved by recording in the state a counter n with the number of failed attempts);
- curPage(bid,p), which holds when browser bid is currently displaying the Web page p;
- inconsistentState, which holds when two browser windows or tabs of the same browser refer to different user sessions.

These elementary state predicates are used below to build more complex LTLR formulas expressing mixed properties that include dependencies among states, actions, and time. These properties intrinsically involve both action-based and state-based aspects which are either not expressible or are difficult to express in other temporal logic frameworks.

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

Property A. *Incorrect login attempts are allowed only k times; then login is denied.*

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

Note the sugared syntax (which is allowed in LTLR) when using relational notation for the state predicates which were defined as boolean functions above. Note also that the property is defined parametrically w.r.t. the number of login attempts, which depends on the execution of a script that sets the counter k with the number of failed attempts.

Property B. *No two administrators can access the administration page simultaneously.*

$$\Box\neg(\mathsf{curPage}(\mathsf{A},\mathsf{Admin}) \wedge \mathsf{curPage}(\mathsf{B},\mathsf{Admin}))$$

This mutual exclusion property sets that a state should not be reached where the current Web page of two users A and B is the administration page.

Property C. *All links refer to existing Web pages* (i.e., absence of broken links).

$$\Box\neg\,\mathsf{curPage}(\mathsf{A},\mathsf{PageNotFound})$$

This is an accessibility property on the Web application. Roughly speaking, a user should never reach the Web page PageNotFound.

Property D. *We do not want to reach a Web application state in which two browser windows refer to different user sessions.*

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

Since inconsistent states are defined when a user is using two browsers with different information in the corresponding browser sessions, this property allows us to solve/handle the multiple windows problem.

The detailed specification of the electronic forum, together with some example properties are available at `http://www.dsic.upv.es/~dromero/web-tlr.html`.

8.3 Web-TLR Graphical Web interface

The WEB-TLR online checker is a Web tool that allows us to specify and verify Web applications in a simple and friendly way. WEB-TLR is equipped with a user-friendly, graphical Web interface that shields the user from unnecessary technical information. Whenever a property is refuted, an interactive slideshow is generated that allows the user to visually reproduce, step by step, the erroneous navigation trace that underlies the failing model checking computation. This provides deep insight into the system behavior, which helps to debug Web applications. In the following we describe the major features of the WEB-TLR online checker.

We consider again the Web application that implements an electronic forum given in Section 8.2. In order to model-check the considered Web application, the user must provide the following inputs by using the WEB-TLR Graphical Web interface: *i*) Web application and state predicates; *ii*) initial Web states; *iii*) selected initial state; *iv*) property (LTLR formula) to be checked. A snapshot of the input form of the WEB-TLR user interface is given in Figure 8.4.

i) **Web application and state predicates.** In this field, two Maude modules must be entered that respectively contain the Web application navigation model and the definitions of the desired state predicates. Optionally, these modules can be loaded from a file.

Figure 8.2 details (the core part of) the Maude module specifying the navigation model of the Electronic Forum example. Figure 8.3 displays the Maude definition of the state predicate curPage(bid,p), which holds when browser bid is currently displaying the Web page p.

ii) **Initial Web states.** This field is filled with a Maude module specifying the initial Web state, including the active browsers, the substitution (predefined answers used to feed user input in forms) associated to each of them, the communication channel, the Web server and the data base connected with the Web application.

The initial state describes the system at the start of the model checking process, and it is possible to define more than one initial

```
(fmod WEBAPP is inc PROTOCOL .
 --- Names of the web pages
 ops INDEX LOGIN ACCESS LOGOUT ADMIN ADDCOMMENT DELCOMMENT VIEWTOPIC
     NEWTOPIC DELTOPIC : -> Qid .

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

 --- The access page processes the login request.
 --- Upon success, the user is redirected to the index page.
 --- Otherwise, the user is redirected back to the login page.
 op accessPage : -> Page .
 eq accessPage = (ACCESS, accessScript, {((s("reg")'== s("yes")) => INDEX)
                   : ((s("reg")'== s("no")) => LOGIN)}, {nav-empty}) .

 op accessScript : -> Script .
 eq accessScript =
   'u := getQuery('user) ;                  --- get user name
   'p := getQuery('pass) ;                  --- get password
   'p1 := selectDB('u) ;                    --- get the password from DB
   if ( 'p = 'p1 ) then                     --- check password
     'r := selectDB( 'u '. s("-role") ) ;   --- get user role
      setSession( s("reg"), s("yes") ) ;    --- set user capabilities
   fi
    .
  --- ...
endfm)
```

Figure 8.2: Maude specification of the navigation model

```
(fmod WEBAPP-CHECK is pr TLR[WEBAPP] .
 subsorts WebState < State .
 vars idb idw : Id .     vars z : Sigma .          vars urls : URL .
 vars sv : Server .      vars ms lms : Message .   vars ss : Session .
 vars brs : Browser .    vars h : History .        vars idlm : Nat .
 vars page : Qid .       vars ba : BrowserActions .

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

 eq  [brs] ba [ms] [sv] |= curPage(idb, page) = false [owise] .
endfm)
```

Figure 8.3: Maude specification of the curPage state predicate

state which are univocally identified by a label, but only one is selected for checking the property. In the electronic forum example, we have defined several alternative initial configurations that the

user can play with.

iii) **Selected initial state.** In this field, we insert the label of the chosen initial Web state.

As an initial Web state example, consider the Web state that is labeled with name initial in Figure 8.1, which logs two administrator users (whose identifiers are bidAlfred and bidAnna, respectively) onto the Electronic forum.

iv) **Property (LTLR formula) to be checked.** In this field, the user has to enter the property s/he wants to check. This property must be expressed as an LTLR formula in Maude notation, and is built up using the LTLR logic operators along with the state predicates introduced in the first field.

By using the state predicates defined above, here we define in LTLR syntax the property to be verified by means of WEB-TLR. For example, the mutual exclusion property *"No two administrators can access the administration page simultaneously"* is given by the following LTLR formula:

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

Finally, when all the textual fields have been filled in, we can automatically check the property by just clicking the button Check, which invokes the Maude built-in operator tlr check [BM08] that supports model checking of rewrite theories w.r.t. LTLR formulas.

The result of the verification is displayed in a new, dynamically generated Web page. Since the above property is not satisfied, a counterexample trace with the erroneous information is delivered. This counterexample is expressed as a navigation trace (rewrite sequence) and can be navigated by using an interactive slideshow, which is illustrated in Figure 8.5.

Roughly speaking, each slide contains a graph and a table. The graph shows which pages are being viewed by which browsers in the sequence of states represented by the table below the graph. An asterisk (∗) adjacent to the `ID` of a browser indicates an HTTP request has been done. A

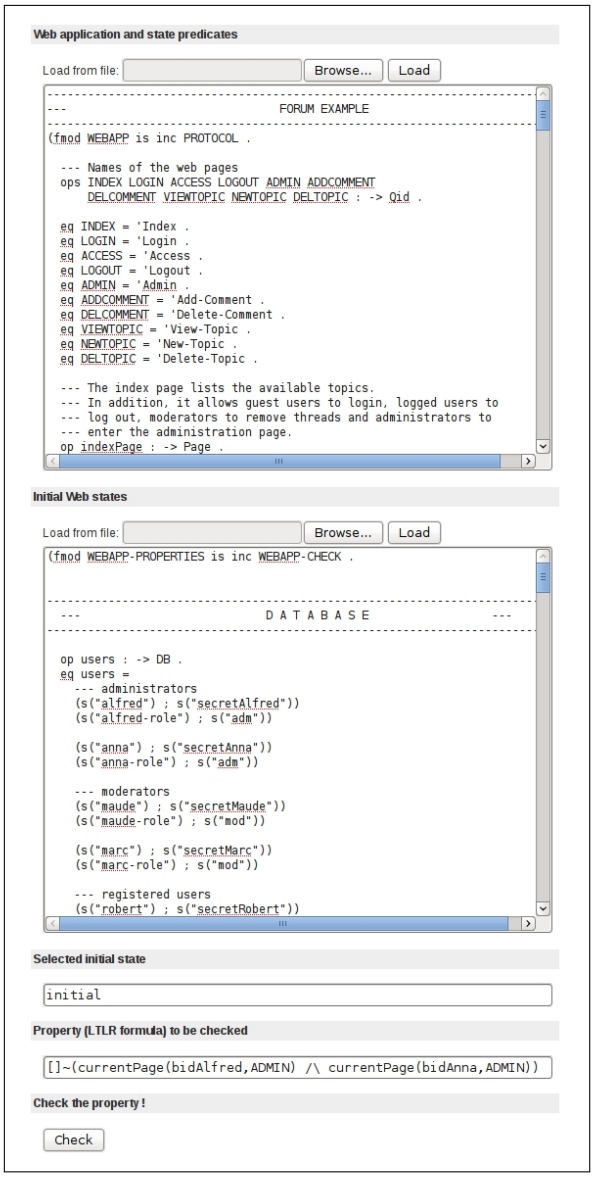


Figure 8.4: Electronic Forum Application in Web-TLR

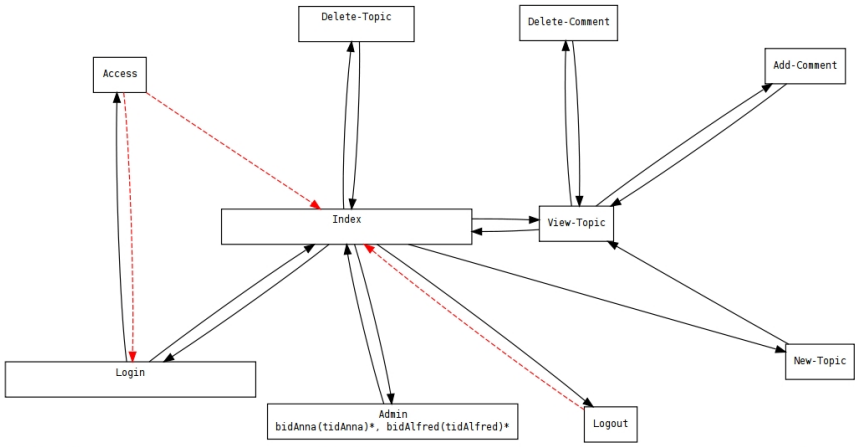
question mark (?) means that the browser is blocked, and no navigation can occur from the current page.

Model checking results

Brief information

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

Slide 17 of 57 First Previous Next Last



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

Figure 8.5: Slideshow of the WEB-TLR execution

8.4 Debugging of Web Applications

Let us discuss how we can extend Web-TLR in order to support the debugging of Web applications. As we mentioned before, if a property is not satisfied, a counter-example trace is returned, which shows the erroneous trace. This trace is expressed as a sequence of rewrite steps that leads from the initial state to the state that violates the property. For example, by applying WEB-TLR to the electronic forum example of Section 8.2, the property $\Box\neg(\mathsf{curPage}(\mathsf{bidAlfred},\mathsf{Admin})\wedge\mathsf{curPage}(\mathsf{bidAnna},\mathsf{Admin}))$ is refuted and one counter-example trace is delivered, which consists of a sequence of 143 states. Figure 8.6 shows one of those 143 states witnessing that a manual debugging of the counter-example trace is impracticable.

In Chapter 7, a *backward-tracing slicing* technique for rewriting theories is developed. Roughly speaking, this technique consists in tracing back, over a rewrite sequence, all the relevant symbols of the term (or state) that we are interested in. This way, the user can pay attention only to those subterms of the state that s/he wants to inspect. This technique has been implemented in a tool that supports the manipulation of traces of rewrite theories that are written in Maude. By coupling WEB-TLR with this tool, we can focus on the relevant information of each Web state and reduce the effort that is required for debugging the Web application via the analysis of a given counter-example trace.

8.4.1 Debugging: A Case Study

Let us consider again the electronic forum example of Section 8.2 with the initial state that consists of two administrator users whose identifiers are $\mathsf{bidAlfred}$ and $\mathsf{bidAnna}$, respectively. Let us also consider the mutual exclusion property

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

which states that "no two administrators can access the administration page simultaneously". Note that the predicate state $\mathsf{curPage}(\mathsf{bidAlfred},\mathsf{Admin})$ holds when the user $\mathsf{bidAlfred}$ logs into the Admin page (a similar interpretation is given to predicate $\mathsf{curPage}(\mathsf{bidAnna},\mathsf{Admin})$. By verifying the above property in WEB-TLR, we get a huge counter-example

{[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 8.6: One state of the counter-example trace of Section 8.2.

trace that proves the property is not satisfied. The trace size weighs around $190kb$.

In the following, we show how we can debug the Web application by reducing and inspecting the counter-example trace delivered by WEB-TLR. First of all, we specify the slicing criterion to be applied in order to reduce the counter-example trace. Then, by running our backward slicing tool, a slice trace is obtained. Finally, we analyze the slice trace and show how this analysis can help the user to locate errors.

Slicing Criterion

In the context of Web-TLR, we recall that the considered *execution trace* $\mathcal{T}$ is the counter-example trace produced as the outcome of the built-in model-checker of Web-TLR. The slicing criterion represents the information that we want to trace back through the execution trace $\mathcal{T}$. The slicing criterion is defined by specifying the positions where the relevant information is located within the state that we are observing.

For example, consider the Web state s_n showed in Figure 8.6. In this Web state the two users, `bidAlfred` and `bidAnna`, are logged into the `Admin` page[2]. Therefore, the considered mutual exclusion property has been violated. Let us assume that we want to diagnose the pieces of information within the execution trace $\mathcal{T}$ that contribute to this mistake. For this purpose, the slicing criterion $\mathcal{O}_{\mathcal{T}}$ can be defined as follows:

$$\{1.1.1,\ 1.1.3,\ 1.2.1,\ 1.2.3\}$$

where 1.1.1 and 1.2.1 are the positions in s_n of the user identifiers `bidAlfred` and `bidAnna` respectively, and 1.1.3 and 1.2.3 are the positions in s_n that indicate the users are logged into the `Admin` page.

Trace Slice

The slicing technique proceeds backwards, from the observable state s_n to the initial state s_0, and recursively generates, for each state s_i, a sliced state $s_i^\bullet$ that only 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 $\mathcal{O}_{\mathcal{T}}$ 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 8.7 shows a part of the trace slice $\mathcal{T}^\bullet$.

Trace Slice Analysis

Let us show how it is possible to trace back the information along the trace slice $\mathcal{T}^\bullet$. In order to facilitate the understanding, the main symbols involved in the description are underlined in Figure 8.7.

[2]The detailed explanation of the structure of a state can be found in Section 6.2.2.

$$\mathcal{T}^\bullet = \ldots \rightarrow s^\bullet_{n-6} \overset{ScriptEval}{\rightarrow} s^\bullet_{n-5} \overset{flat/unflat}{\rightarrow} s^\bullet_{n-4} \overset{ResIni}{\rightarrow} s^\bullet_{n-3}$$

$$\overset{flat/unflat}{\rightarrow} s^\bullet_{n-2} \overset{ResFin}{\rightarrow} s^\bullet_{n-1} \overset{flat/unflat}{\rightarrow} s^\bullet_n$$

where

$s^\bullet_n$= [B(bidAlfred, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗) : B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)]∗ [∗] [∗]

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

$s^\bullet_{n-2}$= [B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗) : B(bidAlfred, tidAlfred, ∗, ∗, ∗, ∗, ∗, ∗, 1)]∗
[m(bidAlfred, tidAlfred, Admin, ∗, ∗, 1) : ∗] [∗]

$s^\bullet_{n-3}$= [B(bidAlfred, tidAlfred, ∗, ∗, ∗, ∗, ∗, ∗, 1) : B(bidAnna, ∗, Admin, ∗, ∗, ∗, ∗, ∗, ∗)]∗
[∗ : m(bidAlfred, tidAlfred, Admin, ∗, ∗, 1)] [∗]

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

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

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

Figure 8.7: Trace slice $\mathcal{T}^\bullet$.

- The sliced state $s^\bullet_n$ is the observable state recording only the relevant information defined by the slicing criterion.

- The slice state $s^\bullet_{n-1}$ is an intermediate state obtained from $s^\bullet_n$ by applying the flat/unflat transformation[3].

- 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 as well. 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,

[3]For more details, see Chapter 7.

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 in the server that is ready to be sent, whereas in the server 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 data base (`DB`) as input. The request message contains the query string that has been sent by the user `bidAlfred` to ask admission into the `Admin` page. Observe that the response message that is showed in the slice state $s^{\bullet}_{n-4}$ is the one given as 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.

Note that the outcome delivered by the operator `evalScript` 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 clearly indicate that the login web script presents an anomalous behavior that should be fixed.

To conclude, by using our backward trace slicing tool, the debugging of counter-example traces given by Web-TLR is more intuitive and easy. The integration of our slicing tool into Web-TLR is developed by using appropriate wrappers and a friendly interface as is described in [ABE+11].

Conclusions

Maintaining the consistency of Web contents is an open and urgent problem since outdated, incorrect, and incomplete information is becoming more and more frequent in the World Wide Web. Furthermore, the dynamic features of Web systems raise new challenges for Web verification techniques that intersect many other areas. In order to promote the technological transfer of such techniques to the industry, it is necessary to provide tools that are able to detect mistakes with precision and to give prompt solutions on real examples. In this thesis, we have presented some techniques that (hopefully) contribute a step forward in the Web verification area.

Static Web Verification

Repairing Faulty Web Sites. We presented a semi–automatic methodology for repairing Web sites that has a number of advantages over other approaches (and hence can be used as a useful complement to them). Here we highlight the main advantages of our repair methodology.

- Our methodology can be smoothly integrated on top of existing rewriting-based Web verification frameworks such as [ABF06], which offers the expressiveness and computational power of functions and allows one to avoid the encumbrances of DTDs and XML rule languages.
- By solving the constraint satisfaction problem associated to the conditions of the Web specification rules, we are also able to aid users to fix erroneous information by suggesting ranges of correct values.
- In contrast to the active database Web management techniques, we are able to predict whether a repair action can raise new errors, and hence assist the user in reformulating the action.

We also described a significant optimization that improves several aspects of our basic repair technique. More specifically,

- we provided a detailed analysis of the errors in a Web site that clarified the relation among them. By exploiting the results of such analysis, we formulated two correction/completeness strategies that reduce the number of repair actions and the amount of information needed to fix a given Web site;
- the considered correction/completeness strategies increase the level of automation of the repairing method, since the user just has to fix a small number of errors to make a Web site correct and complete whit respect to a given Web specification.

Automated Verification of Web Sites. We presented a powerful Web verification/repair engine called Verdi-M [ABE+07], which greatly improves the original Verdi system [BV05]. Verdi-M is implemented in Maude and exploits the specific Maude capabilities that are particularly suitable for our implementation, such as associative commutative pattern matching and metaprogramming. In order to assess the performance of our system, we extensively tested the Verdi-M core engine. The produced benchmark reports an impressive performance, e.g., less than one second for evaluating a tree of 30,000 nodes. We also proposed a service-oriented architecture that makes the Web verification/repair capabilities of the system easily accessible to internet requesters. The resulting prototype WebVerdi-M [ABF+07a] is publicly available together with a set of examples and its XML API. Another important factor that we have considered was how to reduce the learning costs to the user. For this reason, we developed a friendly and innovative interface for our system.

Although WebVerdi-M shows excellent performance for correctness checking, unfortunately, for the verification of completeness, a (finite) fixpoint computation is typically needed, which leads to unsatisfactory performance for XML documents bigger than 1Mb. To overcome this drawback, we proposed a novel abstract methodology for analyzing and verifying Web sites that offsets the high execution costs of analyzing the completeness of complex Web documents. The framework is formalized as a source-to-source transformation that is parametric with respect to

the abstraction, and translates the Web documents and their specifications into constructions of the same language, so that an efficient implementation can be easily derived with very little effort. The key idea for the abstraction is to exploit the substructure similarity that is commonly found in HTML/XML documents.

We also have ascertained the conditions that ensure the correctness of the approximation, so that the resulting abstract rewrite engine safely supports accurate Web site verification. In other words, we can conclude that:

- there are no concrete completeness errors in the case when no abstract completeness errors are detected;
- whenever an abstract correctness error is detected, a corresponding correctness error must exist in the concrete counterpart.

Finally, we tested our approximation with several examples, which demonstrated a considerable improvement in the WebVerdi-M's performance.

Dynamic Web Verification

We formulated a detailed navigation model in rewriting logic that accurately formalizes the behavior of Web applications. The proposed model allows us to specify several critical aspects of Web applications such as concurrent Web interactions, browser navigation features, and Web scripts evaluations in an elegant, high-level rewrite theory. We also coupled our formal specification with LTLR, which is a linear temporal logic designed to model-check rewrite theories. The proposed technique was implemented in the prototype Web-TLR [ABR09; ABER10], which is written in Maude. We conducted several experiments that demonstrated the practicality of our approach.

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

- a rich Web application core model, which considers the communication protocol underlying for Web interactions as well as the common browser navigation features;

- an efficient and accurate model checking of dynamic properties, e.g., reachability of Web pages generated by means of Web script executions, at low cost. The verification includes a property checking analysis that provides diagnostic traces (counter-examples) whenever a given property does not hold;

- a friendly visualization of counter-examples via an interactive slideshow, which allows the user to explore the model by performing forward and backward transitions. At each slide, the interface shows the values of relevant variables of the Web state. This on–the–fly exploration does not require installation of the checker itself and is provided entirely by the graphical Web interface.

Counter-examples often include a huge amount of information, which makes the debugging of a Web application a hard task. In order to ease Web application debugging, we defined a backward trace slicing technique that can significantly reduce the size of the counter-examples given as outcome by Web-TLR. This is achieved by dropping useless data that do not influence the detected wrong application's behavior.

Future Work

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

- In our repairing methodology, the repair actions are based on changing or removing the subterms that produce the failure. In order to preserve the original structure of the Web page, we plan to use our experience in handling XML documents [BR07a] as well as in analyzing the structure of Web pages [AR08; AR10] in order to refine our repairing methodology. Roughly speaking, the idea is to consider repair actions that affect only the label of the nodes where the error occurs. This improvement will allow us to minimize the changes done by a repair action on a wrong Web site.

- With regard to the abstract verification framework, we plan to investigate how to maximize the accuracy of the abstraction function. For example, in [CGJ+00] an adaptive algorithm guided by

spurious counter-examples is proposed. In our framework, these spurious counter-examples correspond to the spurious errors (false alarms) detected into the abstract domain.

- Since Web-TLR works at an abstract level, we plan to complement it by giving support to synthesizing correct-by-construction [PT10; BV07] Web applications. We also plan to deal with client-side scripts, e.g., scripts written in JavaScript or similar languages.

- The prototype of our backward-trace slicing technique showed to be helpful for debugging rewriting theories. We plan to cope with the execution traces of other sophisticated tools developed on top of the language Maude, such as theorem provers, program debuggers, and program certifiers.

Bibliography

[ABBF10] M. Alpuente, D. Ballis, M. Baggi, and M. Falaschi. A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In *Proc. 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*, pages 43–52. ACM, 2010.

[ABE+07] M. Alpuente, D. Ballis, S. Escobar, M. Falaschi, P. Ojeda, and D. Romero. Un motor algebraico para la verificación de sistemas Web en Gverdi. Technical Report DSIC-II/02/07, DSIC, UPV, 2007.

[ABE+11] M. Alpuente, D. Ballis, J. Espert, F. Frechina, and D. Romero. Debugging of Web Applications with WEB-TLR. In *7th Int'l Workshop on Automated Specification and Verification of Web Systems WWV 2011*, 2011. Submitted.

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

[ABER11] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Backward Trace Slicing for Rewriting Logic Theories. In *The 23rd Int'l Conference on Automated Deduction CADE 2011*, LNCS/LNAI. Springer, 2011. To appear.

[ABF06] M. Alpuente, D. Ballis, and M. Falaschi. Rule-based Verification of Web Sites. *Software Tools for Technology Transfer*, 8:565–585, 2006.

[ABF+07a] M. Alpuente, D. Ballis, M. Falaschi, P. Ojeda, and D. Romero. A Fast Algebraic Web Verification Service. In *Proc. of First Int'l Conference on Web Reasoning and Rule*

Systems (RR 2007), volume 4524 of *LNCS*, pages 239–248, 2007.

[ABF+07b] M. Alpuente, D. Ballis, M. Falaschi, P. Ojeda, and D. Romero. Abstract Web Site Verification in WebVerdi-M. Technical Report DSIC-II/19/07, DSIC-UPV, 2007.

[ABF+07c] M. Alpuente, D. Ballis, M. Falaschi, P. Ojeda, and D. Romero. Repair Strategies for Incomplete Web Sites. In *XIII Congreso Argentino de Ciencias de la Computación (CACIC 2007)*, 2007.

[ABF+08] M. Alpuente, D. Ballis, M. Falaschi, P. Ojeda, and D. Romero. An abstract generic framework for web site verification. In *Proc. of the 2008 Int'l Symposium on Applications and the Internet (SAINT 2008)*, pages 104–110. IEEE Computer Society, 2008.

[ABFR06] M. Alpuente, D. Ballis, M. Falaschi, and D. Romero. A Semi-automatic Methodology for Reparing Faulty Web Sites. In *Proc. of the 4th IEEE Int'l Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 31–40. IEEE Computer Society Press, 2006.

[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.

[ACD09] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Modelling methods for web application verification and testing: state of the art. *Softw. Test., Verif. Reliab.*, 19(4):265–296, 2009.

[Alf01] L. Alfaro. Model checking the world wide web. In *Proc. 13th Int'l. Conference on Computer Aided Verification (CAV 2001), Paris, France*, volume 2102 of *LNCS*, pages 337–349, 2001.

[Apaa] Apache Software Foundation. Apache Tomcat project. Available at: `http://tomcat.apache.org/`.

[Apab] Apache Software Foundation. Web Servises – Axis. Available at: http://ws.apache.org/axis/.

[Apt03] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[AR08] M. Alpuente and D. Romero. A Visual Technique for Web Pages Comparison. In *Proc of 4th Int'l Workshop on Automated Specification and Verification of Web Systems (WWV'08). Siena, Italy*, pages 239–253. ENTCS, 2008.

[AR10] M. Alpuente and D. Romero. An abstract generic framework for web site verification. In *Proc. of the 2010 Int'l Symposium on Applications and the Internet (SAINT 2010)*, pages 45–51. IEEE Computer Society, 2010.

[Bal05] D. Ballis. *Rule-based Software Verification and Correction*. PhD thesis, University of Udine and Technical University of Valencia, 2005.

[BBF09] M. Baggi, D. Ballis, and M. Falaschi. Quantitative Pathway Logic for Computational Biology. In *Proc. of 7th Int'l Conference on Computational Methods in Systems Biology (CMSB '09)*, volume 5688 of *LNCS*, pages 68–82. Springer, 2009.

[Bez03] M. Bezem. *TeReSe, Term Rewriting Systems*, chapter Mathematical background (Appendix A). Cambridge University Press, 2003.

[BGK03] P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *Proc. of the 29th Int'l Conference on Very Large Data Bases (VLDB'03)*, pages 141–152, 2003.

[BKdV00] I. Bethke, J. W. Klop, and R. de Vrijer. Descendants and origins in term rewriting. *Inf. Comput.*, 159(1-2):59–124, 2000.

[BM08] K. Bae and J. Meseguer. A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. In *Proc. of the*

9th Int'l Workshop on Rule-Based Programming (RULE'08), ENTCS. Elsevier, 2008.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BP99] L. Bertossi and J. Pinto. Specifying Active Rules for Database Maintenance. In G. Saake, K. Schwarz, and C. Türker, editors, *Transactions and Database Dynamics, 8th Int'l Workshop on Foundations of Models and Languages for Data and Objects*, volume 1773 of *LNCS*, pages 112–129. Springer, 1999.

[BPM+08] T. Bray, J. Paoli, E. Maler, F. Yergeau, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[BR07a] D. Ballis and D. Romero. Filtering of XML documents. In *Proc of 2nd Int'l Workshop on Automated Specification and Verification of Web Systems (WWV'06). Paphos, Cyprus*, pages 19–26. IEEE Computer Society Press, 2007.

[BR07b] D. Ballis and D. Romero. Fixing web sites using correction strategies. In *Proc of 2nd Int'l Workshop on Automated Specification and Verification of Web Systems (WWV'06). Paphos, Cyprus*, pages 11–19. IEEE Computer Society Press, 2007.

[BV05] D. Ballis and J. García Vivó. A Rule-based System for Web Site Verification. In *Proc. of 1st Int'l Workshop on Automated Specification and Verification of Web Sites (WWV'05)*, volume 157(2). ENTCS, Elsevier, 2005.

[BV07] M. Bordin and T. Vardanega. Correctness by construction for high-integrity real-time systems: A metamodel-driven approach. In *Reliable Software Technologies - Ada Europe 2007, 12th Ada-Europe Intl' Conference on Reliable Software Technologies, Geneva, Switzerland, June 25-29, 2007, Proceedings*, volume 4498 of *LNCS*, pages 114–127, 2007.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Appr. of Fixpoints. In *POPL*, pages 238–252, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[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/.

[CDH+07] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. C. Ölveczky. The Maude Formal Tool Environment. In *CALCO*, volume 4624 of *LNCS*, pages 173–178. Springer, 2007.

[CEFN02] L. Capra, W. Emmerich, A. Finkelstein, and C. Nentwich. XLINKIT: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.

[CF04] J. Coelho and M. Florido. Clp(flex): Constraint logic programming applied to xml processing. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, OTM Confederated Int'l Conferences, Agia Napa, Cyprus, October 25-29, 2004, Proceedings, Part II*, volume 3291 of *LNCS*, pages 1098–1112, 2004.

[CF07] J. Coelho and M. Florido. Type-based static and dynamic website verification. In *Int'l Conference on Internet and Web Applications and Services (ICIW 2007), May 13-19,*

2007, Le Morne, Mauritius, page 32. IEEE Computer Society, 2007.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification CAV 2000*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 2nd edition, 2001.

[CR09] F. Chen and G. Rosu. Parametric trace slicing and monitoring. In *TACAS*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.

[DMRT06] F. M. Donini, M. Mongiello, M. Ruta, and R. Totaro. A Model Checking-based Method for Verifying Web Application Design. *ENTCS*, 151(2):19–32, 2006.

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

[Eke03] S. Eker. Associative-Commutative Rewriting on Large Terms. In *Proc. of 14th Int'l Conference, Rewriting Techniques and Applications (RTA '03)*, volume 2706 of *LNCS*, pages 14–29. Springer, 2003.

[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.

[Eng02] J. English. The HXML Haskell Library, 2002. Available at: http://www.flightlab.com/~joe/hxml/.

[FLV08] S. Flores, S. Lucas, and A. Villanueva. Formal verification of websites. *In Proc. 4th Int'l Workshop on Automated Specification and Verification of Web Sites (WWV'08), ENTCS*, 200(3):103–118, 2008.

[FT94] J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proc. of the 6th Int'l Symposium on Programming Language Implementation and Logic Programming, PLILP '94*, pages 415–431, London, UK, 1994. Springer-Verlag.

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

[GHM+07] M. Gudgin, M. Hadley, N. Mendelsohn, Y. Lafon, J. J. Moreau, A. Karmarkar, and H. F. Nielsen. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[GJJ06] T. Le Gall, B. Jeannet, and T. Jéron. Verification of Communication Protocols Using Abstract Interpretation of FIFO Queues. In *Proc. of Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006*, volume 4019 of *LNCS*, pages 204–219. Springer, 2006.

[Gmb] T. GmbH. Validate/Check XML. Available at: http://www.xmlvalidation.com/.

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

[HSP08] M. Haydar, H. Sahraoui, and A. Petrenko. Specification patterns for formal web verification. In *ICWE '08: Proceedings of the 2008 Eighth Int'l Conference on Web Engineering*,

pages 240–246, Washington, DC, USA, 2008. IEEE Computer Society.

[IBM07] IBM. Service Oriented Architecture, 2007. Available at: http://www-306.ibm.com/software/solutions/soa.

[KEG06] N. El Kadhi and H. El-Gendy. Advanced Method for Cryptographic Protocol Verification. *Journal of Computational Methods in Science and Engineering*, 6:109–119, 2006.

[Klo92] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[Leu02] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403, 2002.

[Luc05] S. Lucas. Rewriting-Based Navigation of Web Sites: Looking for Models and Logics. In *Proc. of 1st Int'l Workshop on Automated Specification and Verification of Web Sites (WWV'05)*, volume 157(2). ENTCS, Elsevier, 2005.

[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.

[Mic03] Sun Microsystems. JavaTM 2 SDK, Standard Edition Documentation, Version 1.4.2, 2003. Available at: http://java.sun.com.

[Mic06] Sun Microsystems. JSR 12: JavaTM Data Objects (JDO) Specification, 2006. Available at: http://www.jcp.org/en/jsr/detail?id=12.

[MM08] R. Message and A. Mycroft. Controlling control flow in web applications. *In Proc. 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.

[MT99] E. Mayol and E. Teniente. A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In *Proc. of Advances in Conceptual Modeling: ER '99*, volume 1727 of *LNCS*, pages 62–73. Springer, 1999.

[MyS07] MySQLAB. MySQL, 2007. Available at: `http://www.mysql.com`.

[MZ07] H. Miao and H. Zeng. Model checking-based verification of web application. In *ICECCS '07: Proceedings of the 12th IEEE Int'l Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 47–55, Washington, DC, USA, 2007. IEEE Computer Society.

[NEF03] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *Proc. of the 25th Int'l Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, 2003.

[Osk05] G. Oskoboiny. W3C Markup Validation Service, 2005. Available at: `http://validator.w3.org/`.

[Pem00] S. Pemberton. XHTML™ 1.0: The extensible hypertext markup language - a reformulation of HTML 4 in XML 1.0. first edition of a recommendation, W3C, January 2000. http://www.w3.org/TR/2000/REC-xhtml1-20000126.

[PGI04] N. Polyzotis, M.N. Garofalakis, and Y. E. Ioannidis. Approximate XML Query Answers. In *Proc. of the ACM Int'l Conference on Management of Data (ICMD'04)*, pages 263–374, 2004.

[Pro05] The TJDO Project. TriActive JDO, 2005. Available at: http://tjdo.sourceforge.net.

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

[PT10] I. Poernomo and J. Terrell. Correct-by-construction model transformations from partially ordered specifications in coq. In *Formal Methods and Software Engineering - 12th Intl' Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010*, volume 6447 of *LNCS*, pages 56–73. Springer, 2010.

[Que04] C. Queinnec. Continuations and web servers. *Higher-Order and Symbolic Computation*, 17(4):277–295, 2004.

[Rét87] P. Réty. Improving basic narrowing techniques. In *Proc. the Conference on Rewriting Techniques and Applications*, pages 228–241. Springer LNCS 256, 1987.

[RH05] G. Rosu and K. Havelund. Rewriting-Based Techniques for Runtime Verification. *Autom. Softw. Eng.*, 12(2):151–197, 2005.

[RVCMO09] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative Debugging of Rewriting Logic Specifications. In *Recent Trends in Algebraic Development Techniques, 19th Int'l Workshop, WADT 2008*, volume 5486 of *LNCS*, pages 308–325. Springer, 2009.

[RVMO10] A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative Debugging of Missing Answers for Maude. In *21st Int'l Conference on Rewriting Techniques and Applications, RTA 2010*, volume 6 of *LIPIcs*, pages 277–294. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[SBRS03] J. Scheffczyk, U. M. Borghoff, P. Rödig, and L. Schmitz. Consistent document engineering: formalizing type-safe consistency rules for heterogeneous repositories. In *Proc. of the 2003 ACM Symposium on Document Engineering (DocEng '03)*, pages 140–149. ACM Press, 2003.

[Sol10] AI Internet Solutions. CSE HTML validator, 2010. Available at: `http://www.htmlvalidator.com/`.

[SRBS04a] J. Scheffczyk, P. Rödig, U. M. Borghoff, and L. Schmitz. Managing inconsistent repositories via prioritized repairs. In *Proc. of the 2004 ACM Symposium on Document Engineering (DocEng '04)*, pages 137–146. ACM Press, 2004.

[SRBS04b] J. Scheffczyk, P. Rödig, U. M. Borghoff, and L. Schmitz. S-dags: Towards efficient document repair generation. In *Proc. 2nd Int. Conference on Computing, Communications and Control Technologies*, volume 2, pages 308–313, 2004.

[SWK+02] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for xml data management. In *VLDB 2002, Proceedings of 28th Int'l Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 974–985, 2002.

[Tal08] C. Talcott. Pathway logic. *Formal Methods for Computational Systems Biology*, 5016:21–53, 2008.

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

[Vir94] P. Viry. Rewriting: An effective model of concurrency. In *Proceedings of the 6th Int'l PARLE Conference on Parallel*

Architectures and Languages Europe, pages 648–660, London, UK, 1994. Springer-Verlag.

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 6.2.1, 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 6.2.3, 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

Example: The War of Souls

Let us illustrate our backward trace slicing technique for rewrite theories by means of a nontrivial producer/consumer example.

The War of Souls (WoS) is a game where an angel and a daemon fight to conquer the souls of human beings. Basically, when a human being passes away, his/her soul is sent to heaven or to hell depending on his/her faith as well as the strength of the angel and the daemon in play. Human beings with a higher level of faith are more likely to go to heaven.

This game is specified as a rewrite theory whose implementation, which is written in Maude, is shown in Fig. C.1[1]. The angel (resp. the daemon) is modeled by using the constructor term A(strength) (resp. D(strength)), where A (resp. D) is a constructor function symbol (i.e., a function symbol which doesn't appear as root symbol of the left-hand side of any equation or rule) and strength is a natural number specifying the strength of goodness of the angel (resp. the strength of evilness of the daemon). A human being is formalized by means of the constructor term H(faith), where H is a constructor function symbol and faith is a natural number in the range 0..100 modeling his/her faith. Basically, the dynamic of the game is specified by means of two rules — namely, the creation rule and the death rule. The creation rule generates a new human being with a random faith, provided that there are at least two living human beings. The auxiliary equation newFaith allows us to consider a random value for the faith. The death rule selects a human being H and judges whether H should be sent to heaven or to hell. This decision is

[1] Operator declarations in Maude may include equational attributes, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are used to represent certain kinds of equational axioms in a way that allows Maude to use these equations efficiently in a built-in way. The operator attribute `ctor` declares that the operator is a constructor, as opposed to a function defined by means of rules or equations.

```
mod war-of-souls is inc
 INT + RANDOM + COUNTER .

 sorts Human Angel Daemon Soul .
 subsorts  Human Angel Daemon < Soul .

 vars faith good bad f1 f2 : Int .
 var a : Angel .   var d : Daemon .
 var h h1 h2 : Human .

 op H : Nat -> Human [ctor] .
 op A : Nat -> Angel [ctor] .
 op D : Nat -> Daemon [ctor] .

 --- constants
 ops HEAVEN HELL : -> Soul [ctor] .

 --- equations
 op MAX-FAITH : -> Int .
 eq MAX-FAITH = 100 .

 op . : Soul Soul -> Soul [assoc comm] .

 op newFaith : -> Int .
 eq newFaith = random(counter) rem MAX-FAITH .

 op judgment : Angel Daemon Human -> Soul .
 eq judgment (A(good), D(bad), H(faith)) =
      if ( (good * faith) >=
           (bad * (MAX-FAITH - faith)) )
          then HEAVEN
          else HELL
      fi .

 --- rewrite rules ----
 rl  [creation] : .(H(f1), H(f2)) =>
                  .(H(newFaith), H(f1), H(f2)) .
 rl  [death] : .(h, a, d) =>
                 .(a, d, judgment(a, d, h)) .
endm
```

Figure C.1: Maude specification of the *WoS* game.

taken by using the auxiliary equation judgment that weights the faith of H w.r.t. both the *goodness* of the angel and *evilness* of the daemon involved.

The Backward Trace Slicing Technique in Action

Let us consider the execution trace $\mathcal{T} : t_0 \rightarrow \ldots .. \rightarrow t_9$ given in Fig. C.2, which reaches the state $t_9 = .(\mathtt{HEAVEN}, \mathtt{D}(30), \mathtt{A}(40), \mathtt{H}(70), \mathtt{H}(80), \mathtt{H}(90))$ from the initial state $t_0 = .(\mathtt{A}(40), \mathtt{D}(30), \mathtt{H}(70), \mathtt{H}(80), \mathtt{H}(90))$. This is done by running the `frew` command of Maude, which ensures both local fairness and rule fairness. The bracketed number between the command and the term to be rewritten (see Fig. C.2), provides an upper bound for the number of rule applications that are allowed.

In Fig. C.2, the application of every rule, equation, and axiom is displayed, showing the corresponding substitution, the current state, and the subterm where the next axiom is applied, before and after its application. Note that, since the equational simplification hides some basic steps, the original trace delivered by Maude has been extended with

```
  | Maude> set trace select on .
  | Maude> trace select creation death newFaith judgment .
t0| Maude> frew [3]  .(A(40), D(30), H(70), H(80), H(90)) .
t1| frewrite in war-of-souls : .(D(30), A(40), H(70), H(80), H(90)) .
  | ********** rule
  | rl :(H(f1), H(f2)) => .(H(newFaith), H(f1), H(f2)) [label creation] .
  | f1 --> 70;
  | f2 --> 80
  | .(D(30), A(40), H(70), H(80), H(90))
t2| ================= intermediate term: .(.(D(30), A(40), H(90)), .(H(70), H(80)))
  | --->
t3| .(.(D(30), A(40), H(90)), .(H(newFaith), H(70), H(80)))
  | ********** equation
  | eq newFaith = random(counter) rem MAX-FAITH .
  | empty substitution
  | newFaith
  | --->
  | 44
t4| ================= intermediate term: .(.(D(30), A(40), H(90)), .(H(44), H(70), H(80)))
  | ********** rule
  | rl :(h, d, a) => .(d, a, judgment(a, d, h)) [label death] .
  | h --> H(44); d --> D(30); a --> A(40)
t5| :(D(30), A(40), H(44), H(70), H(80), H(90))
t6| ================= intermediate term: .(.(H(70), H(80), H(90)), .(H(44), D(30), A(40)))
  | --->
t7| .(.(H(70), H(80), H(90)), .(D(30), A(40), judgment(A(40), D(30), H(44))))
  | ********** equation
  | eq judgment(A(good), D(bad), H(faith)) =
  |                     if faith * good >= bad * (MAX-FAITH - faith)
  |                                             then HEAVEN else HELL fi .
  | good --> 40; bad --> 30; faith --> 44
  | judgment(A(40), D(30), H(44))
  | --->
  | HEAVEN
t8| ================= intermediate term: .(.(H(70), H(80), H(90)), .(D(30), A(40), HEAVEN))
t9| result Soul: .(HEAVEN,D(30),A(40),H(70),H(80),H(90))
```

Figure C.2: Trace given by the `frew` command of Maude.

the sequence of intermediate terms that allows one to explicitly handle the B-equivalence by means of flat/unflat transformations as well as the equational computations. Such extension has been implemented by augmenting the original specification with suitable rewrite rules. For example, in Fig. C.2 the term t_4 stems from the term t_3 where the operator `newFaith` has been replaced with its outcome `44`.

We can write the trace of Fig. C.2 simply as follows:

$$\mathcal{T} : t_0 \rightarrow^*_{flat_B} t_1 \rightarrow^*_{unflat_B} t_2 \overset{\mathsf{creation},\sigma_1}{\rightarrow} t_3 \overset{\mathsf{newFaith},\sigma_2}{\rightarrow} t_4 \rightarrow^*_{flat_B} t_5 \rightarrow^*_{unflat_B} t_6 \overset{\mathsf{death},\sigma_3}{\rightarrow} t_7 \overset{\mathsf{judgment},\sigma_4}{\rightarrow} t_8 \rightarrow^*_{flat_B} t_9$$

where $\sigma_1 = \{\mathtt{h}_1/\mathtt{H}(70),\ \mathtt{h}_2/\mathtt{H}(80)\}$, $\sigma_2 = \emptyset$, $\sigma_3 = \{\mathtt{h}/\mathtt{H}(44),\ \mathtt{a}/\mathtt{A}(40),\ \mathtt{d}/\mathtt{D}(30)\}$, and $\sigma_4 = \{\mathtt{good}/40,\ \mathtt{bad}/30,\ \mathtt{faith}/44\}$.

In the following, we apply our slicing technique to the trace $\mathcal{T}$ w.r.t. the rewrite theory $\mathcal{R}$ of Fig. C.1. Specifically, we employ backward trace slicing to observe how the symbol `HEAVEN` of t_9 is produced.

Step 1: Rule and equation labeling.

First of all, we label the rules and the equations according to the proposed labeling procedure.For the sake of readability, the equation `judgment` containing an if-then-else construct has been simplified by splitting it into two different labeled equations representing the two possible branches. Furthermore, in the equation `newFaith`, we consider the same labeling for any randomly created number.

$$
\begin{aligned}
\mathsf{creation}^L &= .^{\mathtt{a}}(\mathtt{H}^{\mathtt{b}}(\mathtt{f1}), \mathtt{H}^{\mathtt{c}}(\mathtt{f2})) \to .^{\mathtt{abc}}(\mathtt{H}^{\mathtt{abc}}(\mathtt{newFaith}^{\mathtt{abc}}), \mathtt{H}^{\mathtt{abc}}(\mathtt{f1}), \mathtt{H}^{\mathtt{abc}}(\mathtt{f2})) \\
\mathsf{death}^L &= .^{\mathtt{a}}(\mathtt{h}, \mathtt{a}, \mathtt{d}) \to .^{\mathtt{a}}(\mathtt{a}, \mathtt{d}, \mathtt{judgment}^{\mathtt{a}}(\mathtt{a}, \mathtt{d}, \mathtt{h})) \\
\mathsf{judgment}^L &= \mathtt{judgment}^{\mathtt{a}}(\mathtt{A}^{\mathtt{b}}(\mathtt{good}), \mathtt{D}^{\mathtt{c}}(\mathtt{bad}), \mathtt{H}^{\mathtt{d}}(\mathtt{faith})) \to \mathtt{HEAVEN}^{\mathtt{abcd}} \\
\mathsf{judgment}^L &= \mathtt{judgment}^{\mathtt{a}}(\mathtt{A}^{\mathtt{b}}(\mathtt{good}), \mathtt{D}^{\mathtt{c}}(\mathtt{bad}), \mathtt{H}^{\mathtt{d}}(\mathtt{faith})) \to \mathtt{HELL}^{\mathtt{abcd}} \\
\mathsf{newFaith}^L &= \mathtt{newFaith}^{\mathtt{a}} \to [\text{a random number}]^{\mathtt{a}}
\end{aligned}
$$

Step 2: Slicing criterion.

Assume we want to discover which pieces of information in $\mathcal{T}$ contribute to the generation of the value `HEAVEN` in t_9. Therefore, since the symbol `HEAVEN` occurs at position 1, we feed our slicing procedure with the slicing criterion $\{1\}$.

Step 3: Rewrite step labeling and sequence of relevant position sets.

Figure C.3 shows the labeling inferred for every rewrite step along with the sequence of relevant position sets $[P_0, \ldots, P_9]$ which have been computed w.r.t. the slicing criterion $\{1\}$.

$\boxed{Step : Slicing\ criterion}$

$P_9 = \{1\}$

$\boxed{Step : t_8 \rightarrow^*_{flat_B} t_9}$

$t_8^{L_8} = .^{\mathtt{a}}(.^{\mathtt{b}}(\mathtt{H}^{\mathtt{c}}(70^{\mathtt{d}}), \mathtt{H}^{\mathtt{e}}(80^{\mathtt{f}}), \mathtt{H}^{\mathtt{g}}(90^{\mathtt{h}})),$
$.^{\mathtt{i}}(\mathtt{D}^{\mathtt{j}}(30^{\mathtt{k}}), \mathtt{A}^{\mathtt{l}}(40^{\mathtt{m}}), \underline{\mathtt{HEAVEN}^{\mathtt{n}}}))$

$t_9^{L_8} = .^{\mathtt{abi}}(\underline{\mathtt{HEAVEN}^{\mathtt{n}}}, \mathtt{D}^{\mathtt{j}}(30^{\mathtt{k}}), \mathtt{A}^{\mathtt{l}}(40^{\mathtt{m}}), \mathtt{H}^{\mathtt{c}}(70^{\mathtt{d}}),$
$\mathtt{H}^{\mathtt{e}}(80^{\mathtt{f}}), \mathtt{H}^{\mathtt{g}}(90^{\mathtt{h}}))$

$P_8 = \{1,\ 2.3\}$

$\boxed{Step : t_7 \overset{\mathsf{judgment},\sigma_4}{\longrightarrow} t_8}$

$t_7^{L_7} = .^{\mathtt{h}}(.^{\mathtt{i}}(\mathtt{H}^{\mathtt{j}}(70^{\mathtt{k}}), \mathtt{H}^{\mathtt{l}}(80^{\mathtt{m}}), \mathtt{H}^{\mathtt{n}}(90^{\mathtt{o}})),$
$.^{\mathtt{p}}(\mathtt{D}^{\mathtt{q}}(30^{\mathtt{r}}), \mathtt{A}^{\mathtt{s}}(40^{\mathtt{t}}),$
$\mathtt{judgment}^{\mathtt{a}}(\mathtt{A}^{\mathtt{b}}(\underline{40^{\mathtt{c}}}), \mathtt{D}^{\mathtt{d}}(\underline{30^{\mathtt{e}}}), \mathtt{H}^{\mathtt{f}}(\underline{44^{\mathtt{g}}}))))$

$t_8^{L_7} = .^{\mathtt{h}}(.^{\mathtt{i}}(\mathtt{H}^{\mathtt{j}}(70^{\mathtt{k}}), \mathtt{H}^{\mathtt{l}}(80^{\mathtt{m}}), \mathtt{H}^{\mathtt{n}}(90^{\mathtt{o}})),$
$.^{\mathtt{p}}(\mathtt{D}^{\mathtt{q}}(30^{\mathtt{r}}), \mathtt{A}^{\mathtt{s}}(40^{\mathtt{t}}), \underline{\mathtt{HEAVEN}^{\mathtt{ceg}}}))$

$P_7 = \{1,\ 2.3.1.1,\ 2.3.2.1,\ 2.3.3.1\}$

$\boxed{Step : t_6 \overset{\mathsf{death},\sigma_3}{\longrightarrow} t_7}$

$t_6^{L_6} = .^{\mathtt{b}}(.^{\mathtt{c}}(\mathtt{H}^{\mathtt{d}}(70^{\mathtt{e}}), \mathtt{H}^{\mathtt{f}}(80^{\mathtt{g}}), \mathtt{H}^{\mathtt{h}}(90^{\mathtt{i}})),$
$.^{\mathtt{a}}(\mathtt{H}^{\mathtt{j}}(\underline{44^{\mathtt{k}}}), \mathtt{D}^{\mathtt{l}}(\underline{30^{\mathtt{m}}}), \mathtt{A}^{\mathtt{n}}(\underline{40^{\mathtt{o}}})))$

$t_7^{L_6} = .^{\mathtt{b}}(.^{\mathtt{c}}(\mathtt{H}^{\mathtt{d}}(70^{\mathtt{e}}), \mathtt{H}^{\mathtt{f}}(80^{\mathtt{g}}), \mathtt{H}^{\mathtt{h}}(90^{\mathtt{i}})),$
$.^{\mathtt{a}}(\mathtt{D}^{\mathtt{l}}(30^{\mathtt{m}}), \mathtt{A}^{\mathtt{n}}(40^{\mathtt{o}}),$
$\mathtt{judgment}^{\mathtt{a}}(\mathtt{A}^{\mathtt{n}}(\underline{40^{\mathtt{o}}}), \mathtt{D}^{\mathtt{l}}(\underline{30^{\mathtt{m}}}), \mathtt{H}^{\mathtt{j}}(\underline{44^{\mathtt{k}}}))))$

$P_6 = \{1,\ 2.1.1,\ 2.2.1,\ 2.3.1\}$

$\boxed{Step : t_5 \rightarrow^*_{unflat_B} t_6}$

$t_5^{L_5} = .^{\mathtt{abi}}(\mathtt{D}^{\mathtt{c}}(\underline{30^{\mathtt{d}}}), \mathtt{A}^{\mathtt{e}}(\underline{40^{\mathtt{f}}}), \mathtt{H}^{\mathtt{j}}(\underline{44^{\mathtt{k}}}), \mathtt{H}^{\mathtt{l}}(70^{\mathtt{m}}),$
$\mathtt{H}^{\mathtt{n}}(80^{\mathtt{o}}), \mathtt{H}^{\mathtt{g}}(90^{\mathtt{h}}))$

$t_6^{L_5} = .^{\mathtt{abi}}(.^{\mathtt{abi}}(\mathtt{H}^{\mathtt{l}}(70^{\mathtt{m}}), \mathtt{H}^{\mathtt{n}}(80^{\mathtt{o}}), \mathtt{H}^{\mathtt{g}}(90^{\mathtt{h}})),$
$.^{\mathtt{abi}}(\mathtt{H}^{\mathtt{j}}(\underline{44^{\mathtt{k}}}), \mathtt{D}^{\mathtt{c}}(\underline{30^{\mathtt{d}}}), \mathtt{A}^{\mathtt{e}}(\underline{40^{\mathtt{f}}})))$

$P_5 = \{1.1,\ 2.1,\ 3.1\}$

$\boxed{Step : t_4 \rightarrow^*_{flat_B} t_5}$

$t_4^{L_4} = .^{\mathtt{a}}(.^{\mathtt{b}}(\mathtt{D}^{\mathtt{c}}(\underline{30^{\mathtt{d}}}), \mathtt{A}^{\mathtt{e}}(\underline{40^{\mathtt{f}}}), \mathtt{H}^{\mathtt{g}}(90^{\mathtt{h}})),$
$.^{\mathtt{i}}(\mathtt{H}^{\mathtt{j}}(\underline{44^{\mathtt{k}}}), \mathtt{H}^{\mathtt{l}}(70^{\mathtt{m}}), \mathtt{H}^{\mathtt{n}}(80^{\mathtt{o}})))$

$t_5^{L_4} = .^{\mathtt{abi}}(\mathtt{D}^{\mathtt{c}}(\underline{30^{\mathtt{d}}}), \mathtt{A}^{\mathtt{e}}(\underline{40^{\mathtt{f}}}), \mathtt{H}^{\mathtt{j}}(\underline{44^{\mathtt{k}}}), \mathtt{H}^{\mathtt{l}}(70^{\mathtt{m}}),$
$\mathtt{H}^{\mathtt{n}}(80^{\mathtt{o}}), \mathtt{H}^{\mathtt{g}}(90^{\mathtt{h}}))$

$P_4 = \{1.1.1,\ 1.2.1,\ 2.1.1\}$

$\boxed{Step : t_3 \overset{\mathsf{newFaith},\sigma_2}{\longrightarrow} t_4}$

$t_3^{L_3} = .^{\mathtt{b}}(.^{\mathtt{c}}(\mathtt{D}^{\mathtt{d}}(\underline{30^{\mathtt{e}}}), \mathtt{A}^{\mathtt{f}}(\underline{40^{\mathtt{g}}}), \mathtt{H}^{\mathtt{h}}(90^{\mathtt{i}})),$
$.^{\mathtt{j}}(\mathtt{H}^{\mathtt{k}}(\underline{\mathtt{newFaith}^{\mathtt{a}}}), \mathtt{H}^{\mathtt{l}}(70^{\mathtt{m}}), \mathtt{H}^{\mathtt{n}}(80^{\mathtt{o}})))$

$t_4^{L_3} = .^{\mathtt{b}}(.^{\mathtt{c}}(\mathtt{D}^{\mathtt{d}}(\underline{30^{\mathtt{e}}}), \mathtt{A}^{\mathtt{f}}(\underline{40^{\mathtt{g}}}), \mathtt{H}^{\mathtt{h}}(90^{\mathtt{i}})),$
$.^{\mathtt{j}}(\mathtt{H}^{\mathtt{k}}(\underline{44^{\mathtt{a}}}), \mathtt{H}^{\mathtt{l}}(70^{\mathtt{m}}), \mathtt{H}^{\mathtt{n}}(80^{\mathtt{o}})))$

$P_3 = \{1.1.1,\ 1.2.1,\ 2.1.1\}$

$\boxed{Step : t_2 \overset{\mathsf{creation},\sigma_1}{\longrightarrow} t_3}$

$t_2^{L_2} = .^{\mathtt{d}}(.^{\mathtt{e}}(\mathtt{D}^{\mathtt{f}}(\underline{30^{\mathtt{g}}}), \mathtt{A}^{\mathtt{h}}(\underline{40^{\mathtt{i}}}), \mathtt{H}^{\mathtt{j}}(90^{\mathtt{k}})),$
$\underline{.^{\mathtt{a}}}(\underline{\mathtt{H}^{\mathtt{b}}}(70^{\mathtt{l}}), \underline{\mathtt{H}^{\mathtt{c}}}(80^{\mathtt{m}})))$

$t_3^{L_2} = .^{\mathtt{d}}(.^{\mathtt{e}}(\mathtt{D}^{\mathtt{f}}(\underline{30^{\mathtt{g}}}), \mathtt{A}^{\mathtt{h}}(\underline{40^{\mathtt{i}}}), \mathtt{H}^{\mathtt{j}}(90^{\mathtt{k}})),$
$.^{\mathtt{abc}}(\mathtt{H}^{\mathtt{abc}}(\underline{\mathtt{newFaith}^{\mathtt{abc}}}), \mathtt{H}^{\mathtt{abc}}(70^{\mathtt{l}}), \mathtt{H}^{\mathtt{abc}}(80^{\mathtt{m}})))$

$P_2 = \{1.1.1,\ 1.2.1,\ 2.1,\ 2.2,\ 2\}$

$\boxed{Step : t_1 \rightarrow^*_{unflat_B} t_2}$

$t_1^{L_1} = \underline{.^{\mathtt{a}}}(\mathtt{D}^{\mathtt{d}}(\underline{30^{\mathtt{e}}}), \mathtt{A}^{\mathtt{b}}(\underline{40^{\mathtt{c}}}), \underline{\mathtt{H}^{\mathtt{f}}}(70^{\mathtt{g}}), \underline{\mathtt{H}^{\mathtt{h}}}(80^{\mathtt{i}}), \mathtt{H}^{\mathtt{j}}(90^{\mathtt{k}}))$

$t_2^{L_1} = .^{\mathtt{a}}(.^{\mathtt{a}}(\mathtt{D}^{\mathtt{d}}(\underline{30^{\mathtt{e}}}), \mathtt{A}^{\mathtt{b}}(\underline{40^{\mathtt{c}}}), \mathtt{H}^{\mathtt{j}}(90^{\mathtt{k}})), \underline{.^{\mathtt{a}}}(\underline{\mathtt{H}^{\mathtt{f}}}(70^{\mathtt{g}}), \underline{\mathtt{H}^{\mathtt{h}}}(80^{\mathtt{i}})))$

$P_1 = \{\lambda,\ 1.1,\ 2.1,\ 3,\ 4\}$

$\boxed{Step : t_0 \rightarrow^*_{flat_B} t_1}$

$t_0^{L_0} = \underline{.^{\mathtt{a}}}(\mathtt{A}^{\mathtt{b}}(\underline{40^{\mathtt{c}}}), \mathtt{D}^{\mathtt{d}}(\underline{30^{\mathtt{e}}}), \underline{\mathtt{H}^{\mathtt{f}}}(70^{\mathtt{g}}), \underline{\mathtt{H}^{\mathtt{h}}}(80^{\mathtt{i}}), \mathtt{H}^{\mathtt{j}}(90^{\mathtt{k}}))$

$t_1^{L_0} = \underline{.^{\mathtt{a}}}(\mathtt{D}^{\mathtt{d}}(\underline{30^{\mathtt{e}}}), \mathtt{A}^{\mathtt{b}}(\underline{40^{\mathtt{c}}}), \underline{\mathtt{H}^{\mathtt{f}}}(70^{\mathtt{g}}), \underline{\mathtt{H}^{\mathtt{h}}}(80^{\mathtt{i}}), \mathtt{H}^{\mathtt{j}}(90^{\mathtt{k}}))$

$P_0 = \{\lambda,\ 1.1,\ 2.1,\ 3,\ 4\}$

Figure C.3: Labeling and sequence of relevant position sets. In order to facilitate the understanding, the terms involved in each step are underlined.

Step 4: Trace slice.

Finally, the trace slice $\mathcal{T}^\bullet$ is obtained by computing the term slices

$$t_0^\bullet = slice(t_0, P_0), \ldots, t_9^\bullet = slice(t_9, P_9)$$

w.r.t the sequence of relevant position sets $[P_0, \ldots, P_9]$. The outcome delivered by our tool is the trace slice $\mathcal{T}^\bullet = t_0^\bullet \rightarrow^* t_9^\bullet$ that is shown in Fig. C.4 together with the corresponding original trace $\mathcal{T}$.

	Original Execution Trace $\mathcal{T}$
t_0	.(A(40), D(30), H(70), H(80), H(90))
t_1	.(D(30), A(40), H(70), H(80), H(90))
t_2	.(.(D(30), A(40), H(90)), .(H(70), H(80)))
t_3	.(.(D(30), A(40), H(90)), .(H(newFaith), H(70), H(80)))
t_4	.(.(D(30), A(40), H(90)), .(H(44), H(70), H(80)))
t_5	.(D(30), A(40), H(44), H(70), H(80), H(90))
t_6	.(.(H(70), H(80), H(90)), .(H(44), D(30), A(40)))
t_7	.(.(H(70), H(80), H(90)), .(D(30), A(40), judgment(A(40), D(30), H(44))))
t_8	.(.(H(70), H(80), H(90)), .(D(30), A(40), HEAVEN))
t_9	.(HEAVEN, D(30), A(40), H(70), H(80), H(90))

	Trace slice $\mathcal{T}^\bullet$
$t_0^\bullet$	.(A(40), D(30), H(•), H(•), •)
$t_1^\bullet$	.(D(30), A(40), H(•), H(•), •)
$t_2^\bullet$	.(.(D(30), A(40), •), .(H(•), H(•)))
$t_3^\bullet$	.(.(D(30), A(40), •), .(H(newFaith), •, •))
$t_4^\bullet$	.(.(D(30), A(40), •), .(H(44), •, •))
$t_5^\bullet$	.(D(30), A(40), H(44), •, •, •)
$t_6^\bullet$	.(.(•, •, •), .(H(44), D(30), A(40)))
$t_7^\bullet$	.(.(•, •, •), .(•, •, judgment(A(40), D(30), H(44))))
$t_8^\bullet$	.(.(•, •, •), .(•, •, HEAVEN))
$t_9^\bullet$	.(HEAVEN, •, •, •, •, •)

Figure C.4: The execution trace $\mathcal{T}$ and its corresponding trace slice $\mathcal{T}^\bullet$.

Trace Slice Analysis

Let us briefly comment on our results. First of all, we note that the trace slice $\mathcal{T}^\bullet$ is much simpler than the original execution trace $\mathcal{T}$, since $\mathcal{T}^\bullet$ only keeps track of those symbols that influence the generation of the value HEAVEN. Also, we can observe that HEAVEN refers to the judgment of the human being H(44) (see term slice $t_7^\bullet$), who has been created by the pair (H(70),H(80)) (through the sliced rewrite step $t_2^\bullet \to t_3^\bullet$).

On one hand, it can be observed that the strength values good and bad (40 and 30, respectively) appear to influence the observed symbol HEAVEN. This is correct, because these values are used in the operation judgment to define the value HEAVEN. Finally, the generation of the observed outcome implies the existence of two human beings, whereas the values of their initial faith are irrelevant for the result, as shown in the term slice $t_0^\bullet$. This is also correct, because these values are not used to compute the faith for H(44), which simply depends on a random value.

Regarding the size reduction of the execution trace, we can conclude the following facts. We denote by $|\,\mathcal{T}\,|$ the size of the trace $\mathcal{T}$, namely

the sum of the number of symbols of all trace states. Then, the reduction given by the slicing technique is: $\frac{|T^{\bullet}|}{|T|} = \frac{75}{139} = 0.54$ (i.e., a reduction of 46%) for this simple example. It is worth noting that, when we fix the slicing criterion $\{2.1, 3.1\}$, which allows us to observe the daemon `D(30)` and angel `A(30)` in play, the reduction achieved is about 71%. In this case, a relevant part of the original execution trace is discarded, since the behaviors of `D(30)` and `A(30)` are not affected by any other operator across the whole trace. This example along with other test cases can be found at the URL http://www.dsic.upv.es/~dromero/slicing.html.

March 29, 1977
Beware of bugs in the above code;
I have only proved it correct, not tried it.
高德纳