Università degli Studi di Siena
Dipartimento di Scienze Matematiche e Informatiche
Dottorato di Ricerca in Informatica,
Logica Matematica e Scienze Cognitive

&

Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Doctorado en Informática

Joint Ph.D. Thesis Siena-UPV

Rule-based Methodologies for the Specification and Analysis of Complex Computing Systems

Candidate:
Michele Baggi

Supervisors:
María Alpuente
Moreno Falaschi

Author's e-mails: baggi@unisi.it
mbaggi@dsic.upv.es

Author's addresses:

Dipartimento di Scienze Matematiche e Informatiche
Università degli Studi di Siena
Pian dei Mantellini, 44
53100 Siena
Italia

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know our place for the first time.

The Four Quartets, **T. S. Eliot**

Noi non smetteremo mai di esplorare
E la fine della nostra ricerca
Sarà arrivare al punto di partenza
E avere conoscenza del nostro posto
Per la prima volta.

Quattro Quartetti, **T. S. Eliot**

Nunca dejaremos de explorar
Y el fin de todas nuestras exploraciones
Será llegar donde empezamos
Y conocer nuestro lugar
Por vez primera.

Cuatro Cuartetos, **T. S. Eliot**

Abstract

From the earliest hardware and software days to the internet era, complexity of computer systems has been something computer scientists, engineers, and programmers have had to deal with. Important fields of research and technology have originated, developed, or matured as a side-effect of this. In this dissertation, we investigate on some of the most challenging, current research directions that are related to the formal specification and verification of complex computer systems.

In this thesis, we focus on distributed systems such as Web systems and biological systems. In order to provide analysis and verification models and tools for these complex computing systems, we use Domain Specific Languages (DSLs). The first part of the thesis is devoted to security aspects and related techniques such as software certification. First, we study access control systems and propose a language for specifying access control policies that are tightly coupled with knowledge bases that provide semantic-aware descriptions of the accessed resources or elements. Also, we develop a novel framework for Code-Carrying Theory, which is a methodology for software certification to secure delivery of code in a distributed environment. Our framework is based on a Fold/Unfold transformation system for rewrite theories. The second part of the thesis focuses on the analysis and verification of Web systems and biological systems. As for web information retrieval, we propose a language for filtering information from big data repositories which uses semantic information retrieved from remote ontologies to refine the filtering process. Also, we study validation methods to check the consistency of web contents with respect to syntax and semantics properties. As our last Web research contribution, we propose a language which allows one to define and automatically check semantic as well as syntactic constraints on the static content of a Web system. Finally, regarding biological systems, we develop a logical formalism for modelling and analysis of quantitative aspects of biological processes, which is based on rewriting logic.

To evaluate the effectiveness of all the proposed methodologies, particular attention has been devoted to the development of prototype systems that have been implemented by using rule-based languages.

Sommario

Dall'epoca dei primi hardware e software fino ai giorni nostri, la complessità dei sistemi di calcolo è sempre stato un problema al quale informatici, ingegneri e programmatori hanno dovuto far fronte. Come risultato di questo sforzo, hanno avuto origine e sono cresciute importanti aree di ricerca. In questa tesi concentriamo la nostra attenzione su alcune delle attuali linee di ricerca relative alla specifica formale e alla verifica di sistemi complessi.

In questa tesi ci siamo focalizzati su sistemi distribuiti quali i sistemi Web e i sistemi biologici. Al fine di sviluppare modelli e strumenti per l'analisi e la verifica di questi sistemi di computazione complessi, abbiamo utilizzato linguaggi di specifica. La prima parte della tesi è dedicata ad aspetti di sicurezza e metodologie affini come la certificazione del software. Inizialmente abbiamo studiato sistemi per il controllo degli accessi alle risorse e abbiamo proposto un linguaggio per specificare politiche di accesso strettamente associate a basi di conoscenza che forniscono una descrizione semantica delle risorse e degli elementi ai quali si accede. Abbiamo anche sviluppato una nuova struttura per il Code-Carrying Theory, una metodologia per la certificazione del software volta a rendere sicuro il trasferimento di codice in un ambiente distribuito. La nostra struttura si basa su un sistema di trasformazione di teorie di riscrittura mediante operazioni di Fold/Unfold. La seconda parte della tesi si focalizza sull'analisi e la verifica di sistemi Web e sistemi biologici. Considerando il problema del Web Information Retrieval, proponiamo un linguaggio per filtrare informazioni da documenti di grandi dimensioni che memorizzano dati in formato XML. Al fine di raffinare il processo di ricerca, il nostro linguaggio è in grado di utilizzare informazione semantica che può reperire accedendo ad ontologie remote. Ci siamo inoltre occupati dei metodi di validazione per la verifica della consistenza del contenuto di sistemi web rispetto a proprietà sintattiche e semantiche date. Il nostro contributo in questo ambito è la proposta di un linguaggio che permette di definire e verificare automaticamente proprietà sintattiche e semantiche sul contenuto statico di sistemi Web. Abbiamo considerato infine i sistemi biologici e abbiamo sviluppato un formalismo, basato sulla logica di riscrittura, per modellare aspetti quantitativi dei processi biologici.

Per valutare l'efficacia delle metodologie proposte, abbiamo dedicato una particolare attenzione allo sviluppo di sistemi prototipo che sono stati implementati utilizzando linguaggi basati su regole.

Resumen

Desde los orígenes del hardware y el software hasta la época actual, la complejidad de los sistemas de cálculo ha supuesto un problema al cual informáticos, ingenieros y programadores han tenido que enfrentarse. Como resultado de este esfuerzo han surgido y madurado importantes áreas de investigación. En esta disertación abordamos algunas de las líneas de investigación actuales relacionada con el análisis y la verificación de sistemas de computación complejos utilizando métodos formales y lenguajes de dominio específico.

En esta tesis nos centramos en los sistemas distribuidos, con un especial interés por los sistemas Web y los sistemas biológicos. La primera parte de la tesis está dedicada a aspectos de seguridad y técnicas relacionadas, concretamente la certificación del software. En primer lugar estudiamos sistemas de control de acceso a recursos y proponemos un lenguaje para especificar políticas de control de acceso que están fuertemente asociadas a bases de conocimiento y que proporcionan una descripción sensible a la semántica de los recursos o elementos a los que se accede. También hemos desarrollado un marco novedoso de trabajo para la *Code-Carrying Theory*, una metodología para la certificación del software cuyo objetivo es asegurar el envío seguro de código en un entorno distribuido. Nuestro marco de trabajo está basado en un sistema de transformación de teorías de reescritura mediante operaciones de plegado/desplegado. La segunda parte de esta tesis se concentra en el análisis y la verificación de sistemas Web y sistemas biológicos. Proponemos un lenguaje para el filtrado de información que permite la recuperación de informaciones en grandes almacenes de datos. Dicho lenguaje utiliza información semántica obtenida a partir de ontologías remotas para refinar el proceso de filtrado. También estudiamos métodos de validación para comprobar la consistencia de contenidos web con respecto a propiedades sintácticas y semánticas. Otra de nuestras contribuciones es la propuesta de un lenguaje que permite definir y comprobar automáticamente restricciones semánticas y sintácticas en el contenido estático de un sistema Web. Finalmente, también consideramos los sistemas biológicos y nos centramos en un formalismo basado en lógica de reescritura para el modelado y el análisis de aspectos cuantitativos de los procesos biológicos.

Para evaluar la efectividad de todas las metodologías propuestas, hemos prestado especial atención al desarrollo de prototipos que se han implementado utilizando lenguajes basados en reglas.

Resum

Des dels orígens del maquinari i el programari fins a l'època actual, la complexitat dels sistemes de càlcul ha suposat un problema al com informàtics, enginyers i programadors han hagut d'enfrontar-se. Com a resultat d'aquest esforç han sorgit i madurat importants àrees d'investigació. En aquesta tesi abordem algunes de les línies d'investigació actuals relacionada amb l'anàlisi i la verificació de sistemes de computació complexos utilitzant mètodes formals i llenguatges de domini específic. Per a proporcionar models i eines per a l'anàlisi i la verificació de sistemes de computació complexos utilitzem llenguatges de domini específic.

En aquesta tesi ens centrem en els sistemes distribuïts, amb un especial interès pels sistemes Web i els sistemes biològics. La primera part de la tesi està dedicada a aspectes de seguretat i tècniques relacionades, concretament la certificació del programari. En primer lloc estudiem sistemes de control d'accés a recursos i proposem un llenguatge per a especificar polítiques de control d'accés que estan fortament associades a bases de coneixement i que proporcionen una descripció sensible a la semàntica dels recursos o elements als quals s'accedeix. També hem desenvolupat un marc nou de treball per a la *Code-Carrying Theory*, una metodologia per a la certificació del programari l'objectiu de la qual és assegurar l'enviament segur de codi en un entorn distribuït. El nostre marc de treball està basat en un sistema de transformació de teories de reescriptura mitjançant operacions de plegat/desplegat. La segona part d'aquesta tesi es concentra en l'anàlisi i la verificació de sistemes Web i sistemes biològics. Proposem un llenguatge per al filtrat d'informació que permet la recuperació d'informacions en grans magatzems de dades. Aquest llenguatge utilitza informació semàntica obtinguda a partir d'ontologies remotes per a refinar el procés de filtrat. També estudiem mètodes de validació per a comprovar la consistència de continguts web pel que fa a propietats sintàctiques i semàntiques. Una altra de les nostres contribucions és la proposta d'un llenguatge que permet definir i comprovar automàticament restriccions semàntiques i sintàctiques en el contingut estàtic d'un sistema Web. Finalment, també considerem els sistemes biològics i ens centrem en un formalisme basat en lògica de reescriptura per al modelatge i l'anàlisi d'aspectes quantitatius dels processos biològics.

Per a avaluar l'efectivitat de totes les metodologies proposades, hem prestat especial atenció al desenvolupament de prototips que s'han implementat utilitzant llenguatges basats en regles.

Acknowledgments

Everyone knows that this lines of acknowledgements are the last ones, following a chronological order, that are commonly written when the thesis is almost finished. However, they are the first lines that open the PhD thesis after the abstract, and, in my opinion, this is what points out their relevance.

Before being a formal and official document, this thesis is the compendium of a three-years-long experience of life that includes work, study, journeys, but also, and especially, a lot of human relations that have flourished and matured with professors, colleagues, and new and old friends. All of this, and much more, goes under the denomination of "postgraduate studies". Therefore, it is not hard to imagine that the list of people I should thank is very long. However, I prefer mentioning here the people that are strictly related to my work at the University, and allow myself the pleasure to thank personally all the friends that helped and supported me with their affection, otherwise this paragraph would become extremely long.

It is useless to say that this thesis would not have been possible without the encouragement, guidance and support of my supervisors Prof. Moreno Falaschi and Prof. María Alpuente Frasnedo, who deserve my deepest gratitude. An the same way I am sincerely thankful to Dott. Demis Ballis, with whom I have worked since the beginning of the PhD, and who always supported my work, both scientifically and morally, especially in those periods full of doubts and difficulties that so frequently come in a PhD student experience. I recall with emotion a dialogue between Demis and I, that took place when I was not already a PhD student. In that occasion he asked me, for the first time, what I would have thought about starting a PhD under the supervision of Moreno Falaschi, that he literally described as a *good scientific father*. Well, now that I have reached the end of my PhD, I can confirm what he said. I can even extend his claim by saying that María has been for me a good scientific mother and Demis has been a scientific brother. Hence, let me thank again my scientific family!!

During my PhD I stayed and worked at the Universities of Siena, of Valencia, and at the University of Udine as aggregated to the Department of "Mathematics and Computer Science". Therefore, there are three groups of colleagues (and friends) that I would like to thank. At the University of Siena I am grateful to Annamaria Pezzotti, Romina D'aurizio, Beate Bruske, Silvia Vecchiato, Filippo DiSanto, and Daniele Marsibilio for their companionship and familiarity during my stays in Siena. At the University of Valencia I would like to thank Sonia Santiago, Beatriz Alarcón,

Antonio Bella, Salvador Tamarit, Daniel Romero, Alexei Lescaylle, José Iborra, Marco Feliú, Raúl Gutiérrez, Santiago Escobar, Mauricio Fernando Alba Castro, and Cesar Ferri for their always warm welcome. In particular, I would thank Sonia, Bea, Toni, and Tama for the everyday lunches at La Vella or at the Conservatorio, with a special thank to Bea for taking care of my correct nutrition. Another special thank to Sonia for the frequent chats that used to keep us away from working, and that used to finish with Sonia's exclamation "the problem is that I love chatting, and, hence, I do not work"! Finally, at the University of Udine I am grateful to Donatella Gubiani, Giuseppina Barbieri, and Fabio Buttussi for their care that made my days at work more pleasant. A special thank to the afternoon tea offered by Donatella and the delicious cakes and sweets made by Giuseppina that she liked to take to our office and share with us.

Out of the academic context, I want to recall here and kindly thank my family: mum, dad, and Raffaele, who share all my successes, difficulties, joys and pains of my life with affection and care. A special mention is due to Mons. Luigi Giussani and all the Italian and Spanish friends of the movement of Communion and Liberation who helped me in my human growth throughout these years.

Finally, I would like to thank that inexpressible Mystery that we use to call God, who leads my steps towards the promised fulfillment of every human life, in one way or another, through all of my days.

Contents

List of Figures

Introduction

From the earliest hardware and software days to the internet era, complexity has been something computer scientists, engineers, and programmers have had to deal with. Important fields of research and technology have originated, developed, or matured as a side-effect of this. Some of these fields, such as computational complexity and planar graphs, address fundamental theories of broad significance. Other directions such as reduced instruction set architectures, compiler-compilers, and high-level languages, are more applied. In this thesis, we focus on some directions that are related to the system's complexity research from the formal specification and analysis perspective by means of rule-based methodologies. We consider in particular Web systems, whose complexity arises due to several factors, such as the large number of hyperlinks, complex interaction, and the increased use of distributed servers. Modeling can help to understand these complex systems, and a considerable effort has been devoted to the specific problem of modelling Web applications. In some cases, new models have been proposed, while in other cases, existing modelling techniques have been adapted from other computing domains. Modeling can help designers during the design phases by formally defining the requirements, providing multiple levels of detail, and giving support for testing prior to implementation. Support from modelling can also be used in later phases to support analysis, validation and verification.

Most of the early literature on this field of Web research concentrates on the process of modelling the design of web applications. Some proposals use reverse engineering methods to extract models from existing web applications in order to support their maintenance and evolution, still others provide analysis models applied to the field of verification and testing of web applications. Among the different possible analysis models, there are some that focus on modelling the navigational aspects of web applications, whereas other models concentrate on solving problems related to the user interaction with the browser in a way that affects the underlying business process. Still other models support the verification of correctness and completeness properties of either static or dynamic web page contents. For a survey on modelling methods for web application verification and testing, we refer to [3]. The validation methods check the consistency of the web content with respect to syntax and semantics. When verifying the completeness of a web application, the model should enforce that a given web page contains some piece of information, that links between web pages exist, and sometimes even check that the web page exists (broken links). Correctness implies that the information provided on a web page is valid w.r.t. the application requirements. Most of the developed approaches (such as [8]), albeit very useful in their specific domains, share the same limitation: the syntactic as well as semantic constraints they specify only rely on the data to be checked.

This thesis proposes a method for verifying static web contents for both syntactic

and semantic properties by using partial rewriting. In our method, web pages are modeled as the ground terms of a term algebra, and the entire web site is represented as a set of ground terms. A checking specification is a pair (I_N, I_M), where I_N is the set of correctness constraints and I_M is the set of completeness constraints, all encoded as partial rewriting rules. Moreover, our method provides ontology reasoning capabilities which allow us to query a (possibly) remote ontology reasoner in order to check semantic properties over the data on interest, and to retrieve semantic information which can be combined with the syntactic one for improving the analysis.

Strictly connected with the verification of organized and semi-structured data, such as web system contents, is the information retrieval problem for such data. With the advent of XML [162] as a widely accepted standard for data representation and exchange, there has been a rapid growth in the amount of XML data available over the internet, so we can talk of XML Information Retrieval. Arguably, growing attention has been dedicated to query and filtering languages as a means to efficiently extract all and only the relevant information from huge data collections. The World Wide Web Consortium has defined XQuery[164] and XPath[163] as standard languages to consult and filter information contained in XML documents, nonetheless a plethora of alternative and worthwhile proposals have been developed independently [45, 119, 64]. Some programming languages supporting XML processing have also been developed, such as VeriFLog [61] which is a tool originally developed for verification of web system contents and data inference. Such languages can be used to consult and query XML documents but provide basically an exact matching engine. Although the languages mentioned above are very advantageous in many applications, they may be of limited use when dealing with data filtering in a pure information retrieval context since they require the user to be aware of the complete XML document structure, and the results that do not exactly match are not delivered. Therefore, in this context, a more flexible matching mechanism which can manage the lack as well as the vagueness of the information is necessary. Such an approximate behavior is not typically implemented in the standard query languages, and actually only few works address this issue [14, 153, 148, 144].

In order to approximate the filtering of XML documents, we propose a declarative language which allows the user to easily select the desired information as well as to remove noisy, spurious data from a given XML document. While XML documents are modeled as ground terms of a term algebra, the patterns of the information we are looking for are modeled as non-ground terms of the same term algebra. An approximate tree embedding algorithm is proposed to execute filtering queries on XML documents in order to recognize the information that the user wants to select or to strike out. Moreover, the filtering process can exploit additional semantic information which can be retrieved by querying (possibly) remote knowledge bases.

In this context, where the widespread use of web applications provides an easy way to share and exchange data as well as resources over the Internet, controlling the user's ability to exercise access privileges on distributed information is a crucial issue. In recent years, there has been a considerable attention to distributed access control, which has rapidly led to the development of several domain specific languages for the specification of access control policies in such heterogeneous environments:

among those, it is worth mentioning the standard XML frameworks XACML [125] and WS-Policy [161]. In the semantic web, resources are annotated with machine-understandable metadata which can be exploited by intelligent agents in order to infer semantic information regarding the resources under examination. Therefore, in this context, application's security aspects should be aware of the semantic nature of the entities into play (e.g. resources, subjects). In particular, it would be desirable to be able to specify access control requirements about resources and subjects in terms of the rich metadata describing them.

As our contribution to improve web security, we present a rule-based language for specifying access control policies which allows security administrators to tightly couple access control rules with knowledge bases that provide semantic-aware description of subjects and resources. Access control policies are modeled as sets of rewrite rules, called policy rules, which may contain queries to knowledge bases. Evaluating an authorization request essentially boils down to rewriting the initial request by using the policy rules until a decision is reached (e.g. *permit*, *deny*, *notApplicable*). Finally, our language is also endowed with policy composition and delegation facilities which are essential aspects of access control in collaborative and distribute environments.

Besides data and resources security, with the advent of the phenomenon of mobile code, code security has also become an important issue. Mobile code is software transferred between systems and executed on a local system without explicit installation or execution by the recipient, even though it is delivered through an insecure network or from an untrusted source. Important examples of mobile code include Web applets, actor-based distributed system software [156], and updates to embedded computers. During delivery, code might be corrupted, or a malicious hacker might change the code. Potential problems can be summarized as security problems (i.e. unauthorized access to data or system resources), safety problems (i.e. illegal operations or illegal access to memory), or functional incorrectness (i.e the delivered code fails to satisfy a required relation between its input and output). Proof-Carrying Code (PCC) [131] and Code-Carrying Theory (CCT) [158] are two alternatives among other solutions to these problems. The basic idea of PCC is that a code consumer does not accept delivery of new code unless it is accompanied by a formal proof of required safety, security, or functional properties that can be checked by the code consumer. One way of doing this is to attach to the code an easily-checkable proof at the code producer's site. Code-Carrying Theory (CCT) is an alternative to PCC with similar goals and technology, but it is based on the idea of proof-based program synthesis [127] rather than program verification. The basic idea is that a set of axioms that define functions are provided by the code producer together with suitable proofs that the defined functions obey certain requirements. The form of the function-defining axioms is such that it is easy to extract executable code from them. Thus, all that has to be transmitted from the producer to the consumer is a theory (a set of axioms and theorems) and a set of proofs of the theorems. There is no need to transmit code explicitly. Concerning certification, we provide an implementation of the CCT methodology that uses a Fold/Unfold transformation framework for rewrite theories, and that reduces the burden on the code producer. In order to achieve this, first we investigate the completeness of Fold/Unfold operations in rewriting logic [117], a

logical formalism where the states of a system are represented as terms of a suitable algebra and the system behavior is described by means of rewrite rules. Rewriting logic is efficiently implemented in the high-performance functional language Maude [59].

Last but not least, we consider biological systems as complex computing systems. Indeed, the bio-systems are much alike distributed computing systems. Both are made of a great number of independent, geographically dispersed, mobile computing agents, that exchange information and proceed by autonomously processing it. The computational approach to the description, simulation and analysis of biological system is increasigly receiving attention both by biologists and by computer scientist as soon as these systems have been seen as computing objects. The interactions among their components is studied within the System Biology field, that seems more accurate than the classical reductionistic one in describing the behavior of biological systems. Such investigations fostered the research and usage of many computational formalisms. Surveys on these formalisms and their use in System Biology can be found in [38].
The seminal paper [90] showed that biological systems have many aspects in common with distributed mobile systems. E.g., a metabolic network is made of billions of components that concurrently interact, in a non-deterministic way and subject to vicinity constraints, just as (far less) mobile devices exchange data via bluetooth with others if close enough, or access to resources if within a specific wifi network. This observation vindicates the usage of process calculi, typically the pi-calculus [123], for specifying cells, in the "cells as computation" paradigm [140]. Other calculi have been put forward later on, among which we only cite a few, which have primitives to directly represent membranes and compartments, like BioAmbients [139], Brane Calculi [50], beta-binders [136].

Petri nets [126] naturally represent biochemical reactions, and thus they are largely used for studying complex biological systems. Actually, they have strong similarities with the graphical language proposed by the Network Visual Designer and often used by biologists to describe metabolic networks. The literature has a vast number of results on both the qualitative as well as the quantitative analysis of the models based on Petri nets and their stochastic versions (SPN). The SPN have been used to study genetic regulation networks in [94]; recent work on biochemical networks are in [53, 111]. Hybrid Functional Petri Nets [4, 118] are a proposal to cope with both the discrete and the continuous aspects that are typical of biological phenomena.

Recently, formalisms and paradigms with a logic basis ([149, 85]) have been successfully used for the specification and study of biological systems. Among the logical formalisms, Pathway Logic [149, 82] (PL) is a symbolic approach to the modelling and analysis of qualitative aspects of biological processes that is based on rewriting logic. The process of application of rewrite rules, from a given initial state, generates computations. In the case of biological processes, these correspond to pathways. Although Pathway Logic may be very useful to model biological processes and provides a simple way to express the system dynamics, it only supports qualitative modelling of the biological events of interests, and it does not provide adequate capabilities to express inhibitory actions occurring in biological reactions. With the aim of overcoming

some of the limitations of Pathway Logic, we present an extension called Quantitative Pathway Logic which provides support for quantitative information such as element concentrations in cell locations, levels of production as well as consumption of elements occurring in a reaction, reaction threshold, *etc.* Moreover, our formalism provides the capability to express inhibitory actions occurring in biological reactions, which are very common e.g. in regulatory networks. In order to manage the different aspects of biological systems, we follow and adapt the Pathway Logic approach by equipping QPL specifications with two equivalent computational models. The former, based on rewriting logic, allows some kinds of model analysis, while the latter is based on Petri nets and can be used to perform network analysis.

In order to provide models for performing analysis and verification of the different considered forms of complex computing systems and the various aspects we focused on, we made use of Domain Specific Languages (DSLs) [155]. Domain-Specific Modeling (DSM) [104] is a way of designing and developing systems that involves the systematic use of Domain Specific Languages to represent the various facets of a system, in terms of models. Such languages tend to support higher-level abstractions than general-purpose modelling languages, and are closer to the problem domain than to the implementation domain. Furthermore, the rules of the domain can be included into the language as constraints, which disallows the specification of illegal or incorrect models. In order to evaluate the effectiveness of all the proposed methodologies, particular attention has been devoted to the development of prototype systems by using rule-based languages. The rule-based paradigm for knowledge representation appears in various forms within computer science. Language issues related to this paradigm appear in production systems [63], parallel program design (e.g. [52]), default reasoning within AI [114], logic programming [15], rewriting [100], active and deductive databases [77], and logics for action and change [143]. There are many benefits in using rule-based systems instead of conventional development tools. The most important are gathered below:

- *Incremental development and rapid prototyping.* The rules can be run and tested the moment they are added to the system. Unlike traditional programming tools such as C++ or C, changes to the rules do not require recompilation, re-linking and re-deploying.

- *Understandable units of business practice.* Rules in the rule-base are self-contained chunks of logic, representing single concepts. This helps their readability and understandability.

- *No control flow.* Unlike a conventional program that usually has a single starting point and a sequence of execution, there is no control flow in the rule-based approach. Rules can start to execute from any point in the rule-base.

- *Consistency.* In comparison to conventional code, incomplete, incorrect, irrelevant or redundant rules are much easier to find, since they stick out from the system.

- *Ability to work with incomplete and missing information.* In many business situations, it is not possible to provide complete and verifiable data. Rule-based systems can deal with such cases of incomplete information.

I.0.1 Outline of the thesis.

The thesis is divided into two parts. The former is devoted to security aspects and software certification, while the latter presents some results on the analysis and verification of distributed systems. In Chapter 1, we provide the necessary notation and preliminary definitions about the term rewriting and the description logic formalisms that will be used in the document. In Chapter 2, we study the Unfold operation based on narrowing over rewrite logic theories, and we propose a Fold/Unfold–based transformation framework for rewrite logic theories that we apply to implement a Code Carrying Theory (CCT) system. Some results presented in this chapter have been published in [7]. Chapter 3 presents a domain specific language for modelling access control policies which is particularly suitable for managing security in distributed environments, since it allows one to evaluate authorization requests according to information retrieved from remote knowledge bases. This work has been published in [30]. Chapter 4 formalizes a domain specific language for filtering information from XML data. The filtering process combines knowledge base reasoning with a flexible pattern-matching engine. The described results have been documented in a number of publications [24, 26, 23]. In Chapter 5, we consider Web systems and we discuss the problem of keeping data correct, consistent and complete w.r.t. some requirements given. We propose a rule-based specification language which allows one to define and automatically check semantic as well as syntactic constraints over the informative content of a Web system. The result of this work was published in [6]. Chapter 6 presents a rule-based formalism which allows one to model and analyze biological processes and reason about quantitative aspects such as element concentration and reaction rates. The proposed formalism appeared in [29].

1

Preliminaries

In this chapter, we provide the basic notation and terminology about rewriting logic and description logic, that are used in the thesis.

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, \leq)$. We assume an S-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets. A variable $x \in \mathcal{V}$ of sort s is denoted by $x :: s$, while by $f :: s_1 \dots s_n \mapsto s$ we represent the type of the operator $f \in \Sigma$ of arity n. $\mathcal{T}_\Sigma(\mathcal{V})_s$ and $\mathcal{T}_{\Sigma s}$ are the sets of terms and ground terms of sort s, respectively. We write $\mathcal{T}_\Sigma(\mathcal{V})$ and $\mathcal{T}_\Sigma$ for the corresponding term algebras. The set of variables occurring in a term t is denoted by $Var(t)$. We write $\overline{o_n}$ for the *list* of syntactic objects $o_1, \dots, o_n$.

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 at position Λ in the term t. 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. Positions are ordered by the *prefix* ordering: $p \leq q$, if $\exists w$ such that $p.w = q$. Two positions q and p are not comparable if $q \not\leq p$ and $p \not\leq q$. $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. Let $\Sigma \cup \{\Box\}$ be a signature such that $\Box \notin \Sigma$. The symbol $\Box$ is called *hole*. A *context* is a term $\gamma \in \mathcal{T}_{\Sigma \cup \{\Box\}}(\mathcal{V})$ with zero or more holes $\Box$. 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 $\gamma[t_1, \dots t_n]$ to denote the term obtained by filling the holes appearing in $\gamma[\,]$ with terms $t_1, \dots, t_n$. Syntactic equality is represented by $\equiv$.

A *substitution* $\sigma \equiv \{x_1/t_1, x_2/t_2, \dots\}$ is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\mathcal{T}_\Sigma(\mathcal{V})$ satisfying the following conditions: (i) $x_i \neq x_j$, whenever $i \neq j$, (ii) $x_i\sigma = t_i$, $i = 1, \dots, n$ and (iii) $x\sigma = x$, for all $x \in \mathcal{V} \setminus \{x_1, \dots, x_n\}$. By ε we denote the *empty* substitution. A substitution σ is called *ground* if for each $x/t \in \sigma$, t is a ground term. A substitution θ is *more general* than a substitution σ, in symbols $\theta \leq \sigma$, if $\sigma = \theta \circ \gamma$ for some substitution γ. Given two terms s and t, a *unifier* for s and t is a substitution σ such that $s\sigma = t\sigma$. An *instance* of a term t is defined as $t\sigma$, where σ is a substitution. The identity substitution is denoted by id. The restriction

of a substitution σ to a set of variables V is defined as

$$\sigma|_V(x) = \begin{cases} \sigma(x) & \text{if } x \in V \\ x & \text{otherwise} \end{cases}$$

Term Rewriting and Rewriting Logic. Term Rewriting Systems [20] (TRS for short) are reduction systems in which rewrite rules apply to terms, and they provide an adequate computational model for functional languages. Rewriting Logic [117] (RL for short) is a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication and interaction, which employs term rewriting modulo equational theories. It is also a flexible logical framework in which many different logical formalisms can be both represented and executed. In this section, we provide a brief overview of this model.

An *(order-sorted) equational theory* is a pair $E \equiv (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of equations of the form $l = r$, $l, r \in \mathcal{T}_\Sigma(\mathcal{V})$, $l \notin \mathcal{V}$, with $Var(r) \subseteq Var(l)$, and B is a collection of equational axioms that express associativity and commutativity (AC) for some defined symbols of Σ. We assume Σ is a partition $\Sigma \equiv \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors*, and symbols $f \in \mathcal{D}$, called *defined symbols*, each of which has a fixed arity, where $\mathcal{D} \equiv \{f \mid f(\bar{t}) = r \in \Delta\}$ and $\mathcal{C} \equiv \Sigma - \mathcal{D}$. Then $\mathcal{T}_\mathcal{C}(\mathcal{V})$ is the set of constructor terms.

The equations in an equational theory E are considered as simplification rules by using them only in the left to right direction, so for any term t, by repeatedly applying the equations as simplification rules, we eventually reach a term to which no further equations apply. The result is called the *canonical form* of t w.r.t. E. This is guaranteed by the fact that E is required to be terminating and Church-Rosser [44]. The set of equations in Δ together with the equational axioms of B in an equational theory E induce a congruence relation on the set of terms $\mathcal{T}_\Sigma(\mathcal{V})$ which is usually denoted by $=_E$. E is a presentation or axiomatization of $=_E$. In abuse of notation, we speak of the equational theory E to denote the theory axiomatized by E. Given an equational theory E, we say that a substitution σ is an *E-unifier* of two terms t and t' if $t\sigma$ and $t'\sigma$ are both reduced to the same canonical form modulo the equational theory (in symbols $t\sigma =_E t'\sigma$). For substitutions σ, ρ and a set of variables V, we define $\sigma|_V =_E \rho|_V$ if $x\sigma =_E x\rho$ for all $x \in V$, and we define $\sigma|_V \leq_E \rho|_V$ if there is a substitution η such that $\rho|_V =_E (\eta \circ \sigma)|_V$. Given two terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{V})$, a set of substitutions $CSU_E(t, t')$ is said to be a *complete* set of unifiers if (i) each $\sigma \in CSU_E(t, t')$ is an E-unifier of t and t', and (ii) for any E-unifier ρ of t and t', there is a $\sigma \in CSU_E(t, t')$ such that $\sigma \leq_E \rho$. For AC theories, a finite complete set of unifiers does exist [21].

A *(order-sorted) rewrite theory* is a triple $\mathcal{R} \equiv (\Sigma, \Delta \cup B, R)$, where R is a set of rewrite rules of the form $l \to r$, $l, r \in \mathcal{T}_\Sigma(\mathcal{V})$, $l \notin \mathcal{V}$, with $Var(r) \subseteq Var(l)$, and Σ is the pairwise disjoint union $\mathcal{D}_1 \uplus \mathcal{D}_2 \uplus \mathcal{C}$ such that $(\mathcal{D}_1 \uplus \mathcal{C}, \Delta \cup B)$ is an order-sorted equational theory, and $\mathcal{D}_2 \equiv \{f \mid f(\bar{t}) \to r \in R\}$ is the set of symbols defined by the rules of R. We omit Σ when no confusion can arise. Throughout this chapter, a rewrite theory is also called a program.

Given a rule $l \to r$, or an equation $l = r$, terms l and r are called the *left-hand side* (or *lhs*) and the *right-hand side* (or *rhs*) of the rule (*resp.* equation). A rule or equation are said to be:

(1) *Non-erasing*, if $Var(l) = Var(r)$.

(2) *Sort preserving*, if for each substitution σ, we have $l\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$ if and only if $r\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$.

(3) *Sort decreasing*, if for each substitution σ, $r\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$ implies $l\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$.

(4) *Left (or right) linear*, if l (*resp.* r) is *linear*, i.e., no variable occurs in the term more than once. It is called *linear* if both l and r are linear.

A set of equations/rules is said to be non-erasing, or sort decreasing, or sort preserving, or (left or right) linear, if each equation/rule in it is so.

An equational theory (*resp.* rewrite theory) is said to be *conditional* if its equations (*resp.* rules) are of the form ($l = r$ if c) (*resp.* $l \to r$ if c), where c is a term representing the condition. Moreover, *labels* may be associated with equations and rules in order to easily identify them, in the form ($label : l = r$) or ($label : l \to r$).

We define the *one-step rewrite relation* on $\mathcal{T}_\Sigma(\mathcal{V})$ as follows: $t \to_R t'$ if there is a position $p \in O_\Sigma(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p \equiv l\sigma$ and $t' \equiv t[r\sigma]_p$. The relation $\to_{R/E}$ for rewriting modulo E is defined as $=_E \circ \to_R \circ =_E$. Let $\to \subseteq A \times A$ be a binary relation on a set A. We denote the transitive closure by $\to^+$, the reflexive and transitive closure by $\to^*$, and rewriting up to normal forms by $\to^!$.

Example 1.1 *Consider the following rewrite theory* $(\Sigma, \Delta \cup B, R)$ *such that* $\mathcal{C} = \{b, c, e\}$, $\mathcal{D}_1 = \{a, d\}$, $\mathcal{D}_2 = \{f\}$, $\Delta = \{a = b, d = e\}$, *and* $R = \{f(b, c) \to d\}$ *where* B *contains the commutativity axiom for* f. *Then we can* R/E*-rewrite term* $f(c, a)$ *to* e *by means of the following* $\to_{R/E}$ *rewrite sequence* $f(c, a) =_\Delta f(c, b) =_B f(b, c) \to_R d =_\Delta e$.

We say that a rewrite theory $\mathcal{R} \equiv (\Sigma, \Delta \cup B, R)$ is *terminating* w.r.t. $\to_{R/E}$, if there exists no infinite rewrite sequence $t_1 \to_{R/E} t_2 \to_{R/E} \ldots$ A rewrite theory is *confluent* w.r.t. $\to_{R/E}$ if, for all terms s, t_1, t_2, such that $s \to^*_{R/E} t_1$ and $s \to^*_{R/E} t_2$, there exists a term t s.t. $t_1 \to^*_{R/E} t$ and $t_2 \to^*_{R/E} t$.

Description logic. Description Logics (DLs) are decidable logic formalisms for representing knowledge of application domains and reasoning about it.

In DL, domains of interest are modeled as knowledge bases (i.e. ontologies) by means of concepts (classes), individuals (instances of classes) and roles (binary predicates).

In this section, we present the decidable description logic $\mathcal{SHOIQD}_n^-$ underlying the OWL-DL [165] framework. For a full discussion about OWL and DL formalisms, we respectively refer to [69] and [19].

Let Σ_D be a signature containing all the symbols of the considered DL language. A concept C is defined using the following constructors of Σ_D.

$$\begin{array}{rcl} C ::= A & | & \text{(atomic concept)} \\ \top & | & \text{(universal concept)} \\ \bot & | & \text{(bottom concept)} \\ \neg C & | & \text{(concept negation)} \\ C_1 \sqcap C_2 & | & \text{(intersection)} \\ C_1 \sqcup C_2 & | & \text{(union)} \\ \exists R.C & | & \text{(full existential quantification)} \\ \forall R.C & | & \text{(value restriction)} \\ \geq_n R & | & \text{(at-least number restriction)} \\ \leq_n R & | & \text{(at-most number restriction)} \\ \{a_1, \ldots, a_n\} & | & \text{(individual set)} \end{array}$$

where A represents an atomic concept, C_1 and C_2 are concepts, R is a role, n is a natural number, and $a_1, \ldots, a_n$ are individuals. We use $\bot$ (resp., $\top$) to abbreviate the concept $C \sqcap \neg C$ (resp., $C \sqcup \neg C$). Besides, we make use of the role constructor $(\cdot)^-$ to define the *inverse* of a role R. The inverse of a role R is still a role and is denoted by R^-. Concepts and roles are related to each other using *terminological axioms* of the form $C_1 \sqsubseteq C_2$ (*. resp* $R_1 \sqsubseteq R_2$) or $C_1 \equiv C_2$ (*resp* $R_1 \equiv R_2$). A *TBox* is a finite set of terminological axioms. Given a concept C, a role R, and two individual a and b, a *concept assertion* is an expression of the form $C(a)$, while the expression $R(a, b)$ denotes a *role assertion*. An *ABox* is a finite set of concept and role assertions. A *knowledge base* $\mathcal{K}$ is a pair (TB, AB), where TB is a TBox and AB is an ABox.

From a semantic point of view, concepts are interpreted over finite subsets of a given domain. Concretely, an *interpretation* $\mathcal{J}$ is a pair $(\Delta^{\mathcal{J}}, \cdot^{\mathcal{J}})$ consisting of a non-empty set $\Delta^{\mathcal{J}}$, called *domain of the interpretation*, and an *interpretation function* $\cdot^{\mathcal{J}}$ which maps each atomic concept to a subset of $\Delta^{\mathcal{J}}$, each role to a subset of $\Delta^{\mathcal{J}} \times \Delta^{\mathcal{J}}$, and each individual to an element of $\Delta^{\mathcal{J}}$. The interpretation function is lifted to the DL constructs mentioned above in the usual way (see Figure 1.1).

$$\begin{array}{l} \top^{\mathcal{J}} = \Delta^{\mathcal{J}}, \\ \bot^{\mathcal{J}} = \emptyset, \\ (\neg C)^{\mathcal{J}} = \Delta^{\mathcal{J}} \setminus C^{\mathcal{J}}, \\ (C_1 \sqcap C_2)^{\mathcal{J}} = C_1^{\mathcal{J}} \cap C_2^{\mathcal{J}}, \\ (C_1 \sqcup C_2)^{\mathcal{J}} = C_1^{\mathcal{J}} \cup C_2^{\mathcal{J}}, \\ (\exists R.C_1)^{\mathcal{J}} = \{a \in \Delta^{\mathcal{J}} \mid \exists b \in \Delta^{\mathcal{J}}\ (a, b) \in R^{\mathcal{J}} \wedge b \in C_1^{\mathcal{J}}\}, \\ (\forall R.C_1)^{\mathcal{J}} = \{a \in \Delta^{\mathcal{J}} \mid \forall b \in \Delta^{\mathcal{J}}\ (a, b) \in R^{\mathcal{J}} \Longrightarrow b \in C_1^{\mathcal{J}}\}, \\ (\geq_n R)^{\mathcal{J}} = \{a \in \Delta^{\mathcal{J}} \mid |\{b \mid (a, b) \in R^{\mathcal{J}}\}| \geq n\}, \\ (\leq_n R)^{\mathcal{J}} = \{a \in \Delta^{\mathcal{J}} \mid |\{b \mid (a, b) \in R^{\mathcal{J}}\}| \leq n\}, \\ (\{a_1, \ldots, a_n\})^{\mathcal{J}} = \{a_1^{\mathcal{J}}, \ldots, a_n^{\mathcal{J}}\}, \\ (R^-)^{\mathcal{J}} = \{(b, a) \in \Delta^{\mathcal{J}} \times \Delta^{\mathcal{J}} \mid (a, b) \in R^{\mathcal{J}}\} \end{array}$$

Figure 1.1: Semantics of DL constructs.

$$\mathcal{J} \models C_1 \sqsubseteq C_2 \iff C_1^{\mathcal{J}} \subseteq C_2^{\mathcal{J}}$$
$$\mathcal{J} \models C_1 \equiv C_2 \iff C_1^{\mathcal{J}} = C_2^{\mathcal{J}}$$
$$\mathcal{J} \models R_1 \sqsubseteq R_2 \iff R_1^{\mathcal{J}} \subseteq R_2^{\mathcal{J}}$$
$$\mathcal{J} \models R_1 \equiv R_2 \iff R_1^{\mathcal{J}} = R_2^{\mathcal{J}}$$
$$\mathcal{J} \models C(a) \iff a^{\mathcal{J}} \in C^{\mathcal{J}}$$
$$\mathcal{J} \models R(a,b) \iff (a^{\mathcal{J}}, b^{\mathcal{J}}) \in R^{\mathcal{J}}$$

Figure 1.2: Satisfaction of DL axioms and DL assertions.

The notion of satisfaction $\models$ of an axiom or assertion α w.r.t. an interpretation $\mathcal{J}$ (in symbols $\mathcal{J} \models \alpha$) is defined in Figure 1.2. An interpretation $\mathcal{J}$ is a *model* of a knowledge base $\mathcal{K} = (\mathrm{TB}, \mathrm{AB})$, which is denoted by $\mathcal{J} \models \mathcal{K}$, iff $\mathcal{J} \models \alpha$, for each $\alpha \in \mathrm{TB} \cup \mathrm{AB}$.

A concept C is *satisfiable* in a knowledge base $\mathcal{K}$ iff there exists a model $\mathcal{J}$ of $\mathcal{K}$ such that $C^{\mathcal{J}} \neq \emptyset$. A concept C_1 *is subsumed* by a concept C_2 in a knowledge base $\mathcal{K}$ (in symbols, $\mathcal{K} \models C_1 \sqsubseteq C_2$) iff for each model $\mathcal{J}$ of $\mathcal{K}$, $C_1^{\mathcal{J}} \subseteq C_2^{\mathcal{J}}$. Subsumption can be reduced to satisfiability, that is, $\mathcal{K} \models C_1 \sqsubseteq C_2$ iff $C_1 \sqcap \neg C_2$ is not satisfiable in $\mathcal{K}$. We assume that concept satisfiability and concept subsumption can be checked by means of a DL reasoner by invoking the basic reasoning services *satisfiable*(C) and *subsume*(C_1, C_2), where C, C_1 and C_2 are concepts. The other reasoning services considered are listed in Table 1.1.

Query construct	**Description**
allConcepts()	All concepts defined in the ontology
allRoles()	All roles defined in the ontology
allIndividuals()	All individuals defined in the ontology
satisfiable(C)	Is C satisfiable?
subsumes(C_1, C_2)	Does $C_2 \sqsubseteq C_1$?
disjoint(C_1, C_2)	Does $C_1 \sqcap C_2 \equiv \varnothing$?
children(C)	All concepts which are children of C
equivalents(C)	All concepts which are equivalent to C
instances(C)	All individuals which belong to C
instanceOf(a,C)	Does a belong to C?
roleFillers(a, R)	All individuals b such that *R(b,a)* holds
related(R)	All pairs *(a,b)* such that *R(a,b)* holds

Table 1.1: Reasoning services

A *DL query* is an expression $DL(\mathcal{K}, r)$ where $\mathcal{K}$ is a knowledge base and r is a reasoning service. Basically, a DL query, when evaluated, executes a given reasoning service against a knowledge base and returns a value, which can be either a boolean constant or a set of values. We call *boolean* (respectively, *non-boolean*) DL query, any DL query whose executions returns a boolean value (respectively, a set of values).

Example 1.2 *Let $\mathcal{H}$ be a knowledge base modeling an healthcare domain. Assume that $\mathcal{H}$ includes the atomic concepts:* patient*,* physician*,* guardian*,* admin*; and the role* assignedTo*, which establishes who are the people designated to take care of a given patient. Moreover, the ABox of $\mathcal{H}$ is populated by the following concept and role assertions:*

$$patient(CharlieBrown), patient(DavidBowie), patient(CharlieChaplin),$$
$$admin(JohnNash), physician(BobMarley), physician(AliceP.Liddell),$$
$$guardian(FrankSinatra), assignedTo(AliceP.Liddell, CharlieBrown),$$
$$assignedTo(FrankSinatra, CharlieChaplin).$$

Now, consider the following DL queries Q_1, Q_2 and Q_3:

$$DL(\mathcal{H}, instance(BobMarley, physician \sqcap \exists assignedTo.\{CharlieBrown\}))$$
$$DL(\mathcal{H}, subsumes(\neg physician, admin)), DL(\mathcal{H}, instances(guardian \sqcup physician)$$

Q_1 and Q_2 are boolean DL queries, while Q_3 is a non-boolean DL query. More specifically, Q_1 asks $\mathcal{H}$ whether BobMarley *is the designated physician of patient* CharlieBrown*, in this case the execution of Q_1 returns* false*, since* CharlieBrown*'s designated physician is* AliceP.Liddell*. Q_2 checks whether concept* admin *is subsumed by concept* ¬physician*, that amounts to saying there are no administrators who are also physicians. Finally, the evaluation of Q_3 computes the set* {FrankSinatra, BobMarley, AliceP.Liddell} *representing all the individuals belonging to the union of concepts* guardian *and* physician*.*

Our description logic formalism is completely ground, that is, logic formulae do not contain variables. In particular, variables cannot appear in DL queries, but sometimes it might be convenient to generalize the notion of DL query by admitting the use of variables. In this way, DL queries may be (i) easily reused, and (ii) ground values associated with the considered variables may be computed at run-time. In light of these considerations, we define the notion of DL query template as follows. A *reasoning service template* is defined as a reasoning service that may contain variables playing the role of placeholders for concepts, roles, and individuals. A *DL query template* is an expression $DL(\mathcal{K}, r)$ where $\mathcal{K}$ is a knowledge base, and r is a reasoning service template. In the same way, it is sometimes useful to allow DL queries to contain function calls computing atomic concepts, roles and individuals. We define *functional service template* a reasoning service template that may contain functional calls playing the role of placeholders for atomic concepts, roles, and individuals. A *DL functional query template* is an expression $DL(\mathcal{K}, r)$ where $\mathcal{K}$ is a knowledge base, and r is a functional service template.

It is worth noting that a DL (functional) query template cannot be executed by a DL reasoner, since only ground formulae without function calls can be evaluated by the reasoner. Thus, in order to make a DL (functional) query template executable using a standard reasoner, we need to made the query ground and to evaluate all the functions calls before sending the query to the reasoner.

Example 1.3 *Consider the knowledge base $\mathcal{H}$ of Example 1.2 and let ++ denote the string concatenation operator. The following expression is a DL functional query template:*

$$DL(\mathcal{H}, instances(physician \sqcap \exists assignedTo.\{X{+}{+}Y\})),$$

where the individual full name is computed by concatenating the values associated with variables X and Y. Note that the evaluation of the template depends on the concrete values assigned to X and Y. For instance, if X was bound to Charlie *and Y was bound to* Brown*, the result of the evaluation would be* {AliceP.Liddell}*, while we would obtain the empty set in the case when X and Y were associated with values* David *and* Bowie *respectively.*

A query *evaluation* function is a mapping $eval\colon \Delta \rightarrow \mathcal{T}_{\Sigma_D}$ (where Δ denotes the set of all possible DL queries) which takes a DL query as input and returns a data term (typically a boolean value or a list of values belonging to the knowledge base of interest). Thus, by $eval(DL(\mathcal{K}, r))$, we denote the *evaluation* of the DL query $DL(\mathcal{K}, r)$, that is, the result of the execution of the reasoning service r against the knowledge base $\mathcal{K}$.

Example 1.4 *Consider the DL queries Q_1, Q_2 and Q_3 of Example 1.2. Then, $eval(Q_1) = false$, $eval(Q_2) = true$, $eval(Q_3) = \{FrankSinatra, AliceP.Liddell, BobMarley\}$.*

I

A Rule-based Approach to Security Analysis and Certification

2

Program Transformation for Software Certification

Transforming programs automatically to optimize their efficiency is one of the most fascinating techniques for rule–based programming languages [68, 160]. One of the most extensively studied program transformation approaches is the so called *fold/ unfold* transformation system [46, 47, 66] (also known as the *rules+strategies* approach [135]). The folding and unfolding transformations were first introduced by Burstall and Darlington [47] and later introduced in logic programming by Komorowski [109]. The combined effect of unification with rewriting by means of narrowing was first proposed in [10] and was also achieved in [71, 72, 138] by means of a superposition procedure for program synthesis. Unlike the case of pure logic or pure functional programs, where unfolding is correct w.r.t. practically all available semantics, unrestricted unfolding using narrowing does not preserve program meaning, even when we consider the normalization semantics (i.e., the set of normal forms) or the evaluation semantics (i.e., the set of values) of the program. In [10], some conditions were ascertained which guarantee that an equivalent program w.r.t. the semantics of computed answers is obtained for functional logic programs.

Fold/Unfold based Program Transformation. Unfolding is essentially the replacement of a call by its body, with appropriate substitutions. Folding is the inverse transformation, i.e., the replacement of some piece of code by an equivalent function call. For functional programs, folding and unfolding steps involve only pattern matching. The fold/unfold transformation approach was first adapted to logic programs by Tamaki and Sato [151] by replacing pattern matching with unification in the transformation rules. Folding and Unfolding are the essential rules in this program transformation approach, but there are other rules which have been usually considered, such as, instantiation, definition introduction/elimination and abstraction (sometimes referred with different names).

When performing program transformation we may end up with a final program which is equal to the initial one, since the folding rule is the inverse of the unfolding rule. Thus, during the transformation process, we need *strategies* which guide the application of the transformation rules and can allow one to derive programs with improved performance. Some popular transformation strategies which have been pro-

posed in the literature are the *composition* and *tupling* strategies. The *composition* strategy [47] is used to avoid the construction of intermediate data structures that are produced by some function g and consumed by another function f. For some class of programs the composition strategy can be applied automatically. The *tupling* strategy [47, 67] proceeds by grouping calls with common arguments together so that their results are computed simultaneously. Unfortunately, the tupling strategy is more involved than the composition strategy and can in general be obtained only semi-automatically (although for particular classes of programs the tupling strategy has been completely automated [55, 56]).

A lot of literature has been devoted to proving the correctness of fold/unfold systems w.r.t. the various semantics proposed for functional programs [47, 110], logic programs [103, 134, 146, 151], functional logic programs [11], and constraint logic programs [84]. Quite often, however, transformations may have to be carried out in contexts in which the function symbols satisfy certain *equational axioms*. For example, in rule-based languages such as ASF+SDF [37], Elan [43], OBJ [93], CafeOBJ [76], and Maude [60], some function symbols may be declared to obey given algebraic laws (the so-called *equational attributes* of OBJ, CafeOBJ and Maude), whose effect is to compute with equivalence classes modulo such axioms while avoiding the risk of non-termination. Similarly, theorem provers, both general first-order logic ones and inductive theorem provers, routinely support commonly occurring equational theories (e.g. associative-commutative theories) for some function symbols. Moreover, several of the above-mentioned languages and provers have an expressive *order-sorted* typed setting with sorts and subsorts (where subsort inclusions form a partial order and are interpreted semantically as set-theoretic inclusions of the corresponding data sets). The unfolding transformation has been scarcely studied so far for rewriting logic theories that may include sorts, rules, equational theories, and algebraic laws (such as commutativity and associativity).

In this chapter, we first formalize a powerful narrowing-based unfolding transformation for rewriting logic theories that preserves the rewriting logic semantics of the original theory. Our technique relies on the fact that rewriting logic also supports the narrowing mechanism [58] that successfully combines term rewriting and unification [87] and is efficiently implemented in the functional programming language Maude [60]. Roughly speaking, unfolding is defined by applying narrowing steps to the right-hand sides of both rules and equations of the rewrite theory under examination in order to obtain the unfolded theory. Narrowing allows us to empower the unfold operation by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework [47] that introduces an instance of an existing rule) into unfolding by means of unification. A related but different unfolding technique for transforming (canonical) conditionals TRSs, is proposed in [10], where the main goal is to preserve the semantics of (narrowing) computed answers. Here, a completeness result is proved for left-linear and L-closed programs, where the closedness notion compares all calls in the right-hand side of the program rules w.r.t. the left-hand side of the rules similarly to the closedness notion used in Partial Evaluation [12, 13]. Then, a generalized notion of unfolding is provided, which, in the case of unconditional programs, keeps the original rule into the transformed program. With this

generalized unfolding operation, completeness holds under less demanding conditions. In this work, we consider possibly non-confluent and non-terminating rewriting logic theories, and we study the unfolding operation w.r.t. the standard rewriting logic semantics of ground normal forms. Thus, in our setting, no notion similar to the L-closedness is needed.

However, there are pathological situations where unfolding may cause incompleteness. Hence, we develop a transformation methodology that is able to determine whether an unfolding operation would cause incompleteness and overcome this problem by deriving a set of new rules that are added to the transformed program in order to preserve the semantics of the original program.

Once we have formalized the unfolding operation, we propose the first fold/unfold framework in the literature that applies to rewriting logic theories [117] and we prove its correctness. Our methodology considers the possibility of transforming the equation set and the rule set of a rewrite theory separately in a way the semantics of ground reducts is proved to be preserved. The auxiliary transformation rules adopted apart from fold and unfold are: definition introduction and elimination, and abstraction. In this approach, the goal of obtaining a correct and efficient program is achieved in two phases, which may be performed by different actors: the first phase consists in writing an initial, maybe inefficient, program whose correctness can be easily shown; the second phase consists in transforming the initial program in order to obtain a more efficient one. This is done by constructing a sequence of equivalent programs —called *transformation sequence* and denoted by $\mathcal{R}_0, \ldots, \mathcal{R}_n$—where each program $\mathcal{R}_i$ is obtained from the preceding ones $\mathcal{R}_0, \ldots, \mathcal{R}_{i-1}$ by using a transformation rule.

A different approach to program transformation is proposed in [54], where a term rewriting transformation framework is formalized by using templates. In this approach, programs are expressed as TRSs [107], and are transformed according to a given program transformation template expressed as a TRS too. Such a template consists of program schemas for input and output programs and a set of equations that the input and output programs must validate to guarantee the correctness of the transformation. A library of templates that matches the structure of the programs is required, otherwise the transformation cannot be applied.

Software certification. With the advent of the phenomenon of *mobile code* the security of software obtained from remote systems has become a critical problem. Mobile code is software transferred between systems and executed on a local system without explicit installation or execution by the recipient even though it is delivered through an insecure network or from an untrusted source. Important examples of mobile code include Web applets, actor-based distributed system software, and updates to embedded computers. During delivery, code might be corrupted, or a malicious hacker might change the code. Potential problems can be summarized as security problems (i.e. unauthorized access to data or system resources), safety problems (i.e. illegal operations or illegal access to memory), or functional incorrectness (i.e the delivered code fails to satisfy a requires relation between its input and output). Hence, code consumers need assurance that the software is not corrupted or harmful, whether intentionally or inadvertently. Furthermore, requirements should be satisfied without

requiring run-time checks, if possible, in order to avoid performance degradation.

Proof-Carrying Code (PCC) [131] and Code-Carrying Theory (CCT) [157, 158] are two alternatives among other solutions to these problems. In comparison with other solutions, these two alternatives can generally provide stronger assurance of secure delivery of code with all required security properties (e.g. no unauthorized access to classified data), safety properties (e.g. no out-of-bounds array-indexing), or functional correctness properties (e.g. an algorithm sorts its input) preserved. The basic idea of PCC is that a code consumer does not accept delivery of new code unless it is accompanied by a formal proof of required safety, security, or correctness properties that can be checked by the code consumer. One way of doing this is to attach to the code an easily-checkable proof at the code producer's site. This proof must be checked by the code consumer and should prove that the code does not violate predefined requirements. Code-Carrying Theory (CCT), is an alternative to PCC with similar goals and technology, but it is based on the idea of proof-based program synthesis rather than program verification. The basic idea is that a set of axioms that defines functions are provided by the code producer as well as proofs that the defined functions obey certain requirements. The form of the function-defining axioms is such that it is easy to extract executable code from them. Thus all that has to be transmitted from producer to consumer is a theory (a set of axioms and theorems) and a set of proofs of the theorems. There is no need to transmit code explicitly.

PCC has been developed and applied primarily as a method for achieving safety, but in principle it can be used with the other forms of requirements. CCT could also be used with any of the three forms of requirements, but in this chapter we focus on applying CCT to functional correctness. In Section 2.5, we show how we can take advantage of the proposed Fold/Unfold transformation framework to implement a CCT methodology that reduces the burden on the code producer. In conjunction with either PCC or CCT, one could employ additional certification techniques, such as encrypted signatures or check-sums; Devanbu et al. [74] discuss such combinations as well as the use of "trusted hardware."

2.1 Narrowing in Rewriting Logic

Consider the rewrite relation $\rightarrow_{R/E}$ introduced in Chapter 1. Since E-congruence classes can be infinite, $\rightarrow_{R/E}$-reducibility is undecidable in general. One way to overcome this problem is to implement R/E-rewriting by a combination of rewriting using oriented equations (oriented from left to right) and rules [159]. We define the relation $\rightarrow_{\Delta,B}$ on $\mathcal{T}_\Sigma(\mathcal{V})$ as follows: $t \rightarrow_{\Delta,B} t'$ if there is a position $p \in O_\Sigma(t)$, $l = r$ in Δ, and a substitution σ such that $t|_p =_B l\sigma$ and $t' = t[r\sigma]_p$. The relation $\rightarrow_{R,B}$ is similarly defined, and we define $\rightarrow_{R\cup\Delta,B}$ as $\rightarrow_{R,B} \cup \rightarrow_{\Delta,B}$. The idea is to implement $\rightarrow_{R/E}$ using $\rightarrow_{R\cup\Delta,B}$.

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:

(i) B is *non-erasing*, and *sort preserving*.

(ii) B has a finitary and complete unification algorithm, which implies that B-matching is decidable, and $\Delta \cup B$ has a complete (but not necessarily finite) unification algorithm.

(iii) Δ is *sort decreasing*, and *confluent and terminating modulo* B.

(iv) $\rightarrow_{\Delta,B}$ is *coherent with* B, i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \rightarrow^+_{\Delta,B} t_2$ and that $t_1 =_B t_3$ implies $\exists t_4, t_5$ such that $t_2 \rightarrow^*_{\Delta,B} t_4$, $t_3 \rightarrow^+_{\Delta,B} t_5$, and $t_4 =_E t_5$.

(v) $\rightarrow_{R,B}$ is *E-consistent with* B, i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \rightarrow_{R,B} t_2$ and that $t_1 =_B t_3$ implies $\exists t_4$ such that $t_3 \rightarrow_{R,B} t_4$, and $t_2 =_E t_4$.

(vi) $\rightarrow_{R,B}$ is *E-consistent with* $\rightarrow_{\Delta,B}$, i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \rightarrow_{R,B} t_2$ and that $t_1 \rightarrow^*_{\Delta,B} t_3$ implies $\exists t_4, t_5$ such that $t_3 \rightarrow^*_{\Delta,B} t_4$, $t_4 \rightarrow_{R,B} t_5$, and $t_5 =_E t_2$[1].

A term t is called a *redex*, if there exist a rule $l \rightarrow r$, or equation $l = r$, and a substitution σ such that $t =_B l\sigma$. A term t without redexes is called a *normal form*. A rewrite theory $\mathcal{R}$ is *weakly normalizing* if every term t has a normal form in $\mathcal{R}$, though infinite rewrite sequences starting from t may exist. A rewrite theory is *sufficiently complete* [96] if enough rules/equations have been specified so that functions of the theory are fully defined on all relevant data (that is, defined symbols do not appear in any ground term in normal form).

Narrowing [87] generalizes term rewriting by allowing free variables in terms (as in logic programming) and by performing unification (at non-variable positions) instead of matching in order to (non–deterministically) reduce a term. The narrowing relation for rewriting logic theories is defined as follows [122].

Definition 2.1 ($R \cup \Delta, B$-Narrowing) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be an order-sorted rewrite theory. The $R \cup \Delta, B$-*narrowing *relation on $\mathcal{T}_\Sigma(\mathcal{V})$ is defined as $t \rightsquigarrow_{\sigma,p,R\cup\Delta,B} t'$ if there exist $p \in O_\Sigma(t)$, a rule $l \rightarrow r$ or equation $l = r$ in $R \cup \Delta$, and $\sigma \in CSU_B(t|_p, l)$ such that $t' = (t[r]_p)\sigma$. $t \rightsquigarrow_{\sigma,p,R\cup\Delta,B} t'$ is also called a $R \cup \Delta, B$-*narrowing step.

Example 2.2 *Consider the rewrite theory of Example 1.1 where we substitute the rule in R with the following rule $f(x, f(y, b)) \rightarrow d$. Then we can perform the narrowing step $f(f(w, z), c) \rightsquigarrow_{\sigma,\Lambda,R\cup\Delta,B} d$, with $\sigma = \{x/c, z/b, w/y\}$, since by the commutativity of f we have that $f(f(w, z), c)\{z/b, w/y\} =_B f(x, f(y, b))\{x/c\}$.*

When it is clear from the context, we omit $(R \cup \Delta, B)$ from the narrowing relation. Narrowing derivations are denoted by $t_0 \rightsquigarrow^*_\sigma t_n$, which is shorthand for the sequence of narrowing steps $t_0 \rightsquigarrow_{\sigma_1,p_1} \ldots \rightsquigarrow_{\sigma_n,p_n} t_n$ with $\sigma = \sigma_n \circ \ldots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). Completeness of narrowing for several meaningful classes of rewriting logic theories (e.g. topmost theories, linear theories, *etc.*) has been studied in [122].

[1] Properties (iv) and (v) can be achieved by a simple preprocessing of rewrite rules, while property (vi) is guaranteed by a discipline that prevents the defined function symbols of Δ to appear within the lhs's of the rules in R. For more details see Section 2.4.

In rewriting logic implementation such as Maude, defined symbols can be given the commutativity axiom or both commutativity and associativity, but not the associativity alone since unification modulo associativity is infinitary, i.e., infinitely many unifiers may exist modulo associativity [21].

In what follows, we always consider weakly normalizing and sufficiently complete rewrite theories. These conditions are essential in order to prove the correctness and completeness of the unfolding operation w.r.t. the considered semantics (i.e., Theorem 2.12).

2.2 The Unfolding Operation

Let us introduce the Unfolding operation.

Definition 2.3 (Unfolding) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a program and let F be an equation (*resp. *rule) of the form $l = r$ (*resp. *$l \to r$) in $\mathcal{R}$. We obtain a new program from $\mathcal{R}$ by replacing F with the set of equations (*resp. *rules)*

$$\{l\sigma = r' \mid r \leadsto_{\sigma,\Delta,B} r' \text{ is a } \Delta, B \text{ narrowing } step\}$$

$$\{l\sigma \to r' \mid r \leadsto_{\sigma,R\cup\Delta,B} r' \text{ is a } R \cup \Delta, B \text{ narrowing } step\}$$

The following example suggests that right linearity must be required for completeness. For the sake of simplicity we omit sort declarations when specifying rewriting logic theories.

Example 2.4 *Consider the following rewrite theory $\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, where $\Sigma_{\mathcal{R}}$ is the signature containing all the symbols of R and*

	R:			R':		
1.	$f(a,a)$	$\to$	a	$f(a,a)$	$\to$	a
2.	$f(b,c)$	$\to$	b	$f(b,c)$	$\to$	b
3.	a	$\to$	b	a	$\to$	b
4.	a	$\to$	c	a	$\to$	c
5.	$g(x)$	$\to$	$f(x,x)$			
6.				$\mathbf{g(a)}$	$\to$	$\mathbf{a}$

We obtain program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from $\mathcal{R}$ by applying an unfolding step over the rule 5 in R, through the narrowing step $f(x,x) \leadsto_{x/a} a$. Let us consider term $g(a)$. In the original program, $g(a)$ can rewrite to the normal form b by the rewrite sequence: (i) $g(a) \to_5 f(a,a) \to_3 f(b,a) \to_4 f(b,c) \to_2 b$. In the transformed program, such a rewrite sequence is no longer possible from term $g(a)$, and, hence, the normal form b is lost.

We consider the standard semantics of rewrite theories given by the following definition.

Definition 2.5 (Program Semantics) *Given a rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, the semantics of $\mathcal{R}$ is the set $\mathtt{gred}(\mathcal{R}) = \{(t,s) \mid t \in \mathcal{T}_\Sigma, t \to^*_{R\cup\Delta,B} s\}$.*

Let us also denote by $\mathtt{gnf}(\mathcal{R})$ ($\subseteq \mathtt{gred}(\mathcal{R})$) the semantics of ground reducts in normal form, and by $(t, s) \in \mathtt{gnf}(E)$ the fact that s is the canonical form of t w.r.t. the equational theory $E = (\Delta \cup B)$.

Since we consider rewrite theories where defined symbols are allowed to be arbitrarily nested in left-hand sides of rules, rule unfolding may cause the loss of completeness for the transformed program w.r.t. the semantics of the original one. Let us illustrate this problem by means of some examples. Since the equational axioms for associativity and commutativity do not affect the incompleteness problem that we want to describe, for the sake of simplicity, in the following examples we consider defined symbols without any equational axiom. A discussion on equational axioms and incompleteness is postponed until Section 2.2.3.

Example 2.6 *Consider the following rewrite theory $\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, where $\Sigma_{\mathcal{R}}$ is the signature containing all the symbols of R and*

	R:			R':		
1.	$g_1(x)$	$\rightarrow$	x	$g_1(x)$	$\rightarrow$	x
2.	$h(x)$	$\rightarrow$	0	$h(x)$	$\rightarrow$	0
3.	$h(g_1(x))$	$\rightarrow$	1	$h(g_1(x))$	$\rightarrow$	1
4.	$f(x)$	$\rightarrow$	$g_1(x)$			
5.				$\mathbf{f(x)}$	$\rightarrow$	$\mathbf{x}$

We get program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from $\mathcal{R}$ by applying an unfolding step over rule 4 in R, through the narrowing step $g_1(x) \leadsto_{\varepsilon} x$. Term $h(f(0))$ can be rewritten in $\mathcal{R}$ to the normal forms 0 or 1 by means of the rewrite sequences $h(f(0)) \rightarrow_4 h(g_1(0)) \rightarrow_1 h(0) \rightarrow_2 0$, and $h(f(0)) \rightarrow_4 h(g_1(0)) \rightarrow_3 1$, respectively. The only possible rewrite sequences from $h(f(0))$ in $\mathcal{R}'$ are $h(f(0)) \rightarrow_2 0$, and $h(f(0)) \rightarrow_5 h(0) \rightarrow_2 0$, thus we miss normal form 1. In fact, symbol g_1 is needed for rule 3 to be applied, and function f provides that occurrence of g_1 needed to reach the normal form 1. However, the unfolding of rule 4 forces the occurrence of symbol g_1 to be evaluated, and, hence, that rewrite sequence is no longer available in $\mathcal{R}'$.

A naïve attempt to identify the rules that are involved in the loss of completeness might be to look for those rules whose left-hand sides contain an instance of the right-hand side of the rule that we want to unfold. In Example 2.6, the right-hand side of rule 4 is embedded in the left-hand side of rule 3. Hence, in order to avoid incompleteness, we could forbid the unfolding operation whenever one such a rule existed in the program. Unfortunately, as shown by Example 2.7, in general, incompleteness can be caused by the interference among several rules, which cannot be identified by using this naïve criterion.

Example 2.7 *Consider the following rewrite theory $\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, where $\Sigma_{\mathcal{R}}$ is the*

signature containing all the symbols of R and

	R:			R':		
1.	$g_1(x,0)$	$\to$	x	$g_1(x,0)$	$\to$	x
2.	$g_1(x,1)$	$\to$	x	$g_1(x,1)$	$\to$	x
3.	$g_1(0,g_1(x,y))$	$\to$	0	$g_1(0,g_1(x,y))$	$\to$	0
4.	$g_2(x)$	$\to$	x	$g_2(x)$	$\to$	x
5.	$h(x,y)$	$\to$	x	$h(x,y)$	$\to$	x
6.	$h(g_2(x),y)$	$\to$	$p(x,y)$	$h(g_2(x),y)$	$\to$	$p(x,y)$
7.	$p(x,y)$	$\to$	x	$p(x,y)$	$\to$	x
8.	$p(g_1(x,y),z)$	$\to$	1	$p(g_1(x,y),z)$	$\to$	1
9.	$k(x)$	$\to$	x	$k(x)$	$\to$	x
10.	$k(g_2(x))$	$\to$	1	$k(g_2(x))$	$\to$	1
11.	$f(x,y)$	$\to$	$g_2(g_1(x,y))$			
12.				$\mathbf{f(x,0)}$	$\to$	$\mathbf{g_2(x)}$
13.				$\mathbf{f(x,1)}$	$\to$	$\mathbf{g_2(x)}$
14.				$\mathbf{f(0,g_1(x,y))}$	$\to$	$\mathbf{g_2(0)}$
15.				$\mathbf{f(x,y)}$	$\to$	$\mathbf{g_1(x,y)}$

We obtain program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from $\mathcal{R}$ by applying an unfolding step over rule 11 in R, through the following narrowing steps: (i) $g_2(g_1(x,y)) \leadsto_{\varepsilon} g_1(x,y)$, (ii) $g_2(g_1(x,y)) \leadsto_{y/0} g_2(x)$, (iii) $g_2(g_1(x,y)) \leadsto_{y/1} g_2(x)$, and (iv) $g_2(g_1(x,y)) \leadsto_{x/0,y/g_1(x',y')} g_2(0)$. The following rewrite sequence can be proved in $\mathcal{R}$: $h(f(0,1),0) \to_{11} h(g_2(g_1(0,1)),0) \to_6 p(g_1(0,1),0) \to_8 1$. In $\mathcal{R}'$ we cannot reach the normal form 1 starting from term $h(f(0,1),0)$ because rules 6 or 8 cannot be applied. This is due to the fact that the occurrences of both symbols g_2 and g_1 is essential for rules 6 and 8 to be applied in order to obtain the normal form 1, while the unfolding step forces these occurrences to be evaluated. Therefore, in the transformed program, the rewrite sequence leading to normal form 1 is no longer viable. In this example, rules 6 and 8 are both involved in the loss of completeness.

The naïve idea outlined above to solve the case in Example 2.6 does not apply to Example 2.7 because the right-hand side of rule 11 does not appear in the left-hand side of any rule; however, it is distributed between the left-hand sides of rules 6 and 8.

In the following, we develop a methodology that is able to identify whether an unfolding operation causes incompleteness, and we overcome this problem by conveniently extending the transformed program. More precisely, according to the identified incompleteness sources, the methodology derives a set of new rules that are added to the transformed program in order to recover the ground semantics of the original program.

2.2.1 Analyzing potential incompleteness

Let $\mathcal{R} = (\Sigma, E, R)$ be a program, let $R^u : lhs_u \to rhs_u \in R$ be the rule we want to unfold and let $\mathcal{R}'$ be the program obtained from $\mathcal{R}$ by performing the unfolding

operation.

Step 1) *Looking for rules that may be involved in incompleteness.*
At the beginning, we look for rules in R whose left-hand side contains a proper subterm rooted by the root symbol of rhs_u. Let $\{R_1, \ldots, R_n\}$ be such a set of rules, and for each $lhs_i, i \in \{1, \ldots, n\}$, let $p_1, \ldots, p_{k_i}$ be the positions in lhs_i where an occurrence of the root symbol of rhs_u has been found. Then we construct the following set of terms $L = \{lhs_i[rhs_u]_{p_j} \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}\}$, where we replace the subterm rooted at position p_j in each lhs_i, with the right-hand side rhs_u. In order to avoid interference among the variables of rhs_u and the variables of the context $lhs_i[\,]_{p_j}$, we consider a variable renaming of rhs_u with fresh variables.

Finally, for each term $lhs_i[rhs_u]_{p_j}$, we try to perform just one narrowing step at the root position using the corresponding rule R_i. In symbols, we try to perform the following narrowing step: $lhs_i[rhs_u]_{p_j} \leadsto_{\sigma_j, \Lambda, (R_i \cup \Delta, B)} r'_j$. We collect the derived terms in a set T of triples of the form $(lhs_i[lhs_u]_{p_j}, \sigma_j, r'_j)$, where the first component is the lhs_i where we replaced the subterm rooted at position p_j with the left-hand side lhs_u. We consistently apply to lhs_u the same variable renaming applied to rhs_u.

Roughly speaking, if the considered narrowing step cannot be done, it follows that no rewrite step can be performed with rule R_i from any instance of term $lhs_i[rhs_u]_{p_j}$, and, hence, there is no incompleteness. Otherwise, the methodology proceeds to restore completeness.

Example 2.8 *Let us again consider the rules of Example 2.7. Recall that the rule for unfolding is $f(x, y) \to g_2(g_1(x, y))$. We first look for rules whose left-hand side contains a proper subterm rooted with symbol g_2, and we find rules 6 and 10. We then construct the set L that contains terms $h(g_2(g_1(w, z)), y)$ and $k(g_2(g_1(w, z)))$, and we try to perform a narrowing step at the root position from each of these terms by using rules 6 and 10, respectively.*
We can perform the following narrowing steps:
$h(g_2(g_1(w, z)), y) \leadsto_{\sigma, \Lambda, R_6} p(g_1(w, z), y)$, *where the computed unifier is* $\sigma = \{x/g_1(w, z), y/y\}$, *and* $k(g_2(g_1(w, z))) \leadsto_{\rho, \Lambda, R_{10}} 1$, *with* $\rho = \{x/g_1(w, z)\}$. *Finally, we construct the triples*
$(h(f(w, z), y), \{x/g_1(w, z), y/y\}, p(g_1(w, z), y))$ *and* $(k(f(w, z)), \{x/g_1(w, z)\}, 1)$.

Step 2) *Restoring Completeness.*
For each triple $(t_1, \sigma, t_2) \in T$, we add rule $t_1\sigma \to t_2$ to $\mathcal{R}'$. This guarantees that the ground semantics of $\mathcal{R}$ is preserved in the new program $\mathcal{R}'$, as stated by Theorem 2.12. In our example, we add rules $h(f(w, z), y) \to p(g_1(w, z), y)$ and $k(f(w, z)) \to 1$ to $\mathcal{R}'$.

Algorith 1 shows the backbone of the procedure that implements the methodology described above. The *restoreCompleteness* procedure takes the initial program $\mathcal{R}$, the transformed program $\mathcal{R}'$, and the right-hand side of the unfolded rule as arguments, and it returns $\mathcal{R}'$ extended with some new rules that are computed as explained above. The *getInvolvedRules* call detects the rules in $\mathcal{R}$ that contain a proper term

whose root symbol is $root(rhs_u)$ in their lhs. $subst\_rhs_u$ replaces the subterms rooted with the function symbol $root(rhs_u)$ in the lhs of the rules by term rhs_u, and *narrowingOneStep* tries to perform a narrowing step from the obtained terms by using the corresponding suspicious rules, obtaining the set of triples $\{(t_1, \sigma, t_2)\}$. Finally, for each one of these triples, the *prodRules* call returns a new rule of the form $t_1\sigma \to t_2$ to be added to the program $\mathcal{R}'$.

Algorithm 1 Procedure to check and restore completeness of unfolding

```
1: procedure RESTORECOMPLETENESS((Σ, E, R), (Σ, E, R'), rhs_u)
2:     {R_1, ..., R_n} ← getInvolvedRules(R, rhs_u|_Λ)
3:     L ← subst_rhs_u({lhs_1, ..., lhs_n}, rhs_u)
4:     {(t_1, σ, t_2)} ← narrowingOneStep(L, {R_1, ..., R_n})
5:     {t_1σ → t_2} ← prodRules({(t_1, σ, t_2)})
6:     return (Σ, E, R' ∪ {t_1σ → t_2})
7: end procedure
```

Example 2.9 *Consider again the Example 2.7. The call* $restoreCompleteness((\Sigma_{\mathcal{R}}, \emptyset, R), (\Sigma_{\mathcal{R}}, \emptyset, R'), g_2(g_1(x, y)))$ *yields* $(\Sigma_{\mathcal{R}}, \emptyset, R' \cup \{h(f(w, z), y) \to p(g_1(w, z), y),\ k(f(w, z)) \to 1\})$.

2.2.2 Methodology optimization

In the methodology above, in order to prevent a possible incompleteness problem, we add a rule of the form $t_1\sigma \to t_2$ to the transformed program $\mathcal{R}'$ for each triple (t_1, σ, t_2) found at Step 1, even if the transformed program is actually complete. Consider again the rules of Example 2.7, and term $k(f(x, y))$. By applying to $k(f(x, y))$ the substitution $\{y/0\}$ computed by the unfolding operation, we can rewrite the obtained term in the transformed program to the normal form 1, by means of the following rewrite sequence: $k(f(x, 0)) \to_{12} k(g_2(x)) \to_{10} 1$. Hence, the rule $k(f(w, z)) \to 1$ added to the transformed program by the methodology is redundant because rule 10 does not provoke incompleteness.

To refine the methodology, we can add an intermediate step that checks whether it is really necessary to add a new rule to the program. Let Σ_u denote the set of substitutions computed by narrowing during the unfolding operation extended with the empty substitution. Then, for each triple $(t_1, \sigma, t_2) \in T$, we want to check whether there exists $\sigma_u \in \Sigma_u$ such that t_2 is reachable from $(t_1\sigma)\sigma_u$ in $\mathcal{R}'$ by rewriting. If that is the case, there is no reason to add a rule that would be redundant; otherwise, this is a symptom of incompleteness, and we can proceed as in Step 2.

Example 2.10 *Consider again the Example 2.7, and the triples* $(h(f(w, z), y), \{x/g_1(w, z), y/y\}, p(g_1(w, z), y))$ *and* $(k(f(w, z)), \{x/g_1(w, z)\}, 1)$ *computed at Step 1. The set* Σ_u *contains the substitutions* $\{\varepsilon\}, \{y/0\}, \{y/1\}$, *and* $\{x/0, y/g_1(x', y')\}$. *Then, we check whether there exists* $\sigma_u \in \Sigma_u$ *such that* $h(f(w, z), y)\sigma_u \to^*_{\mathcal{R}'} p(g_1(w, z), y))$, *and whether there exists* $\sigma_u \in \Sigma_u$ *such that*

$k(f(w,z))\sigma_u \rightarrow^*_{\mathcal{R}'} 1$. *The first reachability goal is unsatisfiable, while the second is satisfied by substitutions* $\{y/0\}, \{y/1\}$, *and* $\{x/0, y/g_1(x', y')\}$.

Hence, the optimized solution is to add only the rule $h(f(w,z),y) \rightarrow (g_1(w,z),y))$ *to the transformed program.*

The reachability problem for rewriting is undecidable in general, but it has been proved to be decidable for particular classes of rewrite theories [88, 120]. For example, in [88] reachability is proved to be decidable for right-linear and right-shallow TRSs. The right-shallow property asks for variables that appear in the right-hand side of the rules to occur at depth 0 or 1. Hence, the proposed refinement has to pay the cost of the additional syntactic restrictions of right-linearity and right-shallowness to be effective. An alternative method to make reachability decidable is presented in [121], where the original rewrite theory is extended by adding a terminating and (ground) Church-Rosser set of extra equations powerful enough to collapse infinite sets of reachable terms into finite sets. Also in this case, several strong conditions are required on the extended rewrite theory in order to make such an analysis effective.

2.2.3 Incompleteness and Equational Axioms

Up to now, we have explained the incompleteness problem that may arise due to the unfolding operation, without considering equational axioms which can be associated with defined symbols. Nevertheless, the unfolding operation uses the $R \cup \Delta, B$ narrowing relation, which takes into account the equational axioms for associativity and commutativity. However, the axioms are not an extra source of incompleteness, as discussed below.

Let us modify the rewrite theory of Example 2.7 by declaring the symbols h, p and g_1 to obey associativity and commutativity. The transformed program will have a higher number of unfolded rules due to the increased number of unifiers computed by narrowing modulo the considered axioms, but exactly the same incompleteness problem arises. The new rules computed by unfolding are:

$$
\begin{array}{lrcl}
12. & f(x,0) & \rightarrow & g_2(x) \\
13. & f(0,x) & \rightarrow & g_2(x) \\
14. & f(x,1) & \rightarrow & g_2(x) \\
15. & f(1,x) & \rightarrow & g_2(x) \\
16. & f(0,g_1(x,y)) & \rightarrow & g_2(0) \\
17. & f(g_1(x,y),0) & \rightarrow & g_2(0) \\
18. & f(g_1(0,x),y) & \rightarrow & g_2(0) \\
19. & f(y,g_1(0,x)) & \rightarrow & g_2(0) \\
20. & f(x,y) & \rightarrow & g_1(x,y)
\end{array}
$$

and allow us to bring back the original semantics for the transformed program. Note that rules 13, 15, 17, 18, and 19 are needed because f is not associative neither commutative.

2.2.4 Completeness of the Transformation

The main result of this section is Theorem 2.12 which states that the unfolding transformation followed by the *restoreCompleteness* procedure preserves the ground semantics of a program. Moreover, the equational unfolding preserves the canonical forms as stated in Theorem 2.11.

Theorem 2.11 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a program, and let $\mathcal{R}' = (\Sigma, \Delta' \cup B, R)$ be the program obtained from $\mathcal{R}$ by unfolding an equation $E^u \in \Delta$. Then, $\mathtt{gnf}(\Delta) =_B \mathtt{gnf}(\Delta')$.*

Theorem 2.12 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a program, and let $\mathcal{R}' = (\Sigma, \Delta \cup B, R')$ be the program obtained from $\mathcal{R}$ by the unfolding of a rule $R^u \in R$ and the* restoreCompleteness *procedure. Then, for each term $t \in \mathcal{T}_\Sigma$, we have that:*

- $t \rightarrow^*_{\mathcal{R}'} s' \Rightarrow t \rightarrow^*_{\mathcal{R}} s$ *and* $s =_{\Delta,B} s'$;
- $t \rightarrow^*_{\mathcal{R}} s \Rightarrow t \rightarrow^*_{\mathcal{R}'} s'$ *and* $\exists s''$ *s.t.* $s \rightarrow^*_{\mathcal{R}} s''$, $s' =_{\Delta,B} s''$.

Basically, Theorem 2.12 states that (i) the ground reducts of the transformed program are exactly the same as in the original one (in symbols, $\mathtt{gred}(\mathcal{R}') \subseteq \mathtt{gred}(\mathcal{R})$), and (ii) for each ground reduct s of the original program, there exists s' in the transformed one such that s can still be reduced to a term that is equivalent to s'. This asymmetry in the result is due to the nature of unfolding. In fact, the unfolding of a rule in the initial program forces some symbols that appear in its right-hand side to be reduced by narrowing, and, hence, a general reduct s obtained by an application of that rule may contain those symbols. Therefore, we need to consider the possibility of some further reduction steps from s in the initial program in order to reduce those symbols and thereby obtain an equivalent term to the one reachable in the transformed program.

This result is obtained as a corollary of Lemma 2.18 which states that the ground normal forms are preserved. The following definitions, propositions and lemmas are auxiliary.

Definition 2.13 (B-Matching) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory. Given two terms t and s (not just variables), we say that t B-matches s at position $p \in O_\Sigma(s)$, if there exists a substitution σ such that $t\sigma =_B s|_p$.*

Proposition 2.14 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory, let t_1, t_2 be two terms such that $Var(t_1) \cap Var(t_2) = \emptyset$, and let $CSU_B(t_1, t_2)$ be the complete set of B-unifiers of t_1 and t_2. Let also θ be a ground substitution such that t_2 B-matches $t_1\theta$ at position Λ. Then, there exists a substitution $\sigma \in CSU_B(t_1, t_2)$, such that the restriction of σ to the variables of t_1 is more general than θ.*

Proof 2.2.1 *From the hypothesis it follows that there exists a substitution ρ such that $t_2\rho =_B t_1\theta$. Since t_1 and t_2 do not have shared variables, we can define a substitution η as the union of θ and ρ, such that $\eta|_{Var(t_1)} = \theta$ and $\eta|_{Var(t_2)} = \rho$. Therefore, η*

is a B-unifier of t_1 and t_2, that is, $t_1\eta =_B t_2\eta$. From the definition of CSU_B, we know that there exists a substitution $\sigma \in CSU_B(t_1, t_2)$ such that $\sigma \leq_B \eta$. Hence, $\sigma|_{Var(t_1)} \leq_B \eta|_{Var(t_1)} = \theta$.

Proposition 2.15 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a program, and let $\mathcal{R}' = (\Sigma, \Delta \cup B, R')$ be the program obtained from $\mathcal{R}$ by the unfolding of a rule $R^u : lhs_u \to rhs_u \in R$ and the* restoreCompleteness *procedure. Then, for each term $lhs_i[rhs_u]_{p_j}$ in L, we have that if $lhs_i[rhs_u]_{p_j} \rightsquigarrow_{\sigma,\Lambda,(R_i \cup \Delta, B)} t$, then $lhs_i[lhs_u]_{p_j}\sigma \to t$ in $\mathcal{R}'$.*

Proof 2.2.2 *The proof follows immediately from the described methodology; indeed, if $lhs_i[rhs_u]_{p_j} \rightsquigarrow_{\sigma,\Lambda,(R_i \cup \Delta, B)} t$, then the triple $(lhs_i[lhs_u]_{p_j}, \sigma, t)$ belongs to set T. Thus, at Step 2 we add the rule $lhs_i[lhs_u]_{p_j}\sigma \to t$ to program $\mathcal{R}'$, which implies the thesis.*

Lemma 2.16 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a program and $\mathcal{R}' = (\Sigma, \Delta \cup B, R')$ the program obtained from $\mathcal{R}$ by the unfolding of a rule $R^u : lhs_u \to rhs_u \in R$ and the* restoreCompleteness *procedure. Let $R_i : lhs_i \to rhs_i$, $i \in \{1, \ldots, n\}$ be the rules returned by the getInvolvedRules call of the* restoreCompleteness *procedure. Let t be a ground term such that $t \to_{R^u,p} t' \to_{R_i,p'} t''$ such that $p' < p$, and the occurrence of $root(rhs_u)$ at some position p_j in lhs_i matches with the occurrence of $root(rhs_u)$ at position p in t'. Then, $t \to^* t''' =_{\Delta \cup B} t''$ in $\mathcal{R}'$.*

Proof 2.2.3 *From the hypothesis it follows that $t|_p =_B lhs_u\theta$ for some grounding substitution θ, and $t|_{p'} =_B lhs_i[lhs_u]_{p_j}\theta$. It follows that t' must be of the form $t[lhs_i[rhs_u]_{p_j}]_{p'}\theta$. Moreover, $lhs_i[rhs_u]_{p_j}\theta$ B-matches with lhs_i. From the methodology described in Section 2.2.1, we know that narrowing computes the complete set of B-unifiers $CSU_B(lhs_i[rhs_u]_{p_j}, lhs_i)$. By Proposition 2.14 it follows that there exists $\sigma \in CSU_B(lhs_i[rhs_u]_{p_j}, lhs_i)$ such that σ is more general than θ (by renaming $lhs_i[rhs_u]_{p_j}$ with t_1 and lhs_i with t_2). Then, there exists a substitution ρ such that $\theta =_B \sigma\rho$. It follows that $t[lhs_i[rhs_u]_{p_j}]_{p'} \rightsquigarrow_{\sigma,p',(R_i \cup \Delta, B)} t^*$ and $t^*\rho =_B t''$. By Proposition 2.15 it follows that $t[lhs_i[lhs_u]_{p_j}]_{p'}\sigma \to^* t^*$ in $\mathcal{R}'$. Hence, $t = t[lhs_i[lhs_u]_{p_j}]_{p'}\theta =_B (t[lhs_i[lhs_u]_{p_j}]_{p'}\sigma)\rho \to^*_{R',\Delta \cup B} t^*\rho =_B t''$.*

Since we ask for Δ to be Church-Rosser and terminating modulo B, the equational unfolding preserves the canonical forms, as stated in Theorem 2.11.

Proof 2.2.4 (Proof of Theorem 2.11) *Let E^u be an equation of the form $(lhs_u = rhs_u)$ and let $E_0, \ldots, E_k$ be the set of equations used to unfold E^u, each one of the form $(l_i = r_i)$ for $i = 0, \ldots, k$. Let $f_1, \ldots, f_n$ with $n \leq k$ be the set of symbols defined by equations $E_1, \ldots, E_k$. Also let $rhs_u \rightsquigarrow_{\sigma_j,\Delta,B} r'_j$ $(j \in \{1, \ldots, n\})$ be the Δ, B-narrowing step such that the result of unfolding E^u using E_j is the equation $E^u_j : (lhs_u\sigma_j = r'_j)$. From the definition and the correctness of narrowing, we recall that:*

(1) $\forall j \in \{1, \ldots, n\} \,.\, rhs_u\sigma_j \to^{E_j} r'_j$

(2) $\forall j \in \{1, \ldots, n\}$ there exists position $p_j \in O_\Sigma(rhs_u)$ such that $rhs_u|_{p_j}\sigma_j =_B l_j\sigma_j$

(3) $\forall j \in \{1,\ldots,n\}\,.\,r'_j = (rhs_u[r_j]_{p_j})\sigma_j$

$\Rightarrow$ *We want to prove that, given any ground term* t*, if* $t \to^!_{\Delta,B} s$*, then* $t \to^!_{\Delta',B} s'$ *and* $s =_B s'$*. From* $t \to^!_{\Delta,B} s$*, the Church-Rosser property, and the termination of* Δ *modulo* B*, there exists a rewrite sequence from* t *to* s *where the left-most inner-most redex is reduced at each step. We will prove the result by induction on the length of this rewrite sequence.*
($n = 0$*.) This case is immediate since* $t =_B s$*.*
($n > 0$*.) Let us decompose the rewriting sequence from* t *to* s *as follows:* $t \to t_1 \to^! s$*. On the rewriting sequence from* t_1 *to* s *we can apply the induction hypothesis, and we now concentrate on the first rewriting step. If* t *rewrites to* t_1 *without using equation* E^u*, the same step can be performed in* Δ' *and the claim holds. Otherwise, there exists a position* $p \in O_\Sigma(t)$ *and a substitution* θ *such that (i)* $lhs_u\theta =_B t|_p$*, (ii)* $t|_p$ *is the left-most inner-most redex, and (iii)* $t_1 = t[rhs_u\theta]_p$*. Note that from (ii) and the sufficient completeness of* Δ*, it follows that (iv)* θ *is a constructor substitution, that is, for each* $x/t \in \theta$*,* t *is a constructor term. From (ii) and (iv), it follows that if* $rhs_u\theta$ *contains a redex, it is the left-most inner-most redex in* t_1 *and its position* p' *belongs to* $O_\Sigma(rhs_u)$*. Since* rhs_u *contains at least one occurrence of the symbols* $f_1,\ldots,f_n$*, and* Δ *is sufficient complete,* $rhs_u\theta$ *contains at least one redex. Let* p' *be the position of the left-most inner-most redex inside* t_1*. Now, consider the following rewrite step* $rhs_u\theta \to_{p',E_j} rhs_u[r_j]_{p'}\theta$*,* $j \in \{1,\ldots,k\}$*, which rewrites the redex in position* p'*. The obtained term* t_2 *is* $t[rhs_u[r_j]_{p'}\theta]_p$*. Since during the unfold operation, we perform narrowing at each possible position in* rhs_u*, the narrowing step* $rhs_u \leadsto_{p',\sigma_j,E_j\cup B} r'_j$ *can be proven in* Δ*. By (iv) and the completeness of narrowing, the substitution computed by narrowing is more general then* θ*, which amounts to saying that there exists a substitution* ρ *such that (v)* $\theta =_B \sigma_j\rho$*. By the definition of unfolding, the equation* $lhs_u\sigma_j = r'_j$ *is one* E^u_j *belonging to* Δ'*. Finally, from (i) and (v), we can apply the equation* E^u_j *to term* t*, thus obtaining* $t[r'_j\rho]_p = t[((rhs_u[r_j]_{p'})\sigma_j)\rho]_p =_B t[rhs_u[r_j]_{p'}\theta]_p = t_2$*, and the claim follows by applying the inductive hypothesis to the rewrite sequence from* t_2 *to* s*.*

$\Leftarrow$ *We want to prove that, given any ground term* t*, if* $t \to^!_{\Delta',B} s'$*, then* $t \to^!_{\Delta,B} s$ *and* $s =_B s'$*. We will prove it by induction on the length of the rewriting sequence in* Δ'*.*
($n = 0$*.) This case is immediate since* $t =_B s'$*.*
($n > 0$*.) Let us decompose the rewriting sequence from* t *to* s' *as follows:* $t \to t_1 \to^! s'$*. On the rewriting sequence from* t_1 *to* s*, we can apply the induction hypothesis, and we now concentrate on the first rewriting step. If* t *rewrites to* t_1 *without using one of the equations* E^u_j*, the same step can be performed in* Δ *and the claim holds. Otherwise, if one of the equations* E^u_j *is used for the last rewriting step, there exists a substitution* θ *such that* $(lhs_u\sigma_j)\theta =_B t|_p$*, and* $t_1 = t[r'_j\theta]_p$*. By* $rhs_u\sigma_j \to_{E_j} r'_j$ *and the stability of rewriting, we have that* $(rhs_u\sigma_j)\theta \to_{E_j} r'_j\theta$*. Therefore,* $t =_B t[lhs_u(\sigma_j\theta)]_p \to_{E^u} t[rhs_u(\sigma_j\theta)]_p = t[(rhs_u\sigma_j)\theta]_p \to_{E_j} t[r'_j\theta]_p = t_1$*, which is a rewrite sequence leading to* t_1 *in* Δ*.*

Before stating and proving Lemma 2.18 let us recall the necessary definition of the antecedent of a position in a term.

Definition 2.17 *Let $R : l \to r$ be a rule in a given rewrite theory and let $t \to_R t'$ be a rewrite step that reduces a redex at position $p \in O_{\Sigma\cup\mathcal{V}}(t)$. According to [142], we say that a position $p' \in O_{\Sigma\cup\mathcal{V}}(t)$ is an* antecedent *of a position $q \in O_{\Sigma\cup\mathcal{V}}(t')$ iff*

(i) q is not comparable with p and $q = p'$, or

(ii) there exists a position o of a variable x in r such that $q = p.o.w$ and $p' = p.u.w$ where u is a position of x in l.

Now we are ready to establish that the rule unfolding transformation followed by the *restoreCompleteness* procedure preserve the semantics of ground normal forms.

Lemma 2.18 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a program, and let $\mathcal{R}' = (\Sigma, \Delta \cup B, R')$ be the program obtained from $\mathcal{R}$ by unfolding a rule $R^u \in R$ and the* restoreCompleteness *procedure. Then, $\mathtt{gnf}(\mathcal{R}) =_{\Delta\cup B} \mathtt{gnf}(\mathcal{R}')$.*

Proof 2.2.5 *Let R^u be a rule of the form $(lhs_u \to rhs_u)$, and let $R_1, \ldots, R_k$ be the set of rules used to unfold rule R^u, each one of the form $(l_i \to r_i)$ for $i = 1, \ldots, k$. Let $f_1, \ldots, f_n$ with $n \leq k$ the set of symbols defined by rules $R_1, \ldots, R_k$. Also let $rhs_u \rightsquigarrow_{\sigma_j, R\cup\Delta, B} r'_j$, $j \in \{1, \ldots, n\}$, be the $R \cup \Delta, B$-narrowing step such that the result of unfolding R^u using R_j is the rule $R^u_j : (lhs_u\sigma_j \to r'_j)$. From the definition and the correctness of narrowing, we recall that:*

(1) $\forall\, j \in \{1, \ldots, n\}\,.\, rhs_u\sigma_j \to^{R_j} r'_j$

(2) $\forall j \in \{1, \ldots, n\}$ there exists position $p_j \in O_\Sigma(rhs_u)$ such that $rhs_u|_{p_j}\sigma_j =_B l_j\sigma_j$

(3) $\forall\, j \in \{1, \ldots, n\}\,.\, r'_j = (rhs_u[r_j]_{p_j})\sigma_j$

$\Rightarrow$ We want to prove that, given any ground term t, if $t \to^!_{\mathcal{R}} s$, then $t \to^!_{\mathcal{R}'} s'$ and $s =_{\Delta\cup B} s'$. We will prove it by induction on the length of the rewrite sequence in $\mathcal{R}$.
($n = 0$.) This case is immediate since $t = s$.
($n > 0$.) Let us decompose the rewrite sequence from t to s as follows: $t \to t_1 \to^! s$. On the rewrite sequence from t_1 to s, we can apply the induction hypothesis, and we now concentrate on the first rewrite step. If t rewrites to t_1 without using rule R^u, the same step can be performed in $\mathcal{R}'$ and the claim holds. Otherwise, we want to describe a procedure to reorder an initial fragment of the rewrite sequence from t to s in such a way it is then trivial to simulate it in $\mathcal{R}'$ and then use the induction hypothesis on the rest of the sequence.

Consider a ground term w and a subsequent application of rules R^u and R_j in $\mathcal{R}$ as follows. If $w|_p =_B lhs_u\theta$, by applying $(\{R^u\}, \Delta \cup B)$, we obtain a $\Delta \cup B$-equivalent term to $w[rhs_u\theta]_p$, which embeds a ground instance of rhs_u. Therefore, this term contains some occurrences of the symbols $f_1, \ldots, f_n$. Then, if we can apply $(\{R_j\}, \Delta\cup B)$ (for some $j \in \{1, \ldots, k\}$) to reduce the redex having

one such symbol as its root, we obtain a $\Delta \cup B$-equivalent term to $w[rhs_u[r_j]_{p_j}\theta]_p$. The key point is to note that this subsequent application of rules R^u and R_j in $\mathcal{R}$ can be simulated in $\mathcal{R}'$ by an application of rule R^u_j. In fact, since the rewrite step using R_j occurs at position $p_j \in O_\Sigma(rhs_u)$, it follows that the left hand side l_j of rule R_j unifies with the subterm $rhs_u|_{p_j}$ by substitution σ_j, which subsumes θ by Proposition 2.14, taking $rhs_u|_{p_j}$ as t_1 and l_j as t_2. Therefore, the narrowing step $rhs_u \leadsto_{\sigma_j,p_j,(R_j\cup\Delta,B)} r'_j$ can be proved in $R, \Delta \cup B$. By the definition of unfolding, the rule $lhs_u\sigma_j \to r'_j$ is one R^u_j belonging to $\mathcal{R}'$. Finally, by applying (R^u_j, Δ) to term w we obtain a $\Delta \cup B$-equivalent term to $w[r'_j\rho]_p = w[((rhs_u[r_j]_{p_j})\sigma_j)\rho]_p = w[rhs_u[r_j]_{p_j}\theta]_p$.

The basic aim of the sequence reordering procedure $reorderSeq$, whose pseudocode is shown in Figure 2.1, is to change the rule application order, thus obtaining an equivalent sequence (in the sense that the same normal form s is reached) where the application of rule R^u is immediately followed by an application of a rule R_j. In the procedure, a rewrite sequence is represented as a list of rewrite steps (R, p) where R is the applied rule and p the position of the reduced redex. Each rewrite step is intended to be followed by a Δ, B normalization. The procedure takes the rewrite sequence starting from the rewrite step using rule R^u as input and returns the reordered rewrite sequence. List s_1 contains the reordered portion of the sequence, which can be easily simulated in $\mathcal{R}'$, while s_2 contains the rest of the sequence (if any). The auxiliary procedure $reorder$ uses two auxiliary lists ns and vs. The former contains the sequence of steps that are moved before (R^u, p), while the latter contains the skipped steps during the reordering that will keep the same position in the final rewrite sequence. The final sequence is made up of the ns list, the consecutive steps (R^u, p), (R_j, p_j), the skipped steps in vs, and the rest of the sequence in ts. There is only one particular case in which the reordering procedure deletes some rewrite steps including the one using rule R^u, which will be discussed later. Let us explain the eight different cases of the ordering procedure in the $reorder$ function.

Case (1) is the easiest one because the applied rule is one R_j, which is used to reduce a redex in $rhs_u\theta$ having one symbol f_i at its root. In this case, the procedure terminates, returning the reordered sequence $ns, (R^u, p), (R_j, p_j), vs, ts$. In case (3), a rule that is different from R^u is used to reduce a redex in the substitution θ. Since the redex belongs to the substitution, this rewrite step is possible before the application of rule R^u at a position q', which is the antecedent of q. Hence, the rewrite step (R, q') is moved at the end of the ns list and the procedure follows with the rest of the sequence. Case (8) is analogous because a rule that is different from R^u is used to reduce a redex that contains the subterm $rhs_u\theta$ in the substitution without erasing it. This rewrite step can also be moved before the application of rule R^u, and, hence, it is put at the end of the ns list. Note that in this case, the antecedent of q is q itself because $q < p$. Case (4) considers a rule that reduces a redex whose root is not in $rhs_u\theta$ nor in a path from p to the term root. This is the case of a skippable rewrite step that is moved at the end of the vs list. Case (6) considers a rewrite step where the reduced

$reorderSeq((R^u, p) : seq) =$ **let** $(s_1, s_2) = reorder((R^u, p), [\,], [\,], seq)$
in $merge(s_1, s_2)$
$reorder((R^u, p), ns, vs, (R, q) : ts) =$
case q of :
$q = p_j \in O_\Sigma(rhs_u)$ and $R = R_j$, **then return** $([ns, (R^u, p), (R_j, p_j)], [vs, ts])$ (2.1)
$q \notin O_\Sigma(rhs_u) > p$ and $R = R^u$, **then let** $(ns_1, ts_1) =$
$reorder((R^u, q), [\,], [\,], ts)$ **in** $reorder((R^u, p), [ns, ns_1], vs, ts_1)$ (2.2)
$q \notin O_\Sigma(rhs_u) > p$, **then** $reorder((R^u, p), [ns, (R, q')], vs, ts)$ (2.3)
$q \not\leq p$ and $q \not> p$, **then** $reorder((R^u, p), ns, [vs, (R, q)], ts)$ (2.4)
$q < p$ and $rhs_u|_\Lambda$ occurs in the lhs of R at non root position
then return $([ns, (R^u, p), (R, q)], [vs, ts])$ (2.5)
$q < p$ and $f_1, \ldots, f_n$ do not appear in the resulting term,
then return $([(R, q)], [vs, ts])$ (2.6)
$q < p$ and $R = R^u$, **then let** $(ns_1, ts_1) = reorder((R^u, p), ns, vs, ts)$
in $reorder((R^u, q), [ns, ns_1], vs, ts_1)$ (2.7)
$q < p$, **then** $reorder((R^u, p), [ns, (R, q)], vs, ts)$ (2.8)
where $q' =$ antecedent of q

Figure 2.1: Rewrite sequence reordering procedure.

redex contains term $rhs_u\theta$ in the substitution but erases it from the term (i.e., the variable that matches the subterm containing $rhs_u\theta$ does not occur in the rhs of the rule). This rule application makes all the rewrite steps stored in ns and the one using R^u useless, so they can be deleted from the sequence and the procedure terminates returning the step (R, q), the skipped steps, and the rest of the sequence. Case (5) considers a rewrite step where the left-hand side of the applied rule R matches the root symbol of rhs_u at some position p. This means that $rhs_u\theta$ is not contained in the matching substitution. We are then in the hypothesis of Lemma 2.16, which states that the two subsequent rewrite steps (R^u, p) and (R, q) can be simulated in $\mathcal{R}'$. Hence, the procedure terminates returning the reordered sequence $ns, (R^u, p), (R, q), vs, ts$. Cases (2) and (7) consider a rewrite step where the same rule R^u is used to reduce a redex that is inside θ or that contains the subterm $rhs_u\theta$, respectively. The basic idea is that when another application of R^u is found, we first terminate the reordering w.r.t. the deeper application of R^u and then we recursively call the reorder function to reorder the sequence w.r.t. the R^u application that is not as deep. In fact, in case (2), we suspend the reordering procedure w.r.t. the considered application of rule R^u, and we recursively call the function to reorder a fragment of the rewrite sequence w.r.t. the deeper R^u application. When the recursive call terminates, we resume the previous call putting the computed list ns_1 at the end of the list ns and following with the computed rest of the sequence ts_1. Case (7) does the reverse, by terminating the current reordering and then recursively calling the function w.r.t. the R^u application that is not as deep.

Termination. *Since we consider programs to be sufficiently complete and the considered rewrite sequence ends with the normal form s, the occurrences of symbols $f_1, \ldots, f_n$ have to be reduced before reaching s by using either a rule R_j as considered in case (2), or a rule that makes them disappear as considered in case (6). In both cases the reorder procedure terminates.*

Correctness. *We want to show that all the rewrite steps contained in list s_1 (which is then merged with the rest of the sequence s_2 in function reorderSeq) can be trivially simulated in $\mathcal{R}'$. List s_1 is the first component of the pair of lists returned by the reorder function. Considering the termination cases, the first component can contain either the step (R, q) (case (6)) where R is different from R^u, or the list $ns, (R^u, p), (R_j, p_j)$ (case (1)), or the list $ns, (R^u, p), (R, q)$ (case (5)). When we apply a rule that is different from R^u it can be trivially simulated in $\mathcal{R}'$ by applying the same rule. Moreover, recall that a subsequent application of rules R^u and R_j can be simulated in $\mathcal{R}'$ by an application of rule R^u_j. Finally, the subsequent steps $(R^u, p), (R, q)$ considered by case (5) can be simulated in $\mathcal{R}'$ by Lemma 2.16. Hence, the correctness holds.*

Reduction of the sequence. *It is easy to see that s_1 is never empty and the rest of the sequence s_2 is strictly shorter that the sequence from t_1 to s. Hence, we can use the inductive hypothesis on s_2.*

$\Leftarrow$ *We want to prove that, given any ground term t, if $t \to^!_{\mathcal{R}'} s'$ then $t \to^!_{\mathcal{R}} s$, and $s =_{\Delta \cup B} s'$. We will prove it by induction on the length of the rewriting sequence*

in $\mathcal{R}'$.
($n = 0$.) This case is immediate since $t = s'$.
($n > 0$.) Let us decompose the rewriting sequence from t to s' as follows: $t \rightarrow t_1 \rightarrow^! s'$. On the rewriting sequence from t_1 to s, we can apply the induction hypothesis, and we now concentrate on the first rewriting step. If t rewrites to t_1 without using one of the rules R^u_j, the same step can be performed in $\mathcal{R}$ and the claim holds. Otherwise, if one of the rules R^u_j is used for the last rewriting step, there exists a substitution θ such that $(lhs_u\sigma_j)\theta =_B t|_p$, and $t_1 = t[r'_j\theta]_p$. By $rhs_u\sigma_j \rightarrow^{R_j} r'_j$ and the stability of rewriting, we have that $(rhs_u\sigma_j)\theta \rightarrow^{R_j} r'_j\theta$. Therefore, $t =_B t[lhs_u(\sigma_j\theta)]_p \rightarrow^R t[rhs_u(\sigma_j\theta)]_p = t_1[(rhs_u\sigma_j)\theta]_p \rightarrow^{R_j} t[r'_j\theta]_p = t_1$, which is a rewriting sequence leading to t_1 in $\mathcal{R}$.

Finally, the main result of the paper immediately follows from the previous Lemma.

Proof 2.2.6 (Proof of Theorem 2.12) *The (Corr.) part of the proof is perfectly equivalent to the ($\Leftarrow$) part of the proof of the Lemma 2.18. For the (Comp.) part, note that since the program is weakly normalizing, if $t \rightarrow^*_{\mathcal{R}} s$, there exists at least a normal form s'' such that $s \rightarrow^*_{\mathcal{R}} s''$, and for Lemma 2.18 $t \rightarrow^*_{\mathcal{R}'} s'$ with $s' =_{\Delta \cup B} s''$.*

Remark. In order to prove Theorem 2.12, stating that the unfolding operation and the *restoreCompleteness* procedure preserve the semantics of ground reducts of the original program, we had to prove that they preserve the semantics of ground normal forms (Lemma 2.18). The reader may think that Theorem 2.12 is just a trivial extension to the semantics of ground reducts, which is mainly based on Lemma 2.18 and that we actually preserve only the semantics of ground normal forms. The fact is that there are cases of rewrite sequences starting from a ground term t where an unfolded symbol is not reduced until a normal form is reached, and since in the transformed program that symbol has been evaluated in advance in the unfolded rules, a rewrite sequence in the transformed program starting from t cannot reach a reduct equivalent to one in the rewrite sequence in the original program until the normal form. However, this is not the general case, as shown in the following example.

Example 2.19 *Let us consider the rules of Example 2.7, and let us recall that the* restoreCompleteness *procedure has extended the set of rules R' with rules* 16. $h(f(w,z),y) \rightarrow p(g_1(w,z),y)$ *and* 17. $k(f(w,z)) \rightarrow 1$. *Consider the following rewrite sequence in $\mathcal{R}$: $h(f(g_2(0),g_1(1,0)),1) \rightarrow_{11} h(g_2(g_1(g_2(0),g_1(1,0))),1) \rightarrow_6 p(g_1(g_2(0),g_1(1,0)),1)$. The same ground reduct can be reached in $\mathcal{R}'$ by a rewrite step using rule* 16.*: $h(f(g_2(0),g_1(1,0)),1) \rightarrow_{16} p(g_1(g_2(0),g_1(1,0)),1)$. Consider also the following rewrite sequence in $\mathcal{R}$: $f(g_2(0),g_1(1,0)) \rightarrow_{11} g_2(g_1(g_2(0),g_1(1,0))) \rightarrow_4 g_1(g_2(0),g_1(1,0))$. The same ground reduct can be reached in the transformed program by a rewrite step using the unfolded rule* 15.*: $f(g_2(0),g_1(1,0)) \rightarrow_{15} g_1(g_2(0),g_1(1,0))$.*

In other words, we do not lose generality by considering rewriting up to normal form in our proof.

2.3 Transforming Rewrite Theories

In this section, we present a fold/unfold-based transformation framework by introducing the transformation rules over rewrite theories and establish the correctness of the transformation system. We divide the transformation process into two steps. At the first step, we disregard the rewrite rules and we only transform the set of equations Δ of the equational theory modulo the set of equational axioms B (which are left unchanged). Then, we consider a new rewrite theory which consists of the transformed equational theory and the original rewrite rules. At the second step, we transform the rules modulo the new equational theory. This two-step process allows one to transform the rewrite rules modulo a fixed, already optimized, equational theory, which cannot change during the transformation of the rewrite rules. This fact results to be particularly helpful in proving the soundness of the whole fold/unfold framework.

A *transformation sequence* of length k for a rewrite theory $(\Sigma, \Delta \cup B, R)$ is a sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1}, \ldots, \mathcal{R}_k)$, $k \geq 0$, where each $\mathcal{R}_j$ is a rewrite theory, such that

- $\mathcal{R}_0 = (\Sigma, E_0, R_0)$, with $E_0 = (\Delta \cup B)$ and $R_0 = R$.
- For each $0 \leq j < i$, $\mathcal{R}_{j+1} = (\Sigma, \Delta_{j+1} \cup B, R_0)$ is derived from $\mathcal{R}_j$ by an application of a transformation rule on the equation set Δ_j.
- For each $i \leq j < k$, $\mathcal{R}_{j+1} = (\Sigma, E_i, R_{j+1})$ is derived from $\mathcal{R}_j$ by an application of a transformation rule on the rule set R_j.

The *transformation rules* are definition introduction, definition elimination, Folding, Unfolding, and abstraction, which are defined as follows.[2]

Definition Introduction. We can obtain program $\mathcal{R}_{k+1}$ by adding to $\mathcal{R}_k$ a set of new equations (*resp.* rules), defining a new symbol f called *eureka*. We consider equations (*resp.* rules) of the form $f(\overline{t_i}) = r_i$ (*resp.* $f(\overline{t_i}) \to r_i$), such that:

(1) f is a function symbol which does not occur in the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_k$ and is declared by $f : s_1 \ldots s_n \to s$ `[Ax]`, where $s_1, \ldots, s_n, s$ are sorts declared in $\mathcal{R}_0$ and `Ax` are equational attributes.

(2) $t_i \in \mathcal{T}_{\mathcal{C}}(\mathcal{V})$, and $Var(\overline{t_i}) = Var(r_i)$, for all i – i.e., the equations/rules are *non-erasing*.

(3) Every defined function symbol occurring in r_i belongs to $\mathcal{R}_0$.

(4) The set of new equations (*resp.* rules) are left linear, sufficient complete and non overlapping. For rules we require also right linearity.

[2] Since the Unfolding operation has been presented and discussed in the previous sections it is not repeated here.

In general, the main idea consists of introducing new auxiliary function symbols which are defined by means of a set of equations/rules whose bodies contain a subset of the functions that appear in the right-hand side of an equation/rule that appears in $\mathcal{R}_0$, whose definition is intended to be improved by subsequent transformation steps. The non overlapping property and the left linearity ensure confluence of eurekas, which is needed to preserve the completeness of the fold operation and will be discussed later. Right linearity on rules is needed to ensure narrowing completeness [122], and left linearity is also needed to preserve the right linearity of rules when doing folding. Consider, for instance, the folding of rule $f(x) \to g(x)$ using the (non left linear) eureka $new(x, x) \to g(x)$, which would produce a new rule $f(x) \to new(x, x)$ which is not right-linear.

Note that, once a transformation is applied to a eureka, the obtained equation/rule is not considered to be a eureka anymore. As we will see later, this is important for the folding operation, since we can only fold non-eureka equations/rules using eureka ones.

The *non-erasing* condition is a standard requirement that avoids the creation of equations/rules with extra-variables when performing folding steps. Consider, for instance, the folding of equation $f(x) = g(x)$ using the (erasing) eureka $new(x, y) = g(x)$, which would produce a new equation $f(x) = new(x, y)$ containing an extra variable in its right-hand side (thus an illegal equation).

Definition Elimination. Let $\mathcal{R}_k$ be the rewrite theory $(\Sigma_k, \Delta_k \cup B_k, R_k)$. We can obtain program $\mathcal{R}_{k+1}$ by deleting from program $\mathcal{R}_k$,

- all equations that define the functions $f_0, \ldots, f_n$, say Δ^f, such that $f_0, \ldots, f_n$ do not occur either in $\mathcal{R}_0$ or in $(\Sigma_k, (\Delta_k \setminus \Delta^f) \cup B_k, R_k)$.
- all rules that define the functions $f_0, \ldots, f_n$, say R^f, such that $f_0, \ldots, f_n$ do not occur either in $\mathcal{R}_0$ or in $(\Sigma_k, \Delta_k \cup B_k, R_k \setminus R^f)$.

Note that the deletion of the equations/rules that define a function f implies that no function calls to f are allowed afterwards. However, subsequent transformation steps (in particular, folding steps) might introduce those deleted functions in the rhs's of the equations/rules, thus producing inconsistencies in the resulting programs. To avoid this, we forbid any folding step after a definition elimination has been performed (this generally boils down to postpone all elimination steps to the end of the transformation sequence).

Folding. Let $F \in \mathcal{R}_k$ be an equation (the "folded equation") of the form $(l = r)$, and let $F' \in \mathcal{R}_j$, $0 \leq j \leq k$, be an equation (the "folding equation") of the form $(l' = r')$, such that $r|_p =_{B_k} r'\sigma$ for some position $p \in O_\Sigma(r)$ and substitution σ. Note that, since we transform the equations of an equational theory, we consider here the congruence relation $=_{B_k}$ modulo the equational axioms B_k (assuming an empty equation set). This is because we cannot consider a congruence modulo an equational theory which is being modified. Moreover, the following conditions must be satisfied:

(1) F is not a eureka.

(2) F' is a eureka.

(3) The substitution σ is sort decreasing, i.e, if $x \in \mathcal{V}_s$, then $x\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_{s'}$ such that $s' \leq s$.

(4) Let $l' = f(\overline{t_n})$ and $r|_p = e$ and let $f(\overline{t_n})$ and e have type s_f and s_e, respectively; then $s_f \leq s_e$.

Then, we can obtain program $\mathcal{R}_{k+1}$ from program $\mathcal{R}_k$ by replacing F with the new equation $(l = r[l'\sigma]_p)$.

Folding can be applied to rules whenever the transformation of the equational theory has been completed. To fold rules we proceed as follows. Let $F \in \mathcal{R}_k$ be a rule (the "folded rule") of the form $(l \to r)$, and let $F' \in \mathcal{R}_j$, $0 \leq j \leq k$, be a rule (the "folding rule") of the form $(l' \to r')$, such that $r|_p =_{E_k} r'\sigma$ for some position $p \in O_\Sigma(r)$ and substitution σ, fulfilling conditions (1) - (4) above. Then, we can obtain program $\mathcal{R}_{k+1}$ from program $\mathcal{R}_k$ by replacing F with the new rule $(l \to r[l'\sigma]_p)$. Note that in this case we use the congruence modulo the equational theory E_k since it does not change any more after this stage.

The need for conditions (1) and (2) is twofold. These conditions forbid self-folding, that is, a folding operation with $F = F'$, thus a rule with the same left and right-hand side cannot be produced, which may introduce infinite loops on derivations and destroy the correctness properties of the transformation system. These conditions also forbid the folding of a eureka, which is meaningless as illustrated in the following example.

Example 2.20 *Consider the following two rules:*

$$\begin{array}{rcll} new & \to & f & \textit{(eureka)} \\ g & \to & f & \textit{(non-eureka)} \end{array}$$

Without conditions (1) and (2), a folding of the eureka rule would be possible, obtaining the new rule $(new \to g)$, which is nothing more than a redefinition of the symbol new. Since transformation rules aim to optimize the original program with the support of eurekas, a folding over a eureka is meaningless or even dangerous.

Finally, conditions (3) and (4) ensure the sort compatibility of both the applied substitutions and the term that is inserted into the folded equation/rule right-hand side.

When presenting the definition introduction operation, we said that eurekas have to be confluent in order to ensure the completeness of the fold operation. We now discuss this point by means of an example.

Example 2.21 *Consider the following rewrite theory*

$\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, *where* $\Sigma_{\mathcal{R}}$ *is the signature containing all the symbols of* R *and*

	R :			R' :		
1.	$f(a,b)$	$\rightarrow$	$g(a,b)$			
2.				$\mathbf{f(a,b)}$	$\rightarrow$	$\mathbf{g(m(a),b)}$
3.	$f(x,y)$	$\rightarrow$	$g(x,y)$	$f(x,y)$	$\rightarrow$	$g(x,y)$
4.	$m(a)$	$\rightarrow$	a	$m(a)$	$\rightarrow$	a
5.	$m(a)$	$\rightarrow$	b	$m(a)$	$\rightarrow$	b
6.	$m(b)$	$\rightarrow$	a	$m(b)$	$\rightarrow$	a
7.	$g(a,x)$	$\rightarrow$	a	$g(a,x)$	$\rightarrow$	a
8.	$g(b,x)$	$\rightarrow$	b	$g(b,x)$	$\rightarrow$	b

We get program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ *from* $\mathcal{R}$ *by applying a fold step to rule* 1 *using the eureka rule* 4. *It is easy to see that in* $\mathcal{R}'$ *we can reduce term* $f(a,b)$ *to the normal forms* a *or* b, *while in* $\mathcal{R}$ *we can reach only the normal form* a. *The point is that in* $\mathcal{R}$, *term* $f(a,b)$ *can reduce only to* $g(a,b)$ *while the fold operation introduces the possibility of rewriting to* $g(b,b)$ *cause the eureka defining* m *is not confluent. This leads to a new solution* b, *thus missing the completeness.*

Abstraction. The set of rules presented so far constitutes the core of our transformation system; however let us mention another useful rule, called *abstraction*, which can be simulated in our settings by applying appropriate definition introduction and folding steps. This rule is usually required to implement tupling, and it consists of replacing, by a new function, multiple occurrences of the same expression e in the right-hand side of an equation/rule. For instance, consider the following equation

$$double\_sum(x,y) = sum(sum(x,y), sum(x,y))$$

where $e = sum(x,y)$. The equation can be transformed into the following pair of equations

$$double\_sum(x,y) = ds\_aux(sum(x,y))$$
$$ds\_aux(z) = sum(z,z)$$

These equations are generated from the original one by a definition introduction of the eureka ds_aux and then by folding the original equation by means of the newly generated eureka.

Note that the abstraction rule applies on equations or rules which are not right-linear, since the same expression e occurs more than once in their rhs. Since we ask for rules to be right-linear for the completeness of the narrowing relation, we may think to use the abstraction rule to preprocess rewrite rules in order to try to make them right-linear.

2.3.1 Correctness of the transformation system

Theorem 2.22 states the main theoretical result for the transformation system based on the elementary rules introduced so far: definition introduction, definition elimination, unfolding, folding, and abstraction. The result is strong correctness of a transformation sequence, i.e., the semantics of the ground reducts `gred(_)` is preserved modulo the equational theory as stated by Theorem 2.22.

Theorem 2.22 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k > 0$, be a transformation sequence. Then, $\mathtt{gnf}(E_0) =_B \mathtt{gnf}(E_k)$, and for all $t \in \mathcal{T}_{\Sigma_0}$, if $(t, s) \in \mathtt{gred}(\mathcal{R}_0)$ then there exist s_1, s_2 such that $(t, s_1) \in \mathtt{gred}(\mathcal{R}_k)$, $(s, s_2) \in \mathtt{gred}(\mathcal{R}_0)$ and $s_1 =_{E_0} s_2$. Viceversa, for all $t \in \mathcal{T}_{\Sigma_0}$, if $(t, s) \in \mathtt{gred}(\mathcal{R}_k)$ then there exist s_1, s_2 such that $(t, s_1) \in \mathtt{gred}(\mathcal{R}_0)$, $(s, s_2) \in \mathtt{gred}(\mathcal{R}_k)$ and $s_1 =_{E_0} s_2$.*

The following example demonstrates that the theorem above cannot be lifted to the non-ground semantics of reducts.

Example 2.23 *Consider the theory $\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$ where*

	R:			R':		
1.	$f(0)$	$\to$	0	$f(0)$	$\to$	0
2.	$f(s(x))$	$\to$	$s(x)$	$f(s(x))$	$\to$	$s(x)$
3.	$g(x)$	$\to$	$f(x)$			
4.				$\mathbf{g(0)}$	$\to$	$\mathbf{0}$
5.				$\mathbf{g(s(x))}$	$\to$	$\mathbf{s(x)}$

We get the rewrite theory $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from $\mathcal{R}$ by applying an unfolding step over rule 3 in R, through the following narrowing steps: (i) $g(x) \leadsto_{x/0} 0$, and (ii) $g(x) \leadsto_{x/s(x')} s(x')$. Then, consider the non-ground term $g(x)$. In $\mathcal{R}$, we have a (one step) derivation from term $g(x)$ to the normal form $f(x)$, whereas in $\mathcal{R}'$ there is no derivation starting from term $g(x)$. So, the reduct $f(x)$ is not preserved by the transformation.
The same example also shows that not even a more restricted non-ground semantics, such as the non-ground normal form semantics, is preserved. Nevertheless, in the reachability context of rewrite theories where confluence or termination are not required, this semantics is neither reasonable nor useful.

In order to prove Theorem 2.22 we need the following auxiliary propositions and lemmas. Let us introduce the key ideas for the proof. We first show that the transformation rules presented above preserve the required properties of a rewrite theory. Then we introduce the notion of *virtual transformation* sequence. The main idea behind this notion is to consider that the last program in a transformation sequence can always be obtained (in an ordered way) by anticipating all the definition introduction steps at the beginning of the sequence and by delaying all the definition elimination steps at the end of the same sequence. Note that is always possible since no folding is allowed after a definition elimination. Thus, we assume that no transformation step changes the signature of the program, i.e., the same set of (new and old) function symbols is fixed throughout. Lemmas 2.26 and 2.27 prove that equational and rule folding preserve the canonical forms of the equational theory and the ground normal forms of the rewrite theory, respectively. Then, Lemma 2.29 combines the results obtained for the folding and unfolding operations and proves the statement of Theorem 2.22 for one transformation step using folding or unfolding. Theorem 2.30 prove the correctness of a virtual transformation sequence and the correctness of the entire transformation sequence follows strait forwardly.

Proposition 2.24 *Let $\mathcal{R}$ be a rewrite theory, and let $\mathcal{R}'$ be the rewrite theory that is obtained from $\mathcal{R}$ by means of the application of one transformation rule selected from introduction, elimination, fold or unfold. Then $\mathcal{R}'$ satisfies all the required restrictions.*

Proof 2.3.1 *It is easy to verify that all the conditions enforced over rewrite theories are preserved by the transformation rules. Since the set of equational axioms B is not changed by transformation rules, condition over B are always preserved. The conditions over coherence and consistence are assured by the preprocessing of the rewrite theories and the constraints over the sets of defined symbols as explained in Section 2.4. Right linearity of equations and rules is preserved since the eurekas have to be linear and the fold operation inserts in the right hand-side a linear instance of the eureka left hand-side. The sort decreasing property is also preserved by fold (by conditions (3) and (4) over the fold operation) and unfold (by the narrowing correctness). The sufficient completeness is preserved by unfold thanks to narrowing completeness.*

Definition 2.25 *Given a transformation sequence of the form $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, we define a* virtual *transformation sequence $(\mathcal{R}'_0, \ldots, \mathcal{R}'_n)$ as a transformation sequence satisfying the following:*

(1) $\mathcal{R}'_0 = \mathcal{R}_0 \cup \mathcal{R}_{new}$, where R_{new} contains all the eureka equations and rules introduced in $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$.

(2) The sequence $(\mathcal{R}'_0, \ldots, \mathcal{R}'_n)$ is constructed by applying only the rules: unfolding, folding, and abstraction[3], in the same order as in the original transformation sequence.

(3) If some definition has been eliminated in $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, then by simply eliminating the same definitions in $\mathcal{R}'_n$ we obtain exactly $\mathcal{R}_k$.

The following lemma proves that equational folding preserves the canonical forms.

Lemma 2.26 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory, and let $\mathcal{R}' = (\Sigma, \Delta' \cup B, R)$ be the rewrite theory obtained from $\mathcal{R}$ by folding an equation $E \in \Delta$. Then, $\mathtt{gnf}(\Delta) =_B \mathtt{gnf}(\Delta')$.*

Proof 2.3.2 *Let E be an equation of the form $(l = r)$ while the eureka E_e, used to fold equation E, be of the form $(l_e = r_e)$. From the definition of the fold operation it follows that $r|_p =_B r_e\sigma$ for some position $p \in O_\Sigma(r)$ and the result of the folding operation of E by E_e is the equation E_f: $(l = r[l_e\sigma]_p)$. Finally, $\Delta' = \Delta - \{E\} \cup \{E_f\}$.*

$\Rightarrow$ We want to prove that, given any ground term t, if $t \to^!_{\Delta,B} s$, then $t \to^!_{\Delta',B} s'$, and $s =_B s'$. We will prove it by induction on the length of the rewriting

[3]In practice, only folding and unfolding rules are considered, since abstraction is recast in terms of definition introduction and folding.

sequence in Δ.
($n = 0$.) This case is immediate since $t =_B s$.
($n > 0$.) Let us decompose the rewriting sequence from t to s as follows: $t \rightarrow t_1 \rightarrow^! s$. On the rewriting sequence from t_1 to s, we can apply the induction hypothesis, and we now concentrate on the first rewriting step. If t rewrites to t_1 without using equation E, the same step can be performed in Δ' and the claim holds. Otherwise, if equation E is used for the first step, it means that (i) $t|_{p'} =_B l\sigma_1$, and (ii) $t_1 = t[r\sigma_1]_{p'}$. Then, considering term t and equation E_f, we note that, by (i), it is possible in Δ' a rewriting step from t using E_f, thus obtaining term $t_2 = t[r[l_e\sigma]_p\sigma_1]_{p'}$. Propagating substitution σ_1 we obtain $t_2 = t[r\sigma_1[l_e\sigma\sigma_1]_p]_{p'}$. Since term l_e is embedded in t_2, a rewriting step using E_e is also possible and we obtain $t_3 = t[r\sigma_1[r_e\sigma\sigma_1]_p]_{p'}$. Since $r|_p =_B r_e\sigma$, we have that $t_3 =_B t[r\sigma_1[r|_p\sigma_1]_p]_{p'} = t[r\sigma_1]_{p'} = t_1$, which completes the proof.

$\Leftarrow$ *We want to prove that, given any ground term t, if $t \rightarrow^!_{\Delta',B} s'$, then $t \rightarrow^!_{\Delta,B} s$ and $s =_B s'$. We will prove it by induction on the length of the rewriting sequence in Δ'.*
($n = 0$.) This case is immediate since $t =_B s'$.
($n > 0$.) Let us decompose the rewriting sequence from t to s' as follows: $t \rightarrow t_1 \rightarrow^! s'$. If t rewrites to t_1 without using equation E_f, the same step can be performed in Δ and, by applying the induction hypothesis on the rewriting sequence from t_1 to s', the claim holds. Otherwise, if equation E_f is used for the first step, note that term t_1 will embed an instance of the left-hand side (l_e) of equation E_e. If the following rewrite step from t_1 in Δ' uses the equation E_e, we can show how the considered rewrite steps $t \rightarrow_{E_f} t_1 \rightarrow_{E_e} t_2$ can be simulated in Δ by only one rewrite step from t using equation E. While, if the rewrite step form t_1 does not use equation E_e, since Δ' is confluent modulo B, we can consider a different rewrite sequence from t to s' where E_e is used to rewrite t_1, and then use the induction on the length of this new rewrite sequence.
So, let us consider that E_e is used to rewrite t_1 in t_2 in the rewire sequence to s'. Then, if $t_1 = t[r[l_e\sigma]_p\sigma_1]_{p'}$, t_2 will be the term $t[r[r_e\sigma]_p\sigma_1]_{p'}$. Applying equation E to term t we will obtain term $t_3 = t[r\sigma_1]_{p'}$. Since $r|_p =_B r_e\sigma$ we have that $t_3 =_B t[r[r_e\sigma]_p\sigma_1]_{p'} = t_2$. The application of the induction hypothesis on the rewrite sequence from t_2 to s' completes the proof.

The following Lemma is auxiliary and proves that the rule folding preserves the semantics of ground normal forms.

Lemma 2.27 *$\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory, and let $\mathcal{R}' = (\Sigma, \Delta' \cup B, R)$ be the rewrite theory obtained from $\mathcal{R}$ by folding a rule R. Then, $\texttt{gred}(\mathcal{R}) =_{\Delta \cup B} \texttt{gred}(\mathcal{R}')$.*

Proof 2.3.3 *Let R be a rule of the form $(l \rightarrow r)$ while the eureka R_e, used to fold rule R, be of the form $(l_e \rightarrow r_e)$. From the definition of the fold operation it follows that $r|_p =_{\Delta \cup B} r_e\sigma$ for some position $p \in O_\Sigma(r)$ and the result of the folding operation of R by R_e is the rule R_f: $(l \rightarrow r[l_e\sigma]_p)$. Finally, $\mathcal{R}' = \mathcal{R} - \{R\} \cup \{R_f\}$.*

$\Rightarrow$ *We want to prove that, given any ground term* t*, if* $t \to^!_{\mathcal{R}} s$*, then* $t \to^!_{\mathcal{R}'} s'$ *and* $s =_{\Delta \cup B} s'$*. We will prove it by induction on the length of the rewriting sequence in* $\mathcal{R}$*.*
($n = 0$*.) This case is immediate since* $t =_{\Delta \cup B} s$*.*
($n > 0$*.) Let us decompose the rewriting sequence from* t *to* s *as follows:* $t \to t_1 \to^! s$*. On the rewriting sequence from* t_1 *to* s*,we can apply the induction hypothesis, and we now concentrate on the first rewriting step. If* t *rewrites to* t_1 *without using rule* R*, the same step can be performed in* $\mathcal{R}'$ *and the claim holds. Otherwise, if rule* R *is used for the first step, it means that (i)* $t|_{p'} =_B l\sigma_1$*, and (ii)* $t_1 =_{\Delta \cup B} t[r\sigma_1]_{p'}$*. Then, considering term* t *and rule* R_f*, we note that, by (i), it is possible in* $\mathcal{R}'$ *a rewriting step from* t *using* R_f*, thus obtaining a term* $t_2 =_{\Delta \cup B} t[r[l_e\sigma]_p\sigma_1]_{p'}$*. Propagating substitution* σ_1 *we obtain* $t_2 =_{\Delta \cup B} t[r\sigma_1[l_e\sigma\sigma_1]_p]_{p'}$*. Since term* l_e *is embedded in* t_2*, a rewriting step using* R_e *is also possible and we obtain* $t_3 =_{\Delta \cup B} t[r\sigma_1[r_e\sigma\sigma_1]_p]_{p'}$*. Since* $r|_p =_{\Delta \cup B} r_e\sigma$*, we have that* $t_3 =_{\Delta \cup B} t[r\sigma_1[r|_p\sigma_1]_p]_{p'} = t[r\sigma_1]_{p'} =_{\Delta \cup B} t_1$*, which completes the proof.*

$\Leftarrow$ *We want to prove that, given any ground term* t*, if* $t \to^!_{\mathcal{R}'} s'$*, then* $t \to^!_{\mathcal{R}} s$ *and* $s =_{\Delta \cup B} s'$*. We will prove it by induction on the length of the rewriting sequence in* $\mathcal{R}'$*.*
($n = 0$*.) This case is immediate since* $t =_{\Delta \cup B} s'$*.*
($n > 0$*.) Let us decompose the rewriting sequence form* t *to* s' *as follows:* $t \to t_1 \to^! s'$*. On the rewriting sequence from* t_1 *to* s*, we can apply the induction hypothesis, and we now concentrate on the first rewriting step. If* t *rewrites to* t_1 *without using rule* R_f*, the same step can be performed in* $\mathcal{R}$ *and the claim holds. Otherwise, if rule* R_f *is used for the first step, term* t_1 *will embed a redex* $(l_e\sigma)\rho$*. More formally,* $t_1 = t[r[l_e\sigma]_p\rho]_{p'}$*. Note that for the required properties on the rules defining an eureka,* R_e *is the only rule applicable to reduce the considered redex, and the eureka symbol can not appear anywhere else, so the incompleteness problems we had with the unfolding operation, cannot occur in this case. The proof follows in a quite similar (but simplified) way to the first part of Proof 2.2.5 using a procedure to reorder a fragment of the rewrite sequence from* t_1 *to* s'*. The procedure is shown in Figure 2.2. Let* p^* *be the absolute position of redex* $(l_e\sigma)\rho$ *in* t_1*.*

The reorde2 *procedure is very similar to the* reorder *procedure of Figure 2.1, but with some simplifications. Let us explain the seven different cases of the* reorder2 *function. Case (1) consider the reduction of redex* $(l_e\sigma)\rho$ *by an application of rule* R_e*. In this case, the procedure terminates, returning the reordered sequence* $ns, (R_f, p'), (R_e, p^*), vs, ts$*. In case (3) a rule that is different from* R_f *is used to reduce a redex in the substitution* $\sigma\rho$*. Note that for the restriction imposed on the left-hand side on an eureka defining rule, there cannot be a redex at a position that is deeper than* p^**, rooted with a symbol of* l_e*. Since the redex belongs to the substitution, this rewrite step is possible before the application of rule* R_f *at a position* q'*, which is the antecedent of* q*. Hence, the rewrite step* (R', q') *is moved at the end of the* ns *list and the procedure follows with the rest*

$reorderSeq((R_f, p') : seq) = \textbf{let } (s_1, s_2) = reorder2((R_f, p'), [\,], [\,], seq)$
$\quad \textbf{in } merge(s_1, s_2)$
$reorder2((R_f, p'), ns, vs, (R', q) : ts) =$
$\quad$ case q of :
$\quad q = p^*$, **then return** $([ns, (R_f, p'), (R_e, p^*)], [vs, ts])$ (2.9)
$\quad q \notin O_\Sigma(l_e) > p^*$ and $R' = R_f$, **then let** $(ns_1, ts_1) =$
$\quad\quad reorder2((R_f, q), [\,], [\,], ts)$ **in** $reorder2((R_f, p'), [ns, ns_1], vs, ts_1)$ (2.10)
$\quad q \notin O_\Sigma(l_e) > p^*$, **then** $reorder2((R_f, p'), [ns, (R', q')], vs, ts)$ (2.11)
$\quad q \not\leq p^*$ and $q \not> p^*$, **then** $reorder2((R_f, p'), ns, [vs, (R', q)], ts)$ (2.12)
$\quad q < p^*$ and $(l_e\sigma)\rho$ does not appear in the resulting term,
$\quad\quad$ **then return** $([(R', q)], [vs, ts])$ (2.13)
$\quad q < p^*$ and $R = R_f$, **then let** $(ns_1, ts_1) = reorder2((R_f, p'), ns, vs, ts)$
$\quad\quad$ **in** $reorder2((R_f, q), [ns, ns_1], vs, ts_1)$ (2.14)
$\quad q < p^*$, **then** $reorder2((R_f, p'), [ns, (R', q)], vs, ts)$ (2.15)
$\quad\quad$ **where** $q' =$ antecedent of q

Figure 2.2: Rewrite sequence reordering procedure 2.

of the sequence. Case (7) is analogous because a rule that is different from R_f is used to reduce a redex that contains the redex $(l_e\sigma)\rho$ in the substitution, without erasing it. This rewrite step can also be moved before the application of rule R_f, and, hence, it is put at the end of the ns list. Note that in this case, the antecedent of q is q itself because $q < p^*$. Case (4) considers a rule that reduces a redex whose root is not in $(l_e\sigma)\rho$ nor in a path from p^* to the term root. This is the case of a skippable rewrite step that is moved at the end of the vs list. Case (5) considers a rewrite step where the reduced redex contains term $(l_e\sigma)\rho$ in the substitution but erases it from the term (i.e., the variable that matches the subterm containing $(l_e\sigma)\rho$ does not occur in the rhs of the rule). This rule application makes all the rewrite steps stored in ns, and the one using R_f, useless, so they can be deleted from the sequence, and the procedure terminates, returning the step (R', q), the skipped steps, and the rest of the sequence. Cases (2) and (6) consider a rewrite step where the same rule R_f is used to reduce a redex that is inside $\sigma\rho$ or that contains the subterm $(l_e\sigma)\rho$. The basic idea is that when another application of R_f is found, we first terminate the reordering w.r.t. the deeper application of R_f and then we recursively call the reorder2 function to reorder the sequence w.r.t. the R_f application that is not as deep. In fact, in case (2), we suspend the reordering procedure w.r.t. the considered application of rule R_f, and we recursively call the function to reorder a fragment of the rewrite sequence w.r.t. the deeper R_f application. When the recursive call terminates,

we resume the previous call putting the computed list ns_1 at the end of the list ns and following with the computed rest of the sequence ts_1.Case (6) does the reverse, by terminating the current reordering and then recursively calling the function w.r.t. the R_f application that is not as deep.

Termination. *Since we consider programs to be sufficiently complete and the considered rewrite sequence ends with the normal form s', the redex $(l_e\sigma)\rho$ has to be reduced before reaching s' by using either rule R_e as considered in case (1), or a rule that makes it disappear as considered in case (5). In both cases the $reorder2$ procedure terminates.*

Correctness. *We want to show that all the rewrite steps contained in list s_1 (which is then merged with the rest of the sequence s_2 in function $reorderSeq$) can be trivially simulated in $\mathcal{R}$. List s_1 is the first component of the pair of lists returned by the $reorder2$ function. Considering the termination cases, the first component can contain either the step (R', q) (case (5)) where R' is different from R_f, or the list $ns, (R_f, p'), (R_e, p^*)$ (case (1)). When we apply a rule that is different from R_f it can be trivially simulated in $\mathcal{R}$ by applying the same rule. Moreover, we can show that the subsequent application of rules R_f and R_e can be simulated in $\mathcal{R}$ by an application of rule R. Hence, the correctness holds.*

Reduction of the sequence. *It is easy to see that s_1 is never empty and the rest of the sequence s_2 is strictly shorter that the sequence from t_1 to s'. Hence, we can use the inductive hypothesis on s_2.*

Now we can show how the rewrite steps $t \rightarrow_{R_f \cup \Delta, B} t_1 \rightarrow_{R_e \cup \Delta, B} t_2$ can be simulated in $\mathcal{R}$ by only one rewrite step from t using rule R, and this will conclude the proof.

If $t_1 =_{\Delta \cup B} t[r[l_e\sigma]_p\rho]_{p'}$, and $t_2 =_{\Delta \cup B} t[r[r_e\sigma]_p\rho]_{p'}$, by applying rule R to term t we obtain term $t_3 =_{\Delta \cup B} t[r\rho]_{p'}$. Since $r|_p =_{\Delta \cup B} r_e\sigma$ we have that $t_3 =_{\Delta \cup B} t[r[r_e\sigma]_p\rho]_{p'} =_{\Delta \cup B} t_2$.

Lemma 2.28 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k > 0$, be a virtual transformation sequence. Then, $\mathtt{gnf}(E_0) =_B \mathtt{gnf}(E_k)$, and $\mathtt{gnf}(\mathcal{R}_0) =_{E_0} \mathtt{gnf}(\mathcal{R}_k)$.*

Proof 2.3.4 *The proof follows immediately from Lemma 2.26 and Theorem 2.11, and Lemma 2.27 and Theorem 2.12, since, at each of the first i-th transformation steps ($0 \leq i \leq k$) a fold or unfold operation is performed over the equations, and at each of the following $k - i$ transformation steps, a fold or unfold operation is performed over rules.*

Lemma 2.29 *Let $\mathcal{R}, \mathcal{R}'$ be two rewrite theories such that $\mathcal{R}'$ is obtained from $\mathcal{R}$ by a* fold *or* unfold *operation over a rule R. Then, for all $t \in \mathcal{T}_\Sigma$, if $(t, s) \in \mathtt{gred}(\mathcal{R})$ then there exist s_1, s_2 such that $(t, s_1) \in \mathtt{gred}(\mathcal{R}')$, $(s, s_2) \in \mathtt{gred}(\mathcal{R})$ and $s_1 =_E s_2$. Viceversa, for all $t \in \mathcal{T}_\Sigma$, if $(t, s) \in \mathtt{gred}(\mathcal{R}')$ then there exist s_1, s_2 such that $(t, s_1) \in \mathtt{gred}(\mathcal{R})$, $(s, s_2) \in \mathtt{gred}(\mathcal{R}')$ and $s_1 =_E s_2$.*

Proof 2.3.5 *We will prove it by induction on the length of the rewrite sequence from t to s in $\mathcal{R}$.*

($\Rightarrow$) (n=0) This case is trivial since $(t,t) \in \mathtt{gred}(\mathcal{R})$ and obviously $(t,t) \in \mathtt{gred}(\mathcal{R}')$. ($n > 0$) Let us decompose the rewriting sequence form t to s as follows: $t \rightarrow t_1 \rightarrow^* s$. On the rewriting sequence from t_1 to s, we can apply the induction hypothesis, thus obtaining terms s_1, s_2 such that $(t_1, s_1) \in \mathtt{gred}(\mathcal{R}')$, $(s, s_2) \in \mathtt{gred}(\mathcal{R})$ and $s_1 =_E s_2$. Since $\mathcal{R}$ is weakly normalizing, there exists a term s_3 in normal form such that $(s_2, s_3) \in \mathtt{gred}(\mathcal{R})$. Therefore, we have a rewrite sequence from t to a normal form s_3 (i.e., $t \rightarrow t_1 \rightarrow^* s \rightarrow^* s_2 \rightarrow^* s_3$), and from Lemma 2.27 and Theorem 2.12, there exists term s_4 in normal form such that $(t, s_4) \in \mathtt{gred}(\mathcal{R}')$ and $s_3 =_E s_4$, and the proof is done.

($\Leftarrow$) It is analogous to the previous direction.

Theorem 2.30 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k > 0$ be a virtual transformation sequence. Then, $\mathtt{gnf}(E_0) =_B \mathtt{gnf}(E_k)$, and for all $t \in \mathcal{T}_{\Sigma_0}$, if $(t,s) \in \mathtt{gred}(\mathcal{R}_0)$ then there exist s_1, s_2 such that $(t, s_1) \in \mathtt{gred}(\mathcal{R}_k)$, $(s, s_2) \in \mathtt{gred}(\mathcal{R}_0)$ and $s_1 =_{E_0} s_2$. Viceversa, for all $t \in \mathcal{T}_{\Sigma_0}$, if $(t,s) \in \mathtt{gred}(\mathcal{R}_k)$ then there exist s_1, s_2 such that $(t, s_1) \in \mathtt{gred}(\mathcal{R}_0)$, $(s, s_2) \in \mathtt{gred}(\mathcal{R}_k)$ and $s_1 =_{E_0} s_2$.*

Proof 2.3.6 *In order to prove the property we just need to show that we can extend the property of Lemma 2.29 to three rewrite theories and hence it will hold for a generic $k > 0$.*
Consider the rewrite theories $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2$. From Lemma 2.28 we have that $\mathtt{gnf}(E_0) =_B \mathtt{gnf}(E_1) =_B \mathtt{gnf}(E_2)$, and $\mathtt{gnf}(\mathcal{R}_0) =_{E_0} \mathtt{gnf}(\mathcal{R}_1) =_{E_0} \mathtt{gnf}(\mathcal{R}_2)$. By Lemma 2.29 we have that for all $t \in \mathcal{T}_{\Sigma_0}$, if $(t,s) \in \mathtt{gred}(\mathcal{R}_0)$ then there exist s_1, s_2 such that $(t, s_1) \in \mathtt{gred}(\mathcal{R}_1)$, $(s, s_2) \in \mathtt{gred}(\mathcal{R}_0)$, and $s_1 =_{E_0} s_2$. For the weak normalization there exists a term s_3 in normal form such that $(s_2, s_3) \in \mathtt{gred}(\mathcal{R}_0)$ and then, there also exists term s_4 in normal form such that $(s_1, s_4) \in \mathtt{gred}(\mathcal{R}_1)$ and $s_3 =_{E_0} s_4$. Since we have a derivation form t to normal form s_4 in $\mathcal{R}_1$, there exists s_5 in normal form such that $(t, s_5) \in \mathtt{gred}(\mathcal{R}_2)$ and $s_4 =_{E_0} s_5$. Summing up we have that $(t,s) \in \mathtt{gred}(\mathcal{R}_0)$ and there exists s_5, s_4 such that $(t, s_5) \in \mathtt{gred}(\mathcal{R}_2)$ and $(s, s_4) \in \mathtt{gred}(\mathcal{R}_0)$.

Finally, since each transformation sequence can be transformed into an equivalent *virtual* transformation sequence (following Definition 2.25) which produces the same output program, the proof of Theorem 2.22 comes directly from Theorem 2.30.

2.4 Coherence and Consistence

We propose here a method to guarantee the coherence of $\rightarrow_{\Delta,B}$ with B and the E-consistence of $\rightarrow_{R,B}$ with B when the associativity (A) and commutativity (C) axioms are declared for a defined symbol. The procedure consists of adding some *extension variables* to each equation or rule having in the left-hand side a topmost symbol which is declared with AC, thus obtaining a new set of *generalized* rules.

For instance, suppose we declare the operator $+$ to be associative and commutative. Consider now a rule $r : x + x \rightarrow 0$, and a term $t : a + (a + b)$. Then $\rightarrow_{r,B}$ is

not AC-coherent since there is no matching between term t and the left-hand side of r, whereas the AC-equivalent term $t' : (a + a) + b$ matches the left-hand side of r by means of substitution $\{x/a\}$. In order to make $\rightarrow_{r,B}$ AC-coherent we need to add the *extension variable* y thus producing the following set of rules:

$$x + x \rightarrow 0$$
$$x + x + y \rightarrow 0 + y.$$

Now, given any term t with topmost symbol $+$, t admits a rewriting step with one of these rules iff for each term t' which is AC-equivalent to t, t' admits a rewriting step too.

In Maude, this generalization does not have to be performed explicitly as a transformation of the specification, because it is achieved implicitly in a built-in, automated way.

For what concern the $\rightarrow_{R,B}$ E-consistence with $\rightarrow_{\Delta,B}$, we want to show how this is guaranteed by the disjointness of the sets of defined symbols $\mathcal{D}_1$ and $\mathcal{D}_2$ (see Chapter 1) and the fact that symbols in $\mathcal{D}_1$ can not appear in the lhs of rewrite rules. Consider a rewrite theory and (i) a rewrite step $t_1 \rightarrow_{R,B} t_2$ that applies a rule $R : l \rightarrow r$ and (ii) $t_1 \rightarrow^*_{\Delta,B} t_3$. From (i) it follows that there exist substitution θ and position p such that $t_1|_p =_B l\theta$ and $t_2 =_{\Delta,B} t_1[r\theta]_p$. Since symbols in $\mathcal{D}_1$ can nor appear in l, all possible steps using Δ, B from t_1 cannot modify the structure of l embedded in t_1, which will then appear unmodified in t_3. Therefore, after the Δ, B reduction steps, there would exists substitution $\rho =_{\Delta,B} \theta$ such that $t_3|_p =_B l\rho$. This implies that it is possible a rewrite step $t_3 \rightarrow_{R,B} t_4$ such that $t_4 =_{\Delta \cup B} t_2$.

2.5 Securing Transfer of Code

Among the many different solutions that have been proposed to tackle the problem of secure the transfer of code from a code producer to a code consumer, we adhere to Code Carrying Theory (CCT) [157, 158] —a program synthesis framework stemming from a pioneering work of Manna and Waldinger [115] in which a theorem proving approach is taken to synthesize correct code from theorems and proofs induced by user specifications. As opposed to more traditional certification approaches such as Proof–Carrying Code [131], where both the code and the certificate are transmitted from the code producer to the code consumer, in CCT only a certificate is transmitted in the form of a theory (a set of axioms and theorems) together with a set of proofs of the theorems; no code needs to be explicitly transmitted. The code consumer would admit new axioms from the code producer only if the associated proofs actually do prove the theorems. If this checking succeeds, then the code consumer can apply a code extractor to the set of function-defining axioms to obtain the executable code. The form of the function-defining axioms is such that it is easy to extract executable code from them. By using the proposed system of Fold/Unfold transformations, which can be applied to a wide class of programs automatically, our CCT methodology greatly reduces the burden on the code producer.

The key idea behind our CCT methodology is as follows. Assuming the code consumer provides the requirements in the form of a rewrite theory, the code producer can (semi-) automatically obtain an efficient implementation of the specified functions by applying a sequence of transformation rules. Moreover, having proved the correctness of the transformation system, the code producer can transmit as the required certificate just a compact representation of the sequence of transformation rules to the consumer so he does not need to manually construct any other correctness proof. By applying the transformation rules to the initial requirements, the code consumer can inexpensively obtain the executable code that can be eventually compiled to a different target language if needed.

In [157, 158], an implementation of CCT has been presented using ATHENA [16, 17], which is a tool that provides a language for both ordinary computation and for logical deduction. To the best of our knowledge, no other implementation of CCT has been proposed in the related literature. The system presented in [158] requires *manually* defining a set of axioms, which are the basis for providing an efficient implementation of the specified functions; then, a proof of the correctness conditions required by the consumer is *manually* constructed using the previously defined axioms.

In this section, we explain how the transformation system presented so far can be employed to implement our CCT approach. The CCT methodology consists of several steps, which are illustrated in Figure 2.5, and summarized below.

(1) **Defining Requirements** (Code Consumer). The code consumer provides the requirements to the code producer in the form of a rewrite theory, specifying the functions of interest with a naïve, non-optimized, even redundant piece of code. The rewrite theory can be written in Maude [59], a high-level specification language that implements rewriting logic [117].

(2) **Defining New Functions** (Code Producer). The code producer has to generate an efficient implementation of the specified functions and a proof that such an implementation satisfies the required specifications. To this aim, the code producer uses the fold/unfold-based transformation system presented in Section 2.3 to (semi-)automatically obtain an efficient implementation of the specified functions. Moreover, some specific strategies such as composition and tupling can be easily automated (see [11] for more details). Subsequently, rather than sending the efficient functions as actual code to the consumer, the producer will send only a certificate consisting of a compact representation of the transformation rule sequence employed to derive the program. The strong correctness of the transformation system ensures that the obtained program is correct w.r.t. the initial consumer specifications, so the code producer does not need to provide extra proofs.

(3) **Code Extraction** (Code Consumer). Assuming the transformation infrastructure is publicly available, once the certificate is received, the code consumer can apply the transformation sequence, described in the certificate, to the requirements, and the final program can be obtained without the need of other auxiliary software for the code extraction.

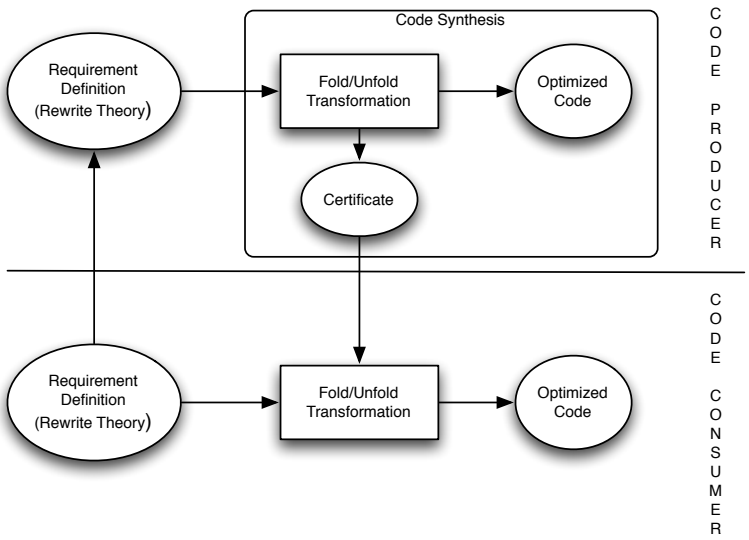


Figure 2.3: Code Carrying Theory Architecture Diagram

Definition 2.32 formalizes the notion of *certificate* for a transformation sequence. In order to build a certificate, we need a way to describe a transformation rule, which is achieved by a transformation rule description.

Definition 2.31 *We associate a* transformation rule description *with each transformation rule, as follows:*

- *Definition Introduction Description:*

  `Intro(Operator Declaration, Equation Set)`
  `Intro(Operator Declaration, Rule Set)`

- *Elimination Description:*

  `Elim(List of function symbols)`

- *Unfolding Description:*

  `Unfold(Unfolded equation id, Unfold position)`
  `Unfold(Unfolded rule id, Unfold position)`

- *Folding Description:*

  `Fold(Folding equation id, Folded equation id, Fold position)`
  `Fold(Folding rule id, Folded rule id, Fold position)`

Note that rules and equations are referenced by an identification label which can be systematically generated and assigned to each rule/equation. We assume that the

identification label for equations (resp. rewrite rules) is of the form `En` (resp. `Rn`), where `n` is a progressive number. More specifically, when a transformation rule is applied to a given rewrite theory and a new equation (resp. rule) is produced, a fresh identification label `En` (resp. `Rn`), is created and associated with the corresponding rule/equation.

It is also worth noting that rule/equation descriptions can precisely identify terms to be folded/unfolded by using the standard notation for term positions.

Definition 2.32 (Certificate) *Let* $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k > 0$, *be a transformation sequence. The* certificate *associated with the transformation sequence* $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$ *is the ordered list of transformation rule descriptions* $(d_1, \ldots, d_k)$ *associated with the transformation rules* $r_1, \ldots, r_k$ *s.t.* $\forall i \in \{1, \ldots, k\}$, r_i *is the transformation rule applied to* $\mathcal{R}_{i-1}$ *to obtain* $\mathcal{R}_i$.

Let us show some selected examples to illustrate this.

Example 2.33 *Let us now consider a simple specification of the Fibonacci function which uses the usual Peano notation to represent natural numbers. The specification is modeled by means of the following naïve equational theory.*

```
     op fib :  Nat -> Nat .
(E1) eq fib(0) = S(0) .
(E2) eq fib(S(0)) = S(0) .
(E3) eq fib(S(S(n))) = fib(S(n)) + fib(n) .
```

Due to the highly recursive nature of this definition of `fib`*, the evaluation of an expression like* `fib(S^50(0))` *will compute many calls to the same instances of the function again and again, and it will expand the original term into a whole binary tree of additions before collapsing it to a number. The exponential number of repeated function calls makes the evaluation of* `fib` *with the above rule very inefficient. Let us transform the previous Fibonacci definition into a more efficient one by using the tupling strategy.*
(1) First, we introduce the following eureka which makes use of the `Pair` *data structure*

```
     sort Pair .
     op ⟨_,_⟩ :  Nat Nat -> Pair .
     op aux :  Nat -> Pair .
(E4) eq aux(n) = ⟨fib(S(n)),fib(n)⟩ .
```

(2) We now unfold the redex `fib(S(n))` *of equation* `(E4)`.

```
(E5) eq aux(0) = ⟨S(0),fib(0)⟩ .
(E6) eq aux(S(n)) = ⟨fib(S(n)) + fib(n),fib(S(n))⟩ .
```

We unfold once again equation `(E5)` *in order to remove the call to* `fib`.

```
(E7) eq aux(0) =⟨S(0),S(0)⟩ .
```

(3) Then, abstraction is applied to equations (E3) *and* (E6) *by means of two new eurekas*

```
     op aux2 :  Pair -> Nat .
     op aux3 :  Pair -> Pair .
(E8) eq aux2(⟨x,y⟩) = x+y .
(E9) eq aux3(⟨x,y⟩) = ⟨x+y,x⟩ .
```

The second step for the abstraction is the folding of equations (E3) *and* (E6) *by means of eurekas* (E8) *and* (E9) *respectively.*

```
(E10) eq fib(S(S(n))) = aux2(⟨fib(S(n)),fib(n)⟩) .
(E11) eq aux(S(n)) = aux3(⟨fib(S(n)),fib(n)⟩) .
```

(4) Finally, the right-hand sides of both equations are folded using the original definition of function aux

```
(E12) eq fib(S(S(n))) = aux2(aux(n)) .
(E13) eq aux(S(n)) = aux3(aux(n)) .
```

The transformed (linear) definition of the equational theory for fib *is as follows.*

```
(E1)  eq fib(0) = S(0) .
(E2)  eq fib(S(0)) = S(0) .
(E12) eq fib(S(S(n))) = aux2(aux(n)) .
(E7)  eq aux(0) = ⟨S(0),S(0)⟩ .
(E13) eq aux(S(n)) = aux3(aux(n)) .
(E8)  eq aux2(⟨x,y⟩) = x+y .
(E9)  eq aux3(⟨x,y⟩) = ⟨x+y,x⟩ .
```

The resulting certificate C *is as follows.*

$$
\begin{aligned}
\mathtt{C} = (&\mathtt{Intro}((\mathtt{op\ aux} : \mathtt{Nat\ ->\ Pair}.),\\
&(\mathtt{eq\ aux}(\mathtt{n}) = \langle \mathtt{fib}(\mathtt{S}(\mathtt{n})), \mathtt{fib}(\mathtt{n})\rangle.)), \mathtt{Unfold}(\mathtt{E4}, [1]),\\
&\mathtt{Unfold}(\mathtt{E5}, [2]), \mathtt{Intro}((\mathtt{op\ aux2} : \mathtt{Pair\ ->\ Nat}.),\\
&(\mathtt{eq\ aux2}(\langle \mathtt{x}, \mathtt{y}\rangle) = \mathtt{x} + \mathtt{y}.)),\\
&\mathtt{Intro}((\mathtt{op\ aux3} : \mathtt{Pair\ ->\ Pair}.),\\
&(\mathtt{eq\ aux3}(\langle \mathtt{x}, \mathtt{y}\rangle) = \langle \mathtt{x} + \mathtt{y}, \mathtt{x}\rangle)), \mathtt{Fold}(\mathtt{E8}, \mathtt{E3}, \Lambda),\\
&\mathtt{Fold}(\mathtt{E9}, \mathtt{E6}, \Lambda), \mathtt{Fold}(\mathtt{E4}, \mathtt{E10}, [1]), \mathtt{Fold}(\mathtt{E4}, \mathtt{E11}, [1])).
\end{aligned}
$$

Example 2.34 *Suppose the code consumer needs a function for computing the sum of the Fibonacci values of the natural numbers in a list. The type of the list of natural numbers is predefined in Maude. The consumer specification is a rewrite theory which consists of the equational theory of Example 2.33 defining the Fibonacci function along with the following set of rules.*

```
     op sum-list :  NatList -> Nat .
(R1) rl sum-list(nil) ⇒ 0 .
(R2) rl sum-list(x xs) ⇒ fib(x) + sum-list(xs) .
```

The equational theory defining the Fibonacci function can be optimized as shown in Example 2.33. The above rules defining the `sum-list` *function can be transformed in a more efficient tail-recursive structure by using our fold/unfold framework as follows. (1) We first introduce the following definition*

```
     op sum-list-aux :  NatList Nat -> Nat .
(R3) rl sum-list-aux(xs,x) => x + sum-list(xs) .
```

(2) By applying the unfold operation over the eureka `(R3)`*, we obtain the following new rules*

```
(R4) rl sum-list-aux(nil,x) => x .
(R5) rl sum-list-aux(y ys,x) => x+fib(y)+sum-list(ys) .
```

(3) Now, by folding rule `(R2)` *and* `(R5)` *using the eureka* `(R3)`*, we obtain the final tail-recursive program.*

```
(R1) rl sum-list(nil) => 0 .
(R6) rl sum-list(x xs) => sum-list-aux(xs,fib(x)) .
(R4) rl sum-list-aux(nil,x) => x .
(R7) rl sum-list-aux(x xs,y) =>
                    sum-list-aux(xs,(fib(x) + y)) .
```

The certificate `C` *is then as follows.*

$$\begin{aligned}\texttt{C} = (&\texttt{Intro}((\texttt{op sum-list-aux} : \texttt{NatList Nat -> Nat}.),\\ &(\texttt{rl sum-list-aux}(\texttt{xs}, \texttt{x}) \Rightarrow \texttt{x} + \texttt{ sum-list}(\texttt{xs}).)),\\ &\texttt{Unfold}(\texttt{R3}, [\texttt{1.2}]), \texttt{Fold}(\texttt{R3}, \texttt{R5}, \Lambda), \texttt{Fold}(\texttt{R3}, \texttt{R2}, \Lambda)).\end{aligned}$$

By applying now the certificate to the initial specification, the code consumer can efficiently obtain the required efficient implementation.

2.6 Implementation

We implemented the transformation framework presented in Section 2.3 in a prototypical system, which consists of about 500 lines of code, written in Maude. Basically, our system allows us to perform the elementary transformation rules over a given initial program. The prototype, named `Meta-Maudest`, is directly accessible through a Web service available at [31]. Alternatively, a stand-alone application can be freely downloaded from `http://users.dimi.uniud.it/~michele.baggi/cct/`. A snapshot of the online Web service is shown in Figure 2.6. The interface allows one to load and execute some predefined examples or load and transform user-defined programs. The transformation sequence appears adjacent to the loaded program and the result of each transformation rule is shown in the textarea below.

In order to implement the transformation rules, we made use of a useful Maude property called *reflection*. Rewriting logic 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 $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a meta-term $\overline{\mathcal{R}}$, any term t, t' in $\mathcal{R}$ as meta-terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{R}, t)$ as a meta-term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, in such a way that we have the following equivalence: $t \rightarrow t'$ in $\mathcal{R}$ iff $\langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$ in $\mathcal{U}$. Thanks to Maude reflection, our program transformation methodology has been easily implemented by manipulating the meta-term representations of rules and equations. In practice, transformation rules presented in Section 2.3 have been implemented as rewrite rules that work and manipulate the meta-term representation of the rewrite theories we want to transform. On the other hand, by virtue of our reflective design, our rewrite theory for program transformation is also available to the level of the CCT infrastructure, which allows us to reuse it in a clear and principled way.

Since the unfolding operation uses narrowing, we employed the `META-E-NARROWING` module, which is part of the Full Maude distribution [152]. The narrowing implemented in Maude is called narrowing *with simplification* because it combines the narrowing relation with rewriting to normal form (represented by $\rightarrow^!$) . The combined relation $(\rightsquigarrow_{\sigma,R,\Delta \cup B}; \rightarrow^!_{\Delta \cup B})$ is defined as $t \rightsquigarrow_{\sigma,R,\Delta \cup B}; \rightarrow^!_{\Delta \cup B} t''$ iff $t \rightsquigarrow_{\sigma,R,\Delta \cup B} t'$, $t' \rightarrow^*_{\Delta \cup B} t''$, and t'' is a normal form. For further details, please refer to [58].

Figure 2.4: Snapshot of the transformation system interface written in Maude.

3

Access Control Policy Specification

The widespread use of web-based applications provides an easy way to share and exchange data as well as resources over the Internet. In this context, controlling the user's ability to exercise access privileges on distributed information is a crucial issue, which requires adequate security and privacy support. In recent years, there has been considerable attention to distributed access control, which has rapidly led to the development of several domain specific languages for the specification of access control policies in such heterogeneous environments: among those, it is worth mentioning the standard XML frameworks XACML [125] and WS-Policy [161].

Term rewriting has been proved successful in formalizing access control to complex systems. For instance, [34] demonstrates that term rewriting is an adequate formalism to model Access Control Lists as well as Role-based Access Control (RBAC) policies. Moreover, it shows how properties of the rewrite relation can enforce policy correctness properties. Also issues regarding policy composition have been investigated within the term rewriting setting. For example, [39] formalizes a higher-order rewrite theory in which access control policies are combined together by means of higher-order operators; then, modularity properties of the theory are used to derive the correctness of the global policy. An alternative methodology for policy composition is presented in [78]: in this approach, composition is achieved by using rewriting strategies that combine rewrite rules specifying individual policies in a consistent, global policy specification.

In the semantic web, resources are annotated with machine-understandable metadata which can be exploited by intelligent agents to infer semantic information regarding the resources under examination. Therefore, in this context, application's security aspects should depend on the semantic nature of the entities into play (e.g. resources, subjects). In particular, it would be desirable to be able to specify access control requirements about resources and subjects in terms of the rich metadata describing them.

In recent years, some efforts have been made towards the integration of semantic-aware data into access control languages. For instance, [65] presents an extension of XACML supporting semantic metadata modeled as RDF statements. In [89, 70, 101]

security ontologies are employed to allow parties to share a common vocabulary for exchanging security-related information. In particular, [101] describes a decentralized framework which allows one to reuse and combine distinct policy languages by means of semantic web technologies. As opposed to our approach, [101] does not define an access control language for policy specification, rather it supports the integration and management of existing policy languages. PeerTrust [92] provides a very interesting mechanism for gaining access to secure information on the web by using semantic annotations, policies and automated trust negotiation.

Description logic (specifically, $\mathcal{ALQ}$ logic) has been used in [167] to represent and reason about the RBAC model: basically, this approach encodes the RBAC model into a knowledge base that is expressed by means of DL axioms, then DL formulae are checked within the knowledge base to verify policy properties (e.g. separation of duty). Ontologies modeled by means of DL statements have been used in [108] in order to specify and check a very expressive subset of the XACML language. In this approach, (a part of) XACML is first mapped to a suitable description logic, then a DL reasoner is employed for analysis tasks such as policy comparison, verification and querying. DL ontologies have also been used in [154] and [102] to describe policy languages for the specification of access restrictions and obligations.

Both term rewriting and description logic provide a declarative framework in which access control specifications can be defined in a concise and simple manner. Besides, these formalisms are both equipped with efficient computational models. Despite they have been extensively used in access control, there exists no attempt to combining term rewriting and description logic in an integrated framework for access control purposes. In this chapter, we propose a rule-based, domain specific language that is well-suited to manage security of semantic web applications. As a matter of fact, it allows security administrators to tightly couple access control rules with knowledge bases (modeled using Description Logic (DL) [19]) that provide semantic-aware descriptions of subjects and resources.

The operational mechanism of our language is based on a rewriting-like mechanism that integrates DL into term rewriting [20]. Specifically, the standard rewrite relation is equipped with reasoning capabilities which allow us to extract semantic information from (possibly remote) knowledge bases in order to evaluate authorization requests. In this setting, access control policies are modeled as sets of rewrite rules, called *policy rules*, which may contain queries expressed in an appropriate DL language. Hence, evaluating an authorization request —specifying the intention of a subject to gain access to a given resource— boils down to rewriting the initial request by using the policy rules until a decision is reached (e.g. *permit*, *deny*, *notApplicable*).

Since policy composition is an essential aspect of access control in collaborative and distribute environments ([39, 78, 108, 42]), our language is also endowed with policy assembly facilities which allow us to glue together several simpler access control policies into a more complex one. To this respect, our language is expressive enough to model all the XACML[125] composition algorithms as well as other conflict-resolution, closure and delegation criteria.

Finally, it is worth noting that our formal framework is particularly suitable for the analysis of policy's domain properties such as *cardinality constraints* and *separation*

of duty. As our rewriting mechanism combines term rewriting with description logic, the analysis of policy specifications can fruitfully exploit both rewriting techniques and DL reasoning capabilities.

3.1 Policy Specification Language

Let Σ_D be the signature defining all the symbols of the Description Logic language of Chapter 1, that is, DL operators, constants, reasoning service constructs, *etc.* A *policy signature* Σ_P is a signature such that $\Sigma_D \subseteq \Sigma_P$ and Σ_P is equipped with the following sorts: *Subject*, *Action*, *Object*, and *Decision*. A term $t \in \mathcal{T}_{\Sigma_P}$ is *pure* if no DL query appears in t.

Our specification language considers a very general access control model, in which policies authorize or prohibit *subjects* to perform *actions* over *objects*. We formalize subjects, actions and objects as terms of a given term algebra which is built out of a policy signature. More formally, given a policy signature Σ_P, a *subject* (resp. *action*, *object*, *decision*) is any pure term in $\mathcal{T}_{\Sigma_P}$ whose sort is *Subject* (resp. *Action*, *Object*, and *Decision*).

The policy behavior is specified by means of rules which are basically rewrite rules whose right-hand sides may contain DL query templates used to extract information from the knowledge bases of interest. Roughly speaking, a policy rule allows one to define what is permitted and what is forbidden using a rewriting-like formalism. Moreover, policy rules can also encode conflict-resolution as well as rule composition operators which can implicitly enforce a given policy behavior (see Section 3.2).

Definition 3.1 *Let Σ_P be a policy signature. A* policy rule *is a rule of the form $\lambda \rightarrow \gamma[q_1, \ldots, q_n]$ where $\lambda \in \mathcal{T}_{\Sigma_P}(\mathcal{V})$, $\gamma[\,]$ is a context in $\mathcal{T}_{\Sigma_P \cup \{\Box\}}(\mathcal{V})$, and each $q_i \in \mathcal{T}_{\Sigma_D}(\mathcal{V})$, $i = 0, \ldots, n$ is a DL query template such that $Var(\gamma[q_1, \ldots, q_n]) \subseteq Var(\lambda)$.*

Given a policy rule $r \equiv f(t_1, \ldots, t_n) \rightarrow \gamma[q_1, \ldots, q_n]$, f is called *defined* symbol for r. Policies are specified by means of sets of policy rules which act over subjects, actions and objects as formally stated in Definition 3.2.

Definition 3.2 *Let Σ_P be a policy signature. An* access control policy *(or simply* policy*) is a triple $(\Sigma_P, P, auth)$, where (i) P is a set of policy rules; (ii) $auth \in \Sigma_P$ is a defined symbol for some policy rule in P such that $auth :: Subject\ Action\ Object \mapsto Decision$. The symbol $auth$ is called a* policy evaluator.

In general, decisions are modeled by means of the constants *permit* and *deny*, which respectively express accessibility and denial of a given resource. Sometimes, it is also useful to include a constant *notApp* to formalize policies which do not allow to derive an explicit decision (that is, policies which are not applicable). This is particularly convenient when composing policies (see [39, 78]). Moreover, thanks to term representation, we can formulate decisions which convey much more information than a simple authorization or prohibition constant. For instance, the starting time and the duration of a given authorization can be easily encoded into a term, e.g. *permit(Starting-time,Duration)* [78].

$$auth(p(n(X), age(Z)), read, rec(n(Y))) \rightarrow \tag{3.1}$$
$$\quad case(DL(\mathcal{H}, instance(X, patient)) \text{ and } Z > 16 \text{ and } X = Y) \Rightarrow permit$$
$$\quad case(DL(\mathcal{H}, instance(X, (\exists assignedTo.\{Y\} \sqcap guardian) \sqcup physician)) \Rightarrow permit$$
$$\quad case(DL(\mathcal{H}, instance(X, admin)) \Rightarrow deny$$
$$auth(p(n(X), age(Z)), write, rec(n(Y))) \rightarrow \tag{3.2}$$
$$\quad case(DL(\mathcal{K}, instance(X, admin)) \Rightarrow deny$$
$$\quad case(DL(\mathcal{H}, instance(X, physician \sqcap \exists assignedTo.\{Y\}))) \Rightarrow permit$$
$$case(true : Condlist, D : Dlist) \rightarrow \ D \tag{3.3}$$
$$case(false : Condlist, D : Dlist) \rightarrow case(Condlist, Dlist) \ (4)$$

Figure 3.1: An access control policy for medical record protection

The following example, which is inspired by the XACML specification in [125], shows how to model a policy for the protection of medical records.

Example 3.3 *Consider the knowledge base $\mathcal{H}$ of Example 1.2 modeling an healthcare domain, and assume that* and, $=$, $>$ *are built-in, infix boolean operators provided with their usual meanings. Let P_H be the set of policy rules of Figure 3.1. Then, $\mathcal{P}_\mathcal{H} \equiv (\Sigma_H, P_H, auth)$, where Σ_H is a policy signature containing all the symbols occurring in P_H, is an access control policy formalizing the following plain English security constraints:*

- *A person, identified by her name, may read any medical record for which she is the designated patient provided that she is over 16 years of age.*
- *A person may read any medical record for which she is the designated guardian or if she is a physician.*
- *A physician may write any medical record for which she is the designated physician.*
- *An administrator shall not be permitted to read or write medical records.*

For the sake of readability, we added some syntactic sugar to rules (1) and (2) of P_H in Figure 3.1. Specifically, the function call $case(cond_1 : \ldots : cond_n, decision_1 : \ldots decision_n)$ has been expanded as follows: case $cond_1 \Rightarrow decision_1 \ldots$ case $cond_n \Rightarrow decision_n$. Note that policy rules (3) and (4) of P_H defining the case *function enforce a conflict resolution operator which simulates the XACML* first-applicable *criterion; that is, only the first condition which is fulfilled derives a decision. Therefore, in this scenario, a 20 year old administrator who is also a patient would be authorized to read her medical record, although administrators are in general not allowed to read any record.*

3.1.1 Policy Evaluation Mechanism

From Chapter 1, we recall that $eval(DL(\mathcal{K}, r))$ denotes the *evaluation* of the DL query $DL(\mathcal{K}, r)$, that is, the result of the execution of the reasoning service r against the knowledge base $\mathcal{K}$.

Definition 3.4 *Let $(\Sigma_P, P, auth)$ be an access control policy and $t, t' \in \mathcal{T}_{\Sigma_P}$ be two terms. Then, t* d-rewrites *to t' w.r.t. P (in symbols, $t \rightarrow_P t'$) iff there exist a rule $\lambda \rightarrow \gamma[q_1, \ldots, q_n] \in P$, a position $u \in O_{\Sigma_P}(t)$, and a substitution σ such that $t_{|u} \equiv \lambda\sigma$ and $t' \equiv t[\gamma\sigma[eval(q_1\sigma), \ldots, eval(q_n\sigma)]]_u$.*

When P is clear from the context, we simply write $\rightarrow$ instead of $\rightarrow_P$. Transitive ($\rightarrow^+$), and transitive and reflexive ($\rightarrow^*$) closures of relation $\rightarrow$, as well as the notions of termination and confluence of $\rightarrow$ are defined in the usual way.

Definition 3.5 *Let $(\Sigma_P, P, auth)$ be an access control policy. Let s be a subject, a be an action, o be an object, and d be a decision in $\mathcal{T}_{\Sigma_P}$. We say that* $(\Sigma_P, P, auth)$ derives *the decision d w.r.t. (s, a, o) iff there exists a finite d-rewrite sequence $auth(s, a, o) \rightarrow_P^+ d$.*

Example 3.6 *Consider the access control policy $\mathcal{P}_{\mathcal{H}}$ of Example 3.3. Then $\mathcal{P}_{\mathcal{H}}$ derives the decision* permit *w.r.t.*
$(p(n(AliceP.Liddell), age(35)), write, rec(CharlieBrown))$, since
$auth(p(n(AliceP.Liddell), age(35)), write, rec(CharlieBrown)) \rightarrow$
$case(false : true, deny : permit) \rightarrow^+ permit$.

The specification language of Section 3.1 allows one to formalize arbitrary access control policies which may be ambiguous or not completely defined, since the rewrite relation $\rightarrow$ might be non-terminating or non-confluent. To avoid such problems, we assume the access control policies meet the following properties:
Totality. Let $\mathcal{P}$ be a policy. $\mathcal{P}$ is *total* iff $\mathcal{P}$ derives a decision d for any triple (s, a, o), where s is a subject, a is an action, and o is an object (that is, there exists a finite d-rewrite sequence $auth(s, a, o) \rightarrow^+ d$, for any (s, a, o)).
Consistency. Let $\mathcal{P}$ be a policy. $\mathcal{P}$ is *consistent* iff $\mathcal{P}$ derives only one decision d for any triple (s, a, o), where s is a subject, a is an action, and o is an object (that is, if $auth(s, a, o) \rightarrow^+ d_1$ and $auth(s, a, o) \rightarrow^+ d_2$, then $d_1 \equiv d_2$).

It is worth noting that, in rewrite-based access control, it is common practice to require policies to be total and consistent. Typically, such constraints are enforced by imposing termination, confluence and sufficient completeness of the rewrite systems underlying the access control requirements (e.g. [34, 39, 78]).

3.2 Policy operators: Composition, Delegation, and Closure

Policy Composition. In distributed environments (e.g. collaborating organizations, large companies made up of several departments, *etc.*) it is crucial to be able to

$$
\begin{aligned}
po(d : dList) &\rightarrow \text{if } d = permit \text{ then } permit \\
&\qquad \text{else } po\_aux(d, dList) \\
po\_aux(x, [\,]) &\rightarrow x \\
po\_aux(x, permit : xs) &\rightarrow permit \\
po\_aux(x, deny : xs) &\rightarrow po\_aux(deny, xs) \\
po\_aux(x, notApp : xs) &\rightarrow po\_aux(x, xs)
\end{aligned}
$$

Figure 3.2: *Permit-overrides* combinator

combine policies in order to protect resources from unauthorized access. Besides, policy composition makes it possible the reuse of security components, which are known to be well specified, to build more complex (and still safe) policies.

In our framework, policy assembling is achieved through policy rules that compose access control policies via policy *combinators*. Basically, policy combinators collect all the decisions taken by local policies and then yield a global decision following the conflict-resolution criterion they encode.

Definition 3.7 *A policy* combinator *is a triple* $(\Sigma_C, C, comb)$*, where* Σ_C *is a policy signature,* C *is a set of policy rules, and* $comb \in \Sigma_C$ *is a defined symbol for some policy rules in* C *such that* $comb :: [Decision] \mapsto Decision$*. The symbol* $comb$ *is called* combination operator*.*

Roughly speaking, combination operators are applied to lists of decisions which derive from the evaluations of local access control policies. The result of such an application is a single decision corresponding to the evaluation of the composition of the considered policies.

This notion of policy combinator is rather powerful, since it allows security administrators to freely define combination criteria according to their needs. Moreover, it is not difficult to see that policy rules can capture the semantics of all the well known combinators of the action control language XACML[125], namely, *permit-overrides*, *deny-overrides*, *first-applicable*, and *only-one-applicable*. To this respect, Example 3.8 shows how to formalize the *permit-overrides* operator within our setting.

Example 3.8 *The XACML permit-overrides combinator is defined as follows. Let* $d_1, \ldots, d_n$ *be a list of decisions which corresponds to the evaluation of* n *access control policies. If there exists* d_i*, for some* $i = 1, \ldots, n$*, equals to* permit*, then, regardless of the other decisions, the combinator returns* permit*. Let* PO *be the set of policy rules of Figure 3.2, where if cond then* exp_1 *else* exp_2 *is assumed to be a built-in conditional construct, and symbols* $[\,]$*,* $:$ *are the usual list constructors. Let* Σ_{PO} *be a policy signature containing all the symbols occurring in* PO*. Then,* $\mathcal{PO} \equiv (\Sigma_{PO}, PO, po)$ *is a policy combinator with combing operator* po *that models the behavior of the XACML permit-overrides combination criterion.*

Starting from (atomic) access control policies, we can assemble more complex policies by applying several policy combinators in a hierarchical way. In other words, access control policies play the roles of basic building blocks which are glued together by means of policy combinators. Composition of policies is formally defined below.

Definition 3.9 *Let $(\Sigma_C, C, comb)$ be a policy combinator, s be a subject, a be an action, and o be an object. Then a composition of policies for (s,a,o) is a term $comb(t_1,\ldots,t_n)$ where each t_i, $i=1,\ldots,n$, is either $auth(s,a,o)$, where auth is the policy evaluator of an access control policy $(\Sigma_P, P, auth)$, or a composition of policies for (s,a,o).*

Basically, evaluating a composition of policies c for a triple (s,a,o) amounts to executing the access control policies and the policy combinators involved in the composition for (s,a,o) by means of the d-rewriting mechanism; that is, we d-rewrite c until we reach a decision. It is worth noting that we cannot simply d-rewrite c w.r.t. the union of all the policy rules involved in the composition, since termination and confluence of the d-rewrite relation are not modular properties. This specifically implies that the termination (resp. confluence) of local policies and combinators does not guarantee the termination (resp. confluence) of the global composition. For instance, it could happen that policy rules defined in distinct local policies $\mathcal{P}_1$ and $\mathcal{P}_2$ interfere in the evaluation of the global composition producing a non-terminating or non-confluent behavior, even if $\mathcal{P}_1$ and $\mathcal{P}_2$ are total and consistent policies. To solve this problem, we follow a bottom-up approach which restricts the application of policy rules in the following way: (i) any authorization request $auth(s,a,o)$ referring to a local policy $\mathcal{P}$ is evaluated using only the policy rules in $\mathcal{P}$; (ii) a combination of policies $comb(t_1,\ldots,t_m)$ referring to a policy combinator $\mathcal{C}$ is evaluated using the policy rules in $\mathcal{C}$ only after the evaluation of terms $t_1,\ldots,t_m$. Such restricted evaluation is formalized in Definition 3.10 using the following auxiliary functions. Let $\mathcal{P} \equiv (\Sigma_P, P, auth)$ be an access control policy, and $\mathcal{C} \equiv (\Sigma_C, C, comb)$ be a policy combinator, then

$$reduce(auth(s,a,o),P) = d \text{ iff } auth(s,a,o) \rightarrow_P^+ d$$
$$reduce(comb(d_1,\ldots,d_n),C) = d \text{ iff } comb(d_1,\ldots,d_n) \rightarrow_C^+ d.$$

where $d, d_1, \ldots, d_n$ are decisions.

Definition 3.10 *Let $(\Sigma_C, C, comb)$ be a policy combinator, s be a subject, a be an action, and o be an object. Then, a composition of policies $comb(t_1,\ldots,t_n)$ for (s,a,o) derives the decision d by evaluating the following function*

$$compute(comb(t_1,\ldots,t_n),C) = reduce(comb(d_1,\ldots,d_n),C)$$
$$\text{where}$$
$$d_i = \begin{cases} reduce(t_i,P) & \text{if } t_i \equiv auth(s,a,o) \\ & \text{w.r.t. some } (\Sigma_P,P,auth) \\ compute(t_i,C') & \text{if } t_i \equiv comb'(t'_1,\ldots,t'_m) \\ & \text{w.r.t. some } (\Sigma_{C'},C',comb') \end{cases}$$

$(\mathbf{r_1})$ $auth_{D_1}(p(n(X)), read, rec(n(Z))) \rightarrow$
$case(DL(\mathcal{K}_1, instance(X, nurse_{D1}))$ and
$DL(\mathcal{K}_2, instance(Z, patient_{D1}))) \Rightarrow permit$
$case(true) \Rightarrow deny$
$(\mathbf{r_2})$ $auth_{D_2}(p(n(X)), read, rec(n(Z))) \rightarrow$
$case(DL(\mathcal{K}_1, instance(X, nurse_{D2}))$ and
$DL(\mathcal{K}_2, instance(Z, patient_{D2}))) \Rightarrow permit$
$case(true) \Rightarrow deny$
$(\mathbf{r_3})$ $auth_A(p(n(X)), Y, rec(n(Y))) \rightarrow$
$case(DL(\mathcal{K}_3, instance(X, employee_A))$ and $Y = read$ and
$DL(\mathcal{K}_2, instance(Z, patient_{D1} \sqcup patient_{D2}))) \Rightarrow permit$
$case(X = deptChief(dep_A)$ and $Y = write$ and
$DL(\mathcal{K}_2, instance(Z, patient_{D1} \sqcup patient_{D2}))) \Rightarrow permit$
$case(true) \Rightarrow deny$
$(\mathbf{r_4})$ $deptChief(Dep) \rightarrow head(DL(\mathcal{K}_3, \exists isChief.\{Dep\}))$

Figure 3.3: Policy rules of Example 3.11

Example 3.11 *Consider a healthcare domain consisting of an administrative department A, and two surgery departments D_1 and D_2. Each department X is modeled as an individual dep_X. Suppose that nurses working in D_1 (resp. D_2) are only allowed to read medical data of patients in D_1 (resp. D_2). Administrative employees of A are allowed to read medical data of any patient, and the chief of A is allowed to write medical data of any patient. Access control policies for D_1, D_2 and A might be specified by using policy rules of Figure 3.3*[1]*; specifically, they can be formalized by $\mathcal{D}_1 \equiv (\Sigma_{D_1}, \{r_1\}, auth_{D1})$, $\mathcal{D}_2 \equiv (\Sigma_{D_2}, \{r_2\}, auth_{D2})$, and $\mathcal{A} \equiv (\Sigma_A, \{r_3, r_4\}, auth_A)$, respectively.*

Now, suppose that Jane *is a nurse working in D_1 with some administrative duties in A. If* Jane *wants to read some medical data about patient* CharlieBrown *belonging to D_2, policy $\mathcal{A}$ will permit it, while policy $\mathcal{D}_2$ will not. Since* Jane *needs to read such medical data to perform her administrative duties, the permit-override policy combinator can be used to solve the conflict. In particular, the composition of policies $po(auth_A(p(n(Jane)), read, rec(n(CharlieBrown))), auth_{D2}(p(n(Jane)), read,$ $rec(n(CharlieBrown))))$ derives the decision permit.*

Policy Delegation. Authorization systems quite commonly support *permission delegation* (see [112, 73]), that is, an identified subject in the authorization sys-

[1] We assume that $\mathcal{K}_1$, $\mathcal{K}_2$, and $\mathcal{K}_3$ are knowledge bases modeling our distributed healthcare domain.

$$
\begin{aligned}
(\mathbf{r_1^*})\ & auth_{D_1}(p(n(X)), Y, rec(n(Z))) \rightarrow \\
& case(DL(\mathcal{K}_1, instance(X, nurse_{D1})) \text{ and } Y = read \text{ and} \\
& \quad DL(\mathcal{K}_2, instance(Z, patient_{D1})) \Rightarrow permit \\
& case(X = Patty \text{ and } Y = write \text{ and } DL(\mathcal{K}_2, instance(Z, patient_{D1})) \Rightarrow \\
& \quad auth_A(p(n(deptChief(dep_A))), write, rec(n(Z))) \\
& case(true) \Rightarrow deny
\end{aligned}
$$

Figure 3.4: The new authorization policy for D_1 implementing a delegation.

$$
\begin{aligned}
clo(s,a,o) &\rightarrow \text{if } (DL(\mathcal{K}, \exists R.\{s\})) = [\,] \text{ then } NotApp \\
&\qquad \text{else } comb_C(applyPol(triples(a, o, DL(\mathcal{K}, \exists R.\{s\})))) \\
applyPol([\,]) &\rightarrow [\,] \\
applyPol((s,a,o) : ts) &\rightarrow auth_P(s,a,o) : applyPol(ts) \\
triples(a, o, [\,]) &\rightarrow [\,] \\
triples(a, o, x : xs) &\rightarrow (x, a, o) : triples(a, o, xs)
\end{aligned}
$$

Figure 3.5: Policy closure rules of Definition 3.13

tem provided with some permissions can delegate (a subset of) its permissions to another identifiable subject (or group of subjects). In our framework, such feature can be implemented as follows. Suppose that the subject s_1, whose permissions are defined by the access control policy $\mathcal{P}_1 \equiv (\Sigma_1, P_1, auth_1)$, wants to delegate subjects in the set S_d to perform actions in the set A_d over objects in the set O_d. Permission delegation can be formalized by a policy $\mathcal{P}$ containing rules of the form $auth_P(s,a,o) \rightarrow case(\langle delegation\_constraint \rangle) \Rightarrow auth_1(s_1,a,o)$, where *delegation_constraint* is a condition that allows us to check whether s,a, and o belongs to the delegation sets S_d, A_d, and O_d. To avoid interferences between applications of rules of $\mathcal{P}$ and $\mathcal{P}_1$, we assume that every authorization request $auth_1(s,a,o)$ is evaluated using only the policy rules in $\mathcal{P}_1$.

Example 3.12 *Consider the healthcare domain of Example 3.11. Suppose that the chief of department A wants to delegate* Patty*, a nurse working in* $D1$*, to write medical data of patients in* $D1$*. We can formalize such a delegation by replacing rule* r_1 *by the new rule* r_1^* *shown in Figure 3.4. Roughly speaking, rule* r_1^* *extends rule* r_1 *by specifying that* Patty *will inherit the chief authorization whenever she wants to write medical data of patients in department* $D1$*.*

Policy Closure. Policy closure operators allow one to infer decisions for a subject s by analyzing decisions for subjects that are semantically related to s, (e.g. if an

employee can read a document, then her boss will). In our framework, such operators can be naturally encoded by exploiting semantic relations conveyed by DL roles. The basic idea is as follows. Consider a role R connecting two subjects s, s' by means of the role assertion $R(s, s')$, and an access control policy $\mathcal{P}$ modeling the access privileges for s' w.r.t. an action a and an object o. If $\mathcal{P}$ derives the decision d w.r.t. (s', a, o), then we infer the same decision d for the subject s. This inference scheme works fine whenever the role R models a *one-to-one* semantic relations (i.e. an injective function). Indeed, when R specifies a *one-to-many* relation between subjects decision conflicts may arise, since several distinct decisions might be computed for distinct subjects s' which are semantically related to s. In this case, a conflict-resolution criterion is needed to infer a decision for subject s. More formally, policy closures are defined as follows.

Definition 3.13 *Let $\mathcal{K}$ be a knowledge base and R be a role in $\mathcal{K}$, $\mathcal{P} \equiv (\Sigma_P, P, auth_P)$ be a policy and $\mathcal{C} \equiv (\Sigma_C, C, comb_C)$ be a policy combinator. A* closure *of $\mathcal{P}$ w.r.t. R and $\mathcal{C}$, is a policy $\mathcal{PC} \equiv (\Sigma_{PC}, PC, clo)$, where Σ_{PC} is the policy signature, PC is the set of policy rules of Figure 3.5, and $clo \in PC$ is the policy evaluator.*

Basically, evaluating *clo* on a triple (s, a, o) amounts to applying the policy evaluator $auth_P$ on the triples in the set $\{(s_1, a, o), \ldots, (s_n, a, o)\}$, where $R(s, s_i)$ holds in $\mathcal{K}$ for all $i \in \{1, \ldots, n\}$. This operation leads to a set of decisions $\{d_1, \ldots, d_n\}$, which are combined together according to the chosen combination operator $comb_C$. The final result of this process is a single decision corresponding to the evaluation of the closure of the policy $\mathcal{P}$ w.r.t. R and $\mathcal{C}$. To avoid interferences between applications of rules of $\mathcal{P}$ and $\mathcal{C}$, we assume that every authorization request $auth_P(s_i, a, o)$ is evaluated using only the rules in $\mathcal{P}$, while the combination $comb_C(d_1, \ldots, d_n)$ is evaluated using the policy rules in $\mathcal{C}$.

Example 3.14 *Consider the access control policy $\mathcal{P}_\mathcal{H}$ specified in Example 3.3 where only designated physicians can write patient medical records. Assume that the knowledge base $\mathcal{H}$ also contains the role* supervises*, which intuitively specifies that fact that some physician may supervise multiple (junior) physicians. Now, we would like to formalize that physicians supervising at least one junior physician can write patient medical records, even if they are not designated. To this end, it suffices to construct the closure of policy $\mathcal{P}_\mathcal{H}$ w.r.t. the role* supervises *and the policy combinator po specified in Example 3.8.*

3.3 Checking Domain Properties of Access Control Policies

In our framework, Description Logic is employed to model the domains to which a given access control policy is applied: subjects, actions, objects, as well as relations connecting such entities can be specified via DL knowledge bases. Therefore, the structure of the policy domains can be naturally analyzed by means of DL reasoning services. More specifically, the idea is to formalize properties over the domains of

interest by means of policy rules. Then, the d-rewriting mechanism can be applied to verify the specified properties.

Definition 3.15 *Let Σ_S be a policy signature. A* domain property specification *of properties $p_1, \ldots, p_n$ is a triple $(\Sigma_S, S, \{p_1, \ldots, p_n\})$, where S is a set of policy rules, and $p_1, \ldots, p_n$ are terms in $\mathcal{T}_{\Sigma_S}$ such that each p_i is an instance of a lhs of some policy rule in S.*

Example 3.16 below shows that domain property specifications are expressive enough to formulate several well known policy constraints such as separation of duty, cardinality constraints, *etc.*

Example 3.16 *Let $\mathcal{H}$ be the knowledge base of Example 1.2 modeling the policy domain of the policy specified in Example 3.3. The following properties*

- ***sep_of_duty.*** *No guardian can be a physician.*
- ***at_most_4.*** *A physician can be assigned at most to four patients.*

can be specified by the domain property specification
$(\Sigma_{\mathcal{H}}, S_{\mathcal{H}}, \{sep\_of\_duty(physician, guardian), at\_most(4)\})$
such that $S_{\mathcal{H}}$ contains

$$sep\_of\_duty(X, Y) \rightarrow DL(\mathcal{H}, subsumes(\neg X, Y))$$
$$at\_most(X) \rightarrow DL(\mathcal{H}, subsumes(\bot, (\geq_{X+1} assignedTo^{-}) \sqcap\ physician)))$$

and $\Sigma_{\mathcal{H}}$ is a policy signature including all the symbols occurring in $S_{\mathcal{H}}$.

In this context, verifying a domain property p amounts to finding a finite d-rewrite sequence which reduces p to the boolean value *true*.

Theorem 3.17 *Let $\mathcal{S} \equiv (\Sigma_S, S, \{p_1, \ldots, p_n\})$ be a domain property specification of properties $p_1, \ldots, p_n$. Then, p_i holds in S iff $p_i \rightarrow_S^+ true$.*

The domain properties $sep\_of\_duty(physician, guardian)$ and $at\_most(4)$ of Example 3.16 hold in $S_{\mathcal{H}}$, in fact, in the knowledge base $\mathcal{H}$ there is no physician who is also guardian, and no physician is assigned to more than four patients.

3.4 Implementation

The proposed access control language has been implemented in the prototype system PAUL, which is written in the functional language Haskell, and whose source code is freely available at [27]. The d-rewriting evaluation mechanism is built around the Haskell's evaluation engine which is based on lazy, higher order, term rewriting. Basically, we integrated DL reasoning capabilities into such an engine by using the DIG interface [35], which is an XML standard for connecting applications to remote DL

reasoners[2]. The DIG interface is capable of expressing the description logic formalized within the OWL-DL [165] framework (namely, $\mathcal{SHOIQD}_n^-$ logic). Therefore, our system fully exploits both the efficiency of Haskell and the reasoning power of $\mathcal{SHOIQD}_n^-$ logic, providing fast evaluations of authorization requests.

To evaluate expressiveness and efficiency of our language, we tested several access control policy specifications, which are available at PAUL's web site [27]. In particular, the considered specifications make use of all the XACML combining operators along with some more complex closure operator, that allows one to infer a decision d for a given authorization request by analyzing decisions which are semantically related to d (e.g. if an employee can read a document, then her boss will). Such semantic relations are extracted from knowledge bases via DL queries.

The rule-based language we proposed is particularly suitable for managing security in the semantic web, where access control information may be shared across multiple sites and depends on the semantic descriptions of the resources to be protected. In fact, the operational engine underlying the language (i.e. *d-rewriting*) allows us to collect semantic metadata from distributed knowledge bases, and to use such data to take decisions w.r.t. the authorization requests under examination. We have also shown that semantic metadata can be exploited both to infer authorization decisions, and to specify and check properties related to the considered security domain.

[2]In our experiments, we have used Pellet [147], an efficient, open-source DL reasoner for the OWL-DL framework.

II

Analysis and Verification of Distributed and Complex Systems

4

Web Systems Filtering

Given the huge amount of date available on the Web, the problem of finding the right information is not so trivial. Information Retrieval (IR) [116] has a high historical importance that has received an even higher attention after the advent of the Web. Actually, after e-mail, using a search engine is today the second activity of Web users, and several Web users perceive a search engine (typically, Google, Yahoo! or MSN Search) as the main access to the Web.

Since the adoption of XML [162] as a widely accepted standard for data representation and exchange has led to a rapid growth in the amount of XML data available over the internet, we can talk of XML Information Retrieval. Nowadays, large-scale XML repositories are constantly browsed, queried and modified by internet users, who typically retrieve a lot of information which is not always possible to absorb in a pleasant and/or understandable fashion. In order to tame the inherent complexity of such a massive amount of data, a lot of decision-support systems to manage and explore XML repositories have been developed.

Arguably, growing attention has been devoted to query and filtering languages as means to efficiently extract all and only the relevant information from huge data collections. As a matter of fact, information frequently appears obscure or difficult to interpret; moreover, most of the time, just a small percentage of the whole amount of the data received is considered interesting by the user. Therefore, query and filtering systems represent a valid way to obtain those contents which best fit user's needs.

The World Wide Web Consortium has defined XQuery[164] and XPath[163] as standard languages to consult and filter information in XML documents, nonetheless a plethora of alternative and worthwhile proposals have been developed independently, e.g. [45, 119, 64]. Basically, they all work by *exactly* matching a given pattern (or path expression) representing the information to be searched for against an XML document. Hence, recognized pattern instances are delivered to the user.

Some programming languages supporting XML processing have also been developed, such as XCentric [62] which is a logic language, extending Prolog with a richer form of unification and regular types, designed specifically for XML processing in logic programming. XCentric is also employed as part of VeriFLog [61] which is a tool for verification of web sites content and data inference. CDuce and XDuce ([36], [99]) are typed functional programming languages, based on pattern matching, designed to

support XML applications. Such languages can be used to consult and query XML documents but provide basically an exact matching behavior.

Although the languages mentioned above are very advantageous in many applications, they may be of limited use when dealing with data filtering in a pure information retrieval context, since (i) they require the user to be aware of the complete XML document structure, (ii) results that are not fully matched are not delivered, (iii) there is no result ranking. Therefore, in this context, a more flexible matching mechanism which can manage the lack as well as the vagueness of the information is necessary. Such an *approximate* behavior is not typically implemented in the standard query languages, and actually only few works address this issue.

For instance, the PIX[14] system is a phrase matching system tailored to XML for searching a given phrase in an XML document. It implements a rough approximate matching method which basically allows to ignore some tags included in the document, while no deletion and renaming of XML items are permitted.

The Flexible XML Search Language XXL [153] is a language designed to query XML documents using the SQL-style (i.e. "Select .. From .. Where .." syntax). XXL uses regular element path expressions and search conditions over element contents, but its mail characteristic is the $\sim$ operator, which can be used for both element comparison and approximate matching of element names. The evaluation of similarity conditions of XXL queries is based on a hierarchical ontology for element names. The result of an XXL query is a ranked list of XML subgraphs based on similarity.

The AQAX [148] system enables fast and accurate approximate answers to complex XML queries, in order to mitigate the increased cost of query evaluation over large semi-structured data stores. Thus, a user can obtain immediate feedback on the query prior to its execution, or may even choose to work with the approximate result if 100% accuracy is not required. The query server relies on the XClusters framework in order to summarize effectively the XML data and to generate approximate query answers. If the user desires it, the server can also retrieve the true results of the query by forwarding to an XML database. The system supports tree-pattern queries with the child and descendant axes, wildcards, branching path predicates, and value predicates on numerical, string, and textual element content.

A more flexible approach is followed in ApproxXQL[145, 144], which is an approximate query language which provides a more sophisticated approximate matching mechanism. It is based on a cost-based query transformation algorithm which allows one to rename, insert and delete XML items in order to find the best match between a pattern and a given XML document. However, this language is still rather simple and does not offer the full expressive power of modern query languages.

In the next sections we present a novel declarative language for approximate filtering of XML documents, which allows the user to easily select the desired information (*positive filtering*) as well as to remove noisy, spurious data (*negative filtering*) from a given XML document. Our language is easy to use and thus can be employed even by those users who are typically not used to express themselves using formal methodologies, since no special expertise is required. Basically, in our approach, XML documents and filtering queries are encoded as tree-shaped terms of a suitable term algebra, then an approximate tree embedding algorithm is employed to execute

filtering queries on XML documents to recognize the information that the user wants to select or to strike out. Our approach is inspired by ApproXQL and extends it in several ways.

- ApproXQL allows to define only ground patterns, while our language provides pattern variables, which can be used to extract parts of the document on which we can perform further tests.
- We add regular expressions and built-in functions to model conditional filtering rules with the aim of refining the approximate search engine.
- Nested filtering queries are allowed, while ApproXQL manages only flat queries.
- ApproXQL does not support negative filtering, while our language does. This feature allows to introduce the expressive power of negation in the language. As a matter of fact, within our framework, we can easily formulate rules to answer queries of the form: "Which people don't have a homepage?"

Besides, our approach improves a filtering framework presented in [33], which formalizes an exact tree embedding algorithm for filtering XML documents.

4.1 The Filtering Language

The filtering language we describe is a declarative, pattern-based language in which we can specify filtering rules as (possibly conditional) patterns. A filtering rule matches an XML document if the pattern is somehow "embedded" into the XML document and fulfills the desired relationships and conditions. Basically, a filtering rule can be formalized by means of the following syntax:

```
{count} <filterop> <pat> in <XML doc> where <cond> (<mode>)
```

which informally says that

1. a pattern `pat` is searched in a document `XML doc`;
2. only detected instances of `pat` which satisfy the given condition `cond` are either extracted (*positive filtering*) or removed (*negative filtering*) from `XML doc` according to the value of the filtering mode `mode`. A filtering mode is a label belonging to the set {`P`,`N`}. Positive filtering rules are identified by means of the filtering mode `P`, while the negative one are denoted by `N`. Whenever a filtering rule does not specify a filtering mode, it has to be considered a positive filtering rule.
3. `count` is an optional operator which allows to count the number of pattern instances detected in the given XML document.

Moreover, several filtering operators `filterop` have been formulated to support approximate as well as exact matching mechanisms. Finally, note that, when no condition is specified, the `where` part of a filtering rule can be omitted.

In the remainder of this section, we present the syntax of each component of a filtering rule providing a brief explanation of the basic constructs of the language.

Filtering operators. We provide four filtering operators which can model both exact and approximative filtering w.r.t. a universal as well as existential semantics.

- `filterOneBest` is an operator that allows one to search for the best approximate match between a given pattern and an XML document. Approximate matching has to be intended modulo renaming, insertion and deletion of pattern items. More precisely, when no exact match is found, either some tag items of the filtering pattern may be renamed or new elements may be inserted/removed in order to find an approximate match. Any insertion/deletion/renaming operation has a fixed cost. `filterOneBest` returns the match with lower cost.

  Informally speaking, given a pattern, we try to generate a result for every position in the document where there is a tag that matches the pattern root. `filterOneBest` then selects the position referring to the document subpart that better matches the pattern. When there is more than one best approximate match, the operator will only deliver the first one it discovered.

- Since there might be several matches with the same cost deriving from the application of distinct sequences of insertion, deletion and renaming operations, the `filterAllBest` operator returns all the best approximate matches found, i.e. all the matches of minimum cost.

- `filterOneExact` is an operator that exactly matches a specified pattern against an XML document. In case that more than one exact match is detected, this operator will only deliver the first one it discovered.

  "Exactly" means that the labels and the structure of the pattern are preserved and precisely recognized inside the XML document. In other words, no renaming, insertion and deletion of pattern items are allowed to "adapt" the pattern to the given document.

- `filterAllExact` returns all the exact matches found.

Patterns. Patterns of filtering rules are used to describe the information we want to detect inside a given XML document. A pattern is built by composing the following syntactical elements.

- *Variables* (we assume to have a countable infinite set of variables {`X,Y,...`}).

- *Text selectors*, that is strings of plain text surrounded by single quotes (e.g. `'Dear friend'`). Text selectors will be matched against the textual part of the XML document.

- *Tag selectors* represent XML tags and are denoted by strings of characters (e.g. `author`, `book`,...). Tag selectors can be followed by the *occurrence* operator `[i]`, where `i`$\in \mathbb{N} \cup \{$last$\}$. Given a sequence of terms (i.e., XML documents) all rooted by a tag `t`, `t[i]` selects the `i`-th term. The keyword `last` is used to select the last element of the sequence.

  Example 4.1 *Consider the XML document*

  ```
  <books>
    <book>'The Lord of the Rings'</book>
    <book>'The Wizard of Oz'</book>
  </books>
  ```

  Then,

  ```
  books(book[2](X))
  ```

  is a pattern selecting the piece of XML `<books><book>'The Wizard of Oz' </book></books>` *(as a side effect the matching mechanism will bind variable* `X` *to* `'The Wizard of Oz'`*).*

  Moreover, tag selectors can be used together with the *synonymity* operator "`$`", which enables the flexible matching of tag selectors. More precisely, given a tag selector `t`, `$t` allows to match `t` against any synonym of `t` which has been defined by the user. Synonyms of tag `t` can be seen as alternative items w.r.t. `t`, that can be employed in an approximate search.

- The containment operator is represented by brackets "`()`" and it is used in combination with tag selectors and boolean operators to define boolean-connected structured patterns. Given a tag selector `t` and $\texttt{pat}_1, \ldots, \texttt{pat}_n$ patterns, the following syntactical expressions are legal patterns:

  - $\texttt{t}(\texttt{pat}_1, \ldots, \texttt{pat}_n)$. The comma "`,`" separator represents the logical conjunctive operator and allows to build conjunctions of patterns. For instance, the pattern `book(title(X),author(Y))` searches for all the books containing both a title `X` *and* an author `Y`.
  - $\texttt{t}(\texttt{pat}_1 | \ldots | \texttt{pat}_n)$. The "`|`" separator allows to model boolean disjunctions of patterns. For example, `book(title(X) | author(Y))` selects all the `book` instances containing a title `X` *or* an author `Y`.
  - $\texttt{t}(\texttt{pat}_1 ? \ldots ? \texttt{pat}_n)$. The separator "`?`" formalizes the boolean *xor* operator. One may use this operator to obtain the evidence of the existence of exactly one of the patterns in the list.

  Operators "`,`" , "`|`" and "`?`" are called *inner boolean operators*. Note that brackets "`()`" are also used to specify precedence in boolean-connected patterns as shown in the following example:

  ```
  pubs(report(author(X),year('2007'))|article(author(Y),year('2007')))
  ```

- Several patterns can be connected together at the root level by means of the outer boolean operators ("`and`", "`or`", "`xor`"). Given patterns $\texttt{pat}_1, \ldots, \texttt{pat}_n$, the syntactical expression $op(\texttt{pat}_1, \ldots, \texttt{pat}_n)$, where $op \in \{\texttt{and}, \texttt{or}, \texttt{xor}\}$ is still a legal pattern.

  Although, outer and inner boolean operators behave very similarly, there is a subtle difference between them. Roughly speaking, when an inner boolean operator is used, it always refers to a parent node explicitly. For example, in the pattern `h(f(X),g(Y))`, the parent node of operator "," is the tag selector `h`, so the patterns `f(X)` and `g(Y)` are somehow connected to the parent node `h`. When an outer operator is used, the parent node is not specified, and the boolean-connected patterns are executed independently.

XML documents and nested filtering rules. Filtering rules work on XML documents. There are three ways to supply an XML document to a filtering rule:

- directly giving the XML code.
  For instance, `filterOneBest a(X) in <a>b</a>`.

- giving the name of a file containing the XML data. In this case, the keyword `file` must precede the file name to be loaded. For instance,

  ```
  filterOneExact a(X) in file 'test.xml'.
  ```

- The execution of a filtering rule generates an XML document. Thus, the outcome of a filtering rule may be employed to feed another filtering rule. Or, equivalently, the result of an inner rule becomes the source document for an outer rule. Our language supports nested filtering rules with an arbitrary level of nesting. As an example, consider

  ```
  filterOneBest a(X) in
      (filterAllBest b(a(X),c())
           in file 'test.xml' where X match [fg]*)
  ```

Conditions. The condition is an optional part of the filtering rule, which can be employed to further refine the search of a given pattern inside an XML document. Formally, a condition is a sequence $\texttt{c}_1, \texttt{c}_2, \ldots, \texttt{c}_n$, where each $\texttt{c}_i$ can be

- a membership test of the form `X match RegExp`, where `X` is a variable occurring in the pattern and `RegExp` is a regular expression[1]. If the variable X is bound to a complex XML subtree (not just a textual node), we build up a string `s` concatenating the labels of all the textual nodes in the subtree, traversing it from left to right, and we subsequently check whether `s` belongs to the language denoted by the considered regular expression `RegExp`.

[1] Regular languages are represented by means of the usual Unix-like regular expressions syntax [132].

- an equation `s=t`, where `s` and `t` are terms built over a set of primitive operators and the set of variables occurring in the pattern. Note that terms may be non linear, that is, they may contain multiple occurrences of the same variable. Our language supports a number of built-in operators to deal with strings and numbers (arithmetic operators, string concatenation, equality over numbers and strings, *etc.*).

Counting the results. By executing a filtering rule on an XML document `doc`, we generate a new XML document containing one or more instances of a given pattern that are embedded into `doc`. However, we might be interested only in the number of embeddings found (e.g. we want to know the number of books written by an author). To model this feature, our language is equipped with the *counting* operator `count` which takes a filtering rule f and a maximum cost c as arguments. The result of applying the `count` operator to f and c is the number of embeddings found by executing f whose cost does not exceed the value c.

Some examples. The following examples formalize, within our framework, three queries (namely, query **Q3**, query **Q5**, and query **Q17**) of the XMark benchmark set [51], which is typically used to evaluate XML filtering and query languages. More precisely,

Q3: Return the IDs of all open auctions whose current increase is at least twice as high as the initial increase.

```
filterAllExact id(Z) in
 (filterAllBest $site($open_auctions(open_auction(id(Z),
                                     bidder[1](increase(X)),
                                     bidder[last](increase(Y)))))
      in file 'auction.xml' where 2*X <= Y (P))
```

Q5: How many sold items cost more than 40?

```
count 0 (filterAllExact price(X) in
  (filterAllBest site($closed_auctions(closed_auction(price(X))))
      in file 'auction.xml' where X >= 40 (P)))
```

Q17: Which people don't have a homepage?

```
filterAllExact site(people(person(name(X)))) in (
   filterAllBest site(people(person(name(X),homepage(Y))))
   in file 'auction.xml'(N))
```

Note that we introduced some occurrences of the "`$`" operator to explicity allow the flexible matching of some tag selectors.

4.2 Filtering is a Tree Embedding Problem

Filtering can be treated as a matching problem over trees. In fact, filtering rule patterns and XML document can be straightforwardly encoded into tree-shaped terms of a suitable term algebra. Note that XML tag attributes can be encoded into common tagged elements and hence translated in the same way (for further information, see [9]).

On the one hand, a pattern can be interpreted as a tree in the following way:

- each variable and text selector is mapped to a leaf node;
- each tag selector and boolean operator is mapped to an inner node;
- the containment operator is interpreted as the standard tree parent-child relation.

The tree representing the pattern is called *pattern tree*. Figure 4.1(a) illustrates the tree encoding of the pattern `h(f('a',X)|g(Y?m('b')))`.

On the other hand, XML documents are provided with a tree-like structure in which plain text elements are mapped to leaf nodes, while tag elements define the inner structure of the tree. Note that XML tag attributes can be considered as common tagged elements, and hence translated in the same way. Precisely, the following piece of XML <tag att$_1$="val$_1$" ... att$_n$="val$_n$"> ... </tag> can be first translated into <tag><att$_1$>val$_1$</att$_1$>...<att$_n$>val$_n$</att$_n$>...</tag>, and next encoded into a tree as described above. The tree representing the XML document is also called *data tree*.

By interpreting patterns and documents as trees, executing a filtering rule boils down to finding one (or all) the matches (i.e., embeddings) of a given pattern tree into a data tree. The recognized subtrees are then either selected or removed from the data tree according to the filtering mode of the rule. Therefore, our filtering mechanism is inspired by and slightly modifies the *unordered path inclusion problem*[105]. Roughly

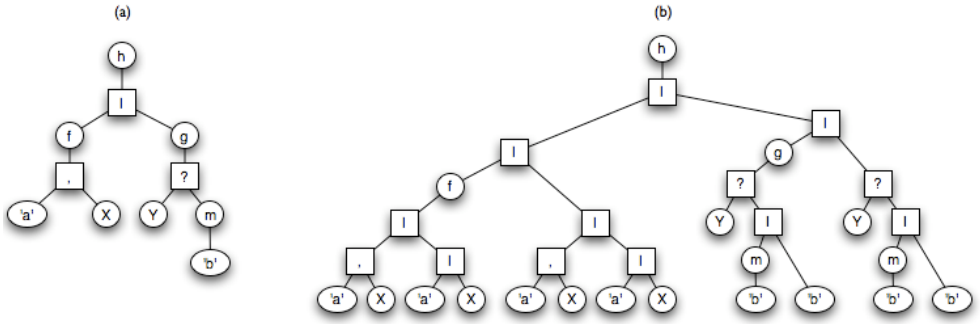


Figure 4.1: Tree encodings of a filtering rule pattern.

speaking, the unordered path inclusion of a tree T_1 in a tree T_2 is defined as an *injective* function from T_1 to T_2 that preserves labels of the nodes and parent-child relationships (i.e. the structure), but not the order of the siblings. Equivalently, it can be considered as a particular instance of the Kruskal's embedding relation[40]. We think that ignoring the order of siblings is favorable or even necessary for filtering XML data, because the ordering of the XML items may not be known to the user.

Following [145], our methodology discards the injectivity property, which is required in the path inclusion problem. Although this can imply a possible loss of precision of the computed results, the efficiency of the matching method is greatly improved, since the computed non-injective embeddings encompasses several injective embeddings at the same time. Moreover, while the unordered path inclusion problem typically searches for *exact* answers (in a sense that the labels and the structure of the pattern tree are precisely embedded in the data tree), our goal is to find a match even when no exact instances of the pattern tree can be recognized inside the data tree.

To find such *approximate* results, we use pattern transformations, which minimally modify the original pattern tree and adapt it to the data tree with the aim of finding the best match which now might be not precise.

A pattern transformation consists of a sequence of basic transformations. Each basic transformation has a cost which is represented by a natural number. The total cost of a sequence of basic transformations is assigned to the matching result of the transformed pattern tree against the data tree and used to rank the result by increasing cost. We consider the following three types of basic pattern transformations.

Renaming. A label l of a pattern tree inner node can be renamed with a new label l' provided that l' is a *synonym* of l. The synonymity relation might be explicitly provided by the user or automatically computed by querying a knowledge base. Renaming is enabled only for tag selectors to which the synonymity operator "`$`" is applied.

Deletion. A pattern tree node (corresponding to either a tag selector or a text selector) can be deleted, whenever it is not the pattern tree root.

Insertion. A new tree node (corresponding to a tag selector) can be inserted into the pattern tree. However, it is not allowed to add a new root or new leaf nodes. In particular, leaf nodes cannot be inserted, because they represent user-dependent data which cannot be automatically inferred by the algorithm.

The costs associated with these basic pattern transformations will be denoted respectively as renaming/deletion/insertion cost. In the next section, we describe a pattern-transformation algorithm which implements the strategy mentioned above.

4.3 An Approximate Tree Matching Algorithm

We start describing a basic procedure for approximate tree matching for ground patterns. Next, we will add all the other components of a filtering rule (variables, condi-

tions,...). The core algorithm is a slightly modified version of the one proposed by Schlieder in [144] which not only finds a single best match, but also allows to find all the best matches of a pattern tree w.r.t. a data tree.

From a theoretical point of view, to evaluate a pattern tree $\mathcal{P}$ against a data tree $\mathcal{D}$, we can follow these steps:

1 Derive from $\mathcal{P}$, every pattern tree $\mathcal{P}'$ which is obtained by applying a sequence of basic transformations to $\mathcal{P}$ (i.e. we compute the *pattern transformation closure* w.r.t. the basic transformations) and compute the corresponding total cost $c_{\mathcal{P}'}$.

2 Find all the exact matches of $\mathcal{P}'$ against $\mathcal{D}$, for each $\mathcal{P}'$.

3 Group the matches found into embedding sets, where an embedding set is a set of matches which refer to the same subtree of the data tree.

4 From each embedding set, choose either one match or all the matches with lower cost $c_{\mathcal{P}'}$ according to the filtering operator applied.

5 Rank the selected matches according to their costs.

Obviously, the brute-force generation of the pattern transformation closure is not feasible, since it would lead to an infinite set of transformed patterns. Nonetheless, in the following, we will show a possible way to solve the problem, which exploits smart representations of data and pattern trees. Basically, all possible node deletions will be encoded in a single pattern tree, while renamings and insertions will be encoded into an index modeling the data tree.

4.3.1 Data Tree Encoding

As shown in Section 4.2, an XML document can be represented by means of a data tree in which leaf nodes represent plain text items and inner nodes represent XML tags. In the following, given a node u of a data tree $\mathcal{D}$ the *position* (or *preorder number*) of u in $\mathcal{D}$ is a natural number $n \geq 1$ assigned to u by a preorder traversal of $\mathcal{D}$. The position of the root node is 1.

In order to construct an approximate match of a pattern tree w.r.t. a data tree, some nodes may need to be inserted into the pattern tree or simply renamed. To avoid the explicit insertions of nodes into a pattern tree, we use a special encoding of the data tree which measures the insertion distance between two nodes in a data tree. Formally, given two nodes u and v of a data tree $\mathcal{D}$ such that u is an ancestor of v, the *insertion distance* between u and v is the sum of the insertion costs of all nodes along the path from u to v (excluding u and v). Moreover, information regarding label renaming of pattern tree nodes is formalized by providing an extensional representation of the synonymity relation (i.e. any node label is decorated with the list of its possible synonyms) along with the associated renaming cost.

The encoding is based on an indexing technique that is inspired by the *partial index* data structure which has been originally introduced in [145] and then successfully employed in the language ApproXQL[144]. The data structure is as follows.

Given a data tree $\mathcal{D}$, for each node u appearing in $\mathcal{D}$, we define a *posting* as a tuple containing the following information:

- pre(u) is the position of u in $\mathcal{D}$;
- dist(u) is the sum of the insertion costs of all ancestors of u in $\mathcal{D}$, which is computed by means of the insertion distance;
- bound(u) is the position of the rightmost leaf of the subtree of $\mathcal{D}$ rooted at u;
- rencost(u) represents the renaming cost of the pattern tree node that matches u,
- embcost(u) stores the cost of embedding a pattern subtree into the subtree of $\mathcal{D}$ rooted at u. The value is zero if u is the match of a pattern tree leaf.
- embtree(u) stores the pattern subtree embedded into the subtree of $\mathcal{D}$ rooted at u whose cost is embcost(u).

The first three fields are computed when building the data tree of the considered XML document and they are not influenced by the execution of the matching algorithm. On the other hand, the last three fields are computed by the matching algorithm and may change during its execution.

We now define a data tree *index*[2] which contains an entry for each label occurring in $\mathcal{D}$. An entry for a label l contains

- a list of all postings referring to nodes in $\mathcal{D}$ with label l. Such a list of postings is sorted by ascending preorder numbers.
- a pair containing a list of all synonyms of label l and the associated renaming cost;

Figure 4.2 illustrates the data tree $\mathcal{D}_{book}$ and the corresponding data tree index for a piece of XML. For the sake of readability, we omitted the postings regarding the leaf nodes of $\mathcal{D}_{book}$. Moreover, we assumed an insertion cost equal to 2 and a renaming cost equal to 6.

Typically, it is preferable to preserving information rather than removing it. Therefore, in order to tune up our filtering system, we considered the following rules of thumb:

- the deletion cost should be two or three times greater than the insertion cost to allow up to two or three insertion operations before preferring to delete a node;
- the renaming cost should be smaller than the deletion cost.

[2]For the sake of efficiency, the implementation indeed uses two indexes to encode the data tree: the former to store the leaf nodes of the data tree (i.e. the plain text elements) and the latter to store the inner nodes (i.e. the XML tags).

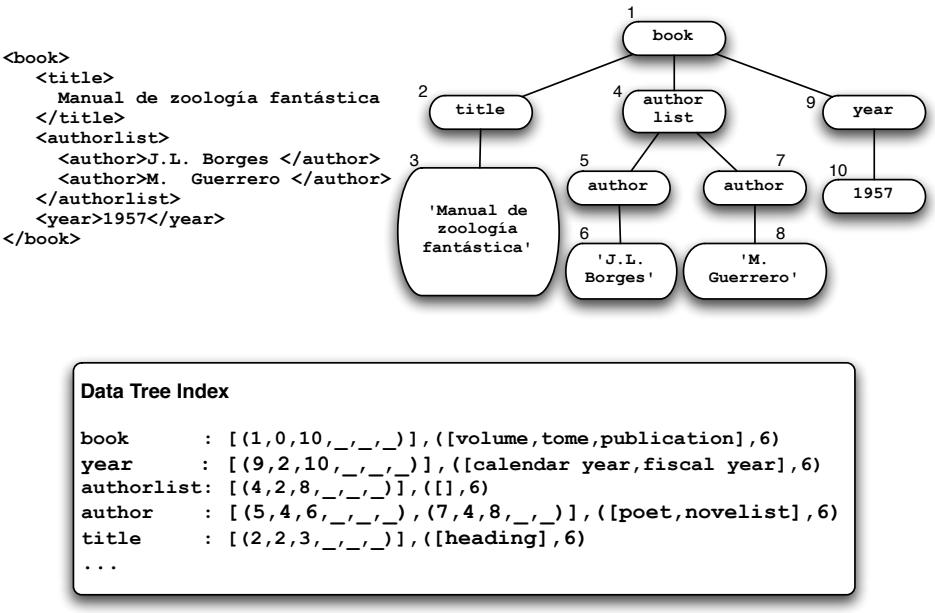


Figure 4.2: Data tree and data tree index for an XML document

4.3.2 Expanded Pattern Tree

The expanded representation of a pattern tree allows one to explicitly encode all possible node deletions of a pattern tree node. More precisely, all permitted deletions of inner pattern tree nodes are represented by transforming the original pattern tree in the following way.

Every inner pattern tree node (except the root) w which represents a tag selector or a "," operator (i.e. boolean conjunction)[3] is replaced by a fresh binary "|"-labelled node w_{or}. The left child of w_{or} refers to w, while the right child represents the fact that the pattern tree node is removed from the pattern tree. Basically, the fresh "|"-labelled (i.e. boolean disjunction) nodes inserted play the role of *choice points* in which the algorithm is called to decide whether to delete a node. Figure 4.1(b) shows the expanded version of the pattern tree depicted in Figure 4.1(a). Observe that we do not have to directly manage leaf deletions, which are hardcoded in the algorithm. Since leaves cannot be renamed, whenever a leaf node n of a pattern tree does not match any leaf node of the data tree, the pattern tree is automatically transformed by deleting n. Besides, the corresponding deletion cost is computed.

Now for every node w of the expanded pattern tree, we define a function delcost(w)

[3]Actually, in our system, we implemented an equivalent, optimized version of the pattern tree expansion which only replaces tag selector nodes, while boolean conjunctive nodes are implicitly managed by the pattern matching mechanism.

which computes the total cost of deleting the node w and all its inner descendants. The returned value is greater than zero only for those right children of "|"-labelled node representing the deletion of an inner node.

In the following, sometimes an expanded pattern tree is simply called pattern tree.

4.3.3 Evaluating an unconditional, positive, ground filtering rule

For the sake of simplicity, we first describe how the methodology works on an unconditional, positive, ground filtering rule, and then we will describe how to implement the other language features. Basically, the problem amounts to finding one or all best approximate matches (i.e. also called embeddings) of a ground pattern against an XML document. We assume to have already generated both the expanded pattern tree $\mathcal{P}$ and the data tree index $\mathcal{D}$ representing the data tree.

The evaluation algorithm is based on the dynamic programming principle. The embedding cost of a subtree of the expanded pattern tree $\mathcal{P}$ rooted at a node w is calculated from

(i) the embedding costs of the subtrees rooted at the children of w;

(ii) the insertion distance between the match of w and the matches of the children of w.

All matches of pattern tree node labels against data tree node labels are stored in posting lists. To find the best approximate embedding of $\mathcal{P}$ in $\mathcal{D}$, the algorithm uses operations on postings. Two types of operations are needed.

1. Given a posting list of potential ancestors and a posting list of potential descendants, the algorithm must find all ancestor-descendant pairs with the smallest embedding cost (*vertical axis*). Such an operation can be performed basically using some information stored in the posting list. Given a posting list S, by S_p, we denote the p-th posting of S. Moreover, $S_{[i,j]}$ represents the sublist of S containing the postings from position i to position j. If U is a posting representing a potential ancestor node u and R is a posting list including n potential descendants of u, recalling that the postings are ordered by preorder numbers, all the n descendants $v_1, \ldots, v_n$ of the node u must reside in the interval $R_{[j,j+n]}$ of the posting list R, since $\text{pre}(u) < \text{pre}(v_i) \wedge \text{bound}(u) \geq \text{pre}(v_i)$. When no deletions are allowed, the smallest embedding cost of u w.r.t. the descendants in R is thus calculated using the following formula

$$\begin{aligned}\text{embcost}(u) = &\min\{\text{embcost}(R_k) + \text{dist}(R_k) \mid j \leq k < j + n\} - \\ &\text{dist}(u) + \text{rencost}(u) + c_{ins}\end{aligned} \tag{4.1}$$

[4]

where c_{ins} is the insertion cost of a node. Whenever we also allow deletions, the smallest embedding cost become the minimum between $\text{delcost}(u)$ and the value computed by the formula (4.1).

[4]Given a data tree node v, which is represented by the posting R_k, $\text{embcost}(R_k)$ (resp., $\text{dist}(R_k)$) stands for $\text{embcost}(v)$ (resp., $\text{dist}(v)$).

2. Given two posting lists that represent the embeddings of two distinct children of a pattern tree node w, the algorithm must find all pairs that belong to the same data node (*horizontal axis*).

   In this case, if w is connected to its children through a boolean conjunction, then the sum of the embedding costs must be calculated, whereas w is connected to its children via a boolean disjunction (i.e. "|" or "?" operator), then we must select the embedding with the cheapest cost.

When evaluating a ground filtering rule, the algorithm visits the pattern tree nodes in depth-first order. During the traversal, it fetches from the data tree index the posting lists belonging to the labels of the visited nodes. Arrived at the leftmost leaf, it joins the posting belonging to this leaf with the posting belonging to the parent node (*vertical axis*) and then proceeds the visit. If a node u has two or more children, then the cheapest combination of matches belonging to u's children is chosen and stored in the posting list generated for u (*horizontal axis*).

The posting lists returned by the algorithm contains the postings representing the transformed pattern that best matches the data tree along with the embedding cost.

4.3.4 Evaluating a generic filtering rule

As shown in Section 4.1, a filtering rule is a quite complex object which may contain non ground patterns and filtering conditions. Moreover, according to the filtering operator and the filtering mode chosen, it can be employed for approximate/exact matching and to implement positive as well as negative filtering. In the following, we briefly discuss how to adapt the algorithm presented in the previous section to cope with such features.

Nonground patterns and filtering conditions. Patterns may contain variables. Therefore, substitutions that bind such variables to subparts of the data tree must be computed during the matching process to find the desired embeddings.

We extend the matching algorithm to deal with variables in the following way. Given a nonground pattern tree $\mathcal{P}$, we consider the pattern $\mathcal{P}'$ which is obtained from $\mathcal{P}$ by removing all the variables in $\mathcal{P}$. For each node labelled with a variable we removed, we record its position into a list. Since $\mathcal{P}'$ is ground, we can apply the previous tree matching algorithm to find the embeddings of $\mathcal{P}'$ into the data tree. Moreover, as we saved the positions of the variables appearing in $\mathcal{P}$, we can compute the embedding substitutions by analyzing the matched subtrees in the data tree and hence selecting those parts which correspond to the variable positions. It can happen that a variable cannot be bound to a subtree, e.g. the algorithm computes an embedding for $\mathcal{P}'$ which requires some node deletions involving an ancestor of a variable node. In this case, such an embedding is simply discarded.

By using this approach, we can thus produce all the possible embedding substitutions for a nonground pattern $\mathcal{P}$ by simply analyzing all the embeddings of $\mathcal{P}'$ in the data tree. We can then apply such substitutions to filtering conditions to generate instantiated conditions and subsequently check their satisfiability.

Approximate and Exact Filtering. The algorithm we described is mainly used for approximate filtering. Nevertheless it can be employed for exact matching. We just have to look for matches with a null embedding cost, since no insertion, deletion, or renaming of nodes is allowed in this case. Therefore, to optimize the search one can think to ignore the basic pattern transformations. Unfortunately, the insertions are implicitly defined in the matching algorithm, and hence we cannot avoid them. However, deletions and renamings can be disabled in the following way: (i) to avoid deletion of nodes, we simply use the original pattern tree instead of using its expanded representation; (ii) to avoid renaming of nodes, the synonymity relation encoded in the data tree index is ignored.

Positive and Negative Filtering. As shown in Section 4.1, our language defines syntax constructs for negative filtering which incorporate the expressive power of negation into our formalism. To enforce such a behaviour, we just execute the negative filtering rule as it was a positive one. We then remove all the embeddings found from the data tree returning the desired filtered outcome. In order to exactly remove all and only the desired information, we allow only to "filter out" exact matches. In other words, any negative filtering is always an exact filtering.

Since negative filtering needs a positive filtering execution along with an entire data tree traversal for removing the embeddings found, it follows that the implementation of negative filtering is less efficient than the implementation of positive filtering (see Section 4.4 for further details).

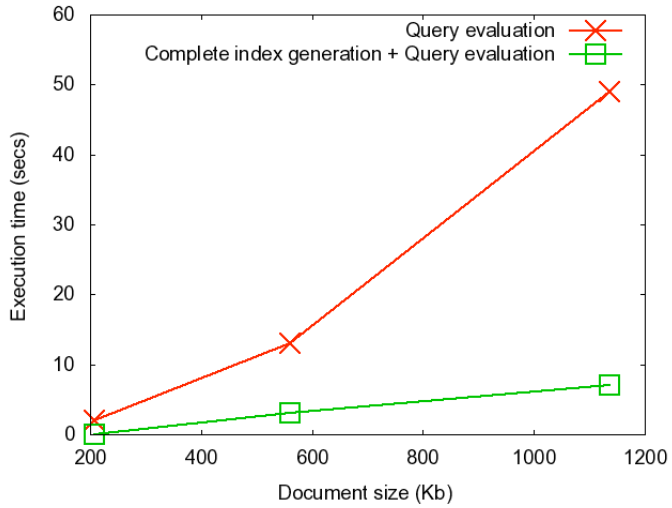
4.4 A Lazy Implementation: an Experimental Evaluation

The proposed filtering language has been implemented in the PHIL system, which is freely available at `http://www.dimi.uniud.it/demis/phil.html`. The implementation has been written in the lazy functional language Haskell with the aim of showing how *laziness* can be particularly fruitful in developing such kind of applications. Lazy functional (and functional logic) languages allow one to somehow "minimize" the amount of information needed to be processed in order to evaluate expressions.

As we have seen in Section 4.3, our methodology employes a quite sophisticated data structure whose whole generation may be very time-expensive. Therefore, exploiting laziness in this context amounts to saying that only the portion of the data tree index which is strictly necessary to evaluate a filtering rule is generated with a consequent gain in the overall system performance. Our claim is supported by the following experiment. We have evaluated a given filtering rule on XML documents of increasing sizes in two distinct ways.

1. We have forced the whole data tree index generation before executing the rule.

2. We have just "lazily" executed the given filtering rule letting Haskell to produce the (portion of) the data structure which is needed to process the rule.



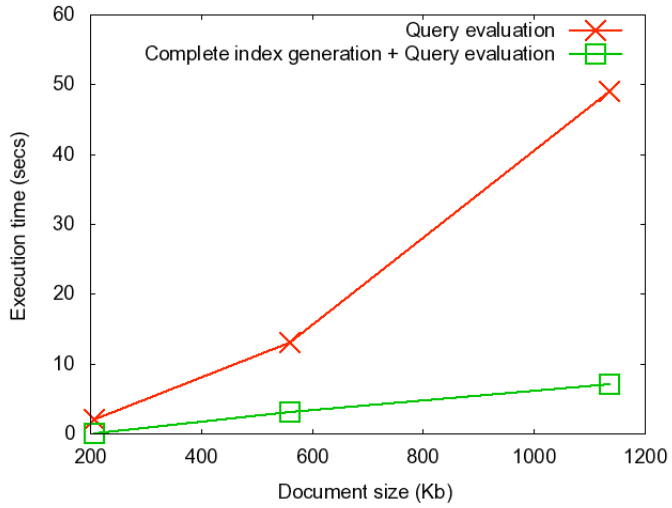
Document size (Kb)	Ex. time: case 1 (s)	Ex. time: case 2 (s)
208	2.459	0.930
560	13.227	3.001
1136	49.212	7,492

Figure 4.3: Experiments with laziness

The results of our experiment are depicted in Figure 4.3 and clearly show how laziness speeds up the query evaluation by automatically avoiding the construction of unnecessary data structure (e.g. for a of 1Mb XML document, the evaluation of the filtering rule in Case 2 is ∼7 times faster than the rule evaluation of Case 1).

Qualitative and Quantitative Analysis. In order to evaluate the usefulness of our approach in a realistic scenario, we have benchmarked our system by using the XMark benchmark suite [51]. The suite offers a set of 20 queries, each of which is intended to challenge a particular primitive of the filtering engine, along with the XML documents generator xmlgen which can be used to produce the synthetic data on which running the experiments. By means of our formalism, we are able to express 17 queries out of 20. The remaining three queries cannot be formalized, since they involve document transformation and computational capabilities which are out of the scope of a simple filtering language, e.g. lexicographic ordering (query n. 19), currency conversion (query n. 18), and output formatting (query n. 10).

From a purely quantitative point of view, we tested the system on a Macbook Intel Core 2 Duo 2Ghz equipped with 2Gb of RAM memory. We defined four filtering rules encompassing all the language features. Specifically, rule $Q1$ models a positive nested filtering rule with regular expressions, rule $Q2$ is a positive nested filtering rule which includes applications of the occurrence operator and arithmetic tests, rule $Q3$ employs



Document size (Kb)	Ex. time Q1 (s)	Ex. time Q2 (s)	Ex. time Q3 (s)	Ex. time Q4 (s)
208	0.985	1.267	1.101	2.183
560	3.046	3.941	3.158	6.812
1136	7.781	8.428	6.647	15.196
3424	43.053	35.248	26.674	65.821
5640	105.358	77.766	54.801	146.839

Figure 4.4: Experimental evaluation of the PHIL System

the counting operator, and finally rule $Q4$ is an example of negative filtering. All the considered rules contain one or more occurrences of the synonymity operator.

Figure 4.4 shows the results obtained by executing the four filtering rules to five different, randomly generated XML documents which have been synthesized by xmlgen data generator. We tuned the generator in order to yield XML documents whose size ranges from 208Kb to almost 6Mb. Execution times of each filtering rule are computed as the average of three filtering rule's runs. The preliminary results are quite encouraging even on experiments that exceed the toy size: all the rules are evaluated in less than three minutes on a repository whose size is almost 6Mb. Moreover, it is a matter of few seconds obtaining an answer on a 1Mb XML document. Finally, note that negative filtering behaves worse than positive filtering. This is mainly due to the fact that, in the current implementation, negative filtering has to traverse the entire data tree (which may easily consist of more than 100000 nodes) in order to get rid of the detected patterns. Therefore, when dealing with large XML documents, the overhead due to the data tree traversal might be considerably high.

4.5 Semantic Filtering via DL Reasoning

Up to now we have presented a declarative pattern-based language for XML filtering, which is endowed with an approximate pattern matching mechanism. PHIL provides an efficient, cost-based query transformation algorithm which allows to automatically rename, insert and delete XML items in order to find the best match between a pattern and a given XML document. Although our approach provides a great degree of flexibility in filtering operations, it offers no support for *semantic* filtering. In fact, XML data can be equipped with a semantics formalized by a given knowledge base which enriches data with meanings and properties.

Therefore, it would be interesting to exploit such knowledge base information to refine the filtering mechanism. In other words, one may be interested in extracting data from an XML repository by querying a knowledge base to which the data are related. As an example, assume that an XML database containing information regarding dishes and wines is given along with a knowledge base specifying wine properties (color, flavor,...). One may want to select all the dishes which are best served with red, structured wines from the database by querying both the database and the given knowledge base and combining their outcomes.

Combining XML query languages with ontologies is a crucial issue for the growth of the Semantic Web. In the last years, many approaches have been followed with the final aim of integrating description logic frameworks into XML query languages. The work more closely related to ours is DIGXcerpt [79], which presents an extension for ontology reasoning built on top of the well-known XML query language Xcerpt [45]. Although DIGXcerpt contains a lot of powerful constructs to search and manipulate XML data, it only allows to perform exact query matching, while our methodology is able to perform a more flexible matching which ranks the query results w.r.t. their similarity cost.

[5] describes a logical framework in which XQuery programs are enriched with

metadata modeled by using the W3C's Resource Description Framework (RDF) [166]. Basically, metadata are retrieved from RDF documents and then integrated into XQuery queries. This approach enables a limited form of data inference from RDF documents, which does not provide the reasoning capabilities of more complex description logic formalisms like the one formalized within the OWL-DL [165] framework.

Finally, the approach which is discussed in [18] provides a general scheme for hybrid integration of rule languages with constrained-based languages such as description logic-based ontologies. The scheme is then particularized to the integration of Datalog and Xcerpt languages with a generic description logic reasoner. Although the proposed methodology is highly flexible and can be easily tailored to different rule languages, it only provides a restricted reasoning support, since only boolean ontology queries are allowed. On the contrary, in our framework, boolean as well as non-boolean ontology queries are definable and naturally integrated in the filtering language.

In the following section, we present an extension of the PHIL language which integrates an approximate pattern-matching mechanism with knowledge base reasoning in order to enable semantic data filtering. Roughly speaking, patterns are searched in an XML document in an approximate way (that is, modulo renaming, deletion and insertion of XML items) using additional information which can be retrieved by querying (possible remote) knowledge bases. Our approach is particularly suitable to information retrieval, since in this context information may be ambiguous or incomplete. The integration of approximate pattern matching with knowledge base reasoning allow to tackle both issues.

4.6 The Extended Filtering Language

The extension to the filtering language PHIL is intended to integrate filtering rules with DL query templates used to infer information from knowledge bases. The way we can take advantage of such additional information is twofold: (i) we can automatize the search of the XML tag synonyms employed by the approximate matching engine when renaming pattern transformations are applied; (ii) we can use boolean DL queries as filtering conditions to refine pattern detection. Hence, the proposed extension affects only the pattern specification and the conditions whose new syntax is presented in what follows.

Figures 4.5–4.6 illustrate the (graph) structure of two knowledge bases over the domains of wines and eating places respectively, that we use in our running example. The former knowledge base contains a set of wine names, which are modeled as individuals. Each wine has color and flavor properties that are formalized by means of concepts. The latter knowledge base classifies distinct eating places

- by kind, using the concepts *Restaurant*, *Tavern*, *Farm house*, and *Pizzeria*;
- by price, using the concepts *Expensive*, *Medium*, *Cheap*.

A role is used to determine the geographic location of each eating place instance.

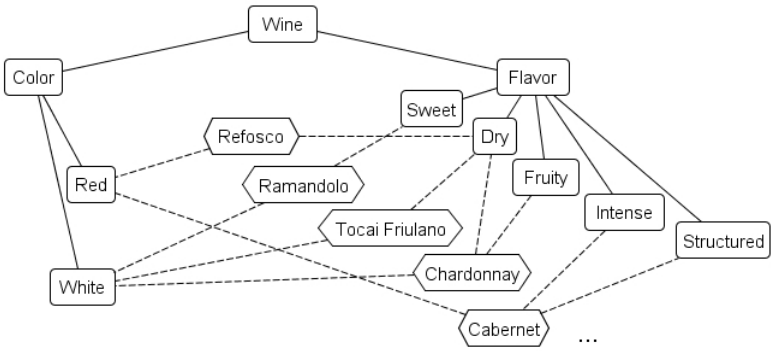


Figure 4.5: A knowledge base about wines

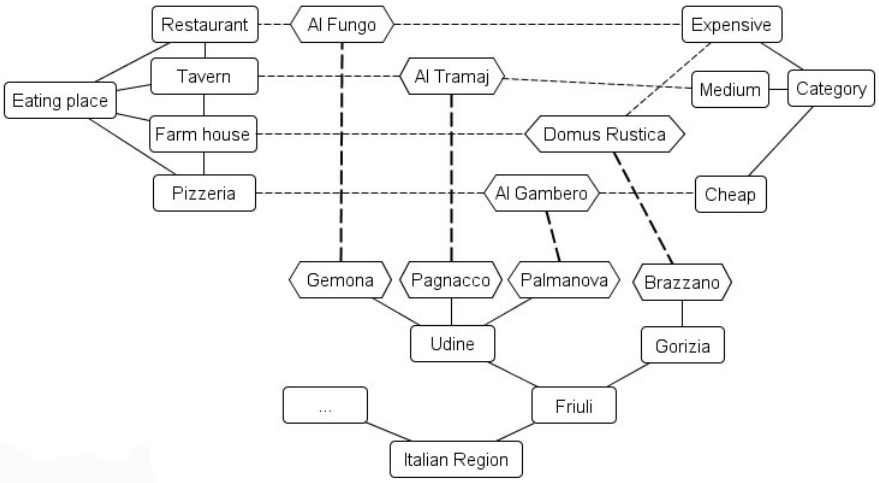


Figure 4.6: A knowledge base about eating places

In Figures 4.5–4.6, we have used rectangles to denote concepts, while hexagons denote individuals. Straight, solid lines connecting concepts represent subconcept relations (e.g. "a tavern is an eating place"). Dashed lines connecting individuals to concepts define membership relations of individuals w.r.t. concepts (e.g. "Chardonnay belongs to white wines"). Finally, bold, dashed lines connecting pairs of individuals formalize roles, that is, binary relations between individuals ("the restaurant called *Al Fungo* is located in a town called *Gemona*").

Patterns. Patterns of filtering rules are XML data used to describe the information we want to detect inside a given XML document. A pattern is built by composing the following syntactical elements.

Variables that play the role of placeholders for unknown pieces of XML code (we assume to have a countable infinite set of variables at hand {`X,Y,...`}).

Text selectors that is, strings of plain text surrounded by single quotes (e.g. `'Dear friend'`). Text selectors will be matched against the textual part of the XML document.

Tag selectors. In Section 4.1 we presented a *synonymity* operator "`$`", which enables the flexible matching of tag selectors. Here we transform the concept of synonymity operator allowing tag selectors to be followed by a non-boolean DL query. More precisely, given a tag selector `t`, by the syntax `t[DL(K, r)]` we retrieve all the synonyms of the tag selector `t` by executing the reasoning service `r` in the knowledge base `K`. Such synonyms are then used by the pattern-matching algorithm to find approximate results.

Example 4.2 *Consider the knowledge base of Figure 4.6. The synonyms retrieved by the tag selector*

```
places[DL(EatKB, children(Eating place))]
```

are
`{Restaurant,Tavern,Farm house,Pizzeria}`.

All the other syntax elements such as the occurrence operator and the boolean operators are left unchanged as are presented in Section 4.1.

Conditions. Conditions are used to refine the search of a given pattern inside an XML document. Roughly speaking, whenever an instance of a pattern is detected, the associated instance of the condition list is evaluated. The detected pattern is then delivered to the user if and only if the instance of the condition list evaluates to true.

Our language is endowed with constructs for specifying several classes of conditions such as

Membership tests which allow one to establish whether a given piece of XML is contained in the language denoted by a given regular expression.

Functional constraints which allow to perform some computations over the extracted XML data and then check the results.

Example 4.3 *Assume that an XML database modeling a price list is given. For each entry of the list, the price without VAT, the VAT, and the total price are defined. Using three variables* `X`, `Y`, *and* `Z`, *we can model the functional constraint* `X+Y=Z` *which verifies that, for each entry, the total price is made up of the price without VAT and the VAT.*

Semantic constraints which allow to check semantic properties of XML documents. Semantic constraints are specified through boolean DL query templates.

Example 4.4 *Consider the knowledge base of Figure 4.5, then the following semantic constraint*

```
DL(WineKB, instance(Y,Red))
```

evaluates to true if and only if the value bound to variable `Y` *is a red wine (i.e. belongs to the concept* `Red`*).*

In the following example, we show a complete filtering rule, which uses several constructs of the language we presented.

Example 4.5 *Consider the knowledge base of Figure 4.5 and Figure 4.6. Assume that an XML database containing a list of eating places is given. For each eating place entry a menu is specified which contains pairs of dishes and suggested wines.*

We would like to retrieve all the main courses which are served with red wine for each eating place in the repository. Since we have different kinds of eating places, we can retrieve all the possible categories of eating place by using the tag selector of Example 4.2, and then exploiting our matching engine to search for the approximate results. Moreover, to check if a wine has red color, we can make use of the semantic constraint of Example 4.4. Therefore, we might specify the filtering rule

```
filterAllBest
  places[DL(EatKB, children(Eating_place))](
         name(Z),
         menu(foodwinepair(
                  dish(type('main'),name(X)),
                  wine(Y)
         ))
  )
  in file 'Menus.xml' where DL(WineKB, instance(Y,Red)) (P)
```

where `Z`,`X`,`Y` *are variables which are bound to eating places, main courses, and wines, respectively.*

4.7 An XML Formalization of the Semantic Filtering Language

In this Section we present a XML formalization of the extended filtering language presented in Section 4.6. Some benefits expressing filtering rules in XML are:

- you don't have to learn a new language;
- you can use your XML editor to edit your filtering rules;
- you can use your XML parser to parse your filtering rules;
- you can check the well-formedness and correctness of your filtering rules providing an XML Schema definition[5];
- you can transform your filtering rules with XSLT.

Before considering the XML syntax of filtering rules, let us introduce a way to model and query ontologies using an XML formalism.

4.7.1 Using DIG to Model and Query Ontologies

In order to connect the filtering engine and the ontology reasoner we make use of the DIG interface [35]. The DIG interface is an API for Description Logic systems which is capable of expressing class and property expressions common to most DLs. In particular, it can model the well-known description logic formalized within the OWL-DL [165] framework, which is supported by several ontology reasoners.

The DIG interface is equipped with four XML languages which are employed to formalize and query ontologies modeling a given application domain. These languages are (i) the *tell* language, (ii) the *concept* language, (iii) the *ask* language, and (iv) the *response* language.[6]

The DIG *concept* and *tell* languages basically contain constructs for describing and then loading an ontology into a reasoner. Roughly speaking, they allow us to formalize the structure of an ontology by defining concepts (classes), roles (relations), individuals (instances of classes), *etc.* A fragment of the wine knowledge base of Figure 4.5 encoded using the DIG languages is shown in Figure 4.7.

The DIG *ask* language provides the notation to formalize statements which are used to query ontologies loaded into ontology reasoners. The *ask* language includes many constructs that can be classified in different categories as shown in Table 4.1. Roughly speaking, *ask* statements allows to infer information regarding concepts, roles and individuals of a given ontology. *Ask* statements can model boolean as well as non-boolean ontology queries. More precisely, a boolean (respectively, non-boolean) ontology query is an *ask* statement that returns a boolean (respectively, non-boolean) value.

[5]The XML Schema defining the extended filtering language is given in Appendix A.1.
[6]The complete DIG formalization is available at [35].

```
...
<defindividual name="Chardonnay"/>
<instanceof>
  <individual name="Chardonnay"/>
  <catom name="white"/>
</instanceof>
<defindividual name="Cabernet"/>
<instanceof>
  <individual name="Cabernet"/>
  <catom name="red"/>
</instanceof>
<instanceof>
  <individual name="Chardonnay"/>
  <catom name="dry"/>
</instanceof>
...
```

Figure 4.7: DIG fragment of the wine knowledge base

Example 4.6 *Consider the ontologies of Figure 4.5 and Figure 4.6. We define an* ask *statement containing the boolean query* `Q1` *and the non-boolean query* `Q2` *as follows (for the sake of readability, here and throughout the whole paper, we omit namespace declarations).*

```
<asks xmlns=...>
 <children id="Q1">
  <catom name="Category"/>
 </children>
 <instance id="Q2">
  <individual name="Ramandolo"/>
  <catom name="white"/>
 </instance>
</asks>
```

`Q1` *allows to retrieve all the concepts which are children of the concept* `Category` *(that is,* `Cheap, Medium, Expensive`*), while* `Q2` *checks whether the individual name* `Ramandolo` *is a white wine.*

Finally, the DIG *response* language formalizes the possible response statements generated after the execution of an *ask* statement (e.g. boolean values, sets of ontology elements, error messages,...)

4.7.2 The Extended DIG Ask Language

DIG *ask* statements are basically ground formulae –that is, formulae not containing variables– of a given description logic. In order to make them more flexible and suitable for our purposes, we adopted a generalized version of *ask* statements, by defining "templates" which

Category	Ask Constructs
Primitive Concept Retrieval	<allConceptNames/ >
	<allRoleNames/ >
	<allIndividuals/ >
Satisfiability	<satisfiable>C< /satisfiable>
	<subsumes>C1 C2< /subsumes>
	<disjoint>C1 C2< /disjoint>
Concept Hierarchy	<parents>C< /parents>
	<children>C< /children>
	<ancestors>C< /ancestors>
	<descendants>C< /descendants>
	<equivalents>C< /equivalents>
Role Hierarchy	<rparents>R< /rparents>
	<rchildren>R< /rchildren>
	<rancestors>R< /rancestors>
	<rdescendants>R< /rdescendants>
Individual Queries	<instances>C< /instances>
	<types>I< /types>
	<instance>I C< /instance>
	<roleFillers>I R< /roleFillers>
	<relatedIndividuals>R < /relatedIndividuals>
	<toldValues>I A< /toldValues>

Table 4.1: DIG Ask Language

(i) can be easily reused in several filtering rules;

(ii) can be instantiated with values computed at run-time.

Therefore, we extend the DIG *ask* language by

- introducing variables into *ask* statements. Variables are employed as placeholders for concepts, roles, and individuals. We denote a variable, whose name is `varName`, by the syntax `var:varName`. *Ask* statements containing variables are called *non-ground ask* statements, and implement DL query templates.

  Example 4.7 *Consider the ontology of Figure 4.5. The following non-ground* ask *statement models a boolean query which checks whether (the value assigned to) the variable* `Y` *is a red wine; that is, it is an instance of the concept* `Red`*.*

```
<asks xmlns=... >
 <instance id="RedWine">
  <individual name="var:Y"/>
  <catom name="Red"/>
 </instance>
</asks>
```

- letting *ask* statements denote non-boolean queries to reference XML tags of a given XML document via the `tag:self` notation. As we will show in Section 4.7.3, a non-boolean ontology query `Q` is typically bound to an XML tag `t`. By using the `tag:self` construct, we can automatically reference `t` inside `Q` without citing it explicitly.

  Example 4.8 *Consider the ontology of Figure 4.6. Assume that a given XML tag* `t` *is associated with the following* ask *statement modeling a non-boolean query* `Q`*.*

  ```
  <asks xmlns=... >
   <children id="Syn">
    <catom name="tag:self"/>
   </children>
  </asks>
  ```

  The query `Q` retrieves all the concepts which are children of the concept `t`.

4.7.3 An XML Syntax for the Filtering Language

Basically, a filtering rule can be formalized by means of the following XML syntax:

```
<rule>
 {<count cost="value"/>}
 <filter> filterop </filter>
 <pattern> XML-pat </pattern>
 <document> XML-doc </document>
 {<conditions> cond-list </conditions>}
 {<mode> mode</mode>}
</rule>
```

The XML elements represented between braces are optional. The operator `count` allows one to count the number of detected pattern instances with similarity cost less then `value` in the given XML document. The `filterop` operators are `filter{One,All}Exact`, `filter{One,All}Best`. `XML-doc` can be specified by an URL referring to an XML document, by some XML code, or even by a nested filtering rule, since the execution of a filtering rule generates an XML document, which can feed another filtering rule. The filtering mode is a label belonging to the set {`P,N`}. Whenever a rule does not specify a filtering mode, it has to be considered a positive

filtering rule (that is, the mode is `P`). `XML-pat` is the information we want to detect inside a given XML document.

We here limit ourselves to specify the syntax of tag selectors and boolean connectors, since variables and text selectors are as usual.

Tag selectors are XML tags. Tag selectors can contain two attributes which enable tag flexible matching (i.e. matching modulo tag renaming). The `ont` attribute specifies an ontology file name, while the `query` attribute specifies the file name of an extended DIG *ask* statement modeling a non-boolean DL query. More precisely, given a tag selector `t`, by the syntax

```
<t ont="ontName" query="queryName">
```

we retrieve all the synonyms of the tag selector `t` by querying the ontology `ontName` via the query `queryName`. As we have seen in Section 4.7.2, the tag selector `t` can be referenced inside the query `queryName` using the `tag:self` notation. During the query execution `tag:self` occurrences are replaced by `t`.

Example 4.9 *Consider the Example 4.2 and let us translate it in the XML format. Consider Figure 4.6 and the extended* ask *statement of Example 4.8. Then, the synonyms retrieved by the tag selector*

```
<eating_place ont="EatOnt" query="Syn">
```

are
`{Restaurant,Tavern,Farm house,Pizzeria}`.

The occurrence operator is represented here by means of the *child* attribute. Given an XML document containing a tag `s` that has n children labeled by the tag `t`, the notation `<t child="i">` selects the `i`-th child labeled with `t`. The keyword `last` is used to select the last element of the sequence.

Boolean connectors define boolean-connected XML patterns. We use `<and>`, `<or>` and `<xor>` tags to express conjunctions and disjunctions of XML patterns. Boolean connectors also replace the "`,`", "`|`" and "`?`" operators.

Conditions. Conditions expressing membership tests and functional constraints are formalized using a simplified version of RuleML [41].

Example 4.10 *Considering the functional condition expressed in Example 4.3, we can formalize it following the RuleML syntax by means of the following XML code.*

```
<Equal>
 <lhs> <Expr>
        <Fun> + </Fun>
        <Var> X </Var>
        <Var> Y </Var>
       </Expr>
 </lhs>
```

```
 <rhs> <Var> Z </Var> </rhs>
</Equal>
```

Semantic conditions are formalized by means of the following syntax:

```
<ontCond ont="ontName" query="queryName">
```

where `ontName` is an ontology file name, and `queryName` represents the file name of an extended DIG *ask* statement modeling a boolean DL query template. Note that DIG *ask* statements may contain variables. In this case, such variables are instantiated with values of the detected pattern instances before sending the statements to the reasoner.

Example 4.11 *Consider the ontology of Figure 4.5 and the extended* ask *statement of Example 4.2 which models the boolean query* `RedWine`. *Therefore, the following semantic constraint*

```
<ontCond ont="wineOnt" query="RedWine">
```

expresses the condition of Example 4.4.

Example 4.12 *The filtering rule of Example 4.5 can then be translated in the following XML format.*

```
<rule>
 <filter>filterAllBest</filter>
 <pattern>
  <eating_place ont="eatOnt" query="Syn">
   <and>
    <name> Z </name>
    <menu>
     <foodwinepair>
      <and>
       <dish>
        <and>
         <type>'main'</type>
         <name>X</name>
        </and>
       </dish>
       <wine>Y</wine>
      </and>
     </foodwinepair>
    </menu>
   </and>
  </eating_place>
 </pattern>
 <document>
  <docFile>'Menus.xml'</docFile>
```

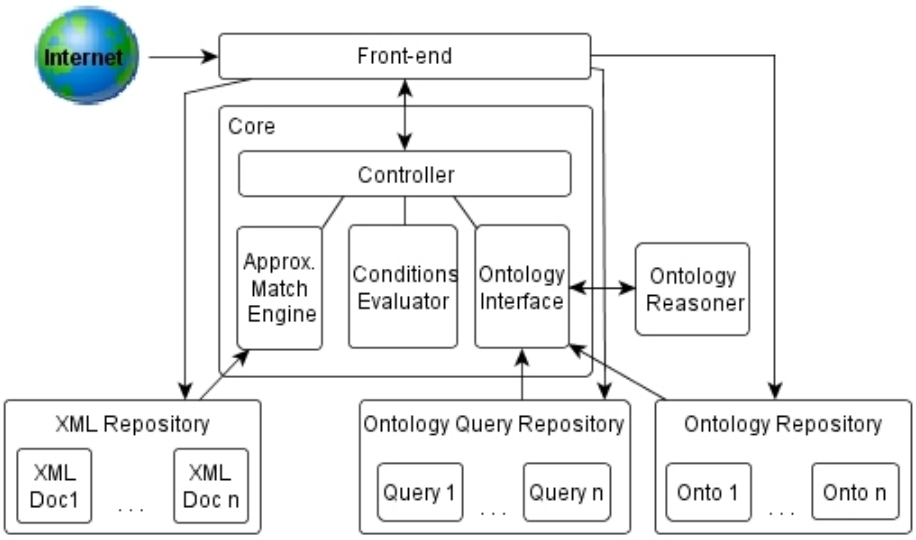


Figure 4.8: System Architecture

```
 </document>
 <conditionClause>
  <ontCond ont="wineOnt" query="redWine"/>
 </conditionClause>
 <mode>P</mode>
</rule>
```

4.8 The XPhil Filtering System

The proposed extended filtering language has been implemented in the XPhil system, which is directly accessible through a Web service available at [25]. Alternatively, a stand-alone application can be freely downloaded from [25] and executed offline. The implementation has been written in the Haskell functional language and consists of about 1,800 lines of code. The sources available at [25] contain the context-free grammar of our language and some parsers for the filtering language and for XML documents. There are some modules for the approximate pattern-matching algorithm and one module to manage the variable associations, one module to interface ontology reasoners and another one to evaluate syntactic and semantic rule conditions.

XPhil supports ontology reasoning via the DIG API. More precisely, it can be connected to any ontology reasoner which is equipped with the DIG interface. In our experiments, we have used XPhil together with Pellet [147], an efficient, open-source ontology reasoner for the OWL-DL[165] framework.

The implemented system is divided into three parts: the front-end, the core and

the ontology reasoner. The core can be in turn divided in a controller, an approximate matching engine, an ontology interface and a condition evaluator. This architecture is represented in Figure 4.8. Users may use the front-end to load XML documents, ontologies and ontology queries, specify and execute the querying rules and view the results. The core is responsible for the execution of rules by means of the controller that manages the interaction of the matching engine, the condition evaluator and the ontology interface. The matching engine is aimed to find the best approximate matches of the rule pattern against the given XML document, the condition evaluator deals with the evaluation of rule conditions and the ontology interface allows one to load and release ontologies in the ontology reasoner and send ontology queries to the ontology reasoner.

When a rule is executed, the controller locates into the rule the required ontologies and asks the ontology interface to load such ontologies in the reasoner. Then, the approximate matching engine is activated to find the best matches of the pattern against the specified XML document. During the matching phase, whenever a tag selector associated with an ontology query is processed, the ontology interface is called to query the appropriate ontology and return the results. Whenever one or more embeddings of the rule pattern are found in the XML document, the rule conditions (if any) have to be verified. The only delivered results will be those embeddings whose condition instances evaluate to true. Membership tests and functional constraints are managed by the condition evaluator, whereas in order to assess semantic conditions, the ontology interface need to be called in order to transfer the query to the ontology reasoner. Finally the controller asks the ontology interface to release all ontologies, packs the results and sends them to the front-end.

Figure 4.9 provides a snapshot of the graphical user interface of the XPhil online system. The interface allows one to load and execute some example rules or build and execute user-defined rules. The interface is divided in six panels whose functions are briefly explained in the following:

- The *Example Rule* panel contains some links to load some rule examples.
- The *XPhil Rule* panel, showing at the beginning the message *No Rule Loaded*, contains a text area where users can edit a rule, otherwise it is possible to load an XML file containing a rule using the *Browse* button. The *Load Rule* button loads the edited or browsed rule into the system and the *No Rule Loaded* message will change into *Rule Loaded*.
- The *XML Document* panel allows one to search an XML document browsing the file system using the *Browse* button and then loading it by means of the *Load Document* button.
- The *Rule Description* panel is usually empty. When an example rule is loaded, this panel may contain a brief rule explanation in natural language.
- The *Ontology* panel allows one to load one or more XML files containing an ontology description specified by means of DIG *tell* statements. The *Browse*

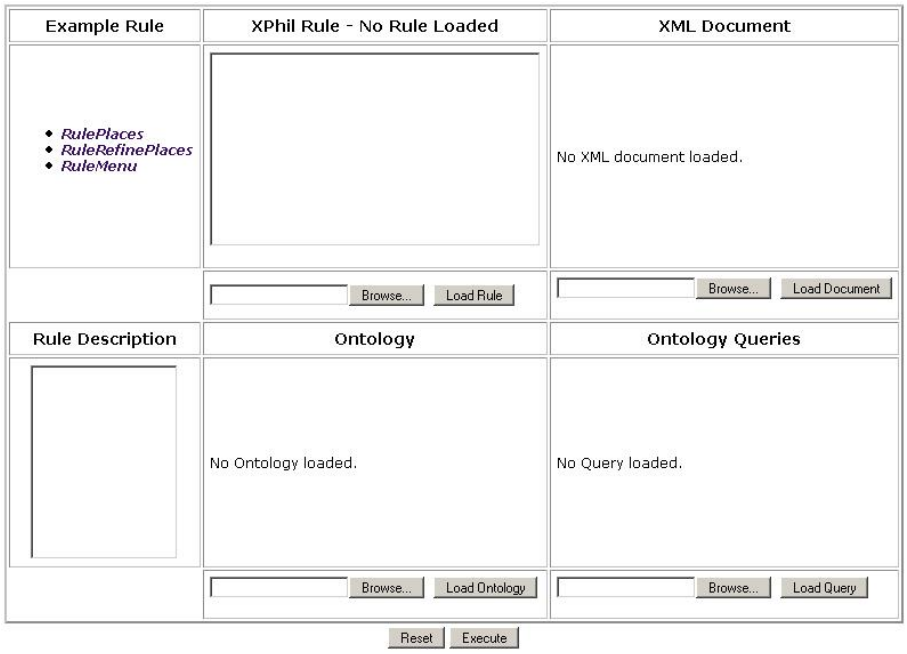


Figure 4.9: Screenshot of the XPhil system online.

button allows one to browse the file system and the *Load Ontology* button loads the specified ontology.

- The *Ontology Queries* panel allows one to load one or more XML files, containing extended DIG *ask* statements modeling boolean ontology queries. The *Browse* button allows one to browse the file system and the *Load Query* button will load the specified query.
- Finally, the *Reset* button is used to clean up all the panels while the *Execute* button executes the specified rule and the result will be displayed in a new window.

5

Web Systems Verification

The increasing complexity of Web systems has turned their design and construction into a challenging problem. Moreover, web systems are very often *collaborative* applications in which many users freely contribute to update their contents (e.g. wikis, blogs, social networks,...). In this scenario, the task of keeping data correct and complete is particularly arduous, because of the very poor control over the content update operations which may easily lead to data inconsistency problems. Systematic, formal approaches can bring many benefits to Web systems construction, and give support for automated Web systems verification.

In recent years, several rule-based methodologies for validating the content of Web systems have been developed. In [61] constraint logic programming is applied to constrain the static content and the structure of a Web site, while [98] defines type systems and type checking techniques which are basically natural generalizations of DTDs and XML Schema definitions for describing and validating the structure of XML documents. Finally, the framework xlinkit [83] allows one to check the consistency of distributed, heterogeneous documents as well as to fix the (possibly) inconsistent information. The specification language is a restricted form of first order logic combined with Xpath expressions [163] where no functions are allowed. With respect to the correctness of Web applications, a symbolic model-checking approach is formalized in [75] which constructs a finite states model of the system in the model checker input language, and then checks the considered properties which are expressed in CTL logic. For a comprehensive survey about the general problem of checking constraints between multiple documents, we refer to [86, 80, 49]. All the mentioned approaches, albeit very useful in their specific domains, share the same limitation: the syntactic as well as semantic constraints they specify only rely on the data to be checked.

In this chapter, we present a rule-based specification language which allows one to formalize and automatically check semantic as well as syntactic properties over the static contents of any Web system. The language provides constructs for specifying two kinds of rules: *correctness* rules and *completeness* rules. The former describe constraints for detecting erroneous information into a given XML repository, while the latter recognize incomplete/missing information. The language is inspired by the GVerdi specification language [9, 32] and extends it in the following ways:

(i) Web contents (typically, XML/XHTML data) are frequently coupled with on-

tologies with the aim of equipping data with semantic information. Our specification language provides ontology reasoning capabilities which allow us to query a (possibly) remote ontology reasoner to check semantic properties over the data of interest, and to retrieve semantic information which may be combined with the syntactic one for improving the analysis.

(ii) We extend the GVerdi specification language with new rule constructs for the definition of conjunctions and disjunctions of patterns which can be recognized inside XML documents. The new constructs increase the expressiveness of the original language, since they enable the specification of a larger set of semantic as well as syntactic constraints.

(iii) Along with the specification language, we formulate a novel verification methodology which automatically checks a specification against the considered Web contents and discovers incorrect as well as incomplete information.

5.1 The Web specification language

Our specification language allows us to formalize and verify properties over the content of a Web system.

Web content denotation. Throughout this paper, we assume that the data to be checked are stored into an XML repository. Let us consider two alphabets T and $\mathcal{T}ag$. We denote the set T^* by $\mathcal{T}ext$. An object $\mathtt{t} \in \mathcal{T}ag$ is called *tag* element, while an element $\mathtt{w} \in \mathcal{T}ext$ is called *text* element. Since XML documents are provided with a tree-like structure, they can be straightforwardly translated into ordinary terms of a given term algebra $\tau(\mathcal{T}ext \cup \mathcal{T}ag)$ as shown in Figure 5.1. Note that XML/XHTML tag attributes can be considered as common tagged elements, and hence translated in the same way.

```
<members>
  <member status="professor">
    <name> mario </name>
    <surname> rossi </surname>
  </member>
  <member status="technician">
    <name> franca </name>
    <surname> bianchi </surname>
  </member>
</members>
```

```
members(
   member(status(professor),
       name(mario),
       surname(rossi)
   ),
   member(status(technician),
       name(franca),
       surname(bianchi)
   )
)
```

Figure 5.1: An XML document and its corresponding encoding as a ground term `p`.

In the following, we will also consider *Web templates*, which are terms of a non-ground term algebra, which may contain variables. Web templates are used for specifying patterns to be recognized in XML repositories. See [9] for more details.

Web specifications. The language provides constructs for specifying two kinds of rules: *correctness* rules and *completeness* rules. The former describe constraints for detecting erroneous information into a given XML repository, while the latter recognize incomplete/missing information. Both kinds of rules may be *conditional*, that is, they can be fired if and only if an associated condition holds.

A *condition* is a finite (possibly *empty*) sequence $c_1, \ldots, c_n$, where each c_i may be (i) a membership test w.r.t. a regular language of the form $X \in \texttt{rexp}$[1], (ii) an equation $s = t$, where s and t are expressions which may contain nested function calls to be evaluated[2], and (iii) a boolean DL functional query template.

Given a substitution σ, which takes an expression and replaces its variables with ground terms and a condition $C \equiv c_1, \ldots, c_n$, we say that C *holds* for σ iff each $c_i\sigma$ is ground and

- if $c_i \equiv X \in \texttt{rexp}$, then $X\sigma \in \mathcal{L}(\texttt{rexp})$, where $\mathcal{L}(\texttt{rexp})$ is the regular language described by `rexp`;
- if $c_i \equiv (s = t)$, then the evaluations of $s\sigma$ and $t\sigma$ compute the same value.
- if $c_i \equiv DL(\mathcal{K}, r)$, a boolean DL functional query template, then $eval(DL(\mathcal{K}, r\sigma))$ returns $true$[3].

Now, we are ready to introduce correctness as well as completeness rules.

Definition 5.1 (Correctness rule) *A* correctness rule *is an expression of the form*

$$\bigwedge_{i=1}^{n} \texttt{l}_i \rightharpoonup \texttt{error} \mid \texttt{C}$$

where each $\texttt{l}_i$ *is a Web template,* `error` *is a reserved constant,* `C` *is a condition.*

Informally, the meaning of a correctness rule $\texttt{l}_1 \wedge \ldots \wedge \texttt{l}_n \rightharpoonup \texttt{error} \mid \texttt{C}$ is as follows. Whenever an instance $\texttt{l}_i\sigma$ of $\texttt{l}_i$ for each $i \in \{1, \ldots, n\}$ is recognized in some XML document `p`, and the rule condition `C` holds for σ, then XML document `p` is signaled as an incorrect document.

Example 5.2 *Consider an XML repository containing academic information along with a knowledge base* `Univ` *modeling such a domain. Suppose we want to verify the following property:* if an associate professor has more than three Ph.D. students, then he cannot teach more than one course. *Then a possible correctness rule formalizing such a property might be*

```
course(cId(X),professor(name(Y)))
∧ course(cId(Z),professor(name(Y))) ⇀ error |
  DL(Univ,instanceOf(Y,AssocProf⊓(≥3hasStd ⊓ ∀hasStd.PhDStudent))),
  X =/= Z
```

[1] Regular languages are denoted by the usual Unix-like regular expression syntax.

[2] In our framework, equation evaluation is handled by standard rewriting [107].

[3] We assume that all the function calls appearing in $r\sigma$ are evaluated before executing the reasoning service.

In order to define completeness rules, we need the following auxiliary notion. Given an expression e, by $Var(e)$ we denote the set of all the variable appearing in e.

Definition 5.3 (Completeness rule) *A* completeness rule *is an expression of the form*

$$\bigwedge_{i=1}^{n} \mathtt{l}_i \rightharpoonup \bigvee_{j=1}^{m} \mathtt{r}_j \mid \mathtt{C}\ \mathtt{containing}\ \mathtt{ct}\ \langle \mathtt{q} \rangle$$

where each $\mathtt{l}_i, \mathtt{r}_j$ *are Web templates,* $\mathtt{C}$ *is a condition,* `containing ct` *is an optional clause, where* `ct` *is a ground term,* $\mathtt{q} \in \{\mathtt{A}, \mathtt{E}\}$*, and* $\bigcup_{j=1}^{m} Var(\mathtt{r}_j) \cup Var(\mathtt{C}) \subseteq Var(\mathtt{l})$.

Completeness rules are called *universal* (resp., *existential*), whenever $\mathtt{q} \equiv \mathtt{A}$ (resp., $\mathtt{q} \equiv \mathtt{E}$).

Intuitively, given an XML repository W, the interpretation of a universal (resp., existential) rule $\mathtt{l}_1 \wedge \ldots \wedge \mathtt{l}_n \rightharpoonup \mathtt{r}_1 \vee \ldots \vee \mathtt{r}_m \,|\, \mathtt{containing}\ \mathtt{ct}\, \langle \mathtt{A} \rangle$ (resp., $\langle \mathtt{E} \rangle$) w.r.t. W is as follows: if an instance $\mathtt{l}_i\sigma$ of $\mathtt{l}_i$ for each $i \in \{1, \ldots, n\}$ is recognized in some $p \in W$ and the condition $\mathtt{C}$ holds for σ, then an instance $\mathtt{r}_j\sigma$ of at least one $\mathtt{r}_j$, $j \in \{1, \ldots, m\}$ must be recognized in *all* (resp. *some*) XML documents of W containing the `ct` term. Roughly speaking, `ct` provides the "scope" of the quantification and allows us to compute the part of the XML repository which is checked by the rule; if `ct` is not specified the rule is applied to the whole repository.

Example 5.4 *Consider again an academic XML repository along with the usual knowledge base* `Univ`*. We want to verify that* for each course, given by a full professor, at least two exam dates must be provided*. A completeness rule formalizing this property might be*

```
course(cId(X)) ⇀ course(cId(X),examDate(),examDate()) |
                 DL(Univ,instanceOf(X,∃CourseGivenBy.FullProf))⟨E⟩
```

Finally, we define a *Web specification* as a pair (I_N, I_M), where I_N is a set of correctness rules, I_M is a set of completeness rules. Given an XML repository and a Web specification, diagnoses are carried out by running the Web specification rules against the XML repository.

Web specifications with meta-symbols. Sometimes, it is particularly fruitful to consider rules containing Web templates which may subsume several meanings. To this purpose, completeness and correctness rules may include special *meta-symbols* into those Web templates which are associated with non-boolean DL functional query templates.

Web specification rules containing meta-symbols have to be pre-processed before being executed on a given XML repository.

The following definition is auxiliary. Let e be a (syntactic) expression of our language, m be a meta-symbol, v be a symbol. By $e[m/v]$ we denote the expression e' obtained from e by replacing each occurrence of m with v.

Basically, we expand each rule r containing meta-symbols as follows:

- for each meta-symbol m appearing in r, we execute the associated ontology query and we collect the results $\{v_1, \ldots, v_n\}$;

- if m appears in the left-hand side of r, we replace r with the rules $r_1[m/v_1], \ldots, r_n[m/v_n]$.
- if m appears in a disjunct ρ of the right-hand side of r, we replace ρ in r with $\rho[m/v_1] \vee \ldots \vee \rho[m/v_n]$.

A full description of the expansion algorithm can be found in Section 5.2.

For the sake of clarity, let us see an example.

Example 5.5 *Consider again an academic XML repository along with the usual knowledge base* `Univ`*. We want to specify that* email or post address have to be specified for each university professor. *Assume that the knowledge base* `Univ` *contains (i) the concept* `contactInfo` *whose subconcepts are* `email` *and* `address`*; and (ii) the concept* `Professor` *whose subconcepts are* `AssociateProf` *and* `FullProf`*.*

We might model the considered property using a universal completeness rule containing two meta-symbols (namely, `contact` *and* `prof`*).*

```
metasymbol contact:  DL(Univ,getChildren("contactInfo"))
metasymbol prof:  DL(Univ,getChildren("Professor"))
prof(name(X)) ⇀ prof(name(X),contact())| containing member() <A>
```

By expanding the considered completeness rule, we generate the following set of rules without meta-symbols.

```
AssociateProf(name(X)) ⇀ AssociateProf(name(X),email()) ∨
                         AssociateProf(name(X),address()) |
                                        containing member() <A>
FullProf(name(X)) ⇀ FullProf(name(X),email()) ∨
                    FullProf(name(X),address()) |
                                        containing member() <A>
```

5.2 Expanding rules with meta-symbols

The main idea is that, to evaluate a rule against a Web repository, the hidden information represented by meta-symbols have to be made explicit. This is achieved by replacing each meta-symbol, associated with a non-boolean ontology query, with the query result. The replacing operation is performed in a different way whether a meta-symbol appears in the left or right hand side of a rule.

In this section we present an algorithm to deal with the expanding operation of a Web specifications rule. The pseudo code is shown in Algorithm 2. Recall that, if e is a (syntactic) expression of our language, m is a meta-symbol and v is a symbol, by $e[m/v]$ we denote the expression e' obtained from e by replacing each occurrence of m with v.

Algorithm 2 An algorithm for translating a specification rule into a set of specification rules without meta-symbols.

```
1: procedure EXPANSION OF META-SYMBOLS(r)
2:     Q ← {r}
3:     NR ← empty set
4:     while notEmpty(Q) do
5:         r ← a rule non deterministically chosen from Q
6:         if r contains meta-symbols in its left or right hand side then
7:             Er ← empty set
8:             t ← a meta-symbol that appear in r
9:             if t appears in both the left and right hand side of r then
10:                Er ← Expansion of a meta-symbol in both sides(t, r)
11:            end if
12:            if t appears in the right hand side of r then
13:                Er ← Expansion of a meta-symbol in the right hand side(t, r)
14:            end if
15:            if t appears in the left hand side of r then
16:                Er ← Expansion of a meta-symbol in the left hand side(t, r)
17:            end if
18:            for all rule in Er do
19:                delete the declaration for meta-symbol t
20:            end for
21:            Q ← Q ∪ Er
22:        else
23:            NR ← NE ∪ {r}
24:        end if
25:        delete r from Q
26:    end while
27:    return NR
28: end procedure
```

Definition 5.6 (Expansion of a meta-symbol in the left hand side) *Let* $r \equiv \bigwedge_{i=1}^{n} \mathtt{l}_i \rightharpoonup Rhs$ *a specification rule and* `metasymbol t: DLquery` *the declaration of a meta symbol appearing in the left hand side. Assume that* $\{\mathtt{s}_1, \ldots, \mathtt{s}_{\mathtt{gt}}\}$ *is the result of the execution of the* `DLquery`*. Then,* r *is replaced by the following set of rules derived from the expansion of* `t`*.*

$$\begin{cases} \bigwedge_{i=1}^{n} \mathtt{l}_i[\mathtt{t}/\mathtt{s}_1] \rightharpoonup Rhs \\ \ldots \\ \bigwedge_{i=1}^{n} \mathtt{l}_i[\mathtt{t}/\mathtt{s}_{\mathtt{gt}}] \rightharpoonup Rhs \end{cases}$$

If the result set of the execution of the `DLquery` *is empty, the rule* r *is simply deleted.*

Example 5.7 *Consider again an academic XML repository along with the usual knowledge base* `Univ`*. We want to verify that* each university professor has an home

page. *Assume that the ontology* `Univ` *contains the concept* `Professor` *whose subconcepts are* `AssociateProf` *and* `FullProf`*. A completeness rule formalizing this property might be*

```
metasymbol prof:  DL(Univ, getChildren(Professor))
prof(name(X)) ⇀ hpage(name(X)) | <E>
```

Using the expansion operation just described we can obtain the following set of rules:

```
AssociateProf(name(X)) ⇀ hpage(name(X)) | <E>
FullProf(name(X)) ⇀ hpage(name(X)) | <E>
```

Definition 5.8 (Expansion of a meta-symbol in the right hand side) *Let* $r \equiv Lhs \rightharpoonup \bigvee_{j=1}^{m} \mathtt{r}_j \mid \mathtt{C}(containing \mathtt{ct})\langle \mathtt{q} \rangle$ *be a completeness specification rule. Assume that* `metasymbol t: DLquery` *is a declaration of a meta-symbol appearing only in the right hand side and* $\{\mathtt{s}_1, \ldots, \mathtt{s}_{gt}\}$ *is the result set of the execution of the* `DLquery`*. For each disjunct* $\mathtt{r}_j$ *in Rhs a new* $\mathtt{nr}_j$ *is built as follows.*

$$\mathtt{nr}_j = \begin{cases} \mathtt{r}_j[\mathtt{t}/\mathtt{s}_1] \vee \ldots \vee \mathtt{r}_j[\mathtt{t}/\mathtt{s}_{gt}] & \textit{if } \mathtt{t} \textit{ appear in } \mathtt{r}_j \\ \mathtt{r}_j & \textit{otherwise} \end{cases}$$

Then, the rule r is replaced by the rule $Lhs \rightharpoonup \bigvee_{j=1}^{m} \mathtt{nr}_j \mid \mathtt{C}\,(containing\ \mathtt{ct})\,\langle \mathtt{q} \rangle$*. If the result set of the execution of the* `DLQuery` *is empty, the rule r is simply deleted.*

Example 5.9 *Consider the Example 5.5 substituting the meta-symbol* `prof` *with a normal tag* `member`*. The rule will become*

```
metasymbol contact:  DL(Univ, getChildren(contactInfo))
member(name(X)) ⇀ member(name(X),contact()) | <A>
```

Expanding this rule we obtain the following new rule

```
member(name(X)) ⇀ member(name(X),email()) ∨
                  member(name(X),address()) | <A>
```

Definition 5.10 (Expansion of a meta-symbol in both sides) *Let* $r \equiv \bigwedge_{i=1}^{n} \mathtt{l}_i \rightharpoonup \bigvee_{j=1}^{m} \mathtt{r}_j \mid \mathtt{C}\,(containing\ \mathtt{ct})\,\langle \mathtt{q} \rangle$ *be a completeness specification rule. Assume that* `metasymbol t: DLquery` *is a declaration of a meta-symbol appearing in the left and right hand side and* $\{\mathtt{s}_1, \ldots, \mathtt{s}_{gt}\}$ *is the result set of the execution of the* `DLquery`*. Then, r is replaced by the following set of rules.*

$$\begin{cases} \bigwedge_{i=1}^{n} \mathtt{l}_i[\mathtt{t}/\mathtt{s}_1] \rightharpoonup \bigvee_{j=1}^{m} \mathtt{r}_j[\mathtt{t}/\mathtt{s}_1] \mid \mathtt{C}\,(containing\ \mathtt{ct})\,\langle \mathtt{q} \rangle \\ \ldots \\ \bigwedge_{i=1}^{n} \mathtt{l}_i[\mathtt{t}/\mathtt{s}_{gt}] \rightharpoonup \bigvee_{j=1}^{m} \mathtt{r}_j[\mathtt{t}/\mathtt{s}_{gt}] \mid \mathtt{C}\,(containing\ \mathtt{ct})\,\langle \mathtt{q} \rangle \end{cases}$$

If the result set of the execution of the `DLquery` *is empty, the rule r is simply deleted.*

The expansion rule of Example 5.5 is an example of application of the expansion of a meta-symbol appearing in the left and right hand side for the meta-symbol `prof`, together with the expansion of a meta-symbol in the right hand side for meta-symbol `contact`.

5.3 Verification Methodology

In this section we present a methodology to automatically verify a given XML repository w.r.t. a Web specification. Without loss of generality, we only consider Web specifications without meta-symbols, since any Web specification with meta-symbols can be transformed into an equivalent one without meta-symbols as explained in the previous section.

We proceed as follows: first we describe the *partial rewriting* [9] mechanism which allows us to detect patterns inside XML documents and rewrite them. Then, we will employ this evaluation mechanism to check correctness and completeness of an XML repository.

Simulation and partial rewriting. Simulation allows us to recognize the structure and the labeling of a given Web template into a particular XML document. It can be formally defined as follows.

Definition 5.11 *The* simulation *relation* $\trianglelefteq \subset \mathcal{T}(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \times \mathcal{T}(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ *in Web templates is the least relation satisfying the rule* $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_m) \trianglelefteq \mathtt{g}(\mathtt{s}_1, \ldots \mathtt{s}_n)$ *iff* $\mathtt{f} \equiv \mathtt{g}$ *and* $\mathtt{t}_i \trianglelefteq \mathtt{s}_{\pi(i)}$*, for* $\mathtt{i} = 1, \ldots \mathtt{m}$*, and some injective function* $\pi : \{1, \ldots \mathtt{m}\} \rightarrow \{1, \ldots, \mathtt{n}\}$.

W.l.o.g., we disregard quantifiers from Web specification rules.

Definition 5.12 (Partial rewriting) *Let* $\mathtt{s}, \mathtt{t} \in \mathcal{T}(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$*. We say that* $\mathtt{s}$ partially rewrites *to* $\mathtt{t}$ *via rule* $r \equiv \bigwedge_{i=1}^{n} \mathtt{l}_i \rightharpoonup \mathtt{Rhs} \mid \mathtt{C}$ *and substitution* σ *(in symbols* $\mathtt{s} \rightharpoonup_r \mathtt{t}$*) if and only if there exist positions* $u_1, \ldots, u_n$ *in* $\mathtt{s}$ *such that*

(i) $\mathtt{l}_i\sigma \trianglelefteq \mathtt{s}|_{u_i}$ *for all* $i \in \{1, \ldots, n\}$*;*

(ii) $\mathtt{C}$ *holds for* σ*;*

(iii) if $\mathtt{Rhs} \equiv \bigvee_{j=1}^{m} \mathtt{r}_j$ *then* $\mathtt{t} \equiv \mathtt{or}(\mathtt{r}_1\sigma, \ldots, \mathtt{r}_m\sigma)$*;*

(iv) if $\mathtt{Rhs} \equiv \mathtt{error}$ *then* $\mathtt{t} = \mathtt{error}(\mathtt{s}, u_1, \ldots, u_n)$.

5.3.1 Detecting correctness errors.

In this subsection, we provide a simple way to detect erroneous or undesirable data included in an XML repository. Our methodology allows us to precisely locate which part of an XML document does not fulfill the Web specification. Let us start by formalizing what does it mean for an XML document to be incorrect w.r.t. a certain rule.

Definition 5.13 *Let* W *be an XML repository and* (I_N, I_M) *be a Web specification. Given* $\mathtt{p} \in W$*, we say that* $\mathtt{p}$ *is* incorrect *w.r.t.* (I_N, I_M)*, if there exists a correctness rule* $r \equiv (\bigwedge_{i=1}^{n} \mathtt{l}_i \rightharpoonup \mathtt{error} \mid \mathtt{C}) \in I_N$ *such that*

(i) $\mathtt{p}$ *partial rewrites to* $\mathtt{error}(p, u_1, \ldots, u_n)$ *via* r *and substitution* σ*;*

(ii) $\mathtt{C}$ *holds for* σ.

We say that $(\bigwedge_{i=1}^{n} \mathtt{l}_i)\sigma$ *is an* incorrectness symptom *for* $\mathtt{p}$ *and* $\mathtt{error}(p, u_1, \ldots, u_n)$ *represents the* correctness error.

To find a correctness error in a document $\mathtt{p}$ we need to recognize first the left hand side $\bigwedge_{i=1}^{n} \mathtt{l}_i$ of a correctness rule into $\mathtt{p}$ by partially rewriting $\mathtt{p}$ via $\bigwedge_{i=1}^{n} \mathtt{l}_i$. If the rule condition holds then the faulty document $\mathtt{p}$ and an incorrectness symptom are supplied to the user. Note that the generated term $\mathtt{error}(p, u_1, \ldots, u_n)$ provides all the needed information to precisely locate the incorrectness symptom inside p.

5.3.2 Detecting completeness errors.

The verification of an XML repository W w.r.t. a set of completeness rules of a Web specification needs a more complex analysis. Essentially, the main idea to diagnose completeness errors is to compute the set of all possible terms that can be derived from W via the completeness rules of a Web specification (I_N, I_M) by means of partial rewriting. These terms can be thought of as requirements to be fulfilled by W (i.e. terms that must be recognized as part of some XML document in the repository). Then, we check whether the computed requirements are satisfied by W using simulation and quantification information. In summary, the method works in two steps, as described below.

- Compute the set of completeness requirements $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ for W w.r.t. I_M;
- Check $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ in W.

We now introduce some semantic foundations we require to formalize the analysis.

Completeness rule semantics. A *completeness requirement* (or simply *requirement*) is a triple $\langle \mathtt{e}, \mathtt{q}, \mathtt{ct} \rangle$, where $\mathtt{e}$ and $\mathtt{ct}$ are ground terms and $\mathtt{q} \in \{\mathtt{A}, \mathtt{E}\}$. A requirement is called *universal* whenever $\mathtt{q} = \mathtt{A}$, while it is called *existential* whenever $\mathtt{q} = \mathtt{E}$. Sometimes the components $\mathtt{q}$, $\mathtt{ct}$ of a requirement can be left undefined, in this case we simply omit them and write $\langle \mathtt{e}, \_, \_ \rangle$. Such requirements are called *initial* requirements.

Let $\langle \mathtt{e}, \mathtt{q}, \mathtt{ct} \rangle$ be a requirement, $r \equiv \mathtt{Lhs} \rightharpoonup \bigvee_{\mathtt{j}=1}^{\mathtt{m}} \mathtt{r}_\mathtt{j} \mid \mathtt{C\ containing\ ct_r}\ \langle \mathtt{q_r} \rangle \in I_M$ be a rule such that $s \equiv \mathtt{e} \rightharpoonup_r \mathtt{or}(\mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{m})$. We define the tree T_s associated with the partial rewriting step s as $\mathtt{or}(\langle \mathtt{h}_1, \mathtt{q_r}, \mathtt{ct_r} \rangle, \ldots, \langle \mathtt{h}_\mathtt{m}, \mathtt{q_r}, \mathtt{ct_r} \rangle)$.

Definition 5.14 (Production step) *Let* (I_N, I_M) *be a Web specification and* $\mathtt{re} \equiv \langle \mathtt{e}, \mathtt{q}, \mathtt{ct} \rangle$ *be a requirement. Let* $s_1, \ldots, s_k$ *be all partial rewriting steps which rewrite* $\mathtt{e}$ *using the rules in* I_M*. Let* $T_{s_1} \ldots, T_{s_k}$ *be the trees associated with the partial rewriting steps* $s_1, \ldots, s_k$*. The* production step *on* $\mathtt{re}$ *w.r.t.* I_M *builds the tree* $\mathtt{re}(T_{s_1}, \ldots, T_{s_k})$.

Note that, if there is no rule $r \in I_M$ such that $\mathtt{e} \rightharpoonup_r \mathtt{t}$, we say that $\mathtt{re} \equiv \langle \mathtt{e}, \mathtt{q}, \mathtt{ct} \rangle$ is *irreducible*. Let us now use the production step to define the maximal derivation tree for a requirement.

Definition 5.15 (Derivation tree) *Given a requirement* $\mathtt{re}$ *and a Web specification* (I_N, I_M)*, a* derivation tree *for* $\mathtt{re}$ *w.r.t. the set* I_M*, is defined as follows:*

- $\mathtt{re}$ *is a derivation tree for* $\mathtt{re}$ *w.r.t. the set* I_M*;*
- *if* T *is a derivation tree for* $\mathtt{re}$ *w.r.t. the set* I_M *and* $\mathtt{re}'$ *is a requirement labeling a leaf of* T*, then the tree* T' *obtained from* T *by replacing* $\mathtt{re}'$ *with the tree generated by applying a production step on* $\mathtt{re}'$ *w.r.t.* I_M*, is a derivation tree for* $\mathtt{re}$ *w.r.t. the set* I_M*.*

A maximal derivation tree $T_{\mathtt{re}}$ *for* $\mathtt{re}$ *w.r.t.* I_M *is a derivation tree where all leaves are labeled with an irreducible requirement and the set of requirements labeling* $T_{\mathtt{re}}$ *nodes is finite.*

It follows that the maximal derivation tree for a requirement $\mathtt{re}$ w.r.t. I_M contains all the requirements that can be derived from $\mathtt{re}$ w.r.t. I_M.

Definition 5.16 *Let* W *be an XML repository,* $\mathtt{p} \in W$*,* (I_N, I_M) *be a Web specification, and* $T_{\mathtt{p}}$ *be the maximal derivation tree for the initial requirement* $\langle \mathtt{p}, \_, \_ \rangle$ *w.r.t.* I_M*. Then the* set of completeness requirements $\mathtt{Req}_{\mathtt{M},\mathtt{p}}$*, for document* $\mathtt{p}$*, is the set of requirements labeling nodes in* $T_{\mathtt{p}}$*.*
The set of completeness requirements $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ *for* W *w.r.t.* I_M *is the set* $\bigcup_{\mathtt{p} \in W} \mathtt{Req}_{\mathtt{M},\mathtt{p}}$.

Since the derivable requirements from a requirement $\mathtt{re}$ w.r.t. I_M could be infinite, a maximal derivation tree for $\mathtt{re}$ might not exist. In Section 5.4, we propose some syntactical restrictions on Web specifications to ensure that the set of derivable requirements for a requirement $\mathtt{re}$ w.r.t. a set of completeness rules is finite and hence a maximal derivation tree for $\mathtt{re}$ exists. Moreover, we provide a way to obtain from a maximal derivation tree, an equivalent finite structure.

Diagnoses of completeness errors. For each completeness requirement, let us define a set TEST containing all documents of the considered XML repository, against which the requirement has to be verified.

Definition 5.17 *Let* W *be an XML repository,* (I_N, I_M) *be a Web specification, and* $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ *be the set of completeness requirements for* W *w.r.t.* I_M*. Let* $\equiv \langle \mathtt{e},\mathtt{q},\mathtt{ct} \rangle \in \mathtt{Req}_{\mathtt{M},\mathtt{W}}$*. The* test set *w.r.t.* $\langle \mathtt{e},\mathtt{q},\mathtt{ct} \rangle$*, is defined as*

$$\mathtt{TEST}_{\langle \mathtt{e},\mathtt{q},\mathtt{ct} \rangle} = \{\mathtt{p} \in W \mid \mathtt{ct} \textit{ not equal to } \_ \rightarrow \mathtt{ct} \trianglelefteq \mathtt{p}\}$$

Now we are able to define what does it mean for a completeness requirement to be not satisfied in an XML repository. We distinguish two cases: the former allows us to discover whether a universal requirement is not fulfilled by a given XML repository, while the latter recognizes unsatisfied existential requirements. In both cases, our analysis provides the missing/incomplete XML documents which are associated with those requirements.

Definition 5.18 (Requirement unsatisfiability) *Let W be an XML repository, (I_N, I_M) be a Web specification and* $\mathtt{re} \equiv \langle \mathtt{e}, \mathtt{q}, \mathtt{ct} \rangle$ *be a requirement.*

- *If* re *is a universal requirement, then* re *is* not satisfied *in W if one of the following conditions hold:*
  1. $\mathtt{TEST}_{\mathtt{re}} = \emptyset$*;*
  2. *there exists* $\mathtt{p} \in \mathtt{TEST}_{\mathtt{re}}$ *s.t.* $\mathtt{e} \not\trianglelefteq \mathtt{p}$.
- *If* re *is an existential requirement, then* re *is* not satisfied *in W if for each* $\mathtt{p} \in \mathtt{TEST}_{\mathtt{re}}$, $\mathtt{e} \not\trianglelefteq \mathtt{p}$.

Vice versa, a universal requirement re is satisfied whenever it is possible to recognize re inside any XML document of the corresponding test set $\mathtt{TEST}_{\mathtt{re}}$. Finally, an existential requirement re is fulfilled, if it is recognized inside (at least) an XML document which belongs to the test set $\mathtt{TEST}_{\mathtt{re}}$.

The requirements which are not fulfilled can be considered as *incompleteness symptoms*. This allows us not only to locate bugs and inconsistencies w.r.t. a given specification, but also to easily repair them by comparing incomplete documents to unsatisfied requirements, since the latter ones provide the missing information which is needed to complete the erroneous document.

To diagnose completeness errors in W, we can proceed as follows. For each $p \in W$, (i) we compute the maximal derivation tree T_p for $\langle \mathtt{p}, \_, \_ \rangle$ w.r.t. I_M, then (ii) we traverse T_p computing the test set for each requirement occurring in T_p, and we check whether the requirements are not satisfied in the sense of Definition 5.18. The verification process terminates delivering the detected completeness errors.

By using the previous definitions, we formalize the following completeness analysis.

Definition 5.19 (Maximal derivation tree analysis) *Let W be an XML repository, $p \in W$, (I_N, I_M) be a Web specification, and T_p be the maximal derivation tree of $\langle \mathtt{p}, \_, \_ \rangle$ w.r.t. I_M. The* completeness analysis *of a maximal derivation tree is inductively defined on the structure of T_p by the following function:*

$$\mathit{Verify}(\mathtt{or}(T_1, \ldots, T_k)) = \begin{cases} \mathtt{error}(\mathtt{e_1}, \ldots, \mathtt{e_k}) & \text{if } \forall i\ rt(T_i) \text{ is not satisfied in } W \\ \mathit{Verify}(T_i)\ \forall i \text{ s.t. } rt(T_i) \text{ is satisfied in } W & \text{otherwise} \end{cases}$$

where $rt(T_i) = \langle \mathtt{e_i}, \mathtt{q_i}, \mathtt{ct_i} \rangle$.

$$\mathit{Verify}(\mathtt{re}(T_1, \ldots, T_k)) = \mathit{Verify}(T_i)\ \forall i = 1, \ldots, k$$

The term $\mathtt{error}(e_1, \ldots, e_k)$ represents a completeness error which means that no disjunct e_i for $i \in \{1, \ldots, k\}$ is recognized into the considered document. Roughly speaking, the execution of $\mathit{Verify}(T_p)$ finds all the completeness errors inside an XML document p w.r.t. I_M. By applying this verification method to all XML documents in W, we can check the completeness of the whole repository W.

5.4 Web Specification Restrictions

In this section, we present some restrictions over Web specifications without meta-symbols, which allow us to ensure that, given a requirement $\mathtt{re}$ and a set of completeness rules I_M, a maximal derivation tree for $\mathtt{re}$ exists.

Let us introduce a more general definition of simulation than the one presented in previous Section.

Definition 5.20 (Simulation*) *The* simulation* *relation* $\trianglelefteq^* \subset \mathcal{T}(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V}) \times \mathcal{T}(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ *in Web templates is the least relation satisfying the following rules:*

- $X \trianglelefteq^* t$, *for each* $X \in \mathcal{V}$, $t \in \mathcal{T}(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$;
- $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_m) \trianglelefteq \mathtt{g}(\mathtt{s}_1, \ldots \mathtt{s}_n)$ *iff* $\mathtt{f} \equiv \mathtt{g}$ *and* $\mathtt{t}_i \trianglelefteq \mathtt{s}_{\pi(i)}$, *for* $\mathtt{i} = 1, \ldots \mathtt{m}$, *and some injective function* $\pi : \{1, \ldots \mathtt{m}\} \to \{1, \ldots, \mathtt{n}\}$.

Definition 5.21 *Let* $\mathtt{s}_1, \mathtt{s}_2 \in \mathcal{T}(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. *We say that* $\mathtt{s}_2$ *partially matches* $\mathtt{s}_1$ *via substitution* σ *iff* $\mathtt{s}_1\sigma \trianglelefteq^* \mathtt{s}_2$.

Definition 5.22 (Rule activation) *Let* $r_1 \equiv \bigwedge_{i=1}^{n_1} \mathtt{l}_i^1 \rightharpoonup \bigvee_{j=1}^{m_1} \mathtt{r}_j^1 \mid \mathtt{C}_1\,\langle \mathtt{q} \rangle\,(\mathtt{ct}_1)$ *and* $r_2 \equiv \bigwedge_{i=1}^{n_2} \mathtt{l}_i^2 \rightharpoonup \bigvee_{j=1}^{m_2} \mathtt{r}_j^2 \mid \mathtt{C}_2\langle \mathtt{q} \rangle(\mathtt{ct}_2)$ *two completeness rules. We say that* $\mathtt{r}_1$ activates $\mathtt{r}_2$, *if* $\exists j_s \in \{1, \ldots, m_1\}$ *and substitution* σ *s.t. for all* $i \in \{1, \ldots, n_2\}$, $\mathtt{l}_i^2$ *partially matches* $\mathtt{r}_{j_s}^1$ *via* σ.

Roughly speaking, the activation between rules r_1 and r_2 means that a partial rewrite step from a term t using rule r_1, generates another term that enables a partial rewrite step using rule r_2.

Definition 5.23 (Bounded substitution) *Let* r_1, r_2 *be two completeness rules such that* r_1 *activates* r_2 *via substitution* σ. *Then,* σ *is called* bounded substitution *if for each variable* x, *either* $x\sigma \in \mathcal{V}$ *or* $x\sigma \in \mathcal{T}_\Sigma$.

Bounded substitutions ensure that a variable cannot be substituted with a term containing a variable at a position greater than Λ.

Definition 5.24 ((Non-)Bounded rule activation) *Let* r_1, r_2 *two completeness rules such that* r_1 *activates* r_2 *via substitution* σ. *If* σ *is a bounded substitution we say that* r_1 *bounded activates* r_2, *otherwise we say that* r_1 *non-bounded activates* r_2.

Definition 5.25 (Bounded activated rules) *Let* I *be a set of completeness rules, we say that* I *is a set of bounded activated rules if, for each* r_i , r_j *in* I, *if* r_i *activates* r_j *then this is a bounded activation.*

Definition 5.26 (Activation graph) *Let* (I_N, I_M) *be a Web specification, the* activation graph *of* (I_N, I_M) *w.r.t. the completeness rules in* I_M *is a direct labeled graph* (V, E) *where*

(i.) if $r \in I_M$ *then* $r \in V$ *is a node of the graph;*

(ii.) if $r_1, r_2 \in I_M$ *and* r_1 *bounded activates* r_2*, then* $(r_1 \mapsto r_2) \in E$*;*

(iii.) if $r_1, r_2 \in I_M$ *and* r_1 *non-bounded activates* r_2*, then* $(r_1 \overset{nb}{\mapsto} r_2) \in E$*.*

Roughly speaking the activation graph of a Web specification describes the dependencies among completeness rules. The circularity of such dependencies is critical for the computation of the maximal derivation trees.

Definition 5.27 (Bounded Specification) *A Web specification is* bounded *if its activation graph either does not contain cycles or no edge in a cycle (also a self-loop) is labeled with nb.*

Proposition 5.37 relates bounded Web specifications with the existence of a finite structure equivalent to maximal derivation trees, which is introduced by Definition 5.36. In order to prove Proposition 5.37, we need the following auxiliary definitions and results.

Definition 5.28 *Given a requirement* $\mathtt{re} \equiv \langle \mathtt{e}, \mathtt{q}, \mathtt{ct} \rangle$*, the* height *of the term* $\mathtt{e}$*,* $height(\mathtt{e})$*, is defined as follows.*

$$height(\mathtt{e}) = \begin{cases} 0 & \text{if } \mathtt{e} \equiv X \in \mathcal{V} \text{ or } \mathtt{e} \equiv c \in \mathcal{T}_\Sigma \\ 1 + \max\{height(t_i) \mid i = 1, \ldots, n\} & \text{if } \mathtt{e} \equiv f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma(\mathcal{V}) \end{cases}$$

From now on we use $height(\mathtt{re})$ *in place of* $height(\mathtt{e})$*.*

We can lift the notion of height to substitutions in the following way.

Definition 5.29 *Given a substitution* $\sigma \equiv \{X_1/t_1, \ldots, X_n/t_n\}$*, the* height *of* σ*,* $height(\sigma)$*, is defined as follows.*

$$height(\sigma) = \max\{height(t_i) \mid i = 1, \ldots, n\}$$

Definition 5.30 (Rewriting chains) *Let* I *be a set of completeness rules, a* rewriting chain *is a partial rewrite sequence*

$$\mathtt{t}_0 \rightharpoonup_{r_0}^{\sigma_0} \mathtt{t}_1 \rightharpoonup_{r_1}^{\sigma_1} \mathtt{t}_2 \rightharpoonup_{r_2}^{\sigma_2} \ldots$$

where $r_j \in I$*,* $j = 0, 1, 2, \ldots$ *and for each* $\mathtt{t}_i$*,* $i = 1, 2, \ldots$*,*

$$\mathtt{t}_i \equiv \mathtt{r}_{i-1}\sigma_{i-1}$$

where $\mathtt{r}_{i-1}$ *is the right-hand side of the rule* r_{i-1}*.*

Proposition 5.31 *Let* I *be a set of completeness rules and*

$$\mathtt{t}_0 \rightharpoonup_{r_0}^{\sigma_0} \mathtt{t}_1 \rightharpoonup_{r_1}^{\sigma_1} \mathtt{t}_2 \rightharpoonup_{r_2}^{\sigma_2} \ldots$$

a rewriting chain, where $r_j \in I$*,* $j = 0, 1, 2, \ldots$ *Then, for each* $\mathtt{t}_i$*,* $i = 1, 2, \ldots$*,*

$$height(\mathtt{t}_i) \leq height(\mathtt{r}_{i-1}) + height(\sigma_{i-1})$$

where $\mathtt{r}_{i-1}$ *is the right-hand side of the rule* r_{i-1}*.*

Proof 5.4.1 *It is a direct consequence of Definition 5.28, Definition 5.29 and Definition 5.30.*

Proposition 5.32 *Let I be a set of bounded activated completeness rules and*

$$\mathtt{t}_0 \rightharpoonup_{r_0}^{\sigma_0} \mathtt{t}_1 \rightharpoonup_{r_1}^{\sigma_1} \mathtt{t}_2 \rightharpoonup_{r_2}^{\sigma_2} \dots$$

a rewriting chain, where $r_j \in I$, $j = 0, 1, 2, \dots$ Then, for each $\mathtt{t}_i$, $i = 1, 2, \dots$,

$$height(\sigma_i) \leq height(\sigma_{i-1})$$

Proof 5.4.2 *Let us focus on the partial rewrite steps*

$$\mathtt{t}_{i-1} \rightharpoonup_{r_{i-1}}^{\sigma_{i-1}} \mathtt{t}_i, \mathtt{t}_i \rightharpoonup_{r_i}^{\sigma_i} \mathtt{t}_{i+1}$$

for a general $i > 0$. By Definition 5.30, $\mathtt{t}_i \equiv \mathtt{r}_{i-1}\sigma_{i-1}$ so that, the partial rewrite step $\mathtt{t}_i \rightharpoonup_{r_i}^{\sigma_i} \mathtt{t}_{i+1}$ can be given on a vertex in $\mathtt{r}_{i-1}$ or on a vertex belonging to a term in $\{t \mid X/t \in \sigma_{i-1}\}$. In the former case, since the set of rules is bounded activated, we must have $height(\sigma_i) = height(\sigma_{i-1})$, in the latter, trivially $height(\sigma_i) \leq height(\sigma_{i-1})$. So finally we have $height(\sigma_i) \leq height(\sigma_{i-1})$.

Lemma 5.33 *Let I be a set of bounded activated completeness rules and*

$$\mathtt{t}_0 \rightharpoonup_{r_0}^{\sigma_0} \mathtt{t}_1 \rightharpoonup_{r_1}^{\sigma_1} \mathtt{t}_2 \rightharpoonup_{r_2}^{\sigma_2} \dots$$

a rewriting chain, where $r_j \in I$, $j = 0, 1, 2, \dots$ Then, for each $\mathtt{t}_i$, $i = 1, 2, \dots$,

$$height(\mathtt{t}_i) \leq height(\mathtt{r}_{i-1}) + height(\sigma_0)$$

where $\mathtt{r}_{i-1}$ is the right-hand side of the rule r_{i-1}.

Proof 5.4.3 *Consider the generic partial rewrite step*

$$\mathtt{t}_{i-1} \rightharpoonup_{r_{i-1}}^{\sigma_{i-1}} \mathtt{t}_i.$$

By Definition 5.30 we have that $\mathtt{t}_i \equiv \mathtt{r}_{i-1}\sigma_{i-1}$ and by Proposition 5.31

$$height(\mathtt{t}_i) \leq height(\mathtt{r}_{i-1}) + height(\sigma_{i-1}).$$

Moreover, by Proposition 5.32 and transitivity, $height(\sigma_{i-1}) \leq height(\sigma_0)$.
Therefore,

$$\begin{aligned} height(\mathtt{t}_i) &\leq height(\mathtt{r}_{i-1}) + height(\sigma_{i-1}) \\ &\leq height(\mathtt{r}_{i-1}) + height(\sigma_0) \end{aligned}$$

and the claim is proved.

Proposition 5.34 *Let I be a set of bounded activated completeness rules. Then, for each requirement $\mathtt{re}$, the set of requirements derivable from $\mathtt{re}$ w.r.t. I is finite.*

Proof 5.4.4 *Let* $H_{\mathtt{re}} = height(\mathtt{re})$ *and* $Req_{\mathtt{re}}$ *be the set of all requirements derivable from* re *w.r.t. the set* I*. First, let us show that each requirement in* $Req_{\mathtt{re}}$ *has a bounded height. Let* $\mathtt{re}' \in Req_{\mathtt{re}}$*. If* $\mathtt{re}' \equiv \mathtt{re}$ *then* $height(\mathtt{re}') = H_{\mathtt{re}}$*. Otherwise, there is a finite rewriting chain starting from* re *leading to* $\mathtt{re}'$*. Let*

$$H_I = \max\{height(\mathtt{r}) \mid \mathtt{r} \text{ is a right hand side of a rule in } I\}$$
$$H_S = \max\{height(\sigma') \mid \mathtt{re} \rightharpoonup^{\sigma'}_{r'} \mathtt{t}, r' \in I\}.$$

By Lemma 5.33, there exists r *right hand side of a rule in* I *such that* $height(\mathtt{re}') \leq height(\mathtt{r}) + height(\sigma_0) \leq H_I + H_S$*.*

Thus, for each $\mathtt{re}' \in Req_{\mathtt{re}}$*, we get* $height(\mathtt{re}') \leq \max\{H_{\mathtt{re}}, H_I + H_S\}$*. Since we have only a finite number of terms whose height is less than or equal to* $\max\{H_{\mathtt{re}}, H_I + H_S\}$*,* $Req_{\mathtt{re}}$ *must be finite.*

Now, we are ready to prove Proposition 5.35.

Proposition 5.35 *Let* (I_N, I_M) *be a bounded Web specification. Then, for each requirement* re*, the set of requirements derivable from* re *w.r.t.* I_M *is finite.*

Proof 5.4.5 *Let* $Req_{\mathtt{re}}$ *be the set of all derivable requirements from* re *w.r.t.* I_M*. If a Web specification* (I_N, I_M) *is bounded there are two cases to be considered.*

- *The activation graph of* (I_N, I_M) *does not contain cycles, so* (I_N, I_M) *is not recursive. It is trivial to see that the set of derivable requirements using a non-recursive set of rules is finite.*
- *The activation graph of* (I_N, I_M) *contains cycles but no edge in such cycles is labeled with nb (i.e. no edge is associated with a non-bound rule activation). Note that, an infinite set of requirements could be generated only by such cycles. Let* C *be a generic cycle in the activation graph and* $I_M|_C$ *the set of completeness rules labeling nodes in* C*. Note that,* $I_M|_C$ *is a set of bounded activated rules. By Proposition 5.34, for all requirement* re*, the set of requirement that can be derived from* re *by rules in* $I_M|_C$ *is finite. It follows immediately that also* $Req_{\mathtt{re}}$ *is finite.*

Let now introduce a graph structure which is a simple variant of a derivation tree. For the sake of simplicity we call *leaf* a graph node with outer degree equals to zero.

Definition 5.36 (Derivation graph) *Given a requirement* re *and a Web specification* (I_N, I_M)*, a derivation graph for* re *w.r.t. the set* I_M*, is defined as follows:*

- re *is a derivation graph for* re *w.r.t. the set* I_M*;*
- *let* G *be a derivation graph for* re *w.r.t. the set* I_M *and* $\mathtt{re}'$ *a requirement labeling a leaf of* G*. Assume that, a production step from* $\mathtt{re}'$ *w.r.t.* I_M *generates a tree* $\mathtt{re}'(T_1, \ldots, T_k)$ *such that* $\{T_1, \ldots, T_i\}$ *appears already in* G *and* $\{T_{i+1}, \ldots, T_k\}$ *do not appear anywhere in* G*, for some* $i \in \{0, \ldots, k\}$*. We can obtain a graph* G' *from* G *by:*

- *replacing* $\mathtt{re}'$ *with the tree* $\mathtt{re}'(T_{i+1}, \ldots, T_k)$*;*
- *for each* $j \in \{1, \ldots, i\}$*, if* T_j *appears in* G *at position* u_j*, adding a back edge from* $\mathtt{re}'$ *to the node at position* u_j*.*

Then the graph G' *is a derivation graph for* $\mathtt{re}$ *w.r.t.* I_M*.*

A finite derivation graph $G_{\mathtt{re}}$ *for* $\mathtt{re}$ *w.r.t.* I_M *is a derivation graph where all leaves are labeled with an irreducible requirement.*

Note that, a finite derivation graph is also a maximal derivation graph, since all leaves are labeled with irreducible requirements. Roughly speaking, a finite derivation graph is a concise form of a maximal derivation tree since a maximal derivation tree can contain infinite nodes labeled with the same requirement, while a finite derivation graph does not contain two nodes labeled with the same requirement since uses back edges to avoid such repetitions. Hence, a maximal derivation tree and the related finite derivation graph share the same information.

Proposition 5.37 relates bounded Web specifications with the existence of finite derivation graphs.

Proposition 5.37 *Let* (I_N, I_M) *be a bounded Web specification. Then, for each requirement* $\mathtt{re}$ *there exists a finite derivation graph containing all the requirements that can be derived from* $\mathtt{re}$ *w.r.t. rules in* I_M*.*

Proof 5.4.6 *Let* $Req_{\mathtt{re}}$ *be the set of all derivable requirements from* $\mathtt{re}$ *w.r.t.* I_M*. Since a derivation graph contains only one node for each requirement derivable from* $\mathtt{re}$*, the graph could be infinite if and only if* $Req_{\mathtt{re}}$ *is infinite. Since the Web specification is bounded, by Proposition 5.35,* $Req_{\mathtt{re}}$ *is finite, so, the derivation graph for* $\mathtt{re}$ *is finite.*

Given a bounded Web specification (I_N, I_M) and an XML document $\mathtt{p}$, we can consider the null requirement $\mathtt{re}_{\mathtt{p}} = \langle \mathtt{p}, \_, \_ \rangle$ and build the finite derivation graph $G_{\mathtt{p}}$ for $\mathtt{re}_p$ w.r.t. I_M. Then, deleting form $G_{\mathtt{p}}$ all the back edges we obtain a tree T containing all the requirements derivable from $\mathtt{re}_{\mathtt{p}}$ and finally $Verify(T)$ will find all the completeness errors inside $\mathtt{p}$ w.r.t. I_M.

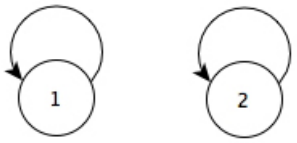
For the sake of clarity let us show an example. Consider a Web specification whose rules are the extended rules of Example 5.5 and the XML document $\mathtt{p}$ of Figure 5.2. The activation graph is very simple, it consists of two nodes, one for each rule, each of which has a self-loop since the left hand side of each rule is partially matched inside its right hand side. Since both the activations are bounded, the Web specification is bounded. The activation graph is shown in Figure 5.3. The maximal derivation tree that can be constructed starting from the requirement $\mathtt{re}_{\mathtt{p}} = \langle \mathtt{p}, \_, \_ \rangle$ is infinite and a small fragment is shown in Figure 5.5. The corresponding finite derivation graph is shown in Figure 5.4 where the labeling function is the same as the one in Figure 5.5. By applying function $Verify$ over the tree obtained from the derivation graph of Figure 5.4 by deleting all its back edges, the following errors are thrown:

```
(AssociateProf(name(Mary),email()) ∨ AssociateProf(name(Mary),address())),
(AssociateProf(name(Luck),email()) ∨ AssociateProf(name(Luck),address())).
```

Summing up, by using bounded specifications, the completeness verification methodology always terminates. Moreover, the information obtained by the completeness analysis can be exploited to repair the considered Web contents by comparing incomplete documents to unsatisfied requirements, since the latter ones provide the missing information which is needed to complete the erroneous document.

```
<members>
  <member>
    <ContractProf>
      <name>Joe</name>
      <email>joe@univ.edu</email>
      <teaching>...</teaching>
    </ContractProf>
  </member>
  <member>
    <AssociateProf>
      <name>Mary</name>
      <phone>1234 5678</phone>
      <courses>...</courses>
    </AssociateProf>
  </member>
  <member>
    <FullProf>
      <name>Paul</name>
      <address>university St. 3, UniTown</address>
      <assistants>
        <AssistantProf>...</AssistantProf>
        <AssociateProf>
          <name>Luck</name>
        </AssociateProf>
      </assistants>
    </FullProf>
  </member>
</members>
```

Figure 5.2: XML document about academic professors.




```
1:  AssociateProf(name(X)) ⇀ AssociateProf(name(X),email()) ∨
                             AssociateProf(name(X),address()) |
                                          containing member() <A>
2:  FullProf(name(X)) ⇀ FullProf(name(X),email()) ∨
                        FullProf(name(X),address()) |
                                          containing member() <A>
```

Figure 5.3: Activation graph for the Web specification.

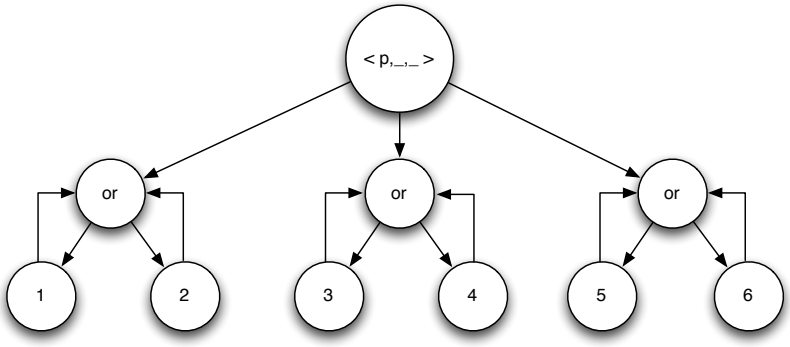
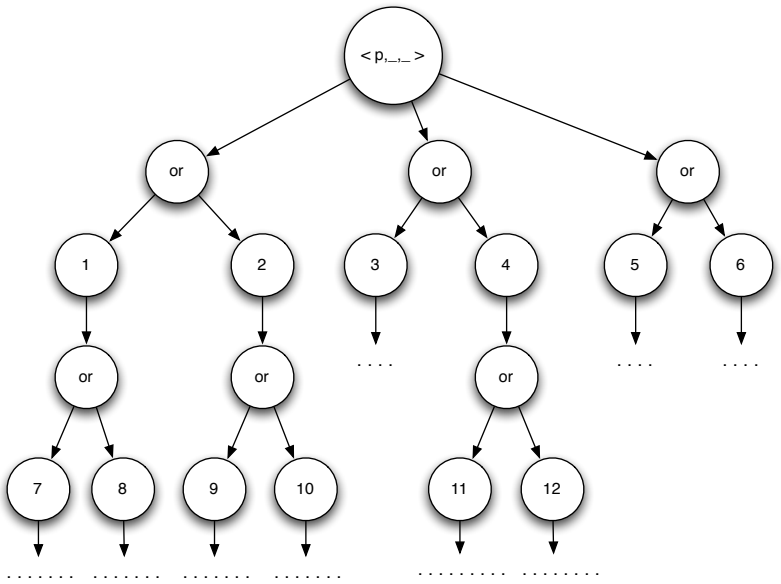


Figure 5.4: Finite derivation graph for the requirement $\mathtt{re}_p$.



1: ⟨ AssociateProf(name(Mary)),email()), A, member() ⟩
2: ⟨ AssociateProf(name(Mary)),address()), A, member() ⟩
3: ⟨ FullProf(name(Paul)),email()), A, member() ⟩
4: ⟨ FullProf(name(Paul)),address()), A, member() ⟩
5: ⟨ AssociateProf(name(Luck)),email()), A, member() ⟩
6: ⟨ AssociateProf(name(Luck)),address()), A, member() ⟩
7: ⟨ AssociateProf(name(Mary)),email()), A, member() ⟩
8: ⟨ AssociateProf(name(Mary)),address()), A, member() ⟩
9: ⟨ AssociateProf(name(Mary)),email()), A, member() ⟩
10: ⟨ AssociateProf(name(Mary)),address()), A, member() ⟩
11: ⟨ FullProf(name(Paul)),email()), A, member() ⟩
12: ⟨ FullProf(name(Paul)),address()), A, member() ⟩

Figure 5.5: Fragment of the maximal derivation tree for the requirement $\mathtt{re}_p$.

6

Biological Systems Modeling and Analysis

Biological systems are complex entities whose characteristic aspects derive from intricate interaction schemes among components and whose behavior is thus not a simple and direct consequence of that of their components. Studying and describing systems with these characteristics require to overcome a reductionist attitude in favor of a systemic approach that is the basis of Systems Biology, where Computer Science plays a primary role.

There are many different computational models of biological processes, depending on the aspects we want to focus on. We can identify two main categories of computational models: mathematical models based on differential equations to model kinetic aspects; and symbolic/logical formalisms to module structure, information flow and processes properties.

Models of process kinetics employ quantitative relations to express the interactions between different parts of the system. Differential equations represent the processes by using experimentally derived or inferred information about concentrations and rates of change of molecules. Since these models require a great amount of detailed quantitative information, meeting the great difficulty of obtaining them, many stochastic variants has been proposed. Analysis of these models by numerical and probabilistic simulation techniques can then be performed.

The advantage of symbolic/logical models over mathematical ones is that they allow us to represent complex biological systems in abstract terms and at different levels of detail, depending on the information available and the questions to be studied. This formalisms provide the means to represent system states and state changes, and analysis tools that are based on computational or logical inference. Symbolic models can be executed in order to simulate the system behavior and properties can be stated by using the associated logical languages and checked by using tools for formal analysis. A variety of formalisms initially developed to model and analyze concurrent computer systems have been employed to develop symbolic models of biological systems, such as: Petri nets [126], the pi-calculus [123] and its stochastic variants [137], membrane calculi[48], statecharts [95], rule-based systems including Rewriting Logic [117] and P-systems [133], and hybrid systems [97].

Among the logical formalisms, Pathway Logic [149, 82] (PL) is a symbolic approach to the modeling and analysis of qualitative aspects of biological processes that is based on rewriting logic [117]. The process of application of rewrite rules, from a given initial state, generates computations. In the case of biological processes, these correspond to pathways. More specifically, a PL model includes the representation of cellular components (proteins, enzymes, etc.), their locations and their state. It also includes representations, as rewrite rules, of basic process steps such as metabolic reactions or intra- and inter- cellular signaling. Execution of the rules allows one to represent and reason about the system dynamics. PL models can be transformed into equivalent models based on Petri nets [150]; this allows one to adopt alternative representations with different expressive capabilities for managing complexity and analyzing properties of the biological processes under examination.
A recent extension of the Pathway Logic [1] has been proposed to represent and reason about semiquantitative and probabilistic aspects of biological processes. Basically, this approach annotates reaction rules with affinity information that can be used to implement distinct simulation strategies which can also include timing information. Although this approach improves expressiveness of standard PL, it only handles a semi-quantitative modeling of biological processes which does not allow us to define complex reaction rates.

Although Pathway Logic may be very useful to model biological processes and provides a simple way to express the system dynamics, it has some important limits: (a) PL only supports qualitative modeling of the biological events of interests. As a matter of fact, it provides no explicit way to add quantitative information to the models such as element concentrations in cell locations, levels of production as well as consumption of elements occurring in a reaction, reaction thresholds, *etc.* (b) PL does not provide adequate capabilities to express inhibitory actions occurring in biological reactions, which are very common e.g. in regulatory networks. Basically, an element acts as an inhibitor in a reaction if its concentration over a given threshold decreases the rate of, or prevents, the reaction.

The possibility to specify qualitative relationships amongst the elements composing the system of interest is particularly useful because the biological data that express quantitative relationships (e.g. kinetic constants or stochastic parameters) are often hard to find or measure due to problems related to wet lab techniques. Our approach grounds on the possibility of choosing amongst various kinds of formalisms that differ in their expression power: expressive formalisms will be used when the available information allows us to effectively exploit their characteristics, whereas less expressive (and less computationally expensive) formalisms will be preferred when the amount of knowledge is limited. In this chapter we provide an extension of Pathway Logic called *Quantitative Pathway Logic* (QPL for short) with the aim of overcoming the mentioned PL limits and obtaining a more precise modeling approach while keeping the possibility to compute with and analyze these complex systems. More specifically, QPL efficiently integrates quantitative data (such as element concentrations, reaction thresholds, production and consumption rates) into PL models. Besides, it allows one to model reaction inhibitors.

To manage the different aspects of biological systems, we equipped QPL speci-

fications with two equivalent computational models following and adapting the PL approach of [150]. This allows one to adopt different representations with different expressive capabilities for handling the complexity of the systems under examination. On the one hand, QPL specifications can be directly formalized and executed by using the Maude rewriting logic formalism. In this way, both model simulation and model search can take advantage of quantitative information to yield more accurate results. On the other hand, QPL models can be translated into an extension of the classical Petri nets called Discrete Functional Petri Nets (DFPN), which are basically Hybrid Functional Petri Nets (HFPN) [129] in which only discrete transitions are authorized. By using this representation our models can be graphically visualized, simulated, and analyzed by means of well known tools (e.g. Cell Illustrator [128]).

6.1 Quantitative Pathway Logic

A QPL model is a rewrite theory $\mathcal{Q} = (\Sigma, E, R)$ which models biological systems. A QPL model is naturally divided in two parts: the *equational part* and the *rule part*. The former allows one to represent the cellular states, while the latter specifies the system dynamics.

The Equational Part. This part corresponds to the equational theory (Σ, E) of the QPL model $\mathcal{Q}$. It provides sorts and operators useful to model molecular components and more in general all the entities which are involved in a biological system.

As in the standard PL framework, the main sorts for entities include `Chemical`, `Protein`, `Complex`, which are all subsorts of sort `Thing` that specifies a generic entity. Cellular compartments are identified by sort `Location`, which provides location names to each compartment, while `Modification` is a sort used to classify Post-transactional protein modifications, which are defined by the operator `[ - ]` (e.g. the term `[EgfR - act]` represents the epidermal growth factor Egf receptor in an activated state).

Besides that, we provide a special sort `QThing` which is represented by the pair $(\texttt{Thing}, \mathbb{R}^+)$, where $\mathbb{R}^+$ specifies the sort for the non-negative real numbers. The sort `QThing` is employed to manage entity concentrations. For instance, `(Erk, 3.3)` might model the fact that the concentration of the the Mitogen-Activated Protein Kinase Erk is 3.3 units. We call *occurrence* to any term of sort `QThing`. A *soup* is a set of occurrences and cellular compartments that is identified by type `Soup`.

Now, a *cell state* is represented by a term of the form `[cellType | locs]`, where `cellType` specifies the type of cell and `locs` represents the contents of a cell organized by cellular compartments (or locations). Each location is modeled by a term of the form { `locName | comp` }, where `locName` is a name that identifies the location (e.g. `CLm` may represent the cell membrane location), and `comp` is a soup in that location.

Figure 6.1 shows a fragment of the Maude functional module that implements the sorts and operations we presented so far.

Note that the Soup constructor `__` is given the equational attributes `comm assoc id:empty`, which allow us to get rid of parentheses and disregard the ordering among the different elements within the list. When the equational part contains axioms for associativity and commutativity of operators, we talk about *AC pattern matching*.

```
fmod CELL is
-- sort declaration
sorts Protein Family Composite DNA Chemical Signature Stimulus Thing
         QThing Location LocName Cell CellType Soup .
subsorts Protein Family Composite DNA Chemical Signature < Thing .
subsorts Thing QThing Location Cell < Soup .

-- Occurrence item: pair of a Thing (i.e. an entity) and its quantity
op (_,_) : Thing Float -> QThing [ctor] .

-- Generic mixture of entity occurrences
op empty : -> Soup .
op __ : Soup Soup -> Soup [assoc comm id: empty ] .

-- Locations in the cell
ops CLo CLm CLi CLc ... : -> LocName .
op {_|_} : LocName Soup -> Location .
ops Cell MuscleCell Fibroblast ... : -> CellType .

-- Cell object type
op [_|_] : CellType Soup -> Cell .
endfm
```

Figure 6.1: Fragment of Maude code that represents cell states.

AC pattern matching is a powerful matching mechanism, which we employ to inspect and extract the partial structure of a term. In particular, we take advantage of it when describing the system dynamics by means of rewrite rules where we want to specify only a partial content of cells.

The Rules part. Given a QPL model $\mathcal{Q} = (\Sigma, E, R)$, the rule part is specified via the set of rewrite rules R, which contains rewrite rules that formalize individual reaction steps. In the case of signal transduction, rewrite rules represent processes such as activation, phosphorylation, complex formation, or translocation. Basically, as in PL, QPL rewrite rules transform a cell state into another one via pattern matching modulo an equational theory. Moreover, this transformation in QPL can take advantage of the quantitative information associated with the entities into play. In this context, it is very easy to define promoters (entities that enables a reaction when their concentration is over a certain threshold), inhibitors (entities that blocks a reaction when their concentration is over a certain threshold), tests (entities not consumed by a reaction), and reaction rates modeled via consumption and production functions. Let us illustrate the modeling capabilities by means of some examples.

Example 6.1 (Promoters and tests) *Consider a reaction modeled by the following rewrite rule.*

$$\begin{aligned}\texttt{ex1}:&\{\texttt{CLi} \mid \texttt{cli}\ (\texttt{A}, \texttt{a})\}\{\texttt{CLm} \mid \texttt{clm}\ ([\texttt{B-GDP}], \texttt{b})(\texttt{D}, \texttt{d})\} \Rightarrow \\ &\{\texttt{CLi} \mid \texttt{cli}\ (\texttt{A}, \texttt{a}/2.0)(\texttt{C}, \texttt{a}/2.0 + \texttt{b})\} \\ &\{\texttt{CLm} \mid \texttt{clm}\ (\texttt{D}, \texttt{d})\}\ \ \texttt{if}\ \ \texttt{a} >= 3.5.\end{aligned}$$

The rule states that, if we detect a cell state in which (1) an entity `A` *with concentration* `a` *is inside the cell membrane (location* `CLi`*); and (2) entities* `[B-GDP]` *(i.e. entity* `B` *bounded to a molecule of Guanosine diphosphate) and* `D` *with concentrations* `b` *and* `d` *are on the border of the cell membrane (location* `CLm`*), then* `A` *promotes the reaction whenever its concentration is greater than or equal* 3.5 *units and a new cell state is generated in which Reactants* `A` *and* `[B-GDP]` *are consumed: the former is consumed according to the consumption function* (`a`/2.0) *and the latter is completely consumed. Entity* `C` *is produced inside the membrane according to the production function* `a`/2.0+`b`*, while* `D` *is a test whose concentration is left unchanged (its presence is necessary for the reaction to take place but there is no need to consume it).*

Note that production functions may depend on concentrations of several reactants. Instead, we assume that the consumption function of a reactant `A` must depend only on the concentration of `A`. Inhibitory behaviors are easily modeled by means of conditional rules. For this purpose, we first define the auxiliary function $\texttt{checkInhibitors}(\texttt{s}, (\texttt{i}_1, \texttt{t}_1) \ldots (\texttt{i}_\texttt{n}, \texttt{t}_\texttt{n}))$ which takes as input a soup `s` and a list of occurrences $(\texttt{i}_\texttt{j}, \texttt{t}_\texttt{j})$, and it returns `true` if there does not exist any occurrence $(\texttt{i}_\texttt{j}, \texttt{c}_\texttt{j})$ in `s` such that $\texttt{c}_\texttt{j} \geq \texttt{t}_\texttt{j}$. Now, an inhibitor $\texttt{i}_\texttt{j}$ is an entity located in some compartment `s` which prevents a reaction to take place whenever its concentration $\texttt{c}_\texttt{j}$ is over a certain threshold $\texttt{t}_\texttt{j}$. This amounts to saying that a set of inhibitors $(\texttt{i}_1, \ldots, \texttt{i}_\texttt{n})$ can block a reaction iff $\texttt{checkInhibitors}(\texttt{s}, (\texttt{i}_1, \texttt{t}_1) \ldots (\texttt{i}_\texttt{n}, \texttt{t}_\texttt{n})) = \texttt{false}$. Therefore,

we can model a reaction containing inhibitors $\mathtt{i}_1$ and $\mathtt{i}_2$ by a rewrite rule of the form $\mathtt{t} \Rightarrow \mathtt{t}'$ `if checkInhibitors(s,`$(\mathtt{i}_1, \mathtt{t}_1)(\mathtt{i}_2, \mathtt{t}_2)$`)`.

Example 6.2 (Inhibitors) *Consider a reaction modeled by the following rewrite rule.*

$$\mathtt{ex2} : \{\mathtt{CLi} \mid \mathtt{cli}\,(\mathtt{A}, \mathtt{a})\} \Rightarrow \{\mathtt{CLi} \mid \mathtt{cli}\,(\mathtt{C}, \mathtt{a})\} \textit{ if } \mathtt{checkInhibitors}\,(\mathtt{cli}, (\mathtt{B}, 4.0))$$

The rule states that, when there exists a cell state in which an entity `A` *with concentration* `a` *appears inside the cell membrane (location* `CLi`*), then the cell state is transformed by consuming the reactant* `A` *completely and produces* `a` *units of entity* `C` *provided that there is no inhibitor* `B` *with concentration greater than* 4.0 *inside the cell membrane (i.e. in the location* `CLi`*). Note that* B *is not consumed by the reaction.*

6.1.1 Simulation and analysis of QPL models

Quantitative Pathway Logic models are rewrite theories, hence *executable* specifications, since they describe system states and provide rules that specify the way in which states may change. In other words, we can directly exploit the Maude rewrite engine to run our models. In particular, Maude supports the *forward simulation* which is the first kind of analysis that can be carried out given such an executable specification. It consists in running the model from a given initial state for a fixed number of steps or until a steady state has been reached. It is very useful for initial exploration of the transition graph but not suitable for understanding the dynamics of systems having infinite behaviors. Maude is also equipped with *forward search* facilities, which allow one to perform a breath-first search of all rewrite paths generated for a given initial state. If the specification is finite, the forward search will find all possible outcomes from a given initial state. Moreover, the search can be constrained to find only states satisfying a given property or until a fixed number of rewrite steps have been performed.

In order to execute both forward simulation and forward search, we need to provide an initial state. Initial states (called *dishes*) are encoded in our rewriting setting by means of terms of the form `PD(out cellstate)`, where `cellstate` represents a cell state and `out` specifies a soup of ligands and other molecular components in the cells surroundings which may interact with the cell.

Example 6.3 *Consider the following dish*

$$\begin{aligned}\mathtt{PD}(\{(\mathtt{Egf}, 2.0)\}[\mathtt{HMEC}|&\{\mathtt{CLo}|\mathtt{empty}\}\{\mathtt{CLm}|(\mathtt{EgfR}, 1.0)(\mathtt{PIP2}, 3.0)\}\\ &\{\mathtt{CLi}|([\mathtt{Hras} - \mathtt{GDP}], 4.0)(\mathtt{Src}, 5.0)\}\\ &\{\mathtt{CLc}|(\mathtt{Gab1}, 5.0)(\mathtt{Grb2}, 6.0)(\mathtt{Pi3k}, 2.0)(\mathtt{Plcg}, 7.0)(\mathtt{Sos1}, 1.0)\}]).\end{aligned}$$

The dish above contains the cell state of a cell of type `HMEC` *that is made up of three locations* `CLo`*,* `CLi` *and* `CLc`*. Each location contains a soup of occurrences. Moreover, the ligand* `Egf` *is present in the cell surroundings with concentration* `2.0`*.*

Now, given an initial state and a QPL model, we can analyze the behavior of the system by means of Maude forward simulation and search capabilities. Let us illustrate it by means of some examples.

Example 6.4 *Assuming that a QPL model for the signal transduction network of the epidermal growth factor receptor (`EgfR`) is given, we may run the model starting from an initial cell state by means of the Maude `rewrite` command (abbreviated `rew`), that is, we may explore the behavior of the specified system for different initial cell states. For instance, the Maude query*

```
rew [100] PD((Egf,2.0) [HMEC | {CLm | (EgfR,1.0) (PIP2,3.0)}
                               {CLi | ([Hras-GDP],4.0) (Src,5.0)}])
```

asks Maude to perform at most 100 rule applications (i.e. rewrite steps) to rewrite the given initial state and returns the final cell state we reached. It is also possible to rewrite without specifying an upper bound on the number of rule applications. Since the model may be non-deterministic (i.e. there might be several computations that start from the same initial state), Maude selects only one of these computations by means of a predetermined rewrite strategy. Therefore, the returned cell state may represent only one of the possible system behaviors.

Maude also provides the `frew` which allows one to implement user-defined rewrite strategies. As explained in Example 6.4, the forward simulation explores only one possible model behavior. In order to analyze all possible model dynamics we may employ the Maude `search` feature as shown in the following example.

Example 6.5 *Assuming the same QPL model of the previous example, we want to know whether —starting from a dish containing a given concentration of the ligand `Egf`— it is possible to produce the Mitogen-Activated Protein Kinase `Erk` (as described in [149]) with a concentration greater than* 3.6 *units. Such analysis can be modeled by means of the following Maude forward search query:*

```
search [1]
 PD((Egf,2.0) [HMEC | {CLo | empty} {CLm | (EgfR,1.0)(PIP2,3.0)}
   {CLi | ([Hras-GDP],4.0)(Src,5.0)}
   {CLc | (Gab1,5.0)(Grb2,6.0)(Pi3k,2.0)(Plcg,7.0)(Sos1,1.0)}])
   =>+ PD(out:Soup [HMEC|cyto:Soup {CLi|(Erk,k)}]) such that k > 3.6 .
```

The term before the right arrow denotes the initial state, while the one after the arrow specifies the pattern of the state we are looking for along with the condition that `Erk` has to appear with a concentration higher than `3.6`.

Finally, also Maude *Model checking* can be employed to analyze QPL models. Model checking enlarges the set of properties which can be investigated. While `search` only concerns with properties of individual states, model checking deals with properties of computations (i.e. pathways). In this context, the model-checker is typically asked to check the assertion that there is no computation starting from the given initial state satisfying the property of interest; thus a path can be extracted from a counterexample, if one is found.

Example 6.6 *By using Maude* `modelCheck` *function we can easily verify whether, starting from a given initial state* `istate` *containing* `Egf`*, it is possible to activate the entity* `Src` *with a concentration greater than* 4.5 *units without having previously produced the entity* `Rala-GDP`*. To this purpose, we define a parametric property* entAct(e,n)*, which is satisfied when the* `CLi` *location contains an occurrence of entity* `e` *with a concentration higher than* `n`*. The property might be specified as follows* `eq PD(out:Soup [HMEC|cyto:Soup {CLi|cli:Soup(e,k)}]) |= entAct(e,n) = k > n` *. Now, we can model the desired analysis by means of the following Maude model-checking query:*

```
red modelCheck(istate,
 []~(<> entAct([Src-act],4.5) /\
       (entAct([Src-act],4.5) |-> entAct(RalaGDP,0.0))))
```

The query is expressed in linear temporal logic[1] *(LTL) and consists of a conjunction of two sub-queries, the former asks for the activation of* `Src` *with a concentration of 4.5 units and the latter asks for a sequence of events (expressed by the* `|->` *operator) where the production of* `Rala` *strictly follows the activation of* `Src`*.*

6.2 Representing QPL models via Discrete Functional Petri Nets

QPL models can be represented by means of Discrete Functional Petri Nets (DFPN) which are restricted HFPNs [129] still able to model quantitative aspects of a given system by means of functional discrete transitions. The advantage of this alternative formalism is twofold. On the one hand, graphical representations are naturally derived from DFPNs, which allow one to visualize the model of interest and to graphically interact with it by using common available tools (e.g. Cell Illustrator). On the other hand, analysis of DFPNs can profit from well-known techniques which have been already developed for the Petri net settings. In what follows, we formalize DFPNs by borrowing terminology and notation from [129].

6.2.1 Discrete Functional Petri Nets

Definition 6.7 (Marking) *Given a finite set of places P, a* marking *of P is a mapping $M : P \to \mathbb{R}^+$. Given $p \in P$, $M(p)$ is called the* mark *of p. We denote the set of all possible markings of P by $\mathcal{M}$. Given two markings M and M' of P, we say that $M \geq M'$ iff for each $p \in P$, $M(p) \geq M'(p)$. Moreover, we say that M and M' are* incomparable *if $M \not\leq M'$ and $M' \not\leq M$.*

Let $\mathcal{M}$ be the set of all markings of the set of places P, we denote the set of all functions mapping a marking of P into a non-negative real number as $\mathcal{F}_P = \{f \mid f : \mathcal{M} \to \mathbb{R}^+\}$. Functions in $\mathcal{F}_P$ are called *update* functions.

[1] Maude supports an extension of LTL called LTLR [22].

Definition 6.8 (Discrete Functional Petri Nets) *A* Discrete Functional Petri Net *(DFPN) is a triple* $\mathcal{P} = (P, T, C)$ *where* $P = \{p_1, \ldots, p_n\}$ *is a non-empty finite set of* places*,* $T = \{t_1, \ldots, t_m\}$ *is a non-empty finite set of* transitions *such that* $P \cap T = \emptyset$*, and* C *is a tuple* (PT, TP, a, w, u) *defined as follows:*

- $PT \subseteq P \times T$ *and* $TP \subseteq T \times P$*. Elements in* PT *(resp.* TP*) are called* input connectors *(resp.* output connectors*). Each connector has a* connector type *which is given by a function* $a\colon PT \cup TP \to \{\texttt{process}, \texttt{test}, \texttt{inhibitor}, \texttt{output}\}$*. Input connectors whose type is* `process` *(resp.* `inhibitor`*,* `test`*) are also called* process *(resp.* inhibitor*,* test*) connectors.*

- $w : PT \to \mathbb{R}^+$ *is a mapping, called* threshold labeling*, that assigns non-negative real numbers to input connectors.*

- $u : PT \cup TP \to \mathcal{F}_P$ *is a partial mapping, called* update labeling*, such that* $u(c)$ *is defined iff* $a(c) \in \{\texttt{process}, \texttt{output}\}$ *(i.e. mapping* u *assigns update functions to process and output connectors only).*

Connectors in a DFPN are labeled by threshold and update labeling. More specifically, threshold labeling puts non-negative real numbers (i.e. *thresholds*) on input connectors (p, t) and are used to fix the minimum threshold on the mark of place p which is required to enable/disable transition t. Update labeling decorates both process and output connectors with update functions and is employed to change the mark of a place p whenever a transition involving p is fired. As we will see in Section 6.3, in order to make backward analysis of DFPNs possible update functions must fulfill the following conditions: (i) for each output connector (t, p), its update function can depend only on the marks of those places p' such that $(p', t) \in PT$; (ii) for each process connector (p, t), its update function can depend only on the mark of p. Update functions fulfilling conditions (i), and (ii) are said to be *well-behaved*.

The enablement relation of a transition t in a DPFN depends on the type of the input connectors (p, t). Basically, an inhibitor connector enables transition t when the mark of p is under a certain threshold; process and test connectors enable transition t whenever the mark of p is over the threshold. More formally,

Definition 6.9 *Let* $\mathcal{P} = (P, T, C)$*, where* $C = (PT, TP, a, w, u)$*, be a DFPN. Given a transition* $t \in T$ *and a marking* $M \in \mathcal{M}$ *of* P*, we say that* t *is* enabled *in* M *iff for each input connector* $c = (p, t) \in PT$ *the following conditions hold:*

1. $M(p) < w(c)$ *if* $a(c) = \texttt{inhibitor}$*.*

2. $M(p) \geq w(c)$ *if* $a(c) \neq \texttt{inhibitor}$*.*

Otherwise transition t *is said to be* disabled *in* M*. We denote the set of all the transitions which are enabled in* M *by* $\mathcal{E}(M)$*.*

Now, given a DFPN $\mathcal{P}$, we can define computations over P as follows.

Definition 6.10 *Let $\mathcal{P} = (P, T, C)$, where $C = (PT, TP, a, w, u)$, be a DFPN. Let $M, M' \in \mathcal{M}$ be two markings of P and t be a transition in T such that $t \in \mathcal{E}(M)$. Then, M* evolves into *M' by using t in $\mathcal{P}$ (in symbols, $\mathcal{P} \vdash M \overset{t}{\mapsto} M'$) iff $M' =$ DFPN_One_Step$(\mathcal{P}, M, t)$, where DFPN_One_Step$(\mathcal{P}, M, t)$ is a function defined as follows:*

DFPN_One_Step$(\mathcal{P}, M, t)$
for each $(p, t) \in PT$ *with* $a(p, t) =$ `process`*:* $M'(p) \leftarrow M(p) - u(p, t)(M)$
for each $(t, p) \in TP$: $M'(p) \leftarrow M(p) + u(p, t)(M)$
return M'

A computation *in $\mathcal{P}$ is a (possibly infinite) sequence $\mathcal{P} \vdash M_0 \overset{t_0}{\mapsto} M_1 \overset{t_1}{\mapsto} M_2 \overset{t_2}{\mapsto} \ldots$ Marking M_0 is called* initial *marking.*

Roughly speaking, a marking M can evolve into a marking M' by firing an enabled transition t that produces M' from M in the following way: for each process connector (p, t), the mark $M(p)$ is consumed by applying the update function labeling (p, t); for each output connector (t, p), the mark $M(p)$ is increased by applying the update function labeling (t, p); for each inhibitor/test connector (p, t), the mark $M(p)$ is left unchanged

Transitive ($\mapsto^+$) and transitive and reflexive ($\mapsto^*$) closures of relation $\mapsto$ are defined in the usual way. Note that several transitions can be enabled at the same time in a DFPN producing non-deterministic computations.

6.2.2 Translating QPL models into DFPNs

Given a QPL model expressed by a rewrite theory $\mathcal{Q} = (\Sigma, E, R)$, we can easily derive a DFPN having the same model behavior which will be denoted by $\mathcal{P}_\mathcal{Q}$. Let us start translating QPL models without entity or cell type variables. Basically, the translation procedure produces a DFPN transition for each rule r that belongs to R. Let us see how the translation of a single rule into a transition proceeds.
Let $r = (l : t \Rightarrow t' \text{ if } c)$, we define $\mathcal{OL}(r) = \{(e, q, loc) \mid$ occurrence (e, q) appears in t in location $loc\}$, $\mathcal{OC}(r) = \{(e, q, loc) \mid$ entity e appears in predicate *checkInhibitors* in c associated to location $loc\}$, and $\mathcal{OR}(r) = \{(e, q, loc) \mid$ occurrence (e, q) appears in t' in the location $loc\}$[2]. First of all, for each $(e, q, loc) \in \mathcal{OL}(r) \cup \mathcal{OR}(r) \cup \mathcal{OC}(r)$, we define a place $\langle e, loc \rangle$ whose marking $M(\langle e, loc \rangle)$ is represented by the placeholder q. Intuitively, each place in the resulting DFPN will model a given entity e that belongs to a compartment loc, while the concentration q provides information regarding the mark of the place.

Then, we generate the transition l, where l is the label of the rule r under examination. Transition l is connected to the generated places via input/output connectors in the following way. For each $(e, q, loc) \in \mathcal{OL}(r)$ we generate an input connector $con = (\langle e, loc \rangle, l)$ whose type $a(con)$ depends on the role of the entity e in the original

[2] As usual locations are identified by their location name.

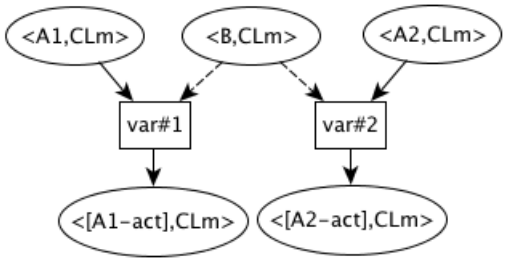


Figure 6.2: Graphical representation of DFPN of Example 6.11.

rule r (i.e. is e a process or a test?). For each $(e, loc) \in \mathcal{OC}(r)$ we generate an input connector $con = (\langle e, loc\rangle, l)$ whose type $a(con) =$ `inhibitor`. For each $(e, q, loc) \in \mathcal{OR}(r)$ if $a(\langle e, loc\rangle, l) \neq$ `test` we generate an output connector $con = (l, \langle e, loc\rangle)$ whose type $a(con) =$ `output`. Finally, thresholds and update functions are defined according to the expressions appearing in the rule condition.

Entity variables, which may appear in QPL rules, may range over a finite set of values (e.g. a class of ligands which may interact with a given receptor). If a rule contains an entity variable, we translate this rule by using as many transitions as the number of values the variable may assume. In this case the transition label consists in the rule label, suffixed with $\sharp 1, \sharp 2, \ldots$ if there are multiple instantiations. Given a rule r and a triple $(e, q, loc) \in \mathcal{OL}(r) \cup \mathcal{OR}(r) \cup \mathcal{OC}(r)$ with e entity variable over a range of values $\{v_1, \ldots, v_n\}$, we produce the set of transitions $\{t_1, \ldots, t_n\}$ labeled with $\{l\sharp 1, \ldots, l\sharp n\}$ where l is the label of rule r. Moreover, we replace (e, q, loc) with the set of triples $\{(v_1, q, loc), \ldots, (v_n, q, loc)\}$ and we proceed by creating the necessary input and output connectors as explained above.

Example 6.11 *Consider an entity type* `A` *that subsumes entities* `A1` *and* `A2` *and the following simple rule with entity variable* `?Var:A`
`var : {CLm|clm (?Var:A,a)(B,b)} ⇒ {CLm|clm ([?Var:A-act],a)(B,b)}`
which represents the activation of an entity of type `A` *whenever entity* `B` *is present. Then there will be produced two distinct transitions (*`var♯1, var♯2`*) and places (*`<A1,CLm>, <A2,CLm>, <[A1-act],CLm>, <[A2-act],CLm>, <B,CLm>`*) connected as shown in Figure 6.2. In figure, places are represented by circles, transitions by rectangles, test edges by dashed arrows and process and output edges by solid arrows.*

Note that rule's variables not representing concentrations or entity type (e.g. `cml`, `cli`, *etc.*) are not encoded into the net $\mathcal{P}_{\mathcal{Q}}$. Such variables have only an auxiliary purpose in the QPL model. Indeed, they are used to enable pattern matching, which is an evaluation mechanism not employed in DFPNs, and hence they do not have a counterpart in the resulting net model.

Figure 6.3 shows a DFPN obtained by the the translation of rules `ex1` and `ex2` of Examples 6.1 and 6.2. The complete description of the translation method is shown in Algorithms 3 and 4. The pseudo-code shown in Algorithm 3 describes the conversion

$H = (P, T, C)$
$P = \{$<A,CLo>, <C,CLo>, <[B-GDP],CLm>, <D,CLm>, <A,CLi>, <B,CLi>, <C,CLi>$\}$
$T = \{$ex1, ex2$\}$
$C = \{In, Out, a, w, u\}$
$In = \{$(<A,CLo>,ex1), (<[B-GDP],CLm>,ex1), (<D,CLm>,ex1), (<A,CLi>,ex2), (<B,CLi>,ex2)$\}$
$Out = \{$(ex1,<C,CLo>), (ex2,<C,CLi>)$\}$

$$a(c) = \begin{cases} \texttt{process} & \text{if } c \in \{\texttt{(<A,CLo>,ex1),(<[B-GDP],CLm>,ex1),(<A,CLi>,ex2)}\} \\ \texttt{test} & \text{if } c = \texttt{(<D,CLm>,ex1)} \\ \texttt{inhibitor} & \text{if } c = \texttt{(<B,CLi>,ex2)} \\ \texttt{output} & \text{if } c \in \{\texttt{(ex1,<C,CLo>), (<ex2,C,CLi>)}\} \end{cases}$$

$$w(c) = \begin{cases} 3.5 & \text{if } c = \texttt{(<A,CLo>,ex1)} \\ 4.0 & \text{if } c = \texttt{(<B,CLi>,ex2)} \end{cases}$$

$$u(c) = \begin{cases} \texttt{A\_CLo\_q}/2.0 & \text{if } c = \texttt{(<A,CLo>,ex1)} \\ \texttt{[B-GDP]\_CLm\_q} & \text{if } c = \texttt{(<[B-GDP],CLm>,ex1)} \\ \texttt{A\_CLo\_q}/2.0 + \texttt{B\_CLm\_q} & \text{if } c = \texttt{(ex1,<C,CLo>)} \\ \texttt{A\_CLi\_q} & \text{if } c = \texttt{(<A,CLi>,ex2)} \\ \texttt{A\_CLi\_q} & \text{if } c = \texttt{(ex2,<C,CLi>)} \end{cases}$$

Figure 6.3: DFPN encoding of rules ex1 and ex2.

of a set of rules where entity variables are considered as normal entities. Algorithm 4 modifies the DFPN produced by Algorithm 3 if the synthesized rule would contain entity variables. More precisely, the DFPN transition corresponding to the considered rule is replaced by a finite set of new transitions where the variable is replaced by the values it can assume.

A PL dish, that is an initial state for a QPL model, is naturally encoded into an initial marking for the resulting DFPN in which we assign a given mark to (some of) the places $\langle e, loc \rangle$ of the net.

6.2.3 Model equivalence.

Given a QPL model $\mathcal{Q}$, the resulting DFPN $\mathcal{P}_\mathcal{Q}$ is equivalent to $\mathcal{Q}$ in the sense that computations are preserved. In other words, any computation over $\mathcal{Q}$ is mapped to a computation over $\mathcal{P}_\mathcal{Q}$ and vice versa. The equivalence holds under certain conditions on the translation that we need to enforce. Our approach follows and adapts the proposal presented in [150] by Talcott and Dill for establishing the equivalence between Pathway Logic and standard Petri nets.

Let $\mathcal{Q}$ be a QPL model and $\mathcal{P}_\mathcal{Q}$ the DFPN obtained from $\mathcal{Q}$. Let $\mathcal{S}_\mathcal{Q}$ be the set of all possible (ground) cell states of $\mathcal{Q}$, and $\mathcal{M}_\mathcal{Q}$ be the set of all possible markings of $\mathcal{P}_\mathcal{Q}$.

We define a mapping $s2m\colon \mathcal{S}_\mathcal{Q} \to \mathcal{M}_\mathcal{Q}$ which maps a cell state $s \in \mathcal{S}_\mathcal{Q}$ into a marking $M_s \in \mathcal{M}_\mathcal{Q}$ such that, for each entity e that appears in a location loc with concentration q, $M_s(\langle e, loc \rangle) = q$. We define the inverse mapping of $s2m$ by $m2s\colon \mathcal{M}_\mathcal{Q} \to \mathcal{S}_\mathcal{Q}$, and thus we have $m2s(s2m(s)) = s$ for any cell state $s \in \mathcal{S}_\mathcal{Q}$. Since

Algorithm 3 Conversion of a set of QPL rules without variables.

procedure QPL2DFPN($\mathcal{R}$) ▷ $\mathcal{R}$ is a set of QPL rules
  $P = T = PT = TP \leftarrow \emptyset$, a, w, u undefined ▷ DFPN initialization
  for each rule `r` $\in \mathcal{R}$ having name `rn`, left/right-hand side `lhs/rhs` and conditions `cond` **do**
    $T \leftarrow T \cup \{$`rn`$\}$
    for each occurrence `(Thing,qexp)` appearing in `r` at location `Loc` **do**
      $P \leftarrow P \cup$ `<Thing,Loc>`
    end for
    for each occurrence `(Thing,qexp)` appearing in `rhs` at location `Loc` **do**
      if `(Thing, qexp)` appears also in `lhs` **then**
        Conn $\leftarrow \{$(`<Thing,Loc>,rn`)$\}$, $PT = PT \cup$ Conn
        if (`qexp < val`) appears in `cond` **then**
          w(Conn) $\leftarrow$ `val`, a(Conn) $\leftarrow$ `inhibitor`
        else if (`qexp > val`) appears in `cond` **then**
          w(Conn) $\leftarrow$ `val`, a(Conn) $\leftarrow$ `test`
        else
          w(Conn) $\leftarrow$ `0`, a(Conn) $\leftarrow$ `test`
        end if
      end if
      if `(Thing, qexp')` appears also in `lhs` with `qexp` $\neq$ `qexp'` **then**
        Conn $\leftarrow \{$(`<Thing,Loc>,rn`)$\}$, $PT \leftarrow PT \cup$ Conn
        a(Conn) $\leftarrow$ `process`
        `qexp` $\leftarrow$ `qexp` where variable has been replaced by `Thing_Loc_q`
        u(Conn) $\leftarrow$ `qexp`
        if (`qexp > val`) appears in `cond` **then**
          w(Conn) $\leftarrow$ `val`
        else
          w(Conn) $\leftarrow$ `0`
        end if
      end if
      if `(Thing, qexp)` does not appear in `lhs` **then**
        Conn $\leftarrow \{$(`rn,<Thing,Loc>`)$\}$, $TP \leftarrow TP \cup$ Conn
        a(Conn) $\leftarrow$ `output`
        `qexp` $\leftarrow$ `qexp` where variables have been replaced by correct names
        u(Conn) $\leftarrow$ `qexp`
      end if
    end for
    for each occurrence `(Thing,qexp)` appearing in `lhs` at location `Loc` **do**
      if `(Thing, qexp')` does not appear also in `rhs` **then**
        Conn $\leftarrow \{$(`<Thing,Loc>,rn`)$\}$, $PT \leftarrow PT \cup$ Conn
        a(Conn) $\leftarrow$ `process`
        u(Conn) $\leftarrow$ `Thing_Loc_q`
        if (`qexp > val`) appears in `cond` **then**
          w(Conn) $\leftarrow$ `val`
        else
          w(Conn) $\leftarrow$ `0`
        end if
      end if
    end for
  end for
  return $H = (P, T, \langle PT, TP, a, w, u\rangle)$
end procedure

Algorithm 4 Procedure to manage entity variables. We consider an entity variable `?var:Type` to appear in rule r, and we consequently modify the DFPN produced by Algorithm 3 as follows.

1: **procedure** DFPNModVar(`?var:Type`,r,H) ▷ $H = (P, T, \langle PT, TP, a, w, u\rangle)$
2: Let `V` be the finite set of values the variable `?var:Type` may assume
3: Let `t` be the petri net transition corresponding to rule r
4: Let (`<?var:Type,Loc>`) the Petri net place representing entity variable `?var:Type` appearing in rule r in location `Loc`
5: Let $C \in PT$ or TP be the connector between place (`<?var:Type,Loc>`) and transition `t`
6: i = 1
7: **for each** `v` $\in$ `V` **do**
8: $T \leftarrow T \cup \{$`t#i`$\}$
9: $P \leftarrow P \cup \{$(`<v,Loc>`)$\}$
10: **if** C is an *input connector* **then**
11: $NC \leftarrow$ (`<v,Loc>,t#i`)
12: $w(NC) \leftarrow w(C)$
13: $PT \leftarrow PT \cup \{NC\}$
14: **else**
15: $NC \leftarrow$ (`t#i, <v,Loc>`)
16: $TP \leftarrow TP \cup \{NC\}$
17: **end if**
18: $a(NC) \leftarrow a(C)$
19: $u(NC) \leftarrow u(C)$
20: i $\leftarrow$ i+1
21: **end for**
22: $T \leftarrow T$ - {t}
23: $P \leftarrow P$ - {(`<?var:Type,Loc>`)}
24: unset ($a(C)$)
25: unset ($w(C)$)
26: unset ($u(C)$)
27: $PT \leftarrow PT$ - C
28: $TP \leftarrow TP$ - C
29: **return** H
30: **end procedure**

cell states are terms which can be built upon contexts and holes, we extend $s2m$ to contexts and holes in such a way that $s2m(C[t]) = s2m(t) \cup s2m(C)$. Therefore, given a place $\langle e, loc \rangle$ of $\mathcal{P}_\mathcal{Q}$, which is obtained from a cell state $s = C[t] \in \mathcal{S}_\mathcal{Q}$, where C is a context with a hole at location loc_C, the mark of $\langle e, loc \rangle$ w.r.t. the mapping $s2m(C[t])$ is defined as

$$s2m(C[t])(\langle e, loc \rangle) = \begin{cases} s2m(t)(\langle e, loc \rangle) & \text{if } loc = loc_C \\ s2m(C)(\langle e, loc \rangle) & \text{otherwise} \end{cases}$$

In order to guarantee the computational equivalence between QPL models and their DFPN counterparts, we need to enforce a constraint over the rules of QPL models. Let us see an example that illustrates some issues related to the model translation which may arise.

Example 6.12 *Consider the two following QPL rules:*

$$\texttt{r1} : \{\texttt{CLm} \mid (\texttt{A}, \texttt{a})\, \{\texttt{CLc} \mid \texttt{cyto}\, (\texttt{B}, \texttt{b})\}\} \Rightarrow \{\texttt{CLm} \mid (\texttt{A}, \texttt{a})\, ([\texttt{B-act}], \texttt{b})\, \{\texttt{CLc} \mid \texttt{cyto}\}\}$$
$$\texttt{r2} : \{\texttt{CLm} \mid \texttt{clm}\, (\texttt{A}, \texttt{a})\, \{\texttt{CLc} \mid \texttt{cyto}\, (\texttt{B}, \texttt{b})\}\} \Rightarrow \{\texttt{CLm} \mid \texttt{clm}\, (\texttt{A}, \texttt{a})\, ([\texttt{B-act}], \texttt{b})\, \{\texttt{CLc} \mid \texttt{cyto}\}\}$$

where `cyto` *and* `clm` *are "anonymous" variables matching any other component located in the cytoplasm or cell membrane, respectively.*

Consider now the state $s = \{\texttt{CLm}|(\texttt{A}, 3.3)(\texttt{C}, 4.7)\{\texttt{CLc}|(\texttt{B}, 0.1)(\texttt{D}, 0.9)\}\}$. *The rule* `r2` *applies to* s *but* `r1` *does not because it lacks variable* `clm` *which enables the pattern matching between* s *and the lhs of* `r1` *on the location* `Clm`. *On the other hand,* `r1` *and* `r2` *are translated into transitions* `t1` *and* `t2` *that are equal modulo renaming of the transition labels. In particular,* `t1` *and* `t2` *connect the same places via the same input/output connectors. Therefore,* `t1` *is enabled whenever* `t2` *is, and vice versa. Clearly, this fact generates a discrepancy between the computational behaviors of the QPL model and the corresponding DFPN.*

To avoid the situation presented in Example 6.12, we basically forbid rules like `r1` from being specified in QPL models.

Definition 6.13 *Let* $\mathcal{Q} = (\Sigma, E, R)$ *be a QPL model. Then* $\mathcal{Q}$ *is* well-specified *iff for each rewrite rule* $r \in R$, *each location Loc appearing in the lhs or rhs of* r *contains a variable loc.*

Note that the QPL model of Example 6.12 is *not* well-specified.

Given a DFPN $H = (P, T, \langle PT, TP, a, w, u \rangle)$ and a transition $t \in T$, we may denote by P_t the set of places affected by t, that is, the set of places whose marking is modified when transition t is fired. The set P_t is exactly the set of places p such that there is a connection $(p, t) \in PT$ or a connection $(t, p) \in TP$. The same idea can be expressed by considering a transition step from a marking M to a marking M' by firing transition t, so that we can decompose M and M' w.r.t. transition t as follows: $M = M(P_t) \cup M(P - P_t)$ and $M' = M'(P_t) \cup M(P - P_t)$. What we want to show now is a parallelism between rewriting steps in a QPL model and transition steps in the corresponding DFPN.

Let $r2t$ be a function that maps each rule $r \in R$ into the corresponding transition $r2t(r)$ of $\mathcal{P}_\mathcal{Q}$. Then, the following result holds:

Proposition 6.14 (Rule2Transition) *Let $\mathcal{Q} = (\Sigma, E, R)$ be a well-specified QPL model, $r \in R$ a rule and s a state. Then,*

$$s \xrightarrow{r} s' \Leftrightarrow s2m(s) \overset{r2t(r)}{\longmapsto} s2m(s')$$

Proof 6.2.1 *($\Rightarrow$)*
The state s rewrites to s' via rule r iff $s = C[\sigma(lhs)]$, $\sigma(cond)$ holds, and $s' = C[\sigma(rhs)]$. It follows immediately that $s2m(s) = s2m(C) \cup s2m(\sigma(lhs))$, where $s2m(\sigma(lhs))$ corresponds to a marking over $P_{r2t(r)}$ and $s2m(C)$ corresponds to a marking over $P - P_{r2t(r)}$. Moreover, since the conversion algorithm maps rule conditions in the threshold function of DFPNs, and $\mathcal{Q}$ is well-specified, if $\sigma(cond)$ holds, then $s2m$ satisfies the conditions stated in Definition 6.9, and, hence, transition $r2t(r)$ is enabled by marking $s2m(s)$. Finally, firing transition t will modify only the marking $s2m(\sigma(lhs))$ in $s2m(\sigma(rhs))$, so the whole marking would be $s2m(C) \cup s2m(\sigma(rhs)) = s2m(s')$.
($\Leftarrow$)
Since marking $s2m(s)$ enables transition $r2t(r)$, the conditions stated in Definition 6.9 are satisfied by marking $s2m(s)$, and moreover we can decompose $s2m(s)$ in a marking M_1 over $P_{r2t(r)}$ and a marking M_2 over $P - P_{r2t(r)}$. For the conversion of QPL rules into transitions, the state $m2s(M_1)$ should be a term that E-matches with lhs, that is, $m2s(M_1) = \sigma(lhs)$. Therefore, $m2s(M_1 \cup M_2)$ should be a term $C[\sigma(lhs)]$. Since $M_1 \cup M_2 = s2m(s)$ we have that $s = C[\sigma(lhs)]$. Since marking $s2m(s)$ enables $r2t(r)$, substitution σ must satisfy condition cond, and, hence, s enables a rewriting step via rule r that produces term $C[\sigma(rhs)]$. Note that, the marking $s2m(s')$ should be the union of marking M_2, which is left unmodified, and a marking M_1' over $P_{r2t(r)}$. For the conversion algorithm, the state $m2s(M_1')$ should be the term $\sigma(rhs)$, so $m2s(M_1' \cup M_2)$ should be the term $C[\sigma(rhs)]$, but $M_1' \cup M_2 = s2m(s')$, and, hence, $s' = C[\sigma(rhs)]$.

The equivalence result between the two computational models is stated by the following theorem.

Theorem 6.15 *Let $\mathcal{Q} = (\Sigma, E, R)$ be a well-specified QPL model and $\mathcal{P}_\mathcal{Q}$ be the DFPN obtained from $\mathcal{Q}$. Then, the following result hold:*

$$\mathcal{Q} \vdash s_0 \xrightarrow{r_1} s_1 \ldots \xrightarrow{r_k} s_k \Leftrightarrow \mathcal{P}_\mathcal{Q} \vdash s2m(s_0) \overset{r2t(r_1)}{\longmapsto} s2m(s_1) \ldots \overset{r2t(r_k)}{\longmapsto} s2m(s_k)$$

where $s_i \in \mathcal{S}_\mathcal{Q}$, $r_j \in R$

Proof. By induction on the computation length k. The base case with $k = 0$ is trivial since the initial marking is $s2m(s_0)$, while the inductive step is given by application of Proposition 6.14.

6.3 Reachability analysis over DFPNs

Given a DFPN description of a QPL model, we can perform some kinds of analysis. First, we can perform a topological analysis of the net by disregarding the quantitative information, in order to find *pathways of interest*, that are sets of transitions that could be possibly enabled in a computation starting from a set of initial places and reaching a set of goals. Then, once obtained a pathway of interest, we can refine the analysis by exploiting the quantitative information by performing a reachability analysis over the subnet.

Reachability analysis. Let us first introduce the backward reachability problem over DFPNs. Given a DFPN H, and a set of goal markings $\mathcal{M}_g$ over H, we want to find the set of all incomparable minimal markings $\mathcal{M}$ over H, such that the *behavior* of the net starting from any marking $M \in \mathcal{M}$ may eventually reach a marking $M' \geq M_g \in \mathcal{M}_g$. The set $\mathcal{M}$ is said to be:

- *incomparable*, that means, for each pair of markings M, $M' \in \mathcal{M}$, M and M' are incomparable;
- *minimal*, which means that for each marking $M \leq M' \in \mathcal{M}$, there is no computation through the net, that starts from M, and leads to a marking $M'' \geq M_g \in \mathcal{M}_g$;

Algorithm 5 An algorithm for backward reachability over DFPNs

```
1: procedure BackwardReach(H,M_g)        ▷ M_g is a set of goal markings for a DFPN H
2:     M ← M_g
3:     M' ← ∅
4:     while M ≠ M' do
5:         M' ← M
6:         M ← BReach_One_Step(H,M')
7:     end while
8:     return M
9: end procedure
10: procedure BReach_One_Step(H,M_g)
11:     calculate the set of minimal incomparable markings M such that for each M ∈ M, M ↦ M' ≥ M_g where M_g ∈ M_g
12:     return M.
13: end procedure
```

This can be performed by using the backward state reachability analysis over well structured transition systems presented in [2]. A sketch of the algorithm for backward reachability over DFPNs (valid for Petri nets too) is presented in pseudo code in Algorithm 5. The basic step in the backward reachability analysis, which is described by procedure *BReachOneStep*, takes a set of goal markings $\mathcal{M}_g$ and computes the set of all possible minimal incomparable markings $\mathcal{M}_{onestep}$ such that

for each $M \in \mathcal{M}_{onestep}$, $M \overset{t}{\mapsto} M' \geq M_g \in \mathcal{M}_g$ by firing a certain transition t once. By applying many times the basic step, we eventually reach a stable set of incomparable minimal markings $\mathcal{M}$, as described by procedure *BackwardReach*. As shown in [2], the *well-structurness* of Petri nets (and DFPNs too) w.r.t. the ordering $\leq$ ensures the termination of the procedure.
Suppose that there are not inhibitor connectors in the considered DFPN subnet, and for each process and output connector its update function is a constant value. When using this constrained DFPNs, the backward reachability procedure is very simple and allows us to compute the set of minimal incomparable markings $\mathcal{M}$.

When more complex update functions are considered, we may obtain a set of sufficient but not minimal markings. More precisely, when the update function is not a constant value but it is expressed as a function of the marking we are looking for, that is, the update value depend on the mark (still unknown) of places, we are not able to derive minimal markings, but we may approximate them. We developed two possible methods to approximate a minimal marking. The first one has a higher convergence speed but the approximation obtainable is quite coarse. The second one has a slow convergence speed but the approximation is finer and sometimes reaches the optimum. Let us describe them with an example.

Example 6.16 *Consider a simple DFPN* $H = (\{P_1, P_2\}, \{t\}, \{\{c_1\}, \{c_2\}, w, u\})$, *where* $c_1 = (P_1, t)$, $c_2 = (t, P_2)$, $w(c_1) = 2.0$, $u(c_1) = P_1/3$, $u(c_2) = P1/2$. *Let us denote the marking as a pair* (P_1, P_2) *and assume our goal* M_g *is* $(0.0, 4.0)$, *which asks for at least 4 tokens in place* P_2.
Method 1. *The first method calculates the minimal marking needed to reach the goal by executing the transition only once. In the example, if we consider the production rate of* c_2 *(i.e.* $u(c_2)$*), we obtain marking* $(8.0, 0.0)$, *since* $(8.0, 0.0) \overset{t}{\mapsto} (8/3, 4.0) \geq (0.0, 4.0)$. *Note that, since this method considers one execution for each transition, it may disregard the consumption rate associated to input arcs (*$u(c_1)$ *in the example).*
Method 2. *The second method employes transitions threshold as the minimum number of tokens to let a transition fire and gets closer to the final goal allowing more than one transition firings. In our example, at the first step we set* P_1*'s value to the transition threshold (i.e. 2.0). If t fires with 2 tokens in* P_1, *it will produce only one token in* P_2, *so that there should be other 3 tokens in* P_2 *to reach the goal. Hence we get marking* $(2.0, 3.0)$. *In fact* $(2.0, 3.0) \overset{t}{\mapsto} (2/3, 4.0) \geq (0.0, 4.0)$. *Now the marking* $(2.0, 3.0)$ *becomes our new goal. At the second step, if we consider the consumption rate of* c_1, *we obtain for* P_1 *a minimum value of 6. If t fires with 6 tokens in* P_1, *it will produce 3 tokens in* P_2 *which is the exact quantity specified in the goal. Hence we get the final marking* $(6.0, 0.0)$, *since* $(6.0, 0.0) \overset{t}{\mapsto} (2.0, 3.0)$. *In this case, the marking* $(6.0, 0.0)$ *is really the minimal marking to reach the goal* $(0.0, 4.0)$.

In order to implement the presented methods we require update functions to be well-behaved. Note that, when a transition threshold is set to 0.0, the second method of analysis cannot be applied, since adopting the threshold value as the number of tokens for a certain place will not let the transition fire at all. So that, only in this case, we allow the user to provide a minimal incremental step $0 < \varepsilon \ll 1$ to let the

analysis proceed. It is clearly possible to combine the two methods during analysis, in order to get a compromise between convergence speed and accuracy. When inhibitor connectors are allowed in Petri nets, the reachability problem becomes much more difficult when even not decidable. In [141], the reachability problem for Petri nets with only one inhibitor connector is shown to be decidable, while the modeling capability of a Petri net with more than one inhibitor connector is shown to be equivalent to that of Turing machines in [124]. Hence, when the considered DFPN subnet contains inhibitor connectors we only provide the topological analysis.

Algoritm 6 shows in pseudo-code the backward reachability analysis procedure for DSPNs without inhibitor connectors. To manage complex update functions, we have implemented the second method explained above. The procedure takes three arguments: a DFPN, a set of goal markings and a positive value $\varepsilon \ll 1$. Starting from $\mathcal{M}_g$ the algorithm computes a sequence of set of markings $\mathcal{M}_g = \mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_{n+1}$ such that for all $i \in \{0, \ldots, n\}$, $\mathcal{M}_{i+1} = BReach\_One\_Step(H, \mathcal{M}_i)$ and $\mathcal{M}_n = \mathcal{M}_{n+1}$. The set $\mathcal{M}_n$ is the final result. At each step i, the set $\mathcal{M}_i$ is stored in queue Q_1 and $\mathcal{M}_{i+1}$ is constructed by using the auxiliary marking M' and then stored in queue Q_2. The core point of the algorithm is the construction of the new marking M' starting from the previous one M, by updating the marks of all input and output places of the considered transition.

Let t be the considered transition. We start updating places p connected by a test connector $c = (p, t)$, setting their mark to the threshold of connector c or the value ε if the threshold is zero. Then we update places p that are connected by a process connector $c = (p, t)$. If c's update function is a constant value, then we just add its value to the mark of the place, while if it is a complex update function we use it to calculate the new mark of the place p. Since the update function is well-behaved, the new mark for p depends only on its previous mark. Finally, we update all places p connected by an output connector $c = (t, p)$ by using c's update function and marking M' where we have already updated the marks of all input places of transition t. Again, since the update functions are well behaved, the marks of output places only depend on the marks of input ones.

6.4 Implementation

We implemented the framework presented so far in a prototypical system written in Maude, which is publicly available along with some examples at [28]. Basically, our system extends the Pathway Logic data structures and mechanisms to cope with quantitative information. Our system allows one to specify and analyze Quantitative Pathway Logic models by means of Maude language features. In particular, it is possible to exploit Maude built-in operators to easily express queries to simulate, search and model-check the models under examination. We tested the prototype on several small/medium size biological systems achieving rather promising results. In the future, we plan to provide a thorough experimental evaluation on real-size case studies assisted by the biologists of our group.

The prototype is also equipped with a model translator which allows one to au-

Algorithm 6 Algorithm for backward reachability over DFPNs without inhibitor connectors.

```
1: procedure BackwardReach(H, M_g, ε)          ▷ H = (P, T, ⟨PT, TP, a, w, u⟩)
2:                                              ▷ M_g is a set of goal markings over H
3:     Q_1 ← ∅
4:     Q_2 ← M_g
5:     while Q_1 ≠ Q_2 do
6:         Q_1 ← Q_2
7:         for each M ∈ Q_2 do                  ▷ This for loop implements the procedure BReach_One_Step
8:             M' ← empty marking
9:             for each place p_x such that M(p_x) ≠ 0 do
10:                M' ← M (*)
11:                for each transition t s.t. (t, p_x) ∈ TP do
12:                    for each test connector (p, t) ∈ PT do
13:                        set M'(p) ← max{w(p, t); ε}
14:                    end for
15:                    for each process connector (p, t) ∈ PT do
16:                        if u(p, t) is constant then
17:                            set M'(p) ← max{w(p, t); u(p, t) + M'(p)}
18:                        else
19:                            set M'(p) ← max{w(p, t); ε; u(p, t)^{-1}(M'(p))}
20:                        end if
21:                    end for
22:                    for each output connector (t, p) ∈ TP do
23:                        set M'(p) ← max{0; M'(p) − u(t, p)(M')}
24:                    end for
25:                end for
26:                if M' has been changed from (*) then
27:                    for each M'' ∈ Q_2 s.t. M' < M'' do
28:                        Q_2 ← Q_2 − {M''}
29:                    end for
30:                    if ∄M'' ∈ Q_2 s.t. M' ≥ M'' then
31:                        Q_2 ← Q_2 ∪ {M'}
32:                    end if
33:                end if
34:            end for
35:        end for
36:    end while
37:    return Q_1
38: end procedure
```

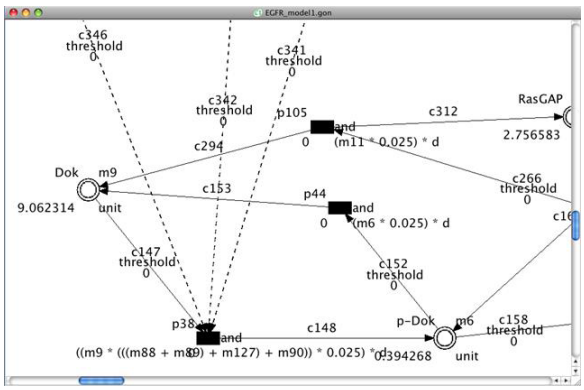


Figure 6.4: Cell Illustrator screenshot of the EgfR pathway model.

tomatically derive the corresponding DFPN from the given QPL model. On the net representation, we can apply well-known Petri net analysis methodologies such as topological analysis for relevant subnet detection, backward/forward reachability analysis, *etc.* Moreover, we provide the possibility to export DFPN descriptions in the *Cell System Markup Language* (CSML) [130], an XML format for modeling biopathways which covers widely used data formats, e.g. CellML[113], SBML[91].

Then, CSML representations of DFPNs can be imported in Cell Illustrator [128], which is a software tool by means of which we can visualize and graphically interact with the DFPN models. In Figure 6.4, a screenshot of the Cell Illustrator application with a small fragment of the EgfR pathway model is shown. Here, discrete transitions are identified by solid rectangles, while continuous places are represented by double circles. Dashed arrows and solid arrows stand for process connectors and test connectors, respectively; finally, labels identify thresholds and update functions.

The QPL formalism we proposed is the first step towards the development of a general Pathway Logic formalism for the full specification of hybrid models (specifying both discrete and continuous components) and their stochastic counterparts.

Conclusions

The success of key human activities ranging from everyday services to research and business relies on the use of ever more sophisticated, feature-rich and complex computer systems. In this thesis, we focused on some directions that are related to the system's complexity research, and we proposed some effective solutions. We addressed different problems related to the specification and analysis of complex systems by using formal methods. More precisely, we studied several aspects of the problem with the aim to provide some formal model of the problem itself in order to support formal analysis and verification. In order to achieve this goal, we made use of rule-based formalisms, such as rewriting logic, as a basis to develop domain-specific languages that support modelling, verification and testing. The main application areas that we considered are distributed systems such as Web systems and biological systems.

The distributed character of the considered systems, which provides an easy way to share and exchange data and resources over the Internet, suggested us the study of security aspects such as the access control problem and the software certification for secure the delivery of code. The first part of this thesis has been devoted to this area. In Chapter 2, we have presented a study on the Unfold operation in rewriting logic, based on narrowing, and provided general conditions that guarantee that the meaning of the program is not modified by the transformation. Then, we proposed the first Fold/Unfold-based program transformation framework for rewriting logic which opens up new applications to program optimization, program synthesis, program specialization and theorem proving for first-order typed rule-based languages such as ASF+SDF, Elan, OBJ, CafeOBJ, and Maude that may include sorts, rules, equations and algebraic laws. The core transformation rules are Fold, Unfold, definition introduction, definition elimination, and abstraction. The correctness of the program transformation framework guarantees that the transformed program is equivalent to the initial one in the sense that it preserves the standard semantics of ground reducts. We have shown that our methodology can be effectively applied to Code-Carrying Theory and may significantly simplify the code producer task. Actually, by applying a built-in strategy such as composition, tupling, etc. the code producer can (semi-)automatically obtain the final improved program. Thanks to the fact that our transformation methodology relies on narrowing, this can be done by adapting the narrowing-based transformation strategies of [11]. Moreover, as an outcome of the transformation process, a compact representation of the sequence of applied transformation rules is delivered as a certificate to the code consumer. The code consumer needs only to apply the certificate to the initial requirements in order to obtain the desired program. This checking can be completely automated. Of course, for the methodology to pay off completely in practice, the transformation system could be instrumented to also provide an estimation of the achieved optimization. Such an

extension can be investigated in future work.

In Chapter 3, we showed how domain specific languages play a key role in access control. On the one hand, they allow security administrators to formally specify precise policy behaviors, and on the other hand, formal methods can be applied to give support to both analysis and verification of access control policies. We proposed a novel rule-based language which is particularly suitable for managing security in distributed applications, where access control information may be shared across multiple sites. In fact, the operational engine underlying the language allows us to collect relevant data from distributed knowledge bases by means of DL queries, and to employ such data to take decisions w.r.t. the authorization requests under examination. Moreover, our formalism can be used to safely build global policies by means of rule-based combining operators. We have also shown how our formalism supports powerful policy analysis methodologies which take advantage of both rewriting and reasoning capabilities. On the practical side, we implemented the theoretical framework into a prototypical tool which we used to experimentally evaluate the proposed approach. Trust is another important topic in access control that should be considered. PeerTrust [92] provides a very interesting mechanism for gaining access to secure information on the web by using semantic annotations, policies and automated trust negotiation. A future work direction is endowing our formalism with constructs to manage trust negotiations. Moreover, narrowing-based analyses in the style of [106] can be integrated in our framework to potentiate its verification capabilities. Hopefully, this might be the base for developing policy repair and optimization techniques.

The second part of the thesis focuses on other important problems concerning the analysis and verification of data in distributed Web systems, and attempts to exploit the knowledge and expertise we acquired on distributed systems to model and analyze biological systems.

The growing complexity of the World Wide Web demands for tools which are able to tame the so-called information overload. To this respect, filtering and query languages allow one to extract relevant and meaningful information within the enormous amount of data available on the Web. In Chapter 4, we firstly presented a declarative XML filtering language which has several advantages w.r.t. other approaches. It is inspired by the approximate pattern-based query language ApproXQL [145] and extends it by introducing a number of new syntax constructs which provide a much more expressive framework (e.g. negative filtering, pattern variables, nested queries, conditional filtering, etc.). Secondly, we endowed our filtering language with the capability to search XML patterns into XML documents w.r.t. semantic criteria. The filtering process is guided by an approximate pattern-matching engine which queries an ontology reasoner to infer semantic information regarding the XML data. Semantic information can be employed to automatize the search of XML tag synonyms used by the matching mechanism when renaming transformations are needed, and to model semantic properties of the data to be extracted. Finally, we implemented the language in the prototype XPhil by using the lazy functional language Haskell, and pointed out the inborn benefits of laziness by means of a thorough experimental evaluation.

In Chapter 5, we faced the verification problem of the static content of Web sys-

tems and we developed a rule-based specification language, inspired by GVerdi [9, 32], that allows us to formalize and automatically check semantic and syntactic properties over Web system contents. The properties that can be specified within our framework are correctness and completeness. Correctness implies that the information provided on a web page is valid w.r.t. the application requirements, while completeness requires web pages to contain some piece of information. We extended the GVerdi language expressiveness by adding new constructs for defining conjunctive as well as disjunctive Web patterns. Moreover, we exploited ontology reasoning capabilities to retrieve semantic information which is useful to refine the verification process. The verification methodology is based on partial rewriting and consists in executing the specification rules against the considered web contents in such a way incorrect or missing information can be identified. We also defined some syntactic restrictions to our specification rules in order to ensure the termination of the verification methodology.

The last contribution of this thesis focuses on the application of formal methodologies to the study of biological systems. We concentrate our focus on a logical formalism called Pathway Logic (PL), first proposed in [81]. PL provides a framework for discrete modelling of biological systems, which does not consider quantitative information. In Chapter 6 we proposed the Quantitative Pathway Logic (QPL) formalism that endows PL with a full handling of quantitative information. QPL efficiently integrates quantitative data (such as substances concentration, reaction rates of substances production and consumption) into PL models. Moreover, it allows one to model inhibitors, that is, substances that decreases the rate of, or prevents, a chemical reaction. QPL models can be directly simulated by using rewriting logic or can be translated into Discrete Functional Petri Nets (DPFN) which are a subclass of Hybrid Functional Petri Nets (HFPN). By using the latter representation, our models can be graphically visualized and simulated by means of well known tools (e.g. Cell Illustrator [128]). Also, it is possible to exploit the DFPN representation in order to perform an approximated rechability analysis [57]. The extension we proposed aims at defining hybrid models (specifying both discrete and continuous components), which on the one hand can take advantage of the expressiveness of HFPNs (or Hybrid Automata), on the other hand guarantee sufficient analysis and verification capabilities. For analysis and verification purposes, it is possible to evaluate the construction of (possibly stochastic) hybrid automata. Moreover, by combining the hybrid extension to the PL probabilistic model, we can obtain a novel calculus called Hybrid Probabilistic Pathway Logic (HPPL) which might be particularly fruitful for the modelling and analysis of complex systems.

A

Some technicalities

A.1 XPhilSchema

This appendix contains the XML code representing the XML Schema of the XML formalization of the semantic filtering language, presented in Section 4.6

```
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:dig="http://dl-web.man.ac.uk/dig/2003/02"
         elementFormDefault="qualified">

<import namespace="http://dl-web.man.ac.uk/dig/2003/02"
        schemaLocation="dig.xsd"/>

<element name="rule" type="ruleType"/>

<complexType name="ruleType">
 <sequence>
  <element name="count" minOccurs="0">
   <complexType>
    <attribute name="cost" type="nonNegativeInteger" use="required"/>
   </complexType>
  </element>
  <element name="filter" type="filterType"/>
  <element name="pattern" type="string"/>
  <element name="document" type="docType"/>
  <element name="conditionClause" type="cond" minOccurs="0"/>
  <element name="mode" type="ruleMode" default="P" minOccurs="0"/>
 </sequence>
</complexType>

<simpleType name="filterType">
  <restriction base="string">
    <enumeration value="filterOneBest"/>
```

```
    <enumeration value="filterAllBest"/>
    <enumeration value="filterOneExact"/>
    <enumeration value="filterAllExact"/>
  </restriction>
</simpleType>

<complexType name="docType">
  <choice>
    <element name="nestRule" type="ruleType"/>
    <element name="xmlFile" type="string"/>
    <element name="docFile" type="anyURI"/>
  </choice>
</complexType>

<complexType name="cond">
    <choice maxOccurs="unbounded">
      <element name="reCond" type="matchPredicate"/>
      <element name="opCond" type="relation"/>
      <element name="ontCond" type="dig:asks"/>
    </choice>
</complexType>

<simpleType name="ruleMode">
  <restriction base="string">
    <pattern value="P|N"/>
  </restriction>
</simpleType>

<simpleType name="Variable">
  <restriction base="string">
    <pattern value="[A-Z]"/>
  </restriction>
</simpleType>

<simpleType name="BoolOperator">
  <restriction base="string">
    <enumeration value="lt"/>
    <enumeration value="gt"/>
    <enumeration value="leq"/>
    <enumeration value="geq"/>
  </restriction>
</simpleType>

<simpleType name="NumOperator">
  <restriction base="string">
```

```
    <enumeration value="+"/>
    <enumeration value="++"/>
    <enumeration value="-"/>
    <enumeration value="*"/>
    <enumeration value="/"/>
  </restriction>
</simpleType>

<complexType name="relation">
  <choice>
    <element name="Equal" type="eq"/>
    <element name="NotEqual" type="eq"/>
    <element name="Predicate" type="predicate"/>
  </choice>
</complexType>

<complexType name="predicate">
  <sequence>
    <element name="Rel" type="BoolOperator"/>
    <element name="lhs" type="genericExpr"/>
    <element name="rhs" type="genericExpr"/>
  </sequence>
</complexType>

<complexType name="matchPredicate">
  <sequence>
    <element name="Predicate" type="matchPred"/>
  </sequence>
</complexType>

<complexType name="matchPred">
  <complexContent>
    <restriction base="predicate">
      <sequence>
        <element name="Rel" fixed="match"/>
        <element name="lhs">
          <complexType>
    <sequence>
      <element name="Var" type="Variable"/>
    </sequence>
  </complexType>
        </element>
        <element name="rhs">
          <complexType>
            <sequence>
```

```
              <element name="Data" type="string"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </restriction>
  </complexContent>
</complexType>

<complexType name="eq">
  <sequence>
    <element name="lhs" type="genericExpr"/>
    <element name="rhs" type="genericExpr"/>
  </sequence>
</complexType>

<complexType name="genericExpr">
  <choice>
    <element name="Expr" type="expr"/>
    <element name="Var" type="Variable"/>
    <element name="Data" type="string"/>
  </choice>
</complexType>

<complexType name="expr">
  <sequence>
    <element name="Fun" type="NumOperator"/>
    <choice minOccurs="2" maxOccurs="2">
        <element name="Var" type="Variable"/>
        <element name="Data" type="string"/>
        <element name="Expr" type="expr"/>
      </choice>
  </sequence>
</complexType>

</schema>
```

Bibliography

[1] A. Abate, Y. Bai, N. Sznajder, C. Talcott, and A. Tiwari. Quantitative and Probabilistic Modeling in Pathway Logic. In IEEE Xplore, editor, *Proc. of the 7th IEEE International Conference on BioInformatics and BioEngineering*, pages 922–929, 2007.

[2] P.A. Abdulla, K. Čerāns, B. Jonsson, and Y. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1-2):109–127, 2000.

[3] M.H. Alalfi, J. R. Cordy, and T. R. Dean. Modelling methods for web application verification and testing: state of the art. *Software Testing, Verification and Reliability*, 19(4):265–296, 2009.

[4] H. Alla and R. David. Continuous and Hybrid Petri Nets. *Journal of Circuits, Systems and Computers*, 8(1):159–188, 1998.

[5] J.M. Almendros. A RDF Query Language Based on Logic Programming. *Electronic Notes in Theoretical Computer Science*, 200(3):67–85, 2008.

[6] M. Alpuente, M. Baggi, D. Ballis, and M. Falaschi. Semantic Verification of Web System Contents. In *Advances in Conceptual Modeling – Challenges and Opportunities*, volume 5232 of *LNCS*, pages 437–446. Springer-Verlag, 2008.

[7] M. Alpuente, M. Baggi, D. Ballis, and M. Falaschi. A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In ACM, editor, *In Proc. of ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM'10)*, pages 43–52, New York, NY, USA, 2010.

[8] M. Alpuente, D. Ballis, and M. Falaschi. Rule-based Verification of Web Sites. In *1st Int'l Symposium on Leveraging Applications of Formal Methods (ISoLA'04)*, pages 81—88, 2004.

[9] M. Alpuente, D. Ballis, and M. Falaschi. Automated Verification of Web Sites Using Partial Rewriting. *Software Tools for Technology Transfer*, 8:565–585, 2006.

[10] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In *6th International Joint Conference on Algebraic and Logic Programming*, pages 1–15. Springer-Verlag, 1997.

[11] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + strategies for transforming lazy functional logic programs. *Theoretical Computer Science*, 311(1-3):479–525, 2004.

[12] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS '98)*, 20(4):768–844, 1998.

[13] M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9–es, September 1998.

[14] S. Amer-Yahia, M. F. Fernández, D. Srivastava, and Y. Xu. Phrase Matching in XML. In *Proc. of 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 177–188, 2003.

[15] K. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[16] K. Arkoudas. *Denotational proof languages*. PhD thesis, Massachusetts Institute of Technology, 2000. Supervisor-Shivers, Olin.

[17] K. Arkoudas. An Athena tutorial, 2005. Available at: `http://www.cag.csail.mit.edu/~kostas/dpls/athena/athenaTutorial.pdf`.

[18] U. Assmann, J. Henriksson, and J. Maluszynski. Combining Safe Rules and Ontologies by Interfacing of Reasoners. In *Principles and Practice of Semantic Web Reasoning*, volume 4187 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[19] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Scheider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[20] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[21] F. Baader and W. Snyder. *Handbook of Automated Reasoning*, chapter Unification Theory. Elsevier and MIT Press, 2001.

[22] K. Bae and J. Meseguer. The Linear Temporal Logic of Rewriting Maude Model Checker. In *Proc. of 8th International Workshop on Rewriting Logic and its Applications (WRLA '10)*, 2010. To appear.

[23] M. Baggi. An Ontology-based System for Semantic Filtering of XML Data. *Electronic Notes in Theoretical Computer Science*, 235:19–33, 2009.

[24] M. Baggi and D. Ballis. PHIL: A Lazy Implementation of a Language for Approximate Filtering of XML Documents. *Electronic Notes in Theoretical Computer Science*, 216:93–109, 2008.

[25] M. Baggi, D. Ballis, and M. Falaschi. XPhil: the extended filtering language. Available at `http://users.dimi.uniud.it/~michele.baggi/xphil.html`, 2008.

[26] M. Baggi, D. Ballis, and M. Falaschi. XML Semantic Filtering via Ontology Reasoning. *Internet and Web Applications and Services, International Conference on*, pages 482–487, 2008.

[27] M. Baggi, D. Ballis, and M. Falaschi. Paul - the Policy Specification and Analysis Language. Available at: `http://users.dimi.uniud.it/~michele.baggi/paul/`, 2009.

[28] M. Baggi, D. Ballis, and M. Falaschi. The QPL System. Available at `http://users.dimi.uniud.it/~michele.baggi/qpl/`, 2009.

[29] M. Baggi, D. Ballis, and M. Falaschi. Quantitative Pathway Logic for Computational Biology. In Pierpaolo Degano and Roberto Gorrieri, editors, *Computational Methods in Systems Biology*, volume 5688 of *Lecture Notes in Computer Science*, pages 68–82. Springer Berlin. Heidelberg, 2009.

[30] M. Baggi, D. Ballis, and M. Falaschi. An access control language based on term rewriting and description logic. In *In Proc. of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP'10), Madrid (Spain)*, 2010.

[31] M. Baggi and P. Lopez. Meta Maudest: Program Transformation for RWL. Available at: `http://users.dsic.upv.es/grupos/elp/maudest/`, 2010.

[32] D. Ballis and J. García Vivó. A Rule-based System for Web Site Verification. In *Proc. of the 1st International Workshop on Automated Specification and Verification of Web Sites (WWV'05), Valencia (Spain), 2005*, volume 157(2). Electronic Notes in Theoretical Computer Science, Elsevier, 2005.

[33] 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 19–28, Washington, DC, USA, 2006. IEEE Computer Society Press.

[34] S. Barker and M. Fernández. Term Rewriting for Access Control. In *Proc. of the 20th Annual IFIP WG 11.3 Conference on Data and Applications Security (DBSec '06)*, volume 4127 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2006.

[35] S. Bechhofer. DIG 2.0: The DIG Description Logic Interface. Available at `http://dig.cs.manchester.ac.uk/`, 2006.

[36] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proc. of 8th ACM SIGPLAN International Conference on Functional Programming, (ICFP'03)*, pages 51–63, 2003.

[37] J.A. Bergstra, J Heering, and P. Klint. *Algebraic Specification.* ACM Press, 1989.

[38] M. Bernardo, P. Degano, and G. Zavattaro, editors. *Proc. of the 8th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Computational Systems Biology*, volume 5016 of *Lecture Notes in Computer Science*, 2008.

[39] C. Bertolissi and M. Fernández. A Rewriting Framework for the Composition of Access Control Policies. In *Proc. of the 10th Int'l ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '08)*, pages 217–225. ACM, 2008.

[40] M. Bezem. *TeReSe, Term Rewriting Systems*, chapter Mathematical background (Appendix A). Cambridge University Press, 2003.

[41] H. Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *Proc. of 14th International Conference on Applications of Prolog, (INAP'01)*, volume 2543 of *Lecture Notes in Computer Science*, pages 5–22, 2001.

[42] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An Algebra for Composing Access Control Policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.

[43] P. Borovanský, C. Kirchner, H. Kirchner, and P. E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.

[44] A. Bouhoula, J.P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.

[45] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[46] R. M. Burstall and J. Darlington. Some transformations for developing recursive programs. *SIGPLAN Not.*, 10(6):465–472, 1975.

[47] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of ACM*, 24(1):44–67, 1977.

[48] N. Busi. Expressiveness issues in brane calculi: A survey. *Electronic Notes in Theoretical Computer Science*, 209:107–124, 2008.

[49] 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.

[50] L. Cardelli. Brane Calculi - Interactions of biological membranes. In *Proc. of Computational Methods in System Biology (CMSB '04)*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–278. Springer Berlin, Heidelberg, 2005.

[51] Centrum voor Wiskunde en Informatica. XMark – an XML Benchmark Project, 2001. Available at: `http://monetdb.cwi.nl/xml/`.

[52] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Boston, MA, USA, 1988.

[53] C. Chaouiya, E. Remy, and D. Thieffry. Petri net modelling of biological regulatory networks. *Journal of Discrete Algorithms*, 6(2):165–177, 2008.

[54] Y. Chiba, T. Aoto, and Y. Toyama. Program Transformation by Templates Based on Term Rewriting. In *Procs. of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, (PPDP '05)*, pages 59–69, New York, NY, USA, 2005. ACM.

[55] W. Chin. Towards an Automated Tupling Strategy. In *Procs. of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, (PEPM '93)*, pages 119–132. ACM, 1993.

[56] W. Chin, A. Goh, and S. Khoo. Effective Optimisation of Multiple Traversals in Lazy Languages. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA (Technical Report BRICS-NS-99-1)*, pages 119–130. University of Aarhus, DK, 1999.

[57] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92)*, pages 343–354, New York, NY, USA, 1992. ACM Press.

[58] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Unification and Narrowing in Maude 2.4. In Ralf Treinen, editor, *Procs. of 20th International Conference on Rewriting Techniques and Applications, (RTA '09), Brasília, Brazil*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer-Verlag, 2009.

[59] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA '03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2003.

[60] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[61] J. Coelho and M. Florido. VeriFLog: A Constraint Logic Programming Approach to Verification of Website Content. In *APWeb International Workshops*, pages 148–156, 2006.

[62] J. Coelho and M. Florido. XCentric: logic programming for XML processing. In *Proc. of 9th ACM International Workshop on Web Information and Data Management, (WIDM'07)*, pages 1–8, 2007.

[63] T.A. Cooper and N. Wogrin. *Rule-based Programming with OPS5.* Morgan Kaufmann, 1988.

[64] A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and Abstract Semantics of a Graphical Query Language. *Theoretical Computer Science*, 275:521–560, 2002.

[65] E. Damiani, S. De Capitani di Vimercati, C. Fugazza, and P. Samarati. Extending Policy Languages to the Semantic Web. In *4th International Conference on Web Engineering (ICWE'04)*, volume 3140 of *Lecture Notes in Computer Science*, pages 330–343. Springer, 2004.

[66] J. Darlington. *A semantic approach to automatic program improvement.* PhD thesis, Department of Machine Intelligence, Edimburgh University, Edimburgh, U.K., 1972.

[67] J. Darlington. Program Transformation. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.

[68] O. De Moore and G. Sittampalam. Generic Program Transformation. In *Advanced Functional Programming*, pages 116–149, 1998.

[69] M. Dean and G. Schreiber. OWL Web Ontology Language Reference — W3C recommendation, 2004. Available at: `http://www.w3.org/TR/owl-ref/`.

[70] G. Denker, L. Kagal, T. W. Finin, M. Paolucci, and K. P. Sycara. Security for DAML Web Services: Annotation and Matchmaking. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 335–350. Springer, 2003.

[71] N. Dershowitz. Computing with Rewrite Systems. *Information and Control*, 64(2-3):122–157, 1985.

[72] N. Dershowitz and U. Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494, 1993.

[73] J. DeTreville. Binder, a Logic-Based Security Language. In *Proc. of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE Computer Society, 2002.

[74] P.T. Devanbu, P. W-L. Fong, and Stubblebine S.G. Techniques for trusted software engineering. In *Proc. of the 20th International Conference on Software Engineering (ICSE '98)*, pages 126–135. IEEE Computer Society, 1998.

[75] E. Di Sciascio, F. M. Donini, M. Mongiello, and G. Piscitelli. Web Applications Design and Maintenance Using Symbolic Model Checking. In *Proc. 7th European Conf. on Software Maintenance and Reengineering*, page 63. IEEE Computer Society, 2003.

[76] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, AMAST Series, 1998.

[77] K.R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of ADBMS features. In Timos Sellis, editor, *Rules in Database Systems*, volume 985 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin, Heidelberg, 1995.

[78] D. J. Dougherty, C. Kirchner, H. Kirchner, and A. S. De Oliveira. Modular Access Control Via Strategic Rewriting. In *Proc of the 12th European Symposium on Research in Computer Security (ESORICS '07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593. Springer, 2007.

[79] W. Drabent and A. Wilk. Extending XML Query Language Xcerpt by Ontology Queries. In *Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)*, pages 447–451. IEEE Computer Society Press, 2007.

[80] S. M. Easterbrook, B. Nuseibeh, and A. Russo. Leveraging Inconsistency in Software Development. *IEEE Computer*, 33(4):24–29, 2000.

[81] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway Logic: Symbolic Analysis of Biological Signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 400–412, 2002.

[82] S. Eker, K. Laderoute, P. Lincoln, and C. Talcott. Pathway Logic: executable models of biological networks. *Electronic Notes in Theoretical Computer Science*, 71, 2002.

[83] E. Ellmer, W. Emmerich, A. Finkelstein, and C. Nentwich. Flexible Consistency Checking. *ACM Transactions on Software Engineering*, 12(1):28–63, 2003.

[84] S. Etalle and M. Gabbrielli. Modular Transformations of CLP Programs. In L. Sterling, editor, *26th International Conference on Logic Programming*. MIT Press, 1995.

[85] F. Fages and S. Soliman. Formal Cell Biology in Biocham. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *Proc. of the 8th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Computational Systems Biology*, volume 5016 of *Lecture Notes in Computer Science*, pages 54–80, 2008.

[86] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *J. ACM*, 49(3):368–406, 2002.

[87] M. Fay. First Order Unification in an Equational Theory. In *Procs. of 4th International Conference on Automated Deduction*, pages 161–167, 1979.

[88] G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. Research Report RR-4970, INRIA, 2003.

[89] T. W. Finin and A. Joshi. Agents, Turst, and Information Access on the Semantic Web. *SIGMOD Record*, 31(4):30–35, 2002.

[90] W. Fontana and L. W. Buss. *Boundaries and Barriers*, chapter The barrier of objects: from dynamical system to bounded organizations, pages 56–116. Addison-Wesley, 1996.

[91] R. Gauges, U. Rost, S. Sahle, and K. Wegner. A model diagram layout extension for SBML. *Bioinformatics*, 22(15):1879–1885, 2006.

[92] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett. No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web. In *The Semantic Web: Research and Applications, First European Semantic Web Symposium (ESWS'04)*, volume 3053 of *LNCS*, pages 342–356. Springer, 2004.

[93] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.

[94] P. Goss and J. Peccoud. Quantitative modelling of stochastic systems in molecular biology by using stochastic Petri nets. In *Proc. of the National Academy of Science USA*, pages 6750–6755, 1998.

[95] D. Harel, Y. Setty, S. Efroni, N. Swerdlin, and I.R. Cohen. Concurrency in biological modeling: Behavior, execution and visualization. *Electronic Notes in Theoretical Computer Science*, 194(3):119–131, 2008.

[96] J. Hendrix, J. Meseguer, and H. Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In U. Furbach and N. Shankar, editors, *Third International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.

[97] T. A. Henzinger. The theory of hybrid automata. *Logic in Computer Science, Symposium on*, 0:278, 1996.

[98] H. Hosoya and B. Pierce. Regular Expressions Pattern Matching for XML. In *Proc. of 25th ACM SIGPLAN-SIGACT Int'l Symp. POPL*, pages 67–80. ACM, 2001.

[99] H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[100] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of ACM*, 27(4):797–821, 1980.

[101] L. Kagal, T. Berners-Lee, D. Connolly, and D. J. Weitzner. Using Semantic Web Technologies for Policy Management on the Web. In *21st national conference on Artificial Intelligence (AAAI'06)*. AAAI Press, 2006.

[102] L. Kagal, T. W. Finin, and A. Joshi. A Policy Based Approach to Security for the Semantic Web. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 402–418. Springer, 2003.

[103] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. In *Future Generation Computer Systems*, pages 413–422. ICOT, 1988.

[104] S. Kelly and J.P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[105] P. Kilpeläinen. Tree Matching Problems with Applications to Structured Text Databases. Ph.d. thesis, University of Helsinki (Finland), 1992.

[106] C. Kirchner, H. Kirchner, and A. S. De Oliveira. Analysis of Rewrite-Based Access Control Policies. *Electronic Notes in Theoretical Computer Science*, 234:55–75, 2009.

[107] 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.

[108] V. Kolovski, J. Hendler, and B. Parsia. Analyzing Web Access Control Policies. In *Proc. of the 16th Int'l Conference on World Wide Web (WWW '07)*, pages 677–686. ACM, 2007.

[109] H.J. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proc. of 9th ACM Symposium on Principles of Programming Languages*, pages 255–267, 1982.

[110] L. Kott. Unfold/fold program transformation. In M. Nivat and J.C. Reynolds, editors, *Algebraic methods in semantics*, chapter 12, pages 411–434. Cambridge University Press, 1985.

[111] C. Li, S. Miyano, and H. Matsuno. Petri net Based Descriptions for Systematic Understanding of Biological Pathways. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E98-A(11):3166–3174, 2006.

[112] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, 2003.

[113] C.M.M. Lloyd, J.R.R. Lawson, P.J.J. Hunter, and P.F.F. Nielsen. The CellML Model Repository. *Bioinformatics*, 2008.

[114] W. Lukaszewicz. *Non-Monotonic Reasoning: Formalization of Commonsense Reasoning.* Ellis Horwood, New York, 1990.

[115] Z. Manna and R.J. Waldinger. Toward automatic program synthesis. *Communication of the ACM*, 14(3):151–165, 1971.

[116] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

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

[118] H. Matsuno, Y. Tanaka, H. Aoshima, A. Doi, M. Matsui, and S. Miyano. Biopathways representation and simulation on hybrid functional Petri net. *In Silico Biology*, 3:32, 2003.

[119] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 2004.

[120] R. Mayr and M. Rusinowitch. Reachability is Decidible for Ground AC Rewrite Systems. In *Proc. of the 3rd INFINITY Workshop*, pages 53–64, 1998.

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

[122] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher Order Symbolic Computation*, 20(1-2):123–160, 2007.

[123] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[124] M.L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[125] T. Moses. eXtensible Access Control Markup Language (XACML) v2.0. Technical report, OASIS, 2005.

[126] T. Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4):541–580, 1989.

[127] D.R. Musser. Automated theorem proving for analysis and synthesis. *Current trends in hardware verification and automated theorem proving*, pages 440–464, 1989.

[128] M. Nagasaki, A. Doi, H. Matsuno, and S. Miyano. Genomic Object Net: A platform for modelling and simulating biopathways. *Applied Bioinformatics*, 2:181–184, 2004.

[129] M. Nagasaki, A. Doi, H. Matsuno, and S. Miyano. A versatile petri net based architecture for modeling and simulation of complex biological processes. *Genome Inform*, 15(1):180–197, 2004.

[130] M. Nagasaki, A. Saito, C. Li, E. Jeong, and S. Miyano. Systematic reconstruction of transpath data into cell system markup language. *BMC Systems Biology*, 2(1), 2008.

[131] G.C. Necula. Proof-Carrying Code. In *Procs. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL '97)*, pages 106–119, New York, NY, USA, 1997. ACM.

[132] The Open Group. Unix Regular Expressions. Available at: `http://www.opengroup.org/onlinepubs/7908799/xbd/re.html`.

[133] G. Paun. *Membrane Computing: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[134] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[135] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[136] C. Priami and P. Quaglia. Beta Binders for Biological Interactions. In *Proc. of Computational Methods in System Biology (CMSB '04)*, volume 3082 of *Lecture Notes in Computer Science*, pages 20–33. Springer Berlin, Heidelberg, 2005.

[137] C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80(1):25–31, 2001.

[138] U. S. Reddy. Rewriting Techniques for Program Synthesis. In *Proc. of Rewriting Techniques and Applications, (RTA'89)*, volume 355 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 1989.

[139] A. Regev, E. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004.

[140] A. Regev and E. Shapiro. Cellular abstractions: Cells as computation. *Nature*, 419(6905):343, 2002.

[141] K. Reinhardt. Reachability in petri nets with inhibitor arcs. *Electronic Notes in Theoretical Computer Science*, 223:239–264, 2008.

[142] P. Réty. Improving Basic Narrowing Techniques and Commutation Properties. In *Rewriting Techniques and Applications*, volume 256 of *LNCS*, pages 228–241, 1987.

[143] E. Sandewall. *Features and Fluents, volume 1.* Oxford University Press, New York, NY, USA, 1994.

[144] T. Schlieder. ApproXQL:Design and Implementation of an Approximate Pattern Matching Language for XML. Technical Report B 01-02, Freie Universität Berlin, 2001.

[145] T. Schlieder and H. Meuss. Querying and Ranking XML documents. *Journal of the American Society for Information Science and Technology JASIST*, 53(6):489–503, 2002.

[146] H. Seki. Unfold/fold Transformation of General Logic Programs for the Well-Founded Semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.

[147] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: a Practical OWL-DL Reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[148] J. Spiegel, E. D. Pontikakis, S. Budalakoti, and N. Polyzotis. AQAX: A System for Approximate XML Query Answers. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB'06), Seoul, Korea*, pages 1159–1162. ACM Press, 2006.

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

[150] C. Talcott and D.L. Dill. Multiple Representations of Biological Processes. *Transactions on Computational Systems Biology*, 2006.

[151] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Procs. of the 2nd International Conference on Logic Programming, (ICLP '84)*, pages 127–139, 1984.

[152] The Maude Team. Full Maude, 2009. Available at: `http://www.lcc.uma.es/~duran/FullMaude/`.

[153] M. Theobald, R. Schenkel, and G. Weikum. TopX and XXL at INEX 2005. In *Initiative for the Evaluation of XML Retrieval (INEX'05)*, pages 282–295. IEEE Computer Society, 2005.

[154] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and J. S. Aitken. KAoS Policy Management for Semantic Web Services. *IEEE Intelligent Systems*, 19(4):32–41, 2004.

[155] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[156] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[157] A. Vargun. *Code-Carrying Theory*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 2006. Adviser-D.R., Musser.

[158] A. Vargun and D.R. Musser. Code-Carrying Theory. In *ACM Symposium on Applied Computing*, pages 376–383, New York, NY, USA, 2008. ACM.

[159] P. Viry. Rewriting: An Effective Model of Concurrency. In *Procs. of the 6th International Conference on Parallel Architectures and Languages Europe, (PARLE '94)*, pages 648–660, London, UK, 1994. Springer-Verlag.

[160] E. Visser. A Survey of Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57(2), 2001.

[161] W3C. Web Services Policy 1.2 - framework (WS-Policy), 2006. Available at: `http://www.w3.org/Submission/WS-Policy/`.

[162] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, 1999. Available at: `http://www.w3.org/`.

[163] World Wide Web Consortium (W3C). XML Path Language (XPath), 1999. Available at: `http://www.w3.org/`.

[164] World Wide Web Consortium (W3C). XQuery: A Query Language for XML, 2001. Available at: `http://www.w3.org/`.

[165] World Wide Web Consortium (W3C). OWL Web Ontology Language Guide, 2004. Available at: `http://www.w3.org/`.

[166] World Wide Web Consortium (W3C). RDF Vocabulary Description Language 1.0: RDF Schema, 2004. Available at: `http://www.w3.org/`.

[167] C. Zhao, N. Heilili, S. Liu, and Z. Lin. Representation and Reasoning on RBAC: A Description Logic Approach. In *Proc. of the 2nd Int'l Colloquium on Theoretical Aspects of Computing (ICTAC '05)*, volume 3722 of *Lecture Notes in Computer Science*, pages 381–393. Springer, 2005.